BUKU AJAR

ALGORITMA PEMROGRAMAN DAN STRUKTUR DATA MENGGUNAKAN PYTHON

Yunianita Rahmawati, S.Kom., M.Kom.

Hamzah Setiawan, S.Kom., M.Kom.

UMSIDA PRESS

BUKU AJAR
ALGORITMA PEMROGRAMAN DAN STRUKTUR DATA MENGGUNAKAN PYTHON

Penulis:

Yunianita Rahmawati, S.Kom., M.Kom.

Hamzah Setiawan, S.Kom., M.Kom.

Anggota APPTI Nomor : 002.018.1.09.2017

Anggota IKAPI Nomor : 218/Anggota Luar Biasa/JTI/2019

Diterbitkan oleh

UMSIDA PRESS

Jl. Mojopahit 666 B Sidoarjo

ISBN: 978-623-464-141-7

Buku Ajar Algoritma Pemrograman Dan Struktur Data Menggunakan Python
Penulis: Yunianita Rahmawati; Hamzah Setiawan
ISBN: 978-623-464-141-7
Editor: M. Tanzil Multazam & Mahardika Darmawan K.W.
Copy Editor: Wiwit Wahyu Wijayanti
Design Sampul dan Tata Letak: Wiwit Wahyu Wijayanti
Penerbit: UMSIDA Press
Redaksi: Universitas Muhammadiyah Sidoarjo Jl. Mojopahit No.666B Sidoarjo, Jawa Timur

Cetakan Pertama, 2026

KATA PENGANTAR

Buku "**Algoritma Pemrograman dan Struktur Data Menggunakan Python**" ini disusun sebagai panduan untuk memahami konsep dasar dalam bidang algoritma dan struktur data. Buku ini ditujukan untuk mahasiswa dan pembaca yang tertarik dalam mempelajari teknik pemrograman dengan menggunakan bahasa pemrograman Python.

Python dipilih karena kemudahan penggunaannya, sintaksis yang jelas, dan banyaknya *library* yang mendukung pemrograman algoritma dan struktur data. Buku ini memberikan penjelasan mendalam tentang dasar-dasar algoritma seperti linked list, stack, queue, binary tree, sorting, dan searching.

Pembaca diharapkan dapat memahami implementasi algoritma dan struktur data serta cara mengaplikasikannya untuk menyelesaikan berbagai masalah komputasi. Buku ini juga dilengkapi dengan contoh kode Python yang dapat langsung dijalankan untuk memperkuat pemahaman dan aplikasi materi yang dipelajari.

Diharapkan buku ini dapat menjadi referensi yang bermanfaat bagi pembaca dalam mengembangkan keterampilan di bidang pemrograman, algoritma, dan struktur data, serta untuk pengembangan keahlian dalam menghadapi tantangan di dunia teknologi informasi yang semakin berkembang pesat.

Ucapan terima kasih yang sebesar-besarnya saya sampaikan kepada Dekan Fakultas Sains dan Teknologi, Kaprodi Informatika, serta Sekprodi Informatika, yang telah memberikan dukungan penuh dalam penyusunan buku ajar ini. Terima kasih atas arahan, dorongan, serta kesempatan yang diberikan untuk menulis dan menyusun buku ini. Kepercayaan dan dukungan dari Bapak/Ibu telah sangat membantu dalam mewujudkan buku ajar ini.

Kami juga mengucapkan terima kasih kepada seluruh dosen dan rekan-rekan di Program Studi Informatika, Universitas Muhammadiyah Sidoarjo, yang telah memberikan kontribusi, baik melalui diskusi, saran, dan masukan yang sangat berarti dalam penyusunan buku ini.

Semoga buku ini dapat memberikan manfaat yang besar bagi mahasiswa dan siapa saja yang ingin mendalami dunia pemrograman, khususnya dalam penguasaan algoritma dan struktur data menggunakan bahasa Python.

Sidoarjo, Mei 2026
Tim Penulis

DAFTAR ISI

DAFTAR TABEL

DAFTAR GAMBAR

BAB 1
Algoritma Pemrograman

Algoritma dan struktur data merupakan konsep paling mendasar dalam komputasi serta menjadi komponen utama dalam pembangunan perangkat lunak yang kompleks. Pemahaman terhadap kedua konsep fondasional ini sangat penting dalam perancangan perangkat lunak, yang mencakup setidaknya tiga karakteristik berikut: (1) bagaimana algoritma memanipulasi informasi yang tersimpan di dalam struktur data, dan (2) bagaimana data diorganisasikan serta ditempatkan di dalam memori (Baka, 2017).

1.1 Pengertian Algoritma

Algoritma berasal dari kata algoris dan ritmis, yang diuangkapkan oleh Abu Ja'far Mohammed Ibn Musa Allah Khowarizmi (825 M), dalam buku Al-Jabr Wa-al Muqabla. Algoritma umumnya didefinisikan sebagai **prosedur yang dirumuskan secara formal untuk melakukan suatu perhitungan** (Thareja, 2014). Karena bersifat formal, prosedur tersebut dapat diimplementasikan menggunakan bahasa formal, yaitu bahasa pemrograman. Secara konseptual, **algoritma berfungsi sebagai cetak biru dalam penyusunan program untuk menyelesaikan masalah tertentu**. Algoritma dipandang sebagai prosedur yang efektif karena menyelesaikan masalah melalui sejumlah langkah yang berhingga; dengan kata lain, algoritma yang terdefinisi dengan baik akan selalu menghasilkan keluaran dan dijamin berhenti (*terminate*).

Algoritma juga berperan penting dalam mendukung *software reuse*. Ketika rancangan solusi telah dirumuskan dalam bentuk algoritma, implementasinya dapat dilakukan dalam berbagai bahasa tingkat tinggi, seperti C, C++, Java, atau python. **Algoritma merupakan seperangkat instruksi untuk menyelesaikan suatu masalah**. Untuk satu masalah yang sama, sering kali tersedia lebih dari satu alternatif algoritma; oleh karena itu, pemilihan algoritma yang paling tepat perlu mempertimbangkan efisiensi, khususnya kompleksitas waktu dan kompleksitas ruang yang menyertainya. Contoh algoritma dalam kehidupan sehari-hari yaitu:

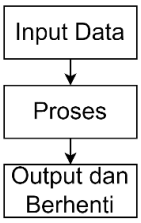
Memasak mi instan:
Step 1: Didihkan air
Step 2: Masukkan mi, masak ±3 menit
Step 3: Siapkan bumbu di piring
Step 4: Tiriskan mi (atau biarkan berkuah sesuai jenis)
Step 5: Campurkan mi dengan bumbu
Step 6: Aduk rata dan sajikan

Mencari buku di rak:
Step 1: Tentukan judul/penulis yang dicari
Step 2: Cari kategori rak yang sesuai
Step 3: Telusuri rak dari urutan abjad atau nomor klasifikasi
Step 4: Jika tidak ditemukan, cek rak lain atau daftar katalog
Step 5: Ambil buku dan catat

Latihan:
Buatlah algoritma lain dalam kehidupan sehari-hari!

Gambar 1.1 menjelaskan **algoritma dalam ilmu komputer** sebagai sekumpulan aturan yang diterapkan dalam suatu program komputer untuk menyelesaikan tugas tertentu. Langkah pertama dalam algoritma ini adalah **input data**, di mana data yang dibutuhkan untuk pemrosesan dimasukkan ke dalam sistem. Setelah itu, proses berikutnya adalah **perhitungan**, di mana data yang telah dimasukkan diproses atau dihitung sesuai dengan aturan atau prosedur yang telah ditentukan dalam algoritma. Langkah terakhir adalah **berhenti ketika jawaban ditemukan atau hasil**, yang berarti algoritma akan berhenti atau selesai setelah memperoleh hasil yang diinginkan. Dengan demikian, algoritma memiliki struktur yang jelas dan sistematis untuk memastikan bahwa tugas dapat diselesaikan dengan efisien dan tepat.

Dua **karakteristik utama** yang menjadikan sebuah algoritma dikategorikan sebagai **algoritma yang baik**, yaitu ***correctness*** (kebenaran) dan **efisiensi**.



Gambar 1.1: Algoritma dalam Ilmu Komputer

1. ***Correctness*** yaitu kemampuan algoritma untuk menghasilkan hasil yang benar dan sesuai dengan tujuan yang telah ditentukan. Artinya, algoritma harus dapat menyelesaikan masalah yang dimaksud secara akurat, tanpa kesalahan atau ketidaksesuaian hasil.
2. **Efisiensi** merupakan **penggunaan sumber daya**, terutama waktu dan memori. Algoritma yang efisien akan menyelesaikan tugasnya dalam waktu yang optimal dan dengan penggunaan memori yang minimal, terutama terhadap data yang besar atau kompleksitas yang tinggi.

Karakteristik penulisan algoritma :

- Tidak menggunakan simbol atau sintaks dari suatu bahasa pemrograman.
- Tidak tergantung dalam suatu bahasa pemrograman.
- Notasi-notasinya dapat digunakan untuk seluruh bahasa manapun

Contoh algoritma dalam menghitung luas segitiga yaitu:

Step 1: Mulai
Step 2: Input nilai alas (a) dan tinggi (t)
Step 3: Hitung luas dengan rumus: Luas=1/2×a×t
Step 4: Tampilkan hasil luas
Step 5: Selesai

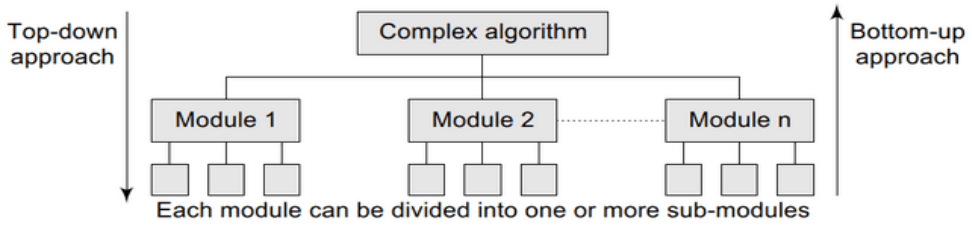
1.2 Pendekatan dalam Merancang Algoritma

Algoritma digunakan untuk memanipulasi data yang terkandung dalam struktur data. Ketika bekerja dengan struktur data, algoritma digunakan untuk melakukan operasi terhadap data yang disimpan. Sebuah algoritma yang kompleks sering kali dibagi menjadi unit-unit yang lebih kecil yang disebut **modul**. Proses pembagian algoritma menjadi modul-modul ini disebut ***modularisasi***. Keuntungan utama dari modularisasi adalah sebagai berikut:

- Membuat algoritma yang kompleks menjadi lebih mudah untuk dirancang dan diimplementasikan.
- Setiap modul dapat dirancang secara independen, sehingga saat merancang satu modul, rincian modul lainnya dapat diabaikan. Langkah tersebut dapat meningkatkan kejelasan desain dan menyederhanakan implementasi, *debugging*, pengujian, dokumentasi, serta pemeliharaan algoritma secara keseluruhan.

Terdapat dua pendekatan utama dalam merancang algoritma yaitu ***top-down approach*** dan ***bottom-up approach***, yang ditampilkan di Gambar 1.2.

Pendekatan **top-down** dimulai dengan **membagi algoritma kompleks menjadi satu atau lebih modul**. Modul-modul ini kemudian dapat didekomposisi lebih lanjut menjadi sub-modul, dan proses dekomposisi ini diulang hingga tingkat kompleksitas modul yang diinginkan tercapai. Method desain *top-down* merupakan bentuk *stepwise refinement*, dimulai dengan modul tingkat atas dan secara bertahap menambahkan modul-modul yang dipanggil oleh modul tersebut. Oleh karena itu, dalam pendekatan *top-down*, dimulai dari desain yang abstrak dan kemudian, setiap langkah, desain ini disempurnakan ke tingkat yang lebih konkret hingga mencapai tingkat yang tidak memerlukan penyempurnaan lebih lanjut.



Gambar 1.2: Pendekatan dalam Merancang Algoritma (Thareja, 2014)

Pendekatan *bottom-up* adalah kebalikan dari *top-down*. Dalam desain *bottom-up*, dimulai dengan merancang modul yang paling dasar atau konkret, lalu melanjutkan ke modul tingkat lebih tinggi. Modul-modul tingkat lebih tinggi diimplementasikan dengan menggunakan operasi yang dilakukan oleh modul-modul tingkat lebih rendah. Dengan demikian, dalam pendekatan ini, sub-modul dikelompokkan bersama untuk membentuk modul tingkat lebih tinggi. Semua modul tingkat lebih tinggi ini digabungkan untuk membentuk modul tingkat yang lebih tinggi lagi. Proses ini diulang hingga desain algoritma lengkap tercapai.

Dalam membandingkan pendekatan top-down dan bottom-up, pilihan pendekatan yang digunakan dapat ditentukan berdasarkan aplikasi yang sedang dikerjakan. Pendekatan top-down lebih dihargai karena kemudahan dalam mendokumentasikan modul-modul, pembuatan kasus uji, implementasi kode, dan debugging. Namun, pendekatan ini juga mendapat kritik karena sub-modul dianalisis secara terisolasi tanpa memperhatikan komunikasi antara modul-modul atau penggunaan kembali komponen-komponennya, dan sedikit perhatian diberikan terhadap data, yang akhirnya mengabaikan konsep *information hiding*. Sebaliknya, pendekatan *bottom-up* untuk *information hiding* karena pertama-tama mengidentifikasi apa yang harus dibungkus dalam sebuah modul dan kemudian menyediakan antarmuka abstrak untuk mendefinisikan batas-batas modul tersebut sebagaimana yang dilihat oleh klien. Meskipun demikian, semua ini sulit dilakukan dalam strategi *bottom-up* yang ketat, sehingga beberapa aktivitas *top-down* tetap perlu dilakukan.

1.3 Struktur Kontrol dalam Algoritma

Sebuah algoritma memiliki jumlah langkah yang terbatas. Beberapa langkah dalam algoritma tersebut mungkin melibatkan pengambilan keputusan dan pengulangan. Secara umum, algoritma dapat menggunakan salah satu dari tiga struktur kontrol berikut:

(a) ***Sequence***, yaitu langkah-langkah yang dieksekusi secara berurutan tanpa percabangan atau pengulangan. Contoh algoritma secara *sequence* dalam menghitung faktorial. Faktorial dari sebuah angka n (dilambangkan dengan n!) adalah hasil perkalian semua angka bulat positif kurang dari atau sama dengan n.

Step 1: Mulai
Step 2: Input nilai n
Step 3: Set faktor_ial=1
Step 4: Untuk i=1 hingga n: faktor_ial= faktor_ial × i
- Step 5: Tampilkan faktor_ial

(b) ***Decision***, algoritma untuk memilih antara dua atau lebih jalur berdasarkan kondisi tertentu, sering kali menggunakan pernyataan *if-else.* Contoh algoritma secara *decision* dalam menentukan kategori usia seseorang, apakah anak-anak, remaja, dewasa, atau lansia.

(c) ***Repetition***, algoritma untuk mengulang serangkaian langkah berdasarkan kondisi tertentu, biasanya menggunakan struktur *loop* seperti *for*, *while*, atau *do-while*. Contoh algoritma secara *repetition* dalam mencetak deret Fibonacci hingga angka ke-n.

Step 1: Mulai
Step 2: Input angka nnn
Step 3: Set a = 0, b = 1, dan i = 1
Step 4: Selama i ≤ n:
- Tampilkan a
- Set temp = a
- Set a = b
- Set b = temp + b
- Increment i = i + 1

Step 5: Selesai

Step 5: Jika 18 ≤ usia < 60, maka
Tampilkan "Dewasa"
Step 6: Jika usia ≥ 60, maka
Tampilkan "Lansia"
Step 7: Selesai

Ketiga struktur kontrol agar algoritma untuk menyelesaikan tugas yang lebih kompleks dengan cara yang lebih terorganisir dan efisien.

Latihan:
Buatlah algoritma lain secara sequence, decision, dan repetition!

1.4 Kompleksitas Worst-case, Average-case, Best-case, dan Amortized Time

Kompleksitas *worst-case, average-case, best-case*, dan *amortized time* dalam konteks algoritma sangat penting untuk memberikan pemahaman yang lebih mendalam mengenai kinerja algoritma di bawah berbagai kondisi input yang berbeda.

Worst-case yaitu perilaku algoritma terkait dengan kasus input yang paling buruk. *Worst-case* dari suatu algoritma merupakan batas atas dari waktu eksekusi untuk setiap input. Oleh karena itu, dengan mengetahui *worst-case*, dapat dipastikan bahwa algoritma tidak akan melebihi batas waktu tersebut.

Average-case adalah perkiraan waktu eksekusi untuk input yang dianggap 'rata-rata'. Waktu eksekusi ini menggambarkan perilaku yang diharapkan dari algoritma ketika input dipilih secara acak dari distribusi tertentu. *Average-case* mengasumsikan bahwa semua input dengan ukuran tertentu memiliki kemungkinan yang sama untuk terjadi.

Best-case merupakan kondisi optimal saat menganalisis kinerja algoritma. Sebagai contoh, kasus terbaik untuk pencarian linear sederhana dalam sebuah array terjadi ketika elemen yang dicari berada dalam posisi pertama dalam daftar. Namun, dalam pengembangan dan pemilihan algoritma untuk menyelesaikan suatu masalah, keputusan tidak pernah sepenuhnya didasarkan terhadap kinerja kasus terbaik. Selalu disarankan untuk memperbaiki kinerja rata-rata dan kinerja kasus terburuk dari suatu algoritma.

Gambar 1.3: Hubungan *Worst-Case, Average-Case,* dan *Best-Case* (Karimov, 2022)

Amortized Time adalah waktu yang diperlukan untuk melakukan urutan operasi terkait, yang dihitung rata-rata atas seluruh operasi yang dilakukan. Analisis teramortisasi menjamin kinerja rata-rata dari setiap operasi meskipun dalam kondisi kasus terburuk.

Hubungan antara *worst-case, average-case, best-case* ditampilkan dalam grafik di Gambar 1.3. Grafik yang ditampilkan menggambarkan kompleksitas waktu algoritma dalam tiga kondisi yang berbeda yaitu *Worst case, Average case,* dan *Best case*. ***Worst case*** menunjukkan bagaimana algoritma berperilaku dalam kondisi terburuk, di mana algoritma membutuhkan waktu eksekusi paling lama. Dalam grafik, *worst case* digambarkan dengan kurva yang paling tinggi, yang menunjukkan bahwa waktu eksekusi meningkat dengan signifikan seiring dengan bertambahnya ukuran input. ***Average case*** menggambarkan waktu eksekusi algoritma dalam kondisi yang lebih realistis, yaitu kondisi rata-rata, yang biasanya lebih baik dibandingkan dengan *worst case*. Kurva Average case terletak di antara *worst case* dan *best case*, mencerminkan kinerja algoritma yang sering ditemui dalam aplikasi sehari-hari. *Best case* menggambarkan waktu eksekusi dalam kondisi optimal, di mana algoritma membutuhkan waktu paling sedikit. ***Best case*** digambarkan dengan kurva yang paling rendah dalam grafik, yang menunjukkan bahwa algoritma beroperasi lebih cepat untuk kondisi yang ideal.

1.5 Ekspresi Kompleksitas Time dan Space

Algoritma terbaik untuk menyelesaikan suatu masalah adalah algoritma yang membutuhkan ruang memori yang lebih sedikit dan waktu eksekusi yang lebih singkat. Namun, dalam praktiknya, merancang algoritma ideal semacam ini bukanlah tugas yang mudah. Sering kali terdapat lebih dari satu algoritma untuk menyelesaikan masalah yang sama, di mana satu algoritma mungkin memerlukan ruang memori yang lebih sedikit, sementara yang lain mungkin memerlukan waktu CPU yang lebih sedikit untuk dieksekusi. Oleh karena itu, tidak jarang suatu hal dikorbankan demi yang lain. Dengan demikian, terdapat kompromi *Time–Space* antar algoritma.

Jika ruang memori menjadi kendala besar, maka seseorang mungkin memilih program yang memerlukan ruang lebih sedikit meskipun mengorbankan lebih banyak waktu CPU. Sebaliknya, jika waktu menjadi kendala utama, maka seseorang mungkin memilih program yang membutuhkan waktu eksekusi minimum meskipun mengorbankan lebih banyak ruang memori.

Kompleksitas *time* dan *space* dapat dinyatakan menggunakan fungsi f(n), di mana n merupakan ukuran input untuk suatu instance masalah yang sedang diselesaikan. Menyatakan kompleksitas diperlukan ketika:

- Diperlukan prediksi terhadap laju pertumbuhan kompleksitas seiring dengan meningkatnya ukuran input dari masalah yang dihadapi.
- Terdapat beberapa algoritma yang dapat menyelesaikan masalah yang sama dan diperlukan pemilihan algoritma yang paling efisien.

Notasi yang paling banyak digunakan untuk menyatakan fungsi f(n) ini adalah notasi **Big O**. Notasi ini memberikan batas atas untuk kompleksitas yang terjadi.

Dalam era kemajuan teknologi komputer yang sangat pesat saat ini, efisiensi algoritma jarang menjadi perhatian utama. Sebaliknya, yang lebih penting adalah mengetahui urutan besar dari algoritma tersebut. Jika terdapat dua algoritma berbeda untuk menyelesaikan masalah yang sama, dimana satu algoritma mengeksekusi dalam 10 iterasi dan yang lainnya dalam 20 iterasi, perbedaan antara kedua algoritma tersebut tidaklah signifikan. Namun, jika algoritma pertama mengeksekusi dalam 10 iterasi dan yang lainnya dalam 1000 iterasi, maka perbedaan tersebut menjadi hal yang perlu diperhatikan.

Jumlah pernyataan yang dieksekusi dalam program untuk n elemen data merupakan fungsi dari jumlah elemen tersebut, yang dinyatakan sebagai f(n). meskipun ada banyak hal yang mempengaruhi kompleksitas, faktor dominan inilah yang paling berpengaruh terhadap kinerja algoritma, terutama dalam hal seberapa cepat algoritma tersebut berjalan ketika ukuran input bertambah besar. Faktor ini dikenal dengan **Big O**, yang dinyatakan sebagai O(n).

Notasi **Big O**, di mana **O** berarti 'order of', berfokus terhadap apa yang terjadi ketika n memiliki nilai yang sangat besar. Sebagai contoh, jika algoritma pengurutan melakukan operasi sebanyak n ^ 2 untuk mengurutkan n elemen, maka algoritma tersebut digambarkan sebagai algoritma O(n ^ 2).

Saat menyatakan kompleksitas menggunakan notasi **Big O**, faktor pengali konstan diabaikan. Sebagai contoh, algoritma O(4n) setara dengan O(n), yang seharusnya dituliskan seperti itu. Jika f(n) dan g(n) adalah fungsi yang didefinisikan sebagai bilangan bulat positif n, maka:

$$f(n) = O(g(n))$$

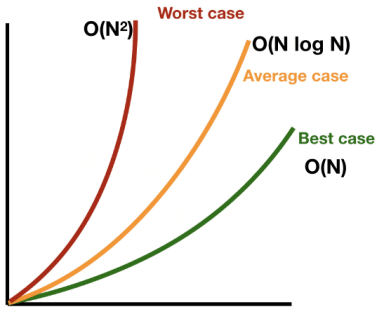
Artinya, f(n) adalah Big–O dari g(n) jika dan hanya jika terdapat konstanta positif c dan n sehingga $f(n) \leq c \cdot g(n)$. Ini berarti bahwa untuk jumlah data yang besar, f(n) akan tumbuh tidak lebih dari faktor konstan dibandingkan dengan g(n). Oleh karena itu, g(n) memberikan batas atas untuk f(n). Perlu dicatat bahwa c adalah suatu konstanta yang bergantung dalam faktor-faktor berikut:

- Bahasa pemrograman yang digunakan,
- Kualitas kompilator atau interpreter,
- Kecepatan CPU,
- Ukuran memori utama dan waktu akses ke memori,
- Pengetahuan programmer, dan
- Algoritma itu sendiri, yang mungkin memerlukan instruksi mesin yang sederhana namun memakan waktu.

Berdasarkan notasi **Big O**, terdapat lima kategori algoritma yang berbeda:

- **Algoritma waktu konstan**: kompleksitas waktu eksekusi yang diberikan adalah **O(1).** Contoh: mengakses elemen tertentu dalam array.
- **Algoritma waktu linier**: kompleksitas waktu eksekusi yang diberikan adalah **O(n).** Contoh: melakukan perulangan melalui elemen array.
- **Algoritma waktu logaritmik**: kompleksitas waktu eksekusi yang diberikan adalah **O(log n).** Contoh: mencari elemen dalam array yang diurutkan.
- **Algoritma waktu polinomial**: kompleksitas waktu eksekusi yang diberikan adalah **O(n^k)** di mana k>1. Contoh: melihat setiap indeks dalam array dua kali.
- **Algoritma waktu eksponensial**: kompleksitas waktu eksekusi yang diberikan adalah **O(2^n).** Contoh: rekursi ganda dalam Fibonacci.

Grafik yang ditampilkan dalam Gambar 1.4 menunjukkan hubungan antara kompleksitas waktu algoritma



Gambar 1.4: Hubungan antara Kompleksitas Waktu Algoritma dengan Notasi Big O (Karimov, 2022)

dengan notasi Big O untuk tiga kondisi yang berbeda yaitu *worst case, average case,* dan *best case*. Grafik tersebut, sumbu horizontal (x-axis) menggambarkan ukuran input (n), sementara sumbu vertikal (y-axis) menggambarkan jumlah operasi atau waktu eksekusi. *Worst case*, diberikan oleh kurva $O(n^2)$ yang menggambarkan kompleksitas waktu kuadratik. Hal ini berarti waktu eksekusi algoritma meningkat secara kuadrat seiring dengan bertambahnya ukuran input. Dalam kondisi ini, algoritma membutuhkan waktu eksekusi yang paling lama. *Average case*, diberikan oleh kurva O(n log n) yang menunjukkan kompleksitas waktu

logaritmik-linear. Waktu eksekusi algoritma meningkat lebih lambat daripada *worst case* karena *average case* menggambarkan kinerja algoritma yang lebih realistis untuk input yang acak dan tidak terlalu buruk. *Best case*, diberikan oleh kurva O(n) yang menunjukkan kompleksitas waktu linear. Dalam kondisi ini, algoritma

Tabel 1.1: Cara Mengukur Kompleksitas Algoritma Menggunakan Notasi Big O

No	Deskripsi	Kompleksitas
Rule 1	Pernyataan penugasan atau *if statement* yang dieksekusi sekali tanpa bergantung terhadap ukuran input atau masalah.	O(1)
Rule 2	Sebuah *for loop* sederhana dari 0 hingga nnn tanpa adanya loop di dalamnya.	O(n)
Rule 3	Sebuah loop bertingkat (*nested loop*) dengan tipe yang sama memerlukan waktu eksekusi yang **kuadratik**.	$O(n^2)$
Rule 4	Sebuah *loop* di mana parameter pengendali dibagi dua setiap langkah eksekusi.	O(logn)
Rule 5	Jika ada beberapa pernyataan yang dieksekusi, jumlahkan kompleksitas waktu masing-masing.	

memerlukan waktu eksekusi yang paling cepat, di mana waktu eksekusi hanya bertambah sebanding dengan ukuran input.

Tabel 1.1 menunjukkan cara mengukur kompleksitas algoritma menggunakan notasi Big O. Kompleksitas $O(1)$ berarti waktu eksekusi tetap konstan, tidak terpengaruh oleh ukuran input. Kompleksitas O(n) berarti waktu eksekusi bertambah sebanding dengan ukuran input n. Kompleksitas $O(n^2)$, yang menunjukkan bahwa waktu eksekusi bertambah secara kuadrat dengan ukuran input n. Kompleksitas O(logn) berarti waktu eksekusi bertumbuh secara logaritmik seiring dengan bertambahnya ukuran input n. Kompleksitas rule 5 kosong mempunyai maksud bahwa jika ada beberapa pernyataan yang terpisah, kompleksitas waktu akan dijumlahkan untuk mendapatkan total kompleksitasnya.

Gambar 1.5 merupakan **diagram kompleksitas waktu algoritma** yang menggambarkan hubungan antara jumlah elemen input (n) dan jumlah operasi yang dibutuhkan untuk menyelesaikan algoritma, yang diwakili oleh sumbu vertikal (operasi) dan sumbu horizontal (elemen). Grafik ini menunjukkan bagaimana algoritma dengan **kompleksitas waktu yang berbeda** berperilaku seiring bertambahnya ukuran input.

1. **O(1) dan O(log n)** (bagian hijau) → **Kompleksitas waktu** algoritma sangat efisien. Waktu eksekusi tidak terpengaruh oleh ukuran input. **O(1)** berarti waktu eksekusi tetap konstan, sementara **O(log n)** menunjukkan algoritma dengan waktu eksekusi yang bertumbuh secara logaritmik, yang sangat efisien terhadap skala input yang besar.

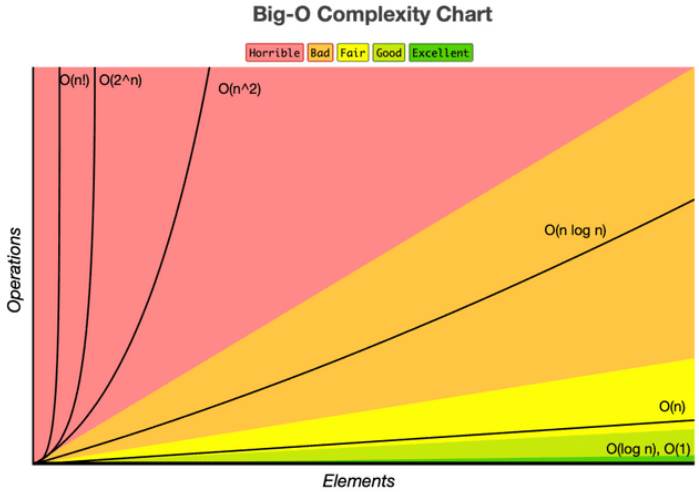
2. **O(n)** (bagian kuning) → Algoritma dengan **kompleksitas waktu linear (O(n))** menunjukkan waktu eksekusi yang bertambah sebanding dengan ukuran input. Meskipun lebih lambat dibandingkan dengan **O(1)** dan **O(log n)**, algoritma dengan **O(n)** masih tergolong efisien untuk sebagian besar aplikasi praktis.
3. **O(n log n)** (bagian oranye). **O(n log n)** adalah **kompleksitas waktu logaritmik-linear** yang menunjukkan waktu eksekusi yang lebih lambat daripada **O(n)** tetapi masih lebih baik dibandingkan **O(n²)**. Algoritma ini biasanya digunakan dalam algoritma pengurutan seperti **merge sort** atau **quick sort**.
4. **O(n²)** (bagian merah muda) → **O(n²)** menunjukkan kompleksitas waktu kuadratik, di mana waktu eksekusi bertambah secara kuadrat seiring dengan bertambahnya ukuran input. Ini adalah kompleksitas waktu yang sangat tidak efisien untuk data dengan ukuran besar, sering ditemukan dalam algoritma dengan dua lapis perulangan, seperti **bubble sort**.
5. **O(2^n) dan O(n!)** (Bagian merah) → **O(2^n)** dan **O(n!)** menunjukkan **kompleksitas waktu eksponensial dan faktorial**, yang sangat tidak efisien. Algoritma dengan kompleksitas ini meningkat sangat cepat dengan bertambahnya ukuran input, sehingga tidak cocok untuk menangani masalah dengan input besar.

Diagram ini menyoroti pentingnya memilih algoritma dengan **kompleksitas waktu yang rendah**, terutama untuk masalah yang melibatkan data dalam jumlah besar. Algoritma dengan **kompleksitas waktu lebih rendah** seperti **O(1)** atau **O(log n)** jauh lebih efisien daripada algoritma dengan **kompleksitas waktu tinggi** seperti **O(n²)** atau **O(2^n)**.

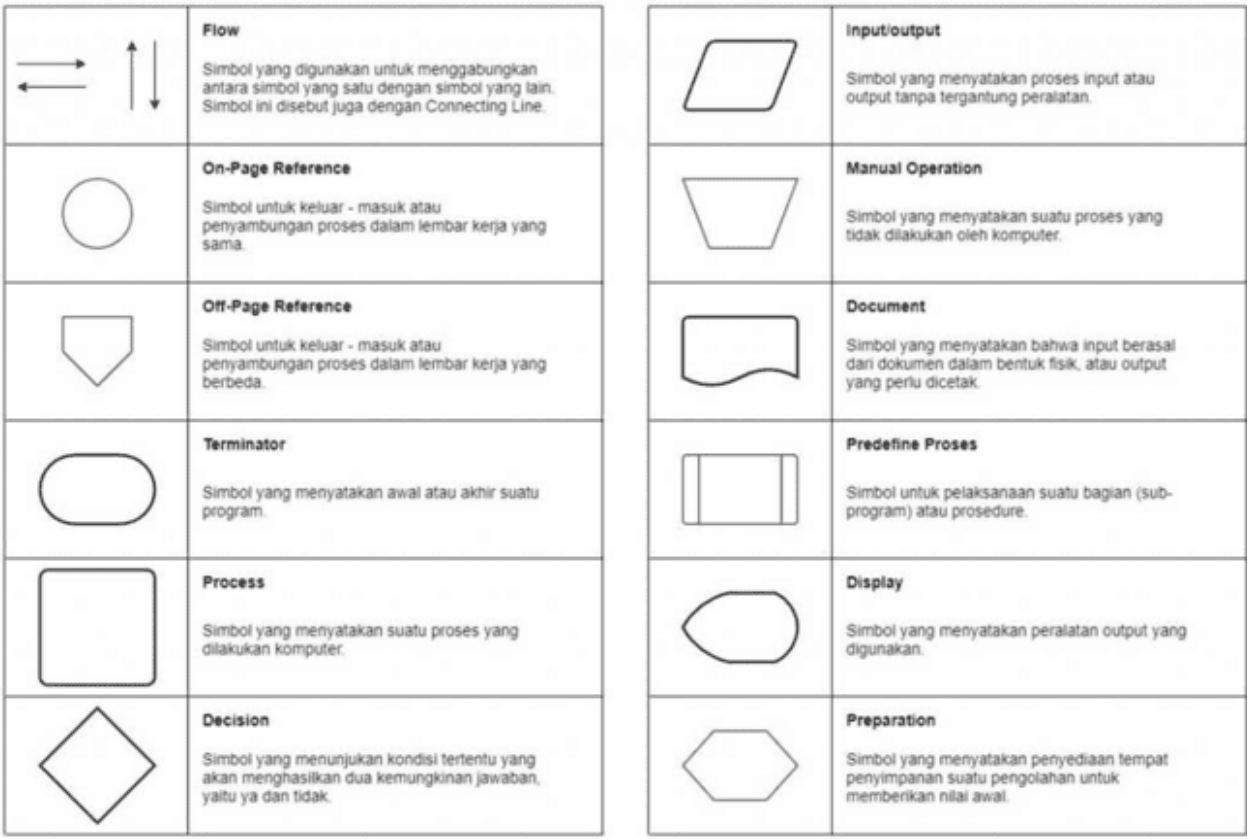
Terdapat beberapa keterbatasan dalam menggunakan notasi **Big O** untuk menyatakan kompleksitas algoritma. Keterbatasan-keterbatasan tersebut adalah sebagai berikut:

- Banyak algoritma yang terlalu sulit untuk dianalisis secara matematis.
- Tidak ada informasi yang cukup untuk menghitung perilaku algoritma untuk kasus rata-rata.
- Analisis **Big O** hanya memberi tahu bagaimana algoritma tumbuh seiring dengan meningkatnya ukuran masalah, namun tidak menunjukkan seberapa efisien algoritma tersebut, karena tidak mempertimbangkan usaha pemrograman.
- **Big O** mengabaikan konstanta-konstanta penting. Sebagai contoh, jika satu algoritma membutuhkan waktu eksekusi **O(n^2)** dan algoritma lainnya membutuhkan waktu **O(100000n^2)** untuk dieksekusi, maka menurut **Big O**, kedua algoritma tersebut memiliki kompleksitas waktu yang setara. Namun, dalam sistem real-time, ini bisa menjadi pertimbangan yang serius.

Perlunya mengabaikan konstanta dan istilah yang tidak dominan dalm pembuatan program yaitu:



Gambar 1.5: Kompleksitas Waktu Eksekusi Algoritma (Karimov, 2022)

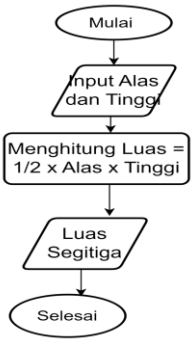


Gambar 1.6: Simbol Flowchart

1) Kode dengan kompleksitas waktu O(N) dapat lebih cepat dibandingkan dengan kode O(1) untuk input tertentu.
   Meskipun notasi Big O menunjukkan bahwa O(N) memiliki waktu eksekusi lebih besar dibandingkan dengan O(1), kenyataannya, kode dengan O(N) bisa jadi lebih efisien dalam beberapa kondisi tertentu, tergantung karakteristik input yang digunakan.
2) Komputer dengan arsitektur yang berbeda memiliki faktor konstan yang bervariasi.
   Setiap komputer dengan spesifikasi dan arsitektur yang berbeda dapat memiliki nilai konstan yang berbeda, yang mempengaruhi kinerja algoritma. Faktor-faktor ini dapat memengaruhi waktu eksekusi secara signifikan, namun tidak tercermin dalam notasi Big O yang lebih memfokuskan terhadap pertumbuhan waktu eksekusi seiring dengan peningkatan ukuran input.
3) Algoritma dengan ide dasar yang sama dan kompleksitas komputasi yang setara mungkin memiliki konstanta yang sedikit berbeda.
   Dua algoritma dengan kompleksitas waktu yang sama, misalnya keduanya O(N), dapat memiliki waktu eksekusi yang berbeda karena perbedaan dalam konstanta atau implementasi rinciannya. Meskipun kompleksitasnya serupa, algoritma yang satu bisa lebih cepat daripada yang lain karena perbedaan dalam pengelolaan konstanta-konstanta tersebut.

1.6 Penyajian Algoritma

Ada beberapa cara yang digunakan untuk menyajikan algoritma, baik dalam bentuk grafis atau tulisan. Dalam bentuk grafis, beberapa method yang dapat digunakan termasuk ***structure chart, hierarchy plus input-process-output, flowchart,*** **dan** ***Nassi-Schneiderman chart***. Sementara itu, dalam bentuk tulisan, method yang digunakan antara lain *English Structure* dan *pseudocode*. Akhirnya, teknik yang digunakan sangat bergantung



Gambar 1.7: Flowchart Menghitung Luas Segitiga

dalam kreatifitas dan kemampuan pemrogram dalam menyajikan algoritma. Seiring berjalannya waktu, banyak method yang berkembang, ***flowchart*** dan ***pseudocode*** menjadi dua method yang paling banyak digunakan.

A. Flowchart

Flowchart adalah diagram yang digunakan untuk menggambarkan alur atau langkah-langkah dalam suatu proses atau algoritma secara visual. Dengan menggunakan simbol-simbol grafis, flowchart memudahkan pemahaman terhadap langkah-langkah yang harus dilakukan dalam sebuah sistem atau algoritma. Flowchart sering digunakan dalam berbagai bidang, termasuk pengembangan perangkat lunak, analisis sistem, dan manajemen proses, karena kemampuannya untuk menyajikan informasi secara jelas dan terstruktur. Dalam flowchart, berbagai simbol digunakan untuk mewakili jenis operasi atau tindakan tertentu, ditampilkan di Gambar 1.6. Contoh flowchart menghitung luas segitiga disajikan di Gambar 1.7.

B. Pseudocode

Pseudocode adalah representasi algoritma yang menggunakan bahasa alami yang terstruktur namun tidak terikat terhadap sintaksis bahasa pemrograman tertentu. Pseudocode digunakan untuk menjelaskan langkah-langkah algoritma secara logis tanpa memperhatikan detail teknis bahasa pemrograman. *Pseudocode* berfungsi sebagai alat untuk merancang algoritma dengan cara yang mudah dipahami oleh manusia, baik oleh programmer maupun non-programmer. Fungsi *Pseudocode* yaitu:

- Memudahkan komunikasi antara pengembang, tim, atau pihak yang tidak terbiasa dengan bahasa pemrograman tertentu.
- Menyederhanakan perancangan algoritma dengan mengabaikan aspek teknis dan detail implementasi yang mungkin membingungkan dalam tahap awal pengembangan.
- Membantu dalam pengujian dan debugging algoritma sebelum implementasi akhir.

Penulisan *pseudocode* tidak terikat oleh aturan yang kaku, tetapi biasanya mengikuti struktur umum berikut:

- Langkah pertama dan terakhir harus jelas, biasanya dengan kata kunci *Start* dan *End* atau *Stop*.
- Gunakan deskripsi yang jelas dan sederhana untuk setiap langkah, sering kali dalam bentuk kalimat perintah.
- Gunakan struktur kontrol seperti *if-else, while*, atau *for* untuk menggambarkan keputusan dan pengulangan.
- Variabel dideklarasikan dan operasi dilakukan menggunakan nama yang jelas tanpa perlu mendetailkan tipe data atau implementasi spesifik.
- *Pseudocode* tidak menggunakan sintaks pemrograman yang sebenarnya, tetapi menggunakan logika yang mudah dimengerti.

Penulisan *pseudocode* yang baik adalah:

- Gunakan bahasa yang mudah dipahami dan menghindari istilah teknis yang berlebihan.
- Gunakan pemisahan blok untuk membedakan langkah-langkah dan struktur kontrol.
- Pilih nama untuk variabel atau proses yang menggambarkan fungsinya secara jelas.

Berikut contoh pseudocode menghitung luas segitiga.

```
Start
 Input nilai alas (a)
 Input nilai tinggi (t)
 luas = (1/2) * a * t
 Output luas
End
```

Latihan:

a. Buatlah flowchart dan pseudocode tentang:
b. Menghitung faktorial.
c. Menentukan bilangan prima.
d. Menampilkan Deret Fibonacci.

BAB 2
Struktur Data

Tujuan dari perancangan program yang baik adalah untuk menciptakan program yang:

- Berfungsi dengan benar,
- Mudah dibaca dan dipahami,
- Mudah diperbaiki (*debug*),
- Mudah dimodifikasi.

Sebuah program harus menghasilkan hasil yang benar, namun selain itu, program tersebut juga harus berjalan dengan efisien. Program dikatakan efisien apabila dapat dieksekusi dalam waktu yang minimal dan menggunakan ruang memori yang minimal. Untuk menghasilkan program yang efisien, penerapan konsep-konsep tertentu dalam pengelolaan data sangat diperlukan. Pengelolaan data merupakan tugas yang kompleks yang mencakup kegiatan seperti pengumpulan data, pengorganisasian data dalam struktur yang sesuai, serta pengembangan dan pemeliharaan rutinitas untuk memastikan kualitas. **Struktur data merupakan bagian yang sangat penting** dalam pengelolaan data, dan dalam buku ini, topik tersebut akan menjadi fokus utama. Secara dasar, struktur data merupakan sekumpulan elemen data yang dikelompokkan dalam satu nama, dan mendefinisikan cara tertentu dalam menyimpan dan mengorganisasi data dalam komputer agar dapat digunakan dengan efisien (Thareja, 2014).

Struktur data digunakan dalam hampir setiap program atau sistem perangkat lunak. Beberapa contoh umum struktur data meliputi array, linked list, queue, stack, pohon biner, dan tabel hash. Struktur data diterapkan secara luas dalam berbagai bidang, seperti:

- Desain kompilator
- Sistem operasi
- Paket analisis statistik
- Sistem manajemen basis data (DBMS)
- Analisis numerik
- Simulasi
- Kecerdasan buatan
- Grafika

Dalam mempelajari DBMS, akan ditemukan bahwa struktur data utama yang digunakan dalam model data Network adalah graf, dalam model data Hierarchical adalah pohon, dan dalam RDBMS adalah array. Struktur data tertentu merupakan komponen penting dalam banyak algoritma efisien karena pemrogram untuk mengelola data dalam jumlah besar dengan mudah dan efisien. Beberapa method desain formal dan bahasa pemrograman menekankan struktur data dan algoritma sebagai faktor utama dalam perancangan perangkat lunak. Hal ini disebabkan karena representasi informasi merupakan aspek mendasar dalam ilmu komputer. Tujuan utama dari sebuah program atau perangkat lunak bukan hanya untuk melakukan perhitungan atau operasi, melainkan untuk menyimpan dan mengambil informasi dengan cepat. Terlepas dari masalah yang dihadapi, penerapan struktur data yang tepat memberikan solusi yang paling efisien. Suatu solusi dikatakan efisien jika dapat menyelesaikan masalah dalam batasan sumber daya yang ditentukan, seperti ruang penyimpanan data yang tersedia dan waktu yang diberikan untuk melaksanakan setiap subtugas. Solusi terbaik adalah yang membutuhkan sumber daya lebih sedikit dibandingkan dengan alternatif yang diketahui. Selain itu, biaya dari suatu solusi adalah jumlah sumber daya yang dikonsumsinya, yang umumnya diukur berdasarkan satu sumber daya utama, seperti waktu, dengan asumsi bahwa solusi tersebut memenuhi batasan sumber daya lainnya.

Saat ini, pemrogram komputer tidak hanya menulis program untuk menyelesaikan masalah, melainkan untuk menulis program yang efisien. Untuk itu, langkah yang diambil adalah menganalisis masalah untuk menentukan tujuan kinerja yang harus dicapai, lalu mencari struktur data yang paling tepat untuk tugas tersebut. Namun, desainer program yang memiliki pemahaman terbatas tentang konsep struktur data sering mengabaikan langkah analisis ini dan menerapkan struktur data yang paling nyaman. Struktur data yang diterapkan tidak tepat untuk masalah yang ada, sehingga dapat mengakibatkan kinerja yang buruk (seperti kecepatan operasi yang lambat). Sebaliknya, jika suatu program mencapai tujuan kinerjanya dengan menggunakan struktur data yang sederhana, maka tidak ada alasan untuk menggunakan struktur data yang lebih kompleks hanya untuk menunjukkan keterampilan pemrogram. Dalam memilih struktur data untuk menyelesaikan suatu masalah, langkah-langkah berikut harus dilakukan:

- Menganalisis masalah untuk menentukan operasi dasar yang harus didukung. Sebagai contoh, operasi dasar dapat mencakup penyisipan, penghapusan, atau pencarian item data dalam struktur data.
- Mengkuantifikasi batasan sumber daya untuk setiap operasi.
- Memilih struktur data yang paling sesuai dengan kebutuhan tersebut.

Pendekatan tiga langkah ini dalam memilih struktur data yang tepat untuk masalah yang dihadapi mendukung pandangan yang berfokus terhadap data dalam proses desain. Dalam pendekatan ini, perhatian pertama tertuju terhadap data dan operasi yang harus dilakukan terhadap data tersebut. Perhatian kedua adalah representasi data, dan perhatian terakhir adalah implementasi dari representasi tersebut. Terdapat berbagai jenis struktur data yang didukung oleh bahasa Python. Sementara satu jenis struktur data hanya menambah item data di awal, jenis lainnya dapat menambahkannya di posisi manapun. Beberapa struktur

Gambar 2.1: Ilustrasi Struktur Data (Karimov, 2022)

data mengakses data secara berurutan, sementara yang lain dapat mengakses data secara acak. Oleh karena

itu, **pemilihan struktur data yang tepat untuk suatu masalah merupakan keputusan penting dan dapat berdampak besar terhadap kinerja program**.

Gambar 2.1 menjelaskan beberapa contoh struktur data, yang merupakan cara yang berbeda untuk mengorganisasi data di dalam komputer agar dapat digunakan secara efektif. Gambar pertama, terdapat ilustrasi mengenai tumpukan kayu yang tidak terorganisir dengan baik, yang menggambarkan data yang belum terstruktur atau tidak terorganisir dengan cara yang efisien. Di sisi lain, gambar kedua menunjukkan tumpukan kayu yang sudah teratur dan rapi, yang menggambarkan bagaimana data yang telah diatur dengan struktur data yang tepat dapat memberikan manfaat dalam pengelolaan dan pemrosesan data. Dengan adanya struktur data yang tepat, data dapat dikelola dengan lebih efisien, mempermudah akses dan pengolahan, serta memaksimalkan penggunaan sumber daya komputer. Sebagai analogi, gambar ketiga memperlihatkan kerumunan orang yang tidak teratur di sebelah kiri dan antrian orang yang teratur di sebelah kanan, yang menggambarkan perbedaan antara data yang tidak terorganisir dan data yang terstruktur dengan baik, di mana data yang terstruktur dapat diproses lebih cepat dan mudah.

2.1 Dasar struktur data

Struktur Data merupakan berbagai cara untuk mengorganisasi data di komputer, yang dapat digunakan secara efektif (Karimov, 2022). Struktur data adalah **blok dasar** dari suatu program (Thareja, 2014). Sebuah program yang dibangun menggunakan struktur data yang tidak tepat mungkin tidak akan berfungsi sesuai dengan yang diharapkan. Oleh karena itu, sebagai seorang programmer, pemilihan struktur data yang paling tepat untuk sebuah program menjadi kewajiban. Istilah **data** merupakan nilai atau sekumpulan nilai. Contoh, nilai suatu variabel atau konstanta (misalnya, nilai ujian siswa, nama karyawan, alamat pelanggan, nilai pi, dan sebagainya).

Sebuah item data yang tidak memiliki elemen data di dalamnya digolongkan sebagai **item elemental**, sementara item yang terdiri dari satu atau lebih elemen data disebut **item grup**. Contohnya, nama seorang siswa dapat dipecah menjadi tiga bagian yaitu nama depan, nama tengah, dan nama belakang. Sedangkan nomor induk siswa biasanya dianggap sebagai satu kesatuan item.

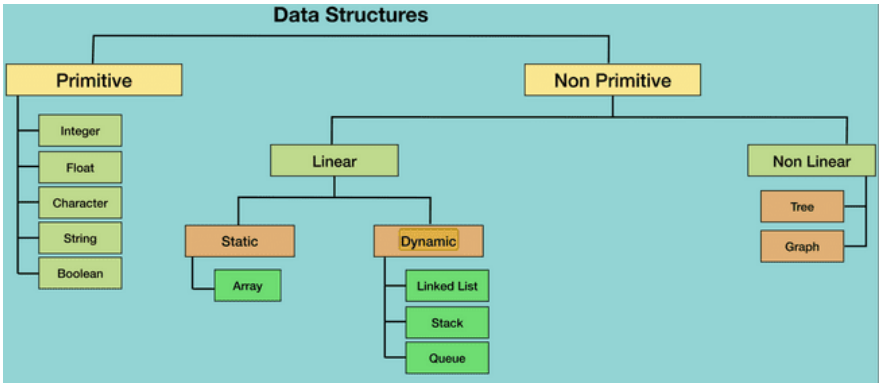
Record adalah kumpulan item data. Sebagai contoh, nama, alamat, mata pelajaran, dan nilai yang diperoleh adalah item data individu. Namun, semua item data ini dapat dikelompokkan bersama untuk membentuk sebuah ***record***. Sebuah **file** adalah kumpulan rekaman yang terkait. Misalnya, jika terdapat 60 siswa di sebuah kelas, maka terdapat 60 rekaman siswa tersebut. Semua rekaman yang terkait ini disimpan dalam sebuah file. Demikian juga, dapat terdapat file yang berisi data semua karyawan di suatu organisasi, file yang berisi data semua pelanggan sebuah perusahaan, file yang berisi data semua pemasok, dan seterusnya.

Selain itu, setiap *record* dalam sebuah file dapat terdiri dari beberapa item data, namun nilai dari suatu item data tertentu secara unik mengidentifikasi rekaman dalam file tersebut. Item data seperti ini disebut ***primary key***, dalam kolom tersebut disebut sebagai **kunci** atau **nilai kunci**. Sebagai contoh, dalam rekaman seorang siswa yang mencakup nomor induk, nama, alamat, mata pelajaran, dan nilai yang diperoleh, kolom nomor induk adalah ***primary key***. Kolom lainnya (nama, alamat, mata pelajaran, dan nilai) tidak dapat berfungsi sebagai kunci utama, karena dua atau lebih siswa bisa memiliki nama yang sama, alamat yang sama (karena tinggal di tempat yang sama), atau terdaftar dalam mata pelajaran yang sama, atau memiliki nilai yang sama.

2.2 Tipe-Tipe Struktur Data

Gambar 2.2 menggambarkan struktur data yang dibagi menjadi dua kategori utama, yaitu primitive dan non-primitive. Struktur data primitive yaitu struktur data jenis ini mencakup tipe data dasar yang langsung digunakan dalam bahasa pemrograman, seperti:

- Integer: angka bulat, seperti 1, 2, 3.
- Float: angka desimal, seperti 3.14, 2.56.
- Character: karakter tunggal, seperti 'a', 'b', '9'.
- String: sekuens karakter, seperti "hello".
- Boolean: nilai logika yang hanya bisa berupa true atau false.



Gambar 2.2: Tipe-Tipe Struktur Data (Karimov, 2022)

Struktur data non-primitive adalah struktur data ini lebih kompleks dan dapat dibagi lagi menjadi dua kategori berdasarkan pengorganisasiannya:

1. Linear asalah data yang disusun dalam urutan linier atau berurutan, yang terdiri dari dua subkategori:
   - Static merupakan struktur data yang memiliki ukuran tetap, seperti array. Dalam array, jumlah elemen yang dapat disimpan telah ditentukan sebelumnya.
   - Dynamic adalah struktur data yang dapat berubah ukurannya, seperti linked list, stack, dan queue. Linked list untuk penyimpanan data yang lebih fleksibel dengan menghubungkan elemen-elemen secara dinamis. Stack dan queue adalah struktur data yang mengatur data dengan prinsip *last in first out* (LIFO) dan *first in first out* (FIFO), masing-masing.
2. Non-Linear yaitu data yang disusun tidak dalam urutan linier, yang mencakup struktur data seperti tree dan graph. Tree mengorganisasi data dalam bentuk hirarki dengan elemen yang terhubung satu sama lain, sedangkan graph menghubungkan elemen-elemen dalam pola yang lebih kompleks dan tidak berurutan.

BAB 3
Bahasa Pemrograman Python

Dalam buku ini, melakukan uji coba pemrograman menggunakan bahasa pemrograman Python. Editornya menggunakan google colab, jupyter, atau visual code. Python adalah bahasa pemrograman yang populer, yang diciptakan oleh Guido van Rossum dan pertama kali dirilis tahun 1991. Bahasa ini digunakan untuk berbagai keperluan, di antaranya pengembangan web (*server-side*), pengembangan perangkat lunak, matematika, dan scripting sistem (Das et al., 2024).

Apa yang dapat dilakukan oleh Python? Python dapat digunakan di server untuk membuat aplikasi web. Python juga dapat digunakan bersama perangkat lunak untuk membuat alur kerja. Selain itu, Python dapat terhubung ke sistem basis data dan juga dapat membaca serta mengubah file. Python juga dapat digunakan untuk menangani data besar dan melakukan perhitungan matematika yang kompleks. Python juga mendukung pembuatan prototipe dengan cepat dan pengembangan perangkat lunak siap produksi.

Mengapa memilih Python? Python dapat dijalankan dalam berbagai platform (Windows, Mac, Linux, Raspberry Pi, dan lainnya). Python memiliki sintaks yang sederhana dan mirip dengan bahasa Inggris. Sintaks Python membuat pengembang menulis program dengan lebih sedikit baris kode dibandingkan dengan beberapa bahasa pemrograman lainnya. Python berjalan dalam sistem interpreter agar kode dieksekusi segera setelah ditulis, sehingga proses pembuatan prototipe dapat dilakukan dengan cepat. Python dapat digunakan secara prosedural, berorientasi objek, maupun fungsional.

Versi utama terbaru Python adalah Python 3, yang akan digunakan dalam tutorial ini. Dalam tutorial ini, Python akan ditulis menggunakan editor teks. Namun, Python juga dapat ditulis dalam *Integrated Development Environment* (IDE) seperti Thonny, Pycharm, NetBeans, atau Eclipse, yang sangat berguna saat mengelola koleksi file Python dalam jumlah besar.

Python dirancang untuk kemudahan dibaca, dengan beberapa kesamaan dengan bahasa Inggris dan pengaruh dari matematika. Python menggunakan baris baru untuk menyelesaikan sebuah perintah, berbeda dengan bahasa pemrograman lain yang sering menggunakan tanda titik koma atau tanda kurung. Python mengandalkan indentasi dan spasi putih untuk mendefinisikan ruang lingkup, seperti ruang lingkup loop, fungsi, dan kelas. Sebaliknya, bahasa pemrograman lain sering menggunakan tanda kurung kurawal untuk tujuan ini.

3.1 Konsep Dasar Pemrograman Python

Sebagai langkah awal, penting untuk memahami konsep dasar Python seperti sintaks dan aturan penulisan kode yang benar. Pemahaman mengenai cara menulis kode Python yang benar sangat penting untuk memulai pemrograman. Python dikenal dengan sintaks yang sederhana dan mudah dibaca. Di antaranya adalah penggunaan indentasi untuk menandai blok kode yang harus dipahami dengan baik.

A. Eksekusi Sintaks Python

Sintaks Python dapat dieksekusi dengan cara menuliskannya langsung di *command line*. Atau dengan membuat file Python di server, menggunakan ekstensi file .py, dan menjalankannya melalui *command line.*

B. Indentasi Python

Indentasi merupakan spasi yang digunakan di awal baris kode. Di beberapa bahasa pemrograman lain, indentasi hanya digunakan untuk memperbaiki keterbacaan kode, namun dalam Python, indentasi sangat penting. Python menggunakan **indentasi** untuk menandakan **sebuah blok kode**. Jika indentasi dilewatkan, Python akan memberikan error. Jumlah spasi yang digunakan tergantung programmer, dengan

empat spasi sebagai penggunaan yang paling umum, namun setidaknya harus menggunakan satu spasi. **Jumlah spasi** yang digunakan dalam **satu** blok **kode harus konsisten**, jika tidak, Python akan memberikan error.

C. Komentar

Komentar dimulai dengan **simbol #**, dan **Python akan menganggap sisa baris sebagai komentar**. Fungsi komentar yaitu:

a. Komentar dapat digunakan untuk menjelaskan kode Python.
b. Komentar dapat sebagai dokumentasi dalam kode.
c. Komentar dapat membuat kode lebih mudah dibaca.
d. Komentar dapat digunakan untuk mencegah eksekusi kode saat pengujian.

Komentar dapat ditempatkan di akhir baris, dan Python akan mengabaikan sisa baris tersebut. Komentar tidak harus berupa teks yang menjelaskan kode, tetapi juga bisa digunakan untuk mencegah Python menjalankan kode.

Python tidak memiliki sintaks khusus untuk komentar multibaris. Agar dapat menambahkan komentar multibaris, dapat menambahkan # di setiap baris. Atau **dapat menggunakan string multibaris atau tanda petik ganda sebanyak tiga kali** (“““) **di awal dan akhir dari suatu pernyataan**. Karena Python akan mengabaikan literal string yang tidak ditugaskan ke variabel. Selama string tersebut tidak ditugaskan ke variabel, Python akan membaca kode tersebut, tetapi kemudian mengabaikannya, dan komentar multibaris telah dibuat.

D. Statement

Program komputer merupakan sekumpulan "instruksi" yang akan "dieksekusi" oleh komputer. Dalam bahasa pemrograman, instruksi-instruksi ini disebut sebagai ***statement***. Di Python, **sebuah pernyataan biasanya berakhir saat barisnya selesai**. Tidak diperlukan penggunaan tanda titik koma (;) seperti dalam bahasa pemrograman lain (Java atau C). Sebagian besar program Python mengandung banyak pernyataan. **Pernyataan-pernyataan tersebut dieksekusi satu per satu, sesuai urutan penulisannya**.

E. Titik Koma

Titik koma bersifat opsional di Python. **Programmer bisa menulis beberapa pernyataan dalam satu baris dengan memisahkannya menggunakan titik koma (;)**, tetapi ini jarang digunakan karena akan membuat kode menjadi sulit dibaca. Contoh“:

```
print("Assalamu’alaikum"); print("Semangat!!!!!!"); print("Selesaikan apa yang kalian mulai!")
```

Namun, jika menempatkan dua pernyataan dalam satu baris tanpa pemisah (baris baru atau titik koma), Python akan memberikan error.

F. Menampilkan Hasil Python

Hasil yang telah diproses perlu ditampilkan agar pengguna dapat mengamati output atau hasil dari program yang telah dijalankan. Salah satu method yang dapat digunakan untuk menampilkan hasil tersebut dalam Python adalah dengan memanfaatkan fungsi print(). Fungsi ini memungkinkan pengguna untuk mencetak teks atau nilai hasil perhitungan ke layar, sehingga memberikan umpan balik langsung terkait proses yang telah dilaksanakan. Dengan demikian, hasil yang diinginkan dapat terlihat dengan jelas, baik dalam bentuk angka, teks, maupun kombinasi keduanya.

♣ **Mencetak Teks**

Fungsi **print()** dapat digunakan untuk menampilkan teks atau nilai output. Fungsi *print()* dapat digunakan sebanyak yang diinginkan. Setiap pemanggilan fungsi ini akan mencetak teks di baris baru secara default.

♣ **Tanda Kutip Ganda**

Teks di Python harus berada di dalam tanda kutip ganda " atau tanda kutip tunggal '. Jika programmer lupa meletakkan teks di dalam tanda kutip, Python akan memberikan error.

♣ **Mencetak Tanpa Baris Baru**

Secara default, **fungsi print() akan diakhiri dengan baris baru**. Jika ingin mencetak beberapa kata dalam satu baris, bisa menggunakan parameter **end=**.

♣ **Mencetak Angka**

Fungsi print() untuk menampilkan angka. Namun, berbeda dengan teks, angka tidak perlu diletakkan dalam tanda kutip ganda. Dapat melakukan perhitungan matematika di dalam fungsi print().

♣ **Menggabungkan Teks dan Angka**

Python dapat menggabungkan teks dan angka dalam satu output dengan memisahkannya menggunakan koma. Contoh:

```
print("Saya belanja bulanan dengan total ", 500000, "Rupiah.")
```

3.2 Variabel

Variabel adalah wadah untuk menyimpan nilai data. **Python tidak memiliki perintah khusus untuk mendeklarasikan variabel**. **Sebuah variabel dibuat ketika pertama kali diberi nilai**. **Variabel tidak perlu dideklarasikan dengan tipe tertentu**, dan tipe data variabel bahkan dapat berubah setelah nilainya ditetapkan. Contoh: `x = "Sally"`.

♥ **Casting**

Apabila ingin menentukan tipe data suatu variabel, hal ini dapat dilakukan dengan menggunakan *casting*. Contoh: `y = int(3)`.

♥ **Cetak Tipe Data**

Fungsi **type()** dapat digunakan untuk mengetahui tipe data dari sebuah variabel. Contoh: `print(type(x))`.

♥ **Penggunaan Tanda Kutip**

Variabel bertipe string dapat dideklarasikan dengan menggunakan tanda kutip tunggal atau ganda.

♥ **Case-Sensitive**

Nama variabel bersifat peka terhadap huruf kapital (***case-sensitive***). Contoh: `a = 4` berbeda dengan `A = "Sally"`.

♥ **Nama Variabel**

Sebuah variabel bisa memiliki nama yang pendek (seperti x dan y) atau nama yang lebih deskriptif (seperti age, carname, total_volume).

♥ **Aturan untuk variabel di Python**

Beberapa aturan yang perlu diperhatikan dalam penamaan variabel di Python adalah sebagai berikut:

(a) Nama variabel harus dimulai dengan huruf atau karakter garis bawah (_).
(b) Nama variabel tidak bisa dimulai dengan angka.
(c) Nama variabel hanya boleh mengandung karakter alfanumerik dan garis bawah (A-Z, 0-9, dan _).
(d) Nama variabel bersifat peka terhadap huruf kapital (***case-sensitive***), (misalnya, age, Age, dan AGE adalah variabel yang berbeda).
(e) Nama variabel tidak boleh merupakan kata kunci di Python.

♥ **Nama Variabel dalam Beberapa Kata**

Nama variabel yang terdiri dari lebih dari satu kata bisa sulit dibaca. Ada beberapa teknik yang bisa digunakan untuk membuatnya lebih mudah dibaca, yaitu:

a. Camel Case adalah setiap kata, kecuali yang pertama, dimulai dengan huruf kapital. Contoh: totalVolume.
b. Pascal Case adalah setiap kata dimulai dengan huruf kapital. Contoh: TotalVolume.
c. Snake Case yaitu setiap kata dipisahkan dengan karakter garis bawah. Contoh: total_volume.

♥ **Banyak Nilai untuk Beberapa Variabel**

Python menetapkan nilai ke beberapa variabel dalam satu baris. Contoh: `x, y, z = 10, 20, 30`. Dalam contoh tersebut, nilai 10 diberikan ke x, nilai 20 diberikan ke y, dan nilai 30 diberikan ke z dalam satu baris.

♥ **Satu Nilai untuk Beberapa Variabel**

Nilai yang sama dapat ditetapkan ke beberapa variabel dalam satu baris. Contoh: `x = y = z = 100`. Di sini, nilai 100 diberikan ke x, y, dan z dalam satu baris.

♥ **Membongkar Koleksi**

Apabila memiliki sekumpulan nilai dalam sebuah *list* atau *tuple*, dapat diekstrak ke dalam variabel menggunakan *unpacking*. Contoh:

```
my_tuple = (1, 2, 3)
x, y, z = my_tuple
```

Dalam contoh ini, nilai dari my_tuple (1, 2, 3) dibongkar dan masing-masing nilai diberikan ke x, y, dan z.

♥ **Menampilkan Nilai Variabel**

Berikut adalah beberapa langkah untuk menampilkan nilai dari sebuah variabel yaitu:

a. Fungsi **print()** sering digunakan untuk menampilkan variabel. Contoh:

```
x = 10
print(x)
```

Fungsi print() digunakan untuk menampilkan nilai dari variabel x.

b. Menampilkan beberapa variabel yang dipisahkan dengan koma. Contoh:

```
x = 10
y = 20
print(x, y)
```

Di sini, fungsi print() digunakan untuk menampilkan dua variabel, x dan y, yang dipisahkan oleh koma.

c. Menggunakan operator + untuk menampilkan beberapa variabel. Contoh:

```
x = "Halo"
y = "Dunia"
print(x + " " + y)
```

Dalam contoh ini, operator + digunakan untuk menggabungkan dua variabel string x dan y menjadi satu kalimat yang ditampilkan oleh print().

d. Operator + untuk angka sebagai operator matematika. Contoh:

```
x = 5
y = 3
print(x + y)
```

Dalam contoh ini, operator + digunakan sebagai operator matematika untuk menjumlahkan angka x dan y, hasilnya adalah 8.

e. Menggabungkan string dan angka dengan operator + yang menghasilkan error. Contoh:

```
x = 10
print("Nilai x adalah: " + x)
```

Jika mencoba menggabungkan string dan angka tanpa mengonversi angka menjadi string, Python akan memberikan error.

f. Menampilkan beberapa variabel dengan berbagai tipe data menggunakan koma. Contoh:

```
x = 10
y = "Python"
z = 3.14
print(x, y, z)
```

Cara terbaik untuk menampilkan beberapa variabel dalam fungsi print() adalah dengan memisahkannya menggunakan koma. Dalam contoh ini, variabel x (angka), y (string), dan z (float) ditampilkan dalam satu baris.

♥ **Variabel Global**

Variabel yang dibuat di luar fungsi (seperti seluruh contoh sebelumnya) dikenal sebagai **variabel global**. Variabel global dapat digunakan oleh siapa saja, baik di dalam fungsi maupun di luar fungsi. Apabila membuat variabel dengan nama yang sama di dalam sebuah fungsi, maka variabel tersebut akan bersifat lokal dan hanya dapat digunakan di dalam fungsi tersebut. Variabel global dengan nama yang sama akan tetap seperti semula, bersifat global dan dengan nilai aslinya. Biasanya, ketika membuat sebuah variabel di dalam sebuah fungsi, variabel tersebut akan bersifat lokal dan hanya bisa digunakan di dalam fungsi itu saja. Untuk membuat variabel global di dalam sebuah fungsi, dapat menggunakan kata kunci **global**. Selain itu, gunakan kata kunci global jika ingin mengubah nilai dari variabel global di dalam sebuah fungsi. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

3.3 Tipe data dalam Python

Dalam pemrograman, tipe data adalah konsep yang sangat penting. Variabel dapat menyimpan data dengan tipe yang berbeda, dan setiap tipe data memiliki kegunaan yang berbeda-beda. Python memiliki beberapa tipe data bawaan yang sudah tersedia secara default dan dibagi ke dalam kategori-kategori yang ditampilkan di Tabel 3.1.

A. Number dalam Python

Beberapa operasi yang dapat dilakukan terhadap number dalam python yaitu:

➢ **Konversi Tipe Data**

Python dapat melakukan konversi dari satu tipe data ke tipe data lainnya menggunakan method *int*(), *float*(), dan *complex*(). Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Tabel 3.1: Tipe Data dalam Python

Tipe Data	*Keyword*	Contoh	Keterangan
Teks	str	x = "Selamat Malam"	Tipe data untuk menyimpan teks atau string. Didefinisikan di dalam tanda kutip, baik tunggal (' ') maupun ganda (" ")
Numerik	int	x = 15	Tipe data untuk angka bulat (integer), baik positif maupun negatif.
Numerik	float	x = 5.5	Tipe data untuk angka desimal (floating point).
Numerik	complex	x = 1j	Tipe data untuk angka kompleks yang memiliki bagian real dan imajiner.
Sequence	list	x = ["apel", "pisang", "jeruk"]	Tipe data yang digunakan untuk menyimpan urutan elemen yang dapat diubah dan bisa berisi berbagai tipe data.
Sequence	tuple	x = ("pepaya", "salak", "anggur")	Tipe data yang mirip dengan list, tetapi bersifat tidak dapat diubah (*immutable*).
Sequence	range	x = range(7)	Tipe data untuk menghasilkan urutan angka dalam rentang tertentu, sering digunakan dalam perulangan.
Mapping	dict	x = {"Nama" : "Rahma", "age" : 14}	Tipe data yang menyimpan pasangan kunci-nilai (*key-value pairs*).
Set	set	x = {"mangga", "kelapa", "manggis"}	Tipe data yang menyimpan kumpulan elemen unik tanpa urutan.
Set	frozenset	x = frozenset ({"tomat", "timun", "bayam"})	Mirip dengan set, tetapi bersifat immutable, artinya tidak dapat diubah setelah dibuat.
Boolean	bool	x = True	Tipe data yang hanya memiliki dua nilai, yaitu True dan False. Digunakan untuk operasi logika.
Binary	bytes	x = b"Surabaya"	Tipe data untuk menyimpan data biner yang tidak dapat diubah (*immutable*).
Binary	bytearray	x = bytearray(25)	ipe data untuk menyimpan data biner yang dapat diubah (*mutable*).
Binary	memoryview	x = memoryview (bytes(50))	Tipe data yang memberikan akses untuk melihat dan memanipulasi data biner tanpa salinan.
None	NoneType	x = None	Tipe data yang digunakan untuk menunjukkan bahwa sebuah variabel tidak memiliki nilai atau kosong. Biasanya digunakan untuk menunjukkan ketidakhadiran nilai atau sebagai nilai default.

➢ **Angka Acak (Random)**

Python tidak memiliki fungsi random() untuk menghasilkan angka acak, namun Python menyediakan modul bawaan bernama random, yang dapat digunakan untuk menghasilkan angka acak. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

B. Boolean dalam Python

Tipe data Boolean hanya memiliki dua nilai yaitu True atau False. Beberapa operasi yang dapat dilakukan terhadap boolean dalam python yaitu:

➢ **Nilai Boolean**

Dalam pemrograman, perlu mengetahui apakah suatu ekspresi bernilai True atau False. Python dapat mengevaluasi ekspresi apa pun dan mendapatkan salah satu dari dua hasil: True atau False. Saat membandingkan dua nilai, ekspresi tersebut dievaluasi dan Python akan mengembalikan hasil Boolean. Contoh: `print(11 == 19)`, maksudnya apakah angka 11 sama dengan 19?, hasilnya 'False', karena 11 tidak sama dengan 19. Begitu juga ketika menjalankan kondisi dalam pernyataan if, Python akan mengembalikan nilai True atau False. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

➢ **Mengevaluasi Nilai dan Variabel**

Fungsi bool() untuk mengevaluasi nilai apa pun dan memberikan hasil True atau False sebagai jawaban.

➢ **Kebanyakan Nilai adalah True**

Hampir semua nilai akan dievaluasi menjadi True jika memiliki konten atau isi. String apa pun bernilai True, kecuali string kosong. Angka apa pun bernilai True, kecuali angka 0. List, tuple, set, dan dictionary apa pun bernilai True, kecuali yang kosong.

➢ **Beberapa Nilai adalah False**

Sebetulnya, hanya sedikit nilai yang akan dievaluasi menjadi False, yaitu nilai-nilai kosong seperti (), [], {}, "", angka 0, dan nilai None. Tentu saja, nilai False juga akan dievaluasi sebagai False. Satu lagi nilai atau objek yang akan dievaluasi sebagai False adalah objek yang dibuat dari kelas dengan fungsi __len__ yang mengembalikan 0 atau False.

➢ **Function yang Mengembalikan Boolean**

Python bisa membuat fungsi yang mengembalikan nilai Boolean. Contoh fungsi yang mengembalikan nilai Boolean:

Python juga memiliki banyak fungsi bawaan yang mengembalikan nilai Boolean, seperti fungsi **isinstance()**, yang digunakan untuk menentukan apakah suatu objek merupakan tipe data tertentu. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

C. *None* dalam Python

None adalah sebuah konstanta khusus di Python yang menunjukkan tidak adanya nilai. Tipe data dari None adalah *NoneType*, dan None adalah satu-satunya instance dari objek NoneType. Variabel dapat diberi nilai *None* untuk menunjukkan "tidak ada nilai" atau "belum diatur". unakan fungsi **type() u**ntuk melihat tipe dari nilai None. Gunakan operator identitas **is** atau **is not** untuk membandingkan sebuah nilai dengan *None*. *None* dapat dievaluasi sebagai False dalam konteks Boolean. *Function* tidak secara eksplisit mengembalikan suatu nilai, secara default akan mengembalikan *None*. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

D. Tanggal di Python

Tanggal di Python bukanlah tipe data tersendiri, tetapi dapat impor modul bernama **datetime** untuk bekerja dengan tanggal sebagai objek tanggal. Tanggal dapat mencakup tahun, bulan, hari, jam, menit, detik, dan mikrodetik. Modul **datetime** memiliki banyak method untuk mengembalikan informasi terkait objek tanggal. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

➢ **Membuat Objek Tanggal**

Untuk membuat sebuah tanggal, dapat menggunakan kelas **datetime()** (konstruktor) dari modul datetime. Kelas datetime() memerlukan tiga parameter untuk membuat sebuah tanggal yaitu tahun, bulan, dan hari. Kelas datetime() juga dapat menerima parameter untuk waktu dan zona waktu (jam, menit, detik, mikrodetik, zona waktu), tetapi parameter tersebut bersifat opsional dan memiliki nilai default 0 (None untuk zona waktu). Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

➢ **Method strftime()**

Objek datetime memiliki sebuah method untuk memformat objek tanggal menjadi string yang dapat dibaca. Method ini disebut strftime(), dan menerima satu parameter, yaitu format, untuk menentukan format dari string yang dikembalikan. Format tanggal dan waktu disajikan di Tabel 3.2.

Tabel 3.2: Format Tanggal dan Waktu dalam Python

Simbol	Deskripsi	Contoh
%a	Hari dalam seminggu, versi pendek	Sen
%A	Hari dalam seminggu, versi lengkap	Rabu
%w	Hari dalam minggu sebagai angka 0-6, dengan 0 sebagai Minggu	3
%d	Hari dalam bulan 01-31	31
%b	Nama bulan, versi pendek	Des
%B	Nama bulan, versi lengkap	Desember
%m	Bulan sebagai angka 01-12	12
%y	Tahun, versi pendek, tanpa abad	18
%Y	Tahun, versi lengkap	2018
%H	Jam 00-23	17
%I	Jam 00-12	5
%p	AM/PM	PM
%M	Menit 00-59	41
%S	Detik 00-59	8
%f	Mikrodetik 000000-999999	548513
%z	Offset UTC	100
%Z	Zona waktu	CST
%j	Nomor hari dalam tahun 001-366	365
%U	Nomor minggu dalam tahun, Minggu sebagai hari pertama minggu	52
%W	Nomor minggu dalam tahun, Senin sebagai hari pertama minggu	52
%c	Versi lokal dari tanggal dan waktu	Sen Dec 31 17:41:00 2018
%C	Abad	20
%x	Versi lokal dari tanggal	12/31/18
%X	Versi lokal dari waktu	17:41:00
%%	Karakter %	%
%G	Tahun ISO 8601	2018
%u	Hari dalam minggu ISO 8601 (1-7)	1
%V	Nomor minggu ISO 8601 (01-53)	1

3.4 Operator dalam Python

Operator digunakan untuk melakukan operasi dalam variabel dan nilai. Python membagi operator-operator ke dalam beberapa kelompok berikut:

A. Operator aritmatika → operator yang digunakan dengan nilai numerik untuk melakukan operasi matematika yang umum seperti penjumlahan, pengurangan, perkalian, dan sebagainya. Operator aritmatika disajikan di Tabel 3.3.

Tabel 3.3: Operator Matematika

Operator	Nama	Contoh	Operator	Nama	Contoh	Operator	Nama	Contoh
+	Penjumlahan	a + b	*	Perkalian		-	Pengurangan	a - b
/	Pembagian (menghasilkan nilai float)	a / b	//	Pembagian floor (menghasilkan nilai integer)	a * b	%	Modulus	a % b
					a // b	**	Eksponen	a ** b

B. Operator *assignment* → operator yang digunakan untuk menetapkan nilai ke variabel. Operator *assignment* ditampilkan di Tabel 3.4.

Tabel 3.4: Operator *Assignment*

Operator	Nama Operator	Contoh
=	Penugasan (*Assignment*)	a = 8
+=	Penugasan penjumlahan (*Addition Assignment*)	a += 4
-=	Penugasan pengurangan (*Subtraction Assignment*)	a -= 2
*=	Penugasan perkalian (*Multiplication Assignment*)	a *= 6
/=	Penugasan pembagian (*Division Assignment*)	a /= 2
%=	Penugasan modulus (*Modulus Assignment*)	a %= 3
//=	Penugasan pembagian *Floor* (*Floor Division Assignment*)	a //= 3
**=	Penugasan eksponen (*Exponentiation Assignment*)	a **= 2
&=	Penugasan bitwise AND (*Bitwise AND Assignment*)	a &= 5
|=	Penugasan bitwise OR (*Bitwise OR Assignment*)	a |= 5
^=	Penugasan bitwise XOR (*Bitwise XOR Assignment*)	a ^= 7
>>=	Penugasan bitwise *Right Shift* (*Right Shift Assignment*)	a >>= 2
<<=	Penugasan bitwise *Left Shift* (*Left Shift Assignment*)	a <<= 3
:=	Operator walrus (*Walrus Operator*)	print(a := 5)

- Penugasan (*assignment*): untuk memberikan nilai dalam variabel.
- Penugasan penjumlahan (*addition assignment*): untuk menambahkan nilai dalam variabel dan langsung menetapkannya.
- Penugasan pengurangan (*subtraction assignment*): untuk mengurangi nilai dalam variabel dan langsung menetapkannya.
- Penugasan perkalian (*multiplication assignment*): untuk mengalikan nilai dalam variabel dan langsung menetapkannya.
- Penugasan pembagian (*division assignment*): untuk membagi nilai dalam variabel dan langsung menetapkannya.
- Penugasan modulus (*modulus assignment*): untuk menghitung sisa bagi dan langsung menetapkannya.
- Penugasan pembagian floor (*floor division assignment*): untuk melakukan pembagian floor dan langsung menetapkannya.
- Penugasan eksponen (*exponentiation assignment*): untuk menghitung eksponen dan langsung menetapkannya.
- Penugasan bitwise and (*bitwise* and *assignment*): untuk operasi bitwise and dalam variabel dan langsung menetapkannya.
- Penugasan bitwise or (*bitwise* or *assignment*): untuk operasi bitwise or dalam variabel dan langsung menetapkannya.
- Penugasan bitwise xor (*bitwise* xor *assignment*): untuk operasi bitwise xor dalam variabel dan langsung menetapkannya.
- Penugasan bitwise *right shift* (*right shift assignment*): untuk operasi bitwise right shift dalam variabel dan langsung menetapkannya.

- Penugasan bitwise *left shift* (*left shift assignment*): digunakan untuk operasi bitwise left shift dalam variabel dan langsung menetapkannya.
- Operator walrus (*walrus operator*): untuk menetapkan nilai dalam variabel di dalam ekspresi yang lebih besar.

C. Operator *comparison* → operator yang digunakan untuk membandingkan dua nilai, seperti lebih besar dari, lebih kecil dari, sama dengan, dan sebagainya. Python dapat menggabungkan beberapa operator perbandingan dalam satu ekspresi. Operator *comparison* disajikan di Tabel 3.5.

Tabel 3.5: Operator Comparison

Operator	Nama	Contoh	Fungsi
==	Sama dengan	a == b	Memeriksa apakah dua nilai sama.
!=	Tidak sama dengan	a != b	Memeriksa apakah dua nilai tidak sama.
>	Lebih besar dari	a > b	Memeriksa apakah suatu nilai lebih besar dari nilai lain.
<	Lebih kecil dari	a < b	Memeriksa apakah suatu nilai lebih kecil dari nilai lain.
>=	Lebih besar atau sama dengan	a >= b	Memeriksa apakah suatu nilai lebih besar atau sama dengan nilai lain.
<=	Lebih kecil atau sama dengan	a <= b	Memeriksa apakah suatu nilai lebih kecil atau sama dengan nilai lain.

D. Operator logika → operator yang digunakan untuk operasi logika seperti *and*, *or*, dan *not*. Operator logika disajikan di Tabel 3.6.

Tabel 3.6: Operator Logika

Operator	Deskripsi	Contoh
and	Mengembalikan True jika kedua pernyataan benar	a < 5 and a < 10
or	Mengembalikan True jika salah satu pernyataan benar	a < 5 or a < 4
not	Membalikkan hasil, mengembalikan False jika hasilnya benar	not(a < 5 and a < 10)

Operator logika digunakan untuk menggabungkan pernyataan kondisi. Python memiliki tiga operator logika yaitu:

- and: Menghasilkan True jika kedua pernyataan benar.
- or: Menghasilkan True jika salah satu dari pernyataan benar.
- not: Membalikkan hasil, menghasilkan False jika hasilnya True.

▪ **Operator and**

Kata kunci **and** adalah operator logika yang digunakan untuk menggabungkan pernyataan kondisi. Kedua kondisi harus benar agar seluruh ekspresi bernilai True. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

▪ **Operator or**

Kata kunci **or** adalah operator logika yang digunakan untuk menggabungkan pernyataan kondisi. Setidaknya satu kondisi harus benar agar seluruh ekspresi bernilai True. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

▪ **Operator not**

Kata kunci not adalah operator logika yang digunakan untuk membalikkan hasil dari pernyataan kondisi. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Menggabungkan Beberapa Operator

Python dapat menggabungkan beberapa operator logika dalam satu ekspresi. Python akan **mengevaluasi not terlebih dahulu, kemudian and, dan terakhir or**. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Menggunakan Tanda Kurung untuk Kejelasan

Saat menggabungkan beberapa operator logika, gunakan tanda kurung untuk membuat maksud lebih jelas dan mengontrol urutan evaluasi.

E. Operator *identity* → operator yang digunakan untuk membandingkan apakah dua objek adalah objek yang sama. Operator *identity* ditampilkan di Tabel 3.7.

Tabel 3.7: Operator *Identity*

Operator	Deskripsi	Contoh
is	Mengembalikan True jika kedua variabel adalah objek yang sama	a is b
is not	Mengembalikan True jika kedua variabel bukan objek yang sama	a is not b

Operator `is` untuk memeriksa apakah kedua variabel menunjuk ke objek yang sama di memori.
Operator `==` untuk memeriksa apakah nilai kedua variabel sama.

F. Operator *membership* → operator yang digunakan untuk memeriksa apakah suatu nilai ada dalam sebuah koleksi, seperti **in** dan **not in**. Operator *membership* ditampilkan di Tabel 3.8.

Tabel 3.8: Operator *Membership*

Operator	Deskripsi	Contoh
in	Mengembalikan True jika sebuah urutan dengan nilai yang ditentukan ada dalam objek	x in y
not in	Mengembalikan True jika sebuah urutan dengan nilai yang ditentukan tidak ada dalam objek	x not in y

G. Operator *bitwise* → operator yang digunakan untuk operasi bit-by-bit dalam angka biner. Operator bitwise ditampilkan di Tabel 3.9.

Tabel 3.9: Operator *Bitwise*

Operator	Nama	Deskripsi	Contoh
&	AND	Menetapkan setiap bit menjadi 1 jika kedua bit bernilai 1	a & b
|	OR	Mengatur setiap bit menjadi 1 jika salah satu dari dua bit bernilai 1	a | b
^	XOR	Menetapkan setiap bit menjadi 1 jika hanya satu dari dua bit bernilai 1	a ^ b
~	NOT	Membalikkan semua bit	~a
<<	Zero fill left shift	Menggeser ke kiri dengan menambahkan nol dari kanan dan membiarkan bit kiri terbuang	a << 2
>>	Signed right shift	Menggeser ke kanan dengan menyalin bit kiri yang paling kiri dan membiarkan bit kanan terbuang	a >> 2

I. Prioritas Operator

Prioritas operator menggambarkan urutan di mana operasi dilakukan. Jika dua operator memiliki prioritas yang sama, maka **ekspresi dievaluasi dari kiri ke kanan**. Urutan prioritas dijelaskan dalam Tabel 3.10, dimulai dengan prioritas tertinggi di bagian atas.

Tabel 3.10: Prioritas Operator

Operator	Deskripsi	Contoh
()	Tanda Kurung (Parentheses)	(a + b) * c
**	Eksponen (Exponentiation)	a ** b
+x, -x, ~x	Plus unari, minus unari, dan bitwise NOT	+a, -b, ~c
* / // %	Perkalian, pembagian, pembagian floor, dan modulus	a * b, c / d, e // f, g % h
+ -	Penjumlahan dan pengurangan	a + b, c - d
<< >>	Pergeseran bit kiri dan kanan (Bitwise left and right shifts)	a << 2, b >> 3
&	Bitwise AND	a & b
^	Bitwise XOR	a ^ b
|	Bitwise OR	a | b
== != > < >= <= is is not in not in	Operator perbandingan, *identity*, dan *membership*	a == b, x is y, z in a
not	Logical NOT	not a
and	AND	a and b
or	OR	a or b

II. Matematika di Python

Python memiliki sekumpulan fungsi matematika bawaan, termasuk modul matematika yang luas, yang melakukan tugas matematika untuk angka. Beberapa fungsi Matematika *Built-in* yaitu:

- Fungsi **min()** dan **max()** dapat digunakan untuk menemukan nilai terendah atau tertinggi dalam sebuah iterable. Iterable adalah objek yang dapat diulang (iterasi) di Python, artinya objek tersebut bisa digunakan dalam loop seperti for. Contoh iterable yang umum adalah list, tuple, string, dan dictionary. Secara teknis, sebuah objek dikatakan iterable jika objek tersebut mendukung operasi iterasi, yaitu dapat diproses satu per satu elemennya. Semua iterable memiliki method khusus bernama __iter__() yang digunakan dalam loop atau struktur lainnya yang membutuhkan iterasi.
- Fungsi **abs()** mengembalikan nilai absolut (positif) dari angka yang diberikan.
- Fungsi **pow(x, y)** mengembalikan nilai x pangkat y (xy).

III. Modul Matematika

Python juga memiliki modul bawaan bernama math, yang memperluas daftar fungsi matematika. Untagar dapatuk menggunakannya, harus mengimpor modul **math**. Setelah mengimpor modul math, maka dapat memulai menggunakan method dan konstanta dari modul tersebut. Sebagai contoh, modul **math.sqrt()** mengembalikan akar kuadrat dari sebuah angka. Modul **math.ceil()** untuk membulatkan angka ke atas ke bilangan bulat terdekat. Modul **math.floor()** membulatkan angka ke bawah ke bilangan bulat terdekat, dan mengembalikan hasilnya. Konstanta **math.pi** untuk mengembalikan nilai PI (3.14). Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

3.5 Decision (Kondisi)

Terdapat dua jenis percabangan dalam pyhton yaitu pernyataan **If** dan **macth**.

I. Kondisi dan Pernyataan dengan If

Python mendukung kondisi logis yang biasa digunakan dalam matematika yaitu:

- Sama dengan: a == b
- Tidak sama dengan: a != b
- Kurang dari: a < b
- Kurang dari atau sama dengan: a <= b
- Lebih besar dari: a > b
- Lebih besar dari atau sama dengan: a >= b

Kondisi-kondisi ini dapat digunakan dalam berbagai cara, yang paling umum adalah "pernyataan if" dan perulangan. **Pernyataan "if" ditulis menggunakan kata kunci if**.

✓ Indentasi

Python mengandalkan indentasi (spasi kosong di awal baris) untuk menentukan cakupan (*scope*) dalam kode. Bahasa pemrograman lain sering menggunakan kurung kurawal untuk tujuan ini. Selain itu, dapat menggunakan spasi atau tab untuk indentasi, tetapi harus menggunakan jumlah indentasi yang sama untuk semua pernyataan dalam blok kode yang sama. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

✓ Bentuk Penulisan IF

Cara penulisan kondisi dan pernyataan If dibedakan menjadi tiga yaitu **if, elif, dan nested if**.

a. Cara Kerja Pernyataan IF

Pernyataan if mengevaluasi sebuah kondisi (ekspresi yang menghasilkan True atau False). Jika kondisi tersebut benar (True), maka blok kode di dalam pernyataan if akan dijalankan. Jika kondisi tersebut salah (False), blok kode tersebut akan dilewati.

♣ Beberapa Pernyataan dalam Blok IF

Programmer dapat memiliki beberapa pernyataan di dalam blok if. Semua pernyataan tersebut harus memiliki tingkat indentasi yang sama. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

♣ IF Singkat

Jika hanya ada satu pernyataan yang ingin dijalankan, pernyataan tersebut bisa ditempatkan dalam baris yang sama dengan pernyataan if. Catatan: **Masih diperlukan tanda titik dua : setelah kondisi**. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

♣ IF ... Else

Jika ada satu pernyataan untuk if dan satu lagi untuk else, keduanya bisa ditempatkan dalam satu baris menggunakan ekspresi kondisional. Ini disebut ekspresi kondisional (kadang dikenal sebagai **"operator ternary"**). Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

♣ Menetapkan Nilai dengan IF ... Else

Python dapat menggunakan if/else dalam satu baris untuk memilih nilai dan menetapkannya ke sebuah variabel. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

♣ **Beberapa Kondisi dalam Satu Baris**

Python dapat menggabungkan beberapa ekspresi kondisional, tetapi tetap jaga agar kode tetap ringkas agar mudah dibaca. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

♣ **Kapan Menggunakan IF Singkat**

Pernyataan if singkat dan operator ternary sebaiknya digunakan ketika:

- Kondisi dan tindakan sederhana.
- Meningkatkan keterbacaan kode.
- Ingin membuat penugasan cepat berdasarkan kondisi.

Penting: Meskipun pernyataan if singkat bisa membuat kode lebih ringkas, hindari penggunaan berlebihan untuk kondisi yang kompleks. Untuk keterbacaan, gunakan pernyataan if-else biasa ketika berhadapan dengan beberapa baris kode atau logika yang rumit. Ternary Operator adalah cara singkat untuk menulis pernyataan if-else dalam satu baris.

♣ **Penggunaan Variabel dalam Kondisi**

Variabel boolean bisa langsung digunakan dalam pernyataan if tanpa operator perbandingan. Python dapat mengevaluasi banyak tipe nilai sebagai True atau False dalam pernyataan if. Angka nol (0), string kosong (""), None, dan koleksi kosong diperlakukan sebagai False. Semua yang lainnya diperlakukan sebagai True. Ini termasuk angka positif (5), angka negatif (-3), dan string non-kosong (bahkan "False" diperlakukan sebagai True karena merupakan string yang tidak kosong). Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

b. Pernyataan Elif di Python

Kata kunci **elif** adalah cara Python untuk mengatakan "jika kondisi sebelumnya tidak benar, maka coba kondisi ini". Kata kunci elif untuk memeriksa beberapa ekspresi untuk nilai True dan menjalankan blok kode segera setelah salah satu kondisi dievaluasi menjadi True.

♣ **Beberapa Pernyataan Elif**

Python mengijinkan memiliki banyak pernyataan elif sesuai kebutuhan. Python akan memeriksa setiap kondisi secara berurutan dan menjalankan yang pertama kali ditemukan benar.

♣ **Cara Kerja Elif**

Saat menggunakan elif, **Python akan mengevaluasi kondisi dari atas ke bawah**. Begitu menemukan kondisi yang benar, Python akan menjalankan blok tersebut dan melewatkan semua kondisi yang tersisa. **Penting**: Hanya kondisi yang pertama kali benar yang akan dijalankan. Meskipun ada beberapa kondisi yang benar, Python akan berhenti setelah menjalankan blok yang cocok pertama kali. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

♣ **Kapan Menggunakan Elif**

Gunakan elif saat memiliki beberapa kondisi yang saling eksklusif untuk diperiksa. Ini lebih efisien dibandingkan menggunakan beberapa pernyataan if terpisah, karena Python akan berhenti memeriksa setelah menemukan kondisi yang benar.

c. Pernyataan IF Bersarang (*Nested* IF)

Python mengijinkan penggunaan pernyataan if di dalam pernyataan if lainnya. Ini disebut sebagai **pernyataan if bersarang**.

♣ **Cara Kerja Pernyataan If Bersarang**

Setiap level dari penyusunan pernyataan if akan menciptakan tingkat pengambilan keputusan yang lebih dalam. **Kode akan dievaluasi mulai dari kondisi terluar hingga ke dalam**. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

♣ **Beberapa Level Penyusunan**

Programmer dapat menyusun pernyataan if hingga sejauh yang diperlukan, tetapi perlu diingat bahwa terlalu banyak level bisa membuat kode menjadi sulit dibaca. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

♣ **Pernyataan If Bersarang vs Operator Logika**

Terkadang, pernyataan if bersarang dapat disederhanakan dengan menggunakan operator logika seperti and. Pilihan ini sesuai dengan logika yang digunakan. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

✓ **Pernyataan Pass di Python**

Pernyataan if tidak boleh dibiarkan kosong, namun jika ingin memiliki pernyataan if tanpa isi, dapat menggunakan pernyataan **pass** untuk menghindari kesalahan. **Pernyataan pass** adalah operasi kosong - tidak ada yang terjadi saat pernyataan ini dieksekusi. Fungsi utama dari pass adalah sebagai pengganti sementara. Pernyataan pass berguna dalam beberapa situasi berikut:

- Saat sedang membuat struktur kode tetapi belum mengimplementasikan logika di dalamnya.
- Ketika sebuah pernyataan diperlukan secara sintaksis tetapi tidak ada aksi yang perlu dilakukan.
- Sebagai pengganti sementara untuk kode di masa depan selama pengembangan.
- Dalam fungsi atau kelas yang kosong yang akan diimplementasikan nanti.

♣ **Pernyataan pass dalam Pengembangan**

Apabila ingin merancang struktur program sebelum mengimplementasikan detailnya, pernyataan pass dapat melakukannya tanpa kesalahan sintaks. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

♣ **Pernyataan Pass vs Komentar**

Komentar diabaikan oleh Python, tetapi pass adalah pernyataan nyata yang tetap dieksekusi (meskipun tidak melakukan apa-apa). Apabila memerlukan pass di tempat yang diharapkan Python untuk sebuah pernyataan, bukan hanya komentar. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

♣ **Pernyataan pass dengan Beberapa Kondisi**

Pernyataan pass dapat digunakan dalam setiap cabang dari pernyataan if-elif-else.

♣ **Pernyataan pass dalam Konteks Lain**

Meskipun di sini fokus terhadap penggunaan pass juga sering digunakan dengan perulangan, fungsi, dan kelas. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

II. Kondisi dan Pernyataan dengan Match

Pernyataan **match** digunakan untuk menjalankan berbagai tindakan berdasarkan kondisi yang berbeda. Alih-alih menulis banyak pernyataan if..else dapat menggunakan pernyataan match. Pernyataan match memilih salah satu dari banyak blok kode untuk dieksekusi. Berikut adalah cara kerjanya:

- Ekspresi match dievaluasi sekali.
- Nilai dari ekspresi dibandingkan dengan nilai terhadap setiap kasus.
- Jika ada yang cocok, blok kode terkait akan dijalankan.

Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=shaing.

♣ **Nilai Default**

Gunakan karakter garis bawah _ **s**ebagai nilai kasus terakhir jika ingin menjalankan blok kode ketika tidak ada kasus lain yang cocok. Nilai _ akan selalu cocok, jadi penting untuk meletakkannya sebagai kasus terakhir agar bertindak sebagai nilai default.

♣ **Menggabungkan Nilai**

Gunakan karakter pipa | sebagai operator OR dalam evaluasi kasus untuk memeriksa lebih dari satu nilai yang cocok dalam satu case.

✓ **Pernyataan If Sebagai *Guards***

Pernyataan if dapat ditambahkan dalam evaluasi kasus sebagai pengecekan kondisi tambahan. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

3.6 Loop (Pengulangan)

Loop dalam Python mempunyai dua kata kunci yaitu while…loop dan for.

A. Perulangan While di Python

Perulangan while dapat mengeksekusi serangkaian pernyataan selama kondisi yang diberikan bernilai benar. **Catatan**: Jangan lupa untuk meningkatkan nilai variabel i, jika tidak, perulangan akan terus berlangsung tanpa henti. Perulangan while membutuhkan variabel yang relevan untuk siap digunakan, dalam contoh perlu mendefinisikan variabel indeks, i, yang diatur ke 1. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Pernyataan break

Pernyataan break dapat menghentikan perulangan meskipun kondisi while masih bernilai benar. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Pernyataan continue

Pernyataan continue dapat menghentikan iterasi saat ini dan melanjutkan ke iterasi berikutnya. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Pernyataan else

Pernyataan else dapat menjalankan blok kode hanya sekali ketika kondisi sudah tidak lagi benar. **Catatan**: Blok else TIDAK akan dieksekusi jika perulangan dihentikan oleh pernyataan break. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

B. Perulangan For di Python

Pernyataan for digunakan untuk melakukan iterasi dalam sebuah urutan (seperti list, tuple, dictionary, set, atau string). Ini berbeda dengan pernyataan for di banyak bahasa pemrograman lainnya, dan lebih mirip dengan method iterator yang ada di bahasa pemrograman berorientasi objek. Dengan perulangan for, dapat dijalankan serangkaian pernyataan, satu kali untuk setiap elemen dalam list, tuple, set, dan sebagainya. Pernyataan for tidak memerlukan variabel indeks untuk diatur sebelumnya. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Melakukan Iterasi dalam String

String pun merupakan objek yang dapat diiterasi, yang berarti dapat menyimpan urutan karakter.

Pernyataan break

Pernyataan break dapat menghentikan perulangan sebelum seluruh elemen terulang. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Pernyataan continue

Pernyataan continue dapat menghentikan iterasi saat ini dan melanjutkan ke iterasi berikutnya. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Fungsi range()

Fungsi range() untuk mengulang sekelompok kode sejumlah kali yang ditentukan. Fungsi range() mengembalikan urutan angka, yang dimulai dari 0 secara default, dan meningkat sebesar 1 (secara default), dan berakhir di angka yang ditentukan. Fungsi range() secara default dimulai dari 0, namun dapat ditentukan nilai awalnya dengan menambahkan paramete, misalnya range(2, 6), yang berarti nilai dari 2 hingga 6 (tetapi tidak termasuk 6). Fungsi range() secara default meningkatkan urutan angka sebesar 1, tetapi dapat ditentukan nilai peningkatannya dengan menambahkan parameter ketiga: range(2, 30, 3). Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Pernyataan else dalam Perulangan For

Kata kunci else dalam perulangan for menentukan blok kode yang akan dieksekusi ketika perulangan selesai. **Catatan**: Blok else TIDAK akan dijalankan jika perulangan dihentikan dengan pernyataan break.

Perulangan Nested For

Perulangan bersarang adalah perulangan di dalam perulangan. "Perulangan dalam" akan dijalankan satu kali untuk setiap iterasi dari "perulangan luar". Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Pernyataan pass

Pernyataan for tidak bisa dibiarkan kosong, tetapi jika karena suatu alasan perulangan for tidak memiliki isi, dapat menggunakan pernyataan pass untuk menghindari kesalahan. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

3.7 Function (Fungsi)

Function_adalah blok kode yang hanya dijalankan ketika dipanggil. Sebuah function dapat mengembalikan data sebagai hasil dari eksekusinya. Function membantu menghindari pengulangan kode yang sama. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Membuat Function

Di Python, function didefinisikan menggunakan kata kunci **def**, diikuti dengan **nama function dan tanda kurung**. Kode di dalam function harus diberi **indentasi**. Python menggunakan indentasi untuk mendefinisikan blok kode.

Memanggil Function

Untuk memanggil function, **cukup tulis nama function diikuti dengan tanda kurung**. Function yang sama bisa dipanggil berkali-kali. Nama function mengikuti aturan yang sama seperti nama variabel di Python:

- Nama function harus dimulai dengan huruf atau garis bawah.
- Nama function hanya boleh mengandung huruf, angka, dan garis bawah.
- Nama function bersifat case-sensitive (misalnya myFunction dan myfunction dianggap berbeda).

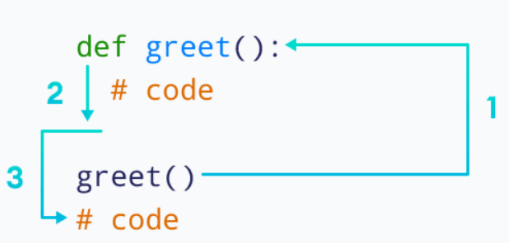
Disarankan untuk menggunakan nama deskriptif yang menjelaskan tujuan dari function tersebut.
Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Cara Kerja Function

Cara kerja function yaitu:

- Ketika function dipanggil, kontrol program akan berpindah ke definisi function tersebut.
- Semua kode yang ada di dalam function akan dieksekusi.
- Setelah eksekusi function selesai, kontrol program akan kembali ke pernyataan berikutnya setelah pemanggilan function.

Cara kerja tersebut diilustrasikan si Gambar 3.1.



Gambar 3.1: Cara Kerja Function (google.com)

Mengapa Menggunakan Function ?

Bayangkan akan mengonversi suhu dari Fahrenheit ke Celsius berkali-kali dalam program. Tanpa function, harus menulis kode perhitungan yang sama berulang-ulang. Dengan function, cukup menulis kode sekali dan dapat menggunakannya lagi kapan pun diperlukan. Selain itu, terdapat beberapa kelebihan menggunakan function yaitu:

- Memudahkan debugging: bagian-bagian kode yang lebih kecil dapat memudahkan proses debugging jika terjadi kesalahan terhadap kode.
- Reusabilitas: function dapat dipanggil kembali dan digunakan kembali di dalam program lain, yang memungkinkan pengembangan perangkat lunak yang lebih cepat dan efisien.
- Penyederhanaan program: function dapat membantu mengurangi kompleksitas program dengan memecahnya menjadi bagian yang lebih sederhana dan mudah dimengerti.
- Modularitas: function memecah kode menjadi bagian yang lebih kecil dan terpisah, yang memudahkan untuk memahami, mengembangkan, dan memelihara kode yang lebih besar.
- Pengurangan duplikasi kode: dengan menggunakan function, dapat menghindari menulis kode yang sama berulang-ulang kali.
- Peningkatan keterbacaan kode: function dapat memberikan nama yang deskriptif dan membuat kode lebih mudah dibaca dan dimengerti.
- Pembagian tugas: function dapat memungkinkan beberapa programer untuk bekerja dalam proyek yang sama, dengan membagi tugas berdasarkan function yang akan ditulis.

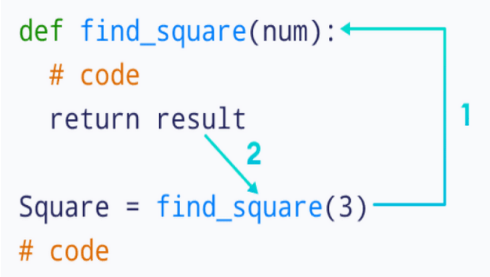
Tipe Function

Terdapat beberapa tipe function yaitu:

- Standard library functions: function -function bawaan yang tersedia di Python untuk digunakan.
- Function yang didefinisikan pengguna: pengguna dapat membuat function sendiri berdasarkan kebutuhan.
- Function rekursif: function yang memanggil dirinya sendiri untuk menyelesaikan suatu masalah.
- Function lambda: function kecil dan anonim yang dapat didefinisikan secara langsung saat dibutuhkan.
- Function tingkat tinggi: function yang menerima fungsi lain sebagai argumen atau mengembalikan function sebagai hasil.

Return Values

Function dapat mengirimkan data kembali ke kode yang memanggilnya menggunakan pernyataan return. Saat sebuah function mencapai pernyataan return, function akan berhenti mengeksekusi dan mengirimkan hasilnya kembali. Return values dapat digunanakan secara langsung. Jika sebuah function tidak memiliki pernyataan return, function tersebut secara default dan akan mengembalikan None. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.



Gambar 3.2: Cara Kerja Return Value (google.com)

Gambar 3.2 menggambarkan sebuah function find_square yang menerima satu argumen yaitu num. Function ini berfungsi untuk menghitung kuadrat dari num dan mengembalikan hasilnya dengan menggunakan perintah return. Dalam contoh ini, fungsi find_square(3) akan menerima nilai 3 sebagai argumen, kemudian menghitung kuadratnya dan mengembalikan hasilnya.

Baris pertama, fungsi find_square didefinisikan dengan num sebagai parameter. Kode dalam function akan melakukan perhitungan yang diperlukan, dan setelah perhitungan selesai, hasilnya dikembalikan dengan perintah return result. Kemudian, baris kedua, fungsi dipanggil dengan argumen 3, dan hasil dari function tersebut disimpan dalam variabel Square.

Function return digunakan untuk mengembalikan nilai hasil perhitungan ke tempat pemanggilan function. Dalam hal ini, nilai Square akan berisi hasil perhitungan kuadrat dari 3. Setelah pemanggilan function selesai, kontrol program akan kembali ke pernyataan setelah pemanggilan function, dan nilai hasil perhitungan dapat digunakan lebih lanjut.

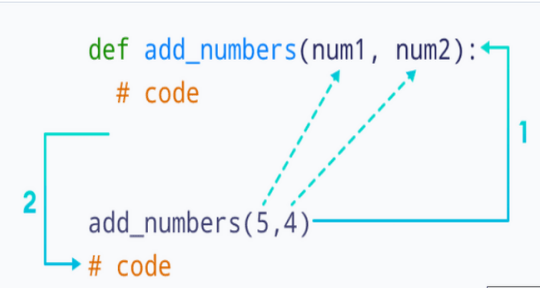
Pernyataan pass

Definisi function tidak boleh kosong. Jika perlu membuat placeholder untuk function tanpa kode, gunakan pernyataan pass. Pernyataan pass sering digunakan selama pengembangan untuk mendefinisikan struktur function terlebih dahulu dan mengimplementasikan detailnya nanti. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

a. Argumen dalam Python

Informasi dapat dikirimkan ke dalam function melalui argumen. Argumen dituliskan setelah nama fungsi, di dalam tanda kurung. Jumlah argumen yang diberikan dapat lebih dari satu, dan dipisahkan menggunakan tanda koma. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Gambar 3.3 menggambarkan cara kerja sebuah fungsi add_numbers yang menerima dua argumen, yaitu num1 dan num2, dan digunakan untuk menjumlahkan keduanya. Bagian pertama, function add_numbers(num1, num2) didefinisikan dengan dua parameter num1 dan num2. Di dalam function tersebut, proses perhitungan dilakukan untuk menjumlahkan kedua nilai yang diterima.



Gambar 3.3: Cara Kerja Argumen (google.com)

Bagian kedua, fungsi dipanggil dengan argumen 5 dan 4 melalui sintaks add_numbers(5, 4), yang berarti nilai 5 disimpan dalam num1 dan nilai 4 disimpan di num2. Function ini kemudian akan melakukan penjumlahan, dan hasil dari operasi tersebut akan dikembalikan oleh function jika ada perintah return di dalam function, meskipun dalam gambar ini return tidak ditampilkan secara eksplisit.

Function return digunakan untuk mengembalikan hasil dari penjumlahan num1 + num2, yang dalam contoh ini adalah 9. Setelah function dipanggil, kontrol program akan kembali ke tempat pemanggilan, dan nilai hasil penjumlahan tersebut dapat digunakan lebih lanjut dalam program.

Parameter vs Argumen

Kedua istilah ini sering dianggap sama, tetapi sebenarnya memiliki perbedaan yaitu:

- Parameter adalah variabel yang ditulis dalam definisi function.
- Argumen adalah nilai nyata yang dikirimkan saat fungsi dipanggil.

Jumlah Argumen

Secara default, function harus dipanggil dengan jumlah argumen yang sesuai. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Parameter dengan Nilai Default

Parameter dapat diberi nilai bawaan. Jika argumen tidak diberikan, maka nilai default yang digunakan. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Keyword Arguments

Argumen dapat dikirimkan menggunakan format **key=value**. Dengan cara ini, urutan tidak berpengaruh. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Positional Arguments

Jika argumen dikirim tanpa menyebutkan nama parameternya, maka disebut argumen posisi. Urutan sangat menentukan. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Menggabungkan Positional dan Keyword

Argumen posisi harus ditulis terlebih dahulu sebelum keyword. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

- **Mengirim Berbagai Tipe Data**

Fuction dapat menerima tipe data apa pun. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

- **Positional-Only Arguments**

Untuk membuat parameter hanya bisa dipanggil secara posisi, tambahkan /. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

- **Keyword-Only Arguments**

Untuk membuat parameter hanya bisa dipanggil dengan keyword, tambahkan *. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

- **Penggabungan Parameter**

Parameter sebelum / hanya posisi, parameter setelah * hanya keyword. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

b. *args dan **kwargs

Secara default, sebuah function harus dipanggil dengan jumlah argumen yang sesuai dengan parameternya. Namun, dalam beberapa kondisi, jumlah argumen yang akan dikirim ke function belum diketahui. Untuk mengatasi hal tersebut, Python menyediakan mekanisme *args dan **kwargs agar function dapat menerima jumlah argumen yang tidak terbatas.

- **Arbitrary Arguments – *args**

Jika jumlah argumen posisi yang akan diterima tidak diketahui, tambahkan tanda bintang * sebelum nama parameter. Dengan cara ini, seluruh argumen posisi yang dikirim akan dikumpulkan dalam bentuk tuple, sehingga dapat diakses satu per satu di dalam function. Istilah ini sering disingkat sebagai *args dalam dokumentasi Python.

- **Apa itu *args?**

Parameter *args berfungsi menerima sejumlah argumen posisi tanpa batas. Di dalam function, args akan menjadi sebuah tuple yang berisi semua argumen yang dikirimkan. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

- **Menggabungkan Parameter Biasa dengan *args**

Parameter biasa harus ditulis sebelum *args. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

- **Arbitrary Keyword Arguments – **kwargs**

Jika jumlah argumen keyword (key=value) yang dikirim tidak diketahui, gunakan dua tanda bintang ** sebelum nama parameter. Semua argumen keyword akan dikumpulkan dalam bentuk dictionary. Istilah ini sering disingkat sebagai **kwargs.

- **Apa itu **kwargs?**

Parameter **kwargs berfungsi menerima jumlah argumen keyword tanpa batas. Di dalam function, kwargs akan menjadi dictionary yang berisi pasangan kunci dan nilai. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Menggabungkan Parameter Biasa dengan **kwargs

Parameter biasa harus ditulis sebelum **kwargs. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Menggabungkan *args dan **kwargs

Keduanya dapat digunakan dalam satu function dengan urutan berikut:

- Parameter biasa
- *args
- **kwargs

Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Unpacking Argumen

Tanda * dan ** juga dapat digunakan saat memanggil function untuk membongkar (unpack) list atau dictionary menjadi argumen terpisah.

Unpacking List dengan *

Unpacking List dengan * dalam Python adalah proses **membuka isi list menjadi elemen-elemen terpisah**. Tanda bintang * digunakan agar setiap item di dalam list diperlakukan sebagai nilai individual, bukan sebagai satu objek list utuh. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

**Unpacking Dictionary dengan **

Kesimpulan: Gunakan *args untuk menerima banyak argumen posisi. Gunakan **kwargs untuk menerima banyak argumen keyword. Gunakan * dan ** dalam definisi fungsi untuk mengumpulkan argumen. Gunakan * dan ** saat memanggil function untuk membongkar argumen. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

c. Scope Fuction

Variabel hanya dapat digunakan di dalam wilayah tempat variabel tersebut dibuat. Konsep ini disebut scope atau cakupan. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Local Scope

Variabel yang dibuat di dalam fuction termasuk dalam cakupan lokal dan hanya dapat digunakan di dalam fuction tersebut.

Function Inside Function (Fuction di dalam Fuction)

Variabel lokal dapat diakses oleh function lain yang berada di dalam function tersebut. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Global Scope

Variabel yang dibuat di luar fuction berada dalam cakupan global dan dapat diakses dari mana saja. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Nama Variabel Sama (Global dan Lokal)

Jika nama variabel sama digunakan di dalam dan di luar fungsi, Python akan menganggapnya sebagai dua variabel berbeda. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Global Keyword

Kata kunci global digunakan untuk membuat atau mengubah variabel global dari dalam fuction. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Nonlocal Keyword

Kata kunci nonlocal digunakan dalam fuction bersarang untuk mengakses variabel dari fuction luar (bukan global). Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Aturan LEGB

Python mencari variabel berdasarkan urutan berikut:

- Local → Di dalam fungsi saat ini.
- Enclosing → Di dalam fungsi luar (nested function).
- Global → Di tingkat modul.
- Built-in → Namespace bawaan Python.

Python menampilkan nilai Local karena pencarian dimulai dari cakupan terdekat terlebih dahulu. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

d. Decorator

Decorator memberikan cara untuk menambahkan perilaku tambahan terhadap suatu fuction tanpa harus mengubah kode asli dari fuction tersebut. Decorator merupakan fuction yang menerima function lain sebagai parameter, lalu mengembalikan function baru.

Basic Decorator (Decorator Dasar)

Decorator didefinisikan terlebih dahulu, kemudian diterapkan menggunakan simbol **@nama_decorator** di atas fungsi. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Multiple Decorator Calls

Satu decorator dapat digunakan dalam beberapa function. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Decorator dalam Fungsi yang Memiliki Argumen

Jika function memiliki parameter, parameter tersebut harus diteruskan ke dalam wrapper. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Menggunakan *args dan **kwargs

Agar decorator dapat menerima berbagai jumlah dan jenis argumen, gunakan *args dan **kwargs. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Decorator dengan Argumen

Decorator dapat memiliki argumen sendiri dengan menambahkan satu lapisan function tambahan. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Multiple Decorators

Beberapa decorator dapat digunakan dalam satu function dengan menumpuknya. Decorator dijalankan dari bawah ke atas. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Menjaga Metadata Fungsi

Ketika function diberi decorator, metadata seperti **__name__ dan __doc__** bisa hilang. Untuk mempertahankannya, gunakan **functools.wraps**. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

e. Lambda

Lambda function adalah function kecil tanpa nama (anonymous function). Lambda dapat menerima banyak argumen, tetapi hanya boleh memiliki satu ekspresi. Contoh di

```
lambda argumen : ekspresi
```

https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Ekspresi tersebut akan dieksekusi dan hasilnya otomatis dikembalikan.

Lambda dapat menerima lebih dari satu argumen.

Mengapa Menggunakan Lambda?

Lambda sering digunakan sebagai function anonim di dalam function lain. Misalnya, membuat function pengali dengan faktor tertentu. Dengan satu definisi function, dapat dibuat beberapa function berbeda.

Lambda dengan map()

Fungsi **map()** menerapkan suatu function ke setiap elemen dalam iterable. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Lambda dengan filter()

Fungsi filter() menghasilkan elemen yang memenuhi kondisi (True). Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Lambda dengan sorted()

Fungsi sorted() dapat menggunakan lambda sebagai key untuk pengurutan khusus. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

f. Generator

Generator adalah fungsi yang dapat berhenti sementara dan melanjutkan kembali eksekusinya. Saat sebuah fungsi generator dipanggil, yang dihasilkan bukan langsung nilai akhir, melainkan objek generator yang bersifat iterator. Kode di dalam generator tidak langsung dijalankan ketika fungsi dipanggil. Eksekusi baru berjalan ketika generator diiterasi (misalnya dengan for atau next()). Generator bertujuan untuk proses iterasi tanpa harus menyimpan seluruh data di memori, karena nilai dihasilkan bertahap (on-the-fly). Berbeda dengan fungsi biasa yang menggunakan return, generator menggunakan kata kunci yield. Kata kunci yield menjadikan fungsi bersifat generator. Saat yield dijalankan:

- Nilai dikembalikan sementara,
- Kondisi (state) fungsi disimpan,
- Eksekusi dapat dilanjutkan lagi dari titik terakhir untuk pemanggilan berikutnya.

Berbeda dengan return yang langsung mengakhiri fungsi, yield hanya menghentikan sementara.

Contoh 1: Generator Dasar. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Contoh 2: Menghemat Memori (Nilai Dibuat Bertahap). Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Contoh 3: next() dan StopIteration. Jika nilai generator sudah habis, pemanggilan next() akan memunculkan StopIteration. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Contoh 4: Generator Expression. Generator dapat dibuat seperti list comprehension, tetapi menggunakan tanda kurung (). Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Contoh 5: Generator Fibonacci (Bisa Berjalan Terus). Generator cocok untuk menghasilkan deret seperti Fibonacci tanpa menyimpan semua nilai.

Contoh 6: Method send() dalam Generator. Method send() dapat mengirim nilai ke dalam generator. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Contoh 7: Method close() untuk Menghentikan Generator. #Method close() untuk Menghentikan Generator. close() menghentikan generator secara paksa. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

3.8 Pengenalan Fitur Dasar Tambahan dalam Python

Berikut penjelasan beberapa fitur tambahan dalam Python yang tidak dapat dikelompokkan ke bagian sebelumnya.

A. PIP

PIP adalah pengelola paket untuk paket Python, atau modul, jika ingin menyebutnya begitu. Catatan: Jika menggunakan versi Python 3.4 atau yang lebih baru, PIP sudah terinstal secara default. Untuk memeriksa apakah PIP sudah terinstal, buka command line dan arahkan ke direktori skrip Python, kemudian ketikkan perintah berikut:

```
pip --version
```

Jika PIP belum terinstal, dapat mengunduh dan menginstalnya dari halaman ini: https://pypi.org/project/pip/.

❖ **Apa itu Package?**

Package berisi semua file yang dibutuhkan oleh sebuah modul. Modul adalah pustaka kode Python dimasukkan ke dalam proyek. Setelah package berhasil diinstal, package tersebut siap untuk digunakan. Misalnya, jika ingin menginstal package camelcase, dapat diimpor ke dalam package:

```
import camelcase
```

Apabila ingin mencari lebih banyak package, kunjungi https://pypi.org/. Perintah uninstall untuk menghapus paket:

```
pip uninstall camelcase
```

PIP akan meminta konfirmasi apakah ingin menghapus package tersebut. Tekan y, dan package akan dihapus. Gunakan perintah list untuk melihat semua package yang terinstal di sistem.

```
pip list
```

Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

B. Input Pengguna

Python memberi pengguna untuk memberikan input. Artinya, program dapat meminta input dari pengguna. Python akan berhenti menjalankan program ketika mencapai fungsi **input()**, dan melanjutkan ketika pengguna sudah memberikan input. Pengguna dapat diminta untuk memasukkan nama di baris baru. Fungsi input() di Python memiliki parameter prompt, yang berfungsi sebagai pesan yang bisa ditampilkan sebelum input pengguna, untuk baris yang sama. Selain itu, pengguna juga dapat menambahkan sebanyak mungkin input yang diperlukan. Python akan berhenti mengeksekusi setiap kali sampai pengguna memberikan input. Input dari pengguna diperlakukan sebagai string. Meskipun pengguna bisa memasukkan angka, Python tetap memperlakukannya sebagai string. Pengguna dapat **mengonversi input menjadi angka menggunakan fungsi float()**. Validasi input merupakan praktik yang baik untuk memvalidasi setiap input dari pengguna. Untuk menghindari kesalahan tersebut, pengguna bisa memeriksa inputnya terlebih dahulu. Jika input tersebut bukan angka, pengguna akan mendapatkan pesan seperti "Input salah, coba lagi", dan diizinkan untuk memberikan input baru. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

C. Try...Except

Blok try untuk menguji sebuah bagian kode untuk kesalahan. Blok except untuk menangani kesalahan yang terjadi. Blok else untuk mengeksekusi kode jika tidak ada kesalahan yang terjadi. Blok finally untuk mengeksekusi kode, terlepas dari hasil yang didapatkan dari blok try dan except. Ketika sebuah kesalahan atau pengecualian terjadi, Python biasanya akan menghentikan program dan menghasilkan pesan kesalahan. Pengecualian ini bisa ditangani menggunakan pernyataan **try**. **Jika ada kesalahan dalam blok try, maka blok except akan dieksekusi**. Tanpa menggunakan blok try, program akan berhenti dan menghasilkan pesan kesalahan. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

❖ **Banyak Pengecualian (Exception)**

Python dapat mendefinisikan dapat mendefinisikan beberapa blok except untuk menangani berbagai jenis kesalahan. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

❖ **Blok else dan finally**

Else digunakan untuk menjalankan kode jika tidak ada kesalahan yang terdeteksi. Finally selalu dijalankan, terlepas apakah kesalahan terjadi atau tidak. Biasanya digunakan untuk membersihkan sumber daya seperti menutup file atau koneksi. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

❖ **Meningkatkan Pengecualian (Raise an Exception)**

Apabila ingin melempar exception, gunakan kata kunci **raise**. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

D. Iterators

Iterator adalah objek yang menyimpan sejumlah nilai yang dapat dihitung. Objek iterator untuk mengakses nilai-nilai tersebut satu per satu. Secara teknis, iterator adalah objek yang mengimplementasikan protokol iterator, yang terdiri dari dua method yaitu **__iter__()** dan **__next__()**. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

❖ **Iterator vs Iterable**

List, tuple, dictionary, dan set adalah objek yang dapat diiterasi (iterable). Objek-objek ini adalah kontainer iterable untuk mendapatkan sebuah iterator darinya. Semua objek ini memiliki method **iter()** yang digunakan untuk mendapatkan sebuah iterator. String juga merupakan objek yang dapat diiterasi dan dapat mengembalikan iterator. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

❖ **Melakukan Perulangan Menggunakan Iterator**

Perulangan for dapat digunakan untuk mengiterasi objek iterable. Pernyataan for sebenarnya membuat objek iterator dan menjalankan method next() untuk setiap iterasi. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

❖ **Membuat Iterator**

Untuk membuat objek atau kelas sebagai iterator harus mengimplementasikan method __iter__() dan __next__() dalam objek tersebut. Sebagai contoh, setiap kelas di Python memiliki method __init__(), yang digunakan untuk melakukan inisialisasi saat objek dibuat. Method __iter__() memiliki fungsi yang mirip, yang digunakan untuk melakukan operasi seperti inisialisasi, namun harus selalu mengembalikan objek iterator itu sendiri. Method __next__() digunakan untuk melakukan operasi, dan harus mengembalikan item berikutnya dalam urutan. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

❖ **StopIteration**

Jika tidak ada kondisi untuk menghentikan perulangan, iterator akan terus berjalan tanpa batas. Untuk mencegah iterasi berjalan selamanya, gunakan pernyataan StopIteration. Dalam method __next__(), dapat menambahkan kondisi terminasi untuk menghentikan iterasi setelah jumlah tertentu. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

E. Module

Modul dapat dianggap sebagai sebuah pustaka kode yang berisi fungsi-fungsi yang ingin digunakan dalam aplikasi. **Modul** adalah file yang menyimpan sekumpulan fungsi yang bisa dimasukkan ke dalam program. Untuk membuat modul yaitu dengan menyimpan kode yang ingin digunakan dalam sebuah file dengan ekstensi .py.

```
# contoh_modul.py
def halo():
 print("Halo Dunia!")
```

❖ Menggunakan Modul

Modul yang telah dibuat, dapat digunakan menggunakan perintah import. Catatan: Saat menggunakan fungsi dari sebuah modul, gunakan sintaks: **nama_modul.nama_fungsi**.

```
import contoh_modul

contoh_modul.halo()

# Output:
# Halo Dunia!
```

❖ Variabel dalam Modul

Modul bisa berisi fungsi-fungsi, seperti yang telah dijelaskan, namun juga dapat menyimpan variabel dengan berbagai jenis tipe (array, dictionary, objek, dan sebagainya).

```
# contoh_modul.py
nama = "John"
umur = 30
```

❖ Penamaan Modul

Nama file modul bisa disesuaikan dengan keinginan, namun file tersebut harus memiliki ekstensi .py agar bisa dikenali sebagai modul oleh Python.

❖ Mengganti Nama Modul

Saat mengimpor modul, nama modul dapat diganti dengan menggunakan kata kunci **as** untuk membuat alias dalam nama modul yang diimpor.

```
import contoh_modul as cm

cm.halo()

# Output:
# Halo Dunia!
```

❖ Modul Bawaan (Built-in)

Python menyediakan berbagai modul bawaan yang dapat langsung digunakan tanpa perlu instalasi tambahan. Contoh penggunaan modul math yang sudah tersedia di Python. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

❖ **Menggunakan Fungsi dir()**

Fungsi dir() digunakan untuk menampilkan daftar nama fungsi atau variabel yang ada dalam suatu modul. Catatan: Fungsi **dir()** dapat digunakan terhadap semua modul, termasuk modul yang dibuat sendiri. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

❖ **Mengimpor Bagian dari Modul**

Jika hanya ingin mengimpor bagian tertentu dari sebuah modul, kata kunci from dapat digunakan. Catatan: Ketika menggunakan from, tidak perlu menuliskan nama modul lagi saat mengakses elemen di dalamnya. Contoh: person1["age"], bukan mymodule.person1["age"].

```
#Mengimpor Bagian dari Modul
from contoh_modul import halo

halo()

# Output:
```

F. Regex

Ekspresi Reguler (RegEx) adalah urutan karakter yang membentuk pola pencarian. Ekspresi Reguler dapat digunakan untuk memeriksa apakah sebuah string mengandung pola pencarian yang ditentukan. Python memiliki package bawaan yang disebut **re**, yang dapat digunakan untuk bekerja dengan Ekspresi Reguler. Awalnya, mengimpor modul re, lalu ekspresi reguler dapat mulai digunakan. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

❖ **RegEx Functions**

Modul re menyediakan serangkaian fungsi untuk melakukan pencarian kecocokan dalam sebuah string ditampilkan dalam Tabel 3.11.

Tabel 3.11: Function Regex

Fungsi	Deskripsi
findall	Mengembalikan daftar yang berisi semua kecocokan
search	Mengembalikan objek Match jika ada kecocokan di mana saja dalam string
split	Mengembalikan daftar di mana string telah dibagi dalam setiap kecocokan
sub	Mengganti satu atau banyak kecocokan dengan sebuah string

❖ **Metakarakter**

Metakarakter adalah karakter dengan makna khusus. Tujuan dari metakarakter dalam ekspresi reguler adalah untuk memberikan kemampuan bagi pengguna untuk menyusun pola pencocokan yang lebih kompleks dan dinamis. Metakarakter adalah simbol atau karakter khusus yang memiliki makna khusus dalam ekspresi reguler dan untuk pencocokan string yang lebih presisi, fleksibel, dan efisien. Jenis karakter metakarakter disajikan di Tabel 3.12.

Tabel 3.12: Metakarakter

Karakter	Deskripsi	Contoh
[]	Sekumpulan karakter	"[p-m]"
\	Menandakan urutan khusus (juga dapat digunakan untuk meloloskan karakter khusus)	"\d"
.	Karakter apapun (kecuali karakter baris baru)	"he..o"
^	Dimulai dengan	"^hai"
$	Diakhiri dengan	"planet$"
*	Nol atau lebih kemunculan	"he.*o"
+	Satu atau lebih kemunculan	"he.+o"
?	Nol atau satu kemunculan	"he.?o"
{}	Jumlah kemunculan yang tepat	"he.{2}o"
()	Menangkap dan mengelompokkan	

❖ Flags

Flags dalam ekspresi reguler digunakan untuk memodifikasi perilaku pencocokan pola. Flags untuk mengubah cara pola dievaluasi tanpa mengubah pola itu sendiri. Tujuan dari flags dalam ekspresi reguler adalah untuk mengubah atau memodifikasi cara pencocokan pola dilakukan, sehingga memberikan fleksibilitas lebih dalam pencocokan string. Flags ini bertujuan untuk menyesuaikan perilaku ekspresi reguler tanpa harus mengubah pola itu sendiri. Jenis flags disajikan di Tabel 3.13.

Tabel 3.13: Flags

Flag	Singkatan	Deskripsi
re.ASCII	re.A	Mengembalikan hanya pencocokan ASCII
re.DEBUG	re.D	Mengembalikan informasi debug
re.DOTALL	re.S	Membuat karakter . mencocokkan semua karakter (termasuk karakter newline)
re.IGNORECASE	re.I	Pencocokan tanpa memperhatikan huruf besar/kecil
re.MULTILINE	re.M	Mengembalikan hanya pencocokan di awal setiap baris
re.NOFLAG		Menyatakan bahwa tidak ada flag yang diatur untuk pola ini
re.UNICODE	re.U	Mengembalikan pencocokan Unicode. Ini adalah default di Python 3. Untuk Python 2, gunakan flag ini untuk mengembalikan hanya pencocokan Unicode
re.VERBOSE	re.X	Penggunaan spasi dan komentar di dalam pola. Membuat pola lebih mudah dibaca

❖ Special Sequences

Urutan khusus adalah tanda backslash (\) diikuti dengan salah satu karakter dalam daftar di bawah ini, yang memiliki makna khusus. Tujuan sequences atau urutan khusus dalam ekspresi reguler adalah untuk pencocokan pola yang lebih spesifik dan fleksibel dalam string. Dengan menggunakan urutan khusus, pengguna dapat mencocokkan posisi tertentu dalam string (seperti awal atau akhir kata), karakter khusus (seperti angka atau spasi), atau karakter tertentu berdasarkan aturan tertentu. Jenis *special sequences* disajikan di Tabel 3.14.

Tabel 3.14: *Special Sequences*

Karakter	Deskripsi	Contoh
\A	Mengembalikan pencocokan jika karakter yang ditentukan ada di awal string	"\AHello"
\b	Mengembalikan pencocokan di mana karakter yang ditentukan ada di awal atau akhir sebuah kata (karakter r di awal memastikan bahwa string diperlakukan sebagai "raw string")	r"\bhello"
\B	Mengembalikan pencocokan di mana karakter yang ditentukan ada, tetapi TIDAK di awal (atau akhir) sebuah kata (karakter r di awal memastikan bahwa string diperlakukan sebagai "raw string")	r"hello\B"

Karakter	Deskripsi	Contoh
\d	Mengembalikan pencocokan di mana string mengandung angka (angka dari 0-9)	"\d"
\D	Mengembalikan pencocokan di mana string TIDAK mengandung angka	"\D"
\s	Mengembalikan pencocokan di mana string mengandung karakter spasi putih	"\s"
\S	Mengembalikan pencocokan di mana string TIDAK mengandung karakter spasi putih	"\S"
\w	Mengembalikan pencocokan di mana string mengandung karakter kata (karakter dari a-z, angka dari 0-9, dan karakter underscore _)	"\w"
\W	Mengembalikan pencocokan di mana string TIDAK mengandung karakter kata	"\W"
\Z	Mengembalikan pencocokan jika karakter yang ditentukan ada di akhir string	"world\Z"

❖ Set

Set adalah sekumpulan karakter di dalam sepasang **tanda kurung siku []** dengan makna khusus. Tujuan dari Set dalam ekspresi reguler untuk pencocokan karakter berdasarkan kriteria tertentu, seperti mencocokkan salah satu dari beberapa karakter atau mencocokkan karakter dalam rentang tertentu. Dengan Set, dapat membuat pola pencocokan yang lebih fleksibel dan kuat. Karakter set ditampilkan di Tabel 3.15.

Tabel 3.15: Karakter Set

Set	Deskripsi
[arn]	Mengembalikan pencocokan jika salah satu karakter yang ditentukan (a, r, atau n) ada.
[a-n]	Mengembalikan pencocokan untuk karakter huruf kecil, yang urutannya alfabetik antara a dan n.
[^arn]	Mengembalikan pencocokan untuk karakter apapun KECUALI a, r, dan n.
[0123]	Mengembalikan pencocokan jika salah satu angka yang ditentukan (0, 1, 2, atau 3) ada.
[0-9]	Mengembalikan pencocokan untuk angka apapun antara 0 hingga 9.
[0-5][0-9]	Mengembalikan pencocokan untuk angka dua digit dari 00 hingga 59.
[a-zA-Z]	Mengembalikan pencocokan untuk karakter apapun yang alfabetik antara a dan z, baik huruf kecil maupun huruf besar.
[+]	Dalam himpunan, +, *, ., |, (), $,{} tidak memiliki arti khusus, jadi [+] berarti: kembalikan kecocokan untuk karakter + apa pun dalam string.

❖ Fungsi findall()

Fungsi findall() mengembalikan daftar yang berisi semua kecocokan. Daftar tersebut mencantumkan kecocokan sesuai urutan ditemukannya. Jika tidak ada kecocokan yang ditemukan, sebuah daftar kosong akan dikembalikan. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

❖ Fungsi search()

Fungsi search() mencari kecocokan dalam string dan mengembalikan objek Match jika ada kecocokan. Jika ditemukan lebih dari satu kecocokan, hanya kecocokan pertama yang akan dikembalikan. Jika tidak ada kecocokan yang ditemukan, fungsi ini akan mengembalikan nilai None. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

❖ Fungsi split()

Fungsi split() mengembalikan daftar di mana string telah dibagi di setiap kecocokan. Jumlah kemunculan yang akan diproses dapat dikontrol dengan menetapkan parameter maxsplit. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

❖ **Fungsi sub()**

Fungsi sub() mengganti kecocokan dengan teks yang diinginkan. Jumlah penggantian dapat dikendalikan dengan menetapkan parameter count. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

❖ **Objek Match**

Objek Match adalah objek yang berisi informasi tentang pencarian dan hasil yang ditemukan. Catatan: Jika tidak ada kecocokan, nilai None akan dikembalikan alih-alih objek Match. Objek Match memiliki properti dan method yang digunakan untuk mengambil informasi tentang pencarian dan hasilnya:

- .span() mengembalikan tuple yang berisi posisi awal dan akhir dari kecocokan.
- .string mengembalikan string yang diberikan ke dalam fungsi.

.group() mengembalikan bagian string di mana kecocokan ditemukan. Contoh di https://colab.research.google.com/drive/18lSZOmZ15lmYs725hAoVzueNGdxvbsAG?usp=sharing.

Latihan:
Soal 1: Percabangan Sederhana
Buatlah program yang meminta pengguna memasukkan angka dan mencetak apakah angka tersebut positif, negatif, atau nol.

Soal 2: Percabangan If-Else
Buatlah program yang meminta pengguna memasukkan sebuah angka. Jika angka tersebut lebih besar dari 10, cetak "Angka lebih besar dari 10", jika tidak, cetak "Angka lebih kecil atau sama dengan 10".

Soal 3: Percabangan Elif
Buatlah program yang meminta pengguna memasukkan sebuah angka dan menentukan apakah angka tersebut genap atau ganjil menggunakan if, elif, dan else.

Soal 4: Pengulangan While (Sederhana)
Buatlah program yang meminta pengguna memasukkan angka, dan program akan terus meminta input angka hingga pengguna memasukkan angka negatif.

Soal 5: Pengulangan For
Buatlah program yang menampilkan semua angka dari 1 hingga 10 menggunakan pengulangan for.

Soal 6: Pengulangan dan Percabangan
Buatlah program yang meminta pengguna memasukkan 5 angka. Program harus mencetak "Angka genap" jika angka tersebut genap, dan "Angka ganjil" jika angka tersebut ganjil.

Soal 7: Nested If-Else (Percabangan Bersarang)
Buatlah program yang meminta pengguna memasukkan usia. Jika usia lebih besar dari atau sama dengan 18, program harus mencetak "Anda sudah dewasa", tetapi jika usia lebih kecil dari 18, program harus memeriksa apakah usia lebih besar dari atau sama dengan 13. Jika iya, cetak "Anda remaja", jika tidak, cetak "Anda masih anak-anak".

Soal 8: Pengulangan dengan Kondisi
Buatlah program yang meminta pengguna memasukkan angka positif dan terus meminta input sampai pengguna memasukkan angka yang lebih besar dari 100.

Soal 9: Program Dengan Nested Loop
Buatlah program yang mencetak pola bintang (*) seperti berikut menggunakan pengulangan:
*
**
***
****
*****

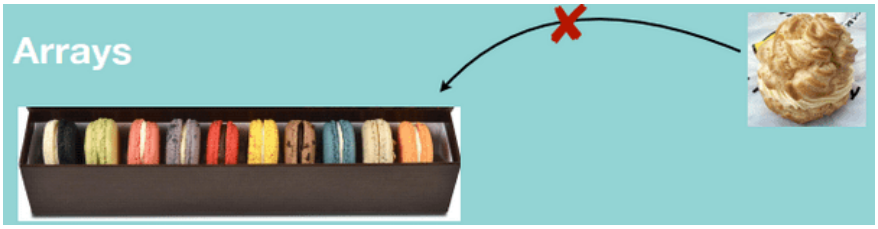
Soal 10: Program Menggunakan While dan If untuk Menentukan Kelipatan
Buatlah program yang meminta pengguna memasukkan angka, kemudian menampilkan semua kelipatan 3 hingga angka yang dimasukkan. Program harus terus meminta input angka sampai angka yang dimasukkan lebih besar dari 0.

BAB 4
Array dalam Python

Bahasa pemrograman Python memiliki beberapa tipe data yang termasuk dalam kategori sequence. Tipe data ini mengadopsi konsep dasar yang serupa dengan array. Selain itu, array berfungsi sebagai struktur data dasar yang memungkinkan penyimpanan elemen-elemen data secara berurutan, serta mendukung penyimpanan beberapa data dalam satu waktu. Meskipun demikian, terdapat perbedaan signifikan antara array dan tipe data sequence lainnya dalam Python, baik dari segi implementasi maupun fleksibilitas penggunaannya. Selain itu, tipe data mapping dianggap sebagai bagian dari struktur data fundamental dalam Python, sehingga penting juga untuk dibahas. Struktur ini sangat efisien untuk pencarian dan pengelolaan data, di mana setiap elemen dapat diakses langsung melalui kunci yang unik, bukan melalui urutan. Penggunaan mapping sangat luas dalam berbagai aplikasi, seperti pengolahan data, pemrograman berbasis objek, dan pengelolaan konfigurasi, menjadikannya salah satu tipe data yang tidak dapat dipisahkan dalam pengembangan aplikasi Python.

4.1 Array

Array adalah **kumpulan elemen data yang serupa**. Elemen data ini memiliki **tipe data yang sama**. Elemen-elemen array disimpan di lokasi memori yang berurutan dan dirujuk oleh sebuah indeks (juga dikenal sebagai subskrip). Subskrip adalah angka ordinal yang digunakan untuk mengidentifikasi sebuah elemen array (Thareja, 2014). Setiap elemen dapat diidentifikasi menggunakan satu indeks atau kunci. Elemen-elemen dalam array disusun sehingga posisi setiap elemen dapat dihitung berdasarkan indeksnya menggunakan rumus matematika (Karimov, 2022).

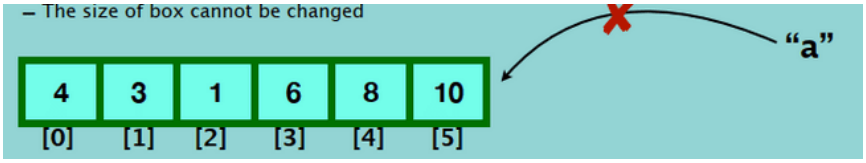


Gambar 4.1: Ilustrasi Array dengan Roti Macaron (Karimov, 2022)

Sebagai ilustrasi, di Gambar 4.1 terdapat sebuah kotak yang digunakan untuk menampung roti macaron. Array dianalogikan sebagai kotak tersebut, yang menyimpan elemen secara berurutan. Setiap roti macaron mewakili elemen dalam array. Roti macaron memiliki ukuran, bentuk, tekstur, dan bahan yang seragam, dengan perbedaan terletak dalam warna rotinya. Demikian pula, elemen-elemen dalam array memiliki tipe data yang sama agar dapat disimpan dalam satu array. Apabila dimasukkan roti yang berbeda, baik dari segi tekstur maupun ukuran, hal tersebut tidak diperkenankan karena perbedaan ukuran rotinya. Begitu pula dengan array, yang tidak dapat menyimpan data dengan tipe data yang berbeda. Setiap roti macaron dapat diidentifikasi secara unik berdasarkan lokasinya dalam kotak. Begitu juga dengan elemen dalam array, yang dapat diidentifikasi secara unik berdasarkan indeksnya, dan ukuran array yang bersifat tetap serta tidak dapat diubah.

Gambar 4.2 menggambarkan konsep array dalam pemrograman. Terdapat sebuah array yang terdiri dari beberapa elemen yang disusun secara berurutan, yaitu angka 4, 3, 1, 6, 8, dan 10. Setiap elemen di dalam array ini diakses menggunakan indeks yang dimulai dari 0 hingga 5 (yang ditulis didalam kururng siku [ ]). Misalnya, elemen pertama (4) terletak di indeks 0, elemen kedua (3) terletak di indeks 1, dan seterusnya. Di sebelah kanan gambar, terdapat tanda silang yang menunjukkan bahwa elemen yang tidak

sesuai, seperti huruf "a", tidak bisa dimasukkan ke dalam array ini karena array hanya dapat menyimpan elemen dengan tipe data yang sama. Selain itu, ukuran array tidak dapat diubah setelah array didefinisikan. Artinya, jumlah elemen dalam array bersifat tetap dan tidak dapat ditambah atau dikurangi setelah pembuatan array tersebut.



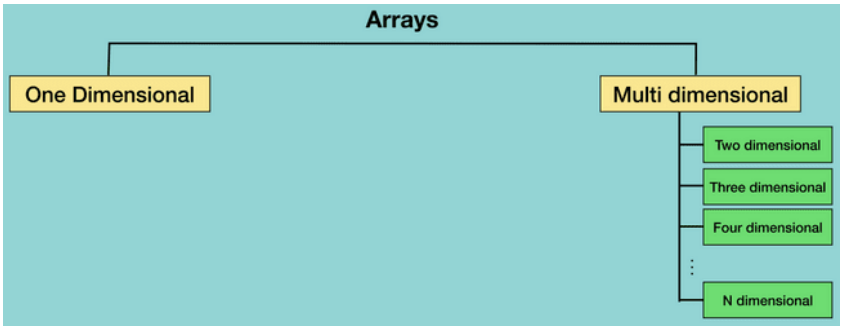
Gambar 4.2: Konsep Array (Karimov, 2022)

Ada beberapa alasan mengapa memerlukan array, diantaranya:

- ❖ Penyimpanan data yang lebih efisien
  Menyimpan beberapa nilai dalam satu variabel, sehingga menghemat ruang memori dan membuat program lebih efisien.
- ❖ Mengakses data dengan mudah
  Mengakses data dalam array dengan mudah menggunakan indeks atau nomor urut elemen.
- ❖ Memproses data secara mudah
  Melakukan perulangan untuk mengakses semua elemen dalam array.
- ❖ Struktur data yang fleksibe
  Array untuk membangun struktur data yang lebih kompleks, seperti matriks, list, dan queue.

Array umumnya digunakan ketika ingin menyimpan sejumlah besar data dengan tipe yang sama (Thareja, 2014). Namun, array memiliki beberapa keterbatasan sebagai berikut:

- ❖ Array memiliki ukuran tetap.
- ❖ Elemen data disimpan di lokasi memori yang berurutan, yang mungkin tidak selalu tersedia.
- ❖ Penambahan dan penghapusan elemen dapat menjadi masalah karena elemen-elemen harus dipindahkan dari posisi semula.



Gambar 4.3: Jenis Array (Karimov, 2022)

Gambar 4.3 menjelaskan jenis-jenis array dalam pemrograman. Array dibagi menjadi dua kategori utama yaitu:

- *One Dimensional Array* adalah array yang memiliki satu dimensi, berupa satu baris atau kolom. Elemen-elemen dalam array ini diakses menggunakan satu indeks tunggal.
- *Multi Dimensional Array* adalah array yang memiliki lebih dari satu dimensi. Dalam kategori ini, terdapat berbagai tipe array berdasarkan jumlah dimensi, seperti:
  - *Two dimensional array*: array yang terdiri dari baris dan kolom, yang membentuk sebuah matriks.

- *Three dimensional array*: array yang terdiri dari beberapa lapisan dua dimensi, membentuk struktur tiga dimensi.
- *Four dimensional array* dan N dimensional array merupakan array dengan lebih dari tiga dimensi, digunakan dalam aplikasi yang membutuhkan penyimpanan data dalam bentuk yang lebih kompleks dan multidimensional.

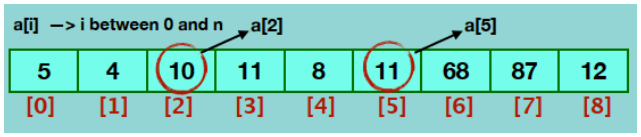
Keyword untuk membangun array dalam pemrograman python yaitu:

- Array untuk membentuk array dalam Python, dilakukan dengan mengimpor modul array yaitu menggunakan perintah **import array**. Modul ini menyediakan cara dasar untuk bekerja dengan array dalam Python, namun fungsionalitasnya terbatas jika dibandingkan dengan pustaka lainnya.
- Numpy untuk operasi array yang lebih canggih. Penggunaan NumPy dilakukan dengan mengimpor pustaka yaitu menggunakan perintah **import numpy**. NumPy menawarkan lebih banyak fungsionalitas, seperti mendukung array multidimensi dan menyediakan berbagai fungsi untuk manipulasi data numerik secara efisien.

A. Array Satu Dimensi

Array 1 dimensi adalah struktur data yang menyimpan sekumpulan elemen yang berurutan dalam satu baris atau satu kolom. Setiap elemen dalam array ini dapat diakses menggunakan indeks tunggal yang menunjukkan posisi elemen tersebut dalam array. Secara sederhana, array 1 dimensi dapat dianggap sebagai daftar atau urutkan data yang memiliki satu kolom atau satu baris yang elemen-elemennya saling berurutan. Ciri-ciri Array 1 Dimensi yaitu:

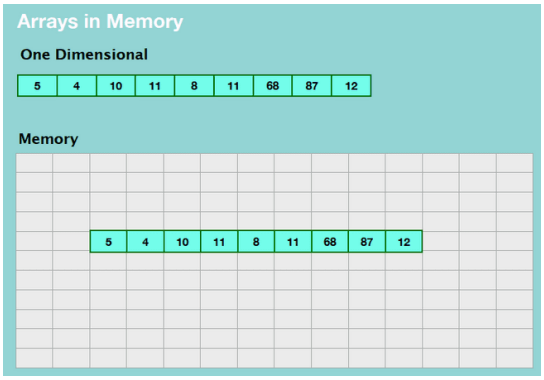
✓ Hanya memiliki satu dimensi atau satu baris atau satu kolom.
✓ Elemen-elemen di dalam array terurut secara berurutan.
✓ Akses elemen dilakukan menggunakan indeks tunggal.



Gambar 4.4: Array Satu Dimensi (Karimov, 2022)

Gambar 4.4 menunjukkan array satu dimensi yang terdiri dari elemen-elemen berupa 5, 4, 10, 11, 8, 11, 68, 87, dan 12. Setiap elemen dalam array ini diakses menggunakan indeks mulai dari 0 hingga 8. Elemen dengan nilai 10 berada di indeks 2, dan elemen dengan nilai 11 ada di indeks 5. Indeks dalam array adalah antara 0 hingga n-1, di mana n adalah jumlah total elemen dalam array. Misalnya, indeks 0 merupakan elemen pertama (5), sedangkan indeks 5 adalah elemen keenam (11). Secara default, indeks array dimulai dengan angka 0. Indeks ini memberikan cara untuk mengakses elemen dalam array secara efisien.

Gambar 4.5 menjelaskan bagaimana array satu dimensi tersebut disimpan dalam memori. Elemen-elemen array tersebut disusun secara berurutan dalam baris memori yang kontigu, yang menunjukkan bahwa elemen-elemen array ditempatkan dalam blok memori yang berdekatan satu sama lain. Ini adalah cara penyimpanan yang efisien dalam memori komputer, di mana setiap elemen dapat diakses dengan indeks yang sesuai. Penyimpanan yang kontigu memungkinkan pengaksesan data secara cepat dan mudah.



Gambar 4.5: Penyimpanan Array Satu Dimensi di Memori (Karimov, 2022)

Berikut cara membuat, mengakses elemen, menghapus, dan menambah elemen array satu dimensi di python dapat dilihat https://colab.research.google.com/drive/14tR9SP3nt101Bq0J5wmDBsQgobXq5Edm?usp=sharing.

Latihan:
Soal 1: Array Satu Dimensi
Buatlah program Python untuk:
- Membuat array satu dimensi yang berisi angka 1 hingga 10.
- Menampilkan elemen pertama dan terakhir dari array tersebut.
- Menambah elemen 11 ke dalam array.
- Menghapus elemen ke-5 dalam array.
- Mengubah nilai elemen ke-3 menjadi 20.

Soal 2: Array Satu Dimensi dan Dua Dimensi
Buatlah program Python untuk:
- Membuat array satu dimensi yang berisi 5 angka yang dimasukkan oleh pengguna.
- Membuat array dua dimensi 2x2 yang berisi nilai yang dimasukkan oleh pengguna.
- Tampilkan elemen terhadap baris pertama dan kolom kedua dari array dua dimensi.

Latihan:
Soal 3: Array Satu Dimensi
Buatlah program Python untuk:
- Membuat array satu dimensi yang berisi angka acak dari 1 hingga 100 sebanyak 20 elemen.
- Menampilkan elemen terhadap indeks genap.
- Menghitung jumlah total dari semua elemen dalam array.
- Menyaring elemen yang lebih besar dari 50 dan menampilkannya.

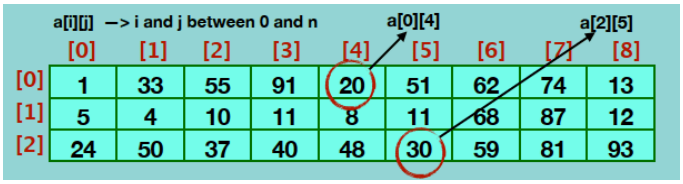
B. Array Dua Dimensi

Array dua dimensi adalah struktur data yang menyimpan elemen-elemen dalam bentuk tabel, terdiri dari **baris** dan **kolom**. Setiap elemen dalam array ini dapat diakses menggunakan dua indeks yaitu satu untuk memilih **baris** dan satu lagi untuk memilih **kolom**. Konsep ini mirip dengan matriks dalam matematika, di mana data disusun dalam bentuk tabel yang memiliki lebih dari satu dimensi. Ciri-ciri array dua dimensi yaitu:

- Mempunyai dua indeks, untuk mengakses elemen dalam array dua dimensi, diperlukan dua indeks. Indeks pertama untuk memilih baris dan indeks kedua untuk memilih kolom.

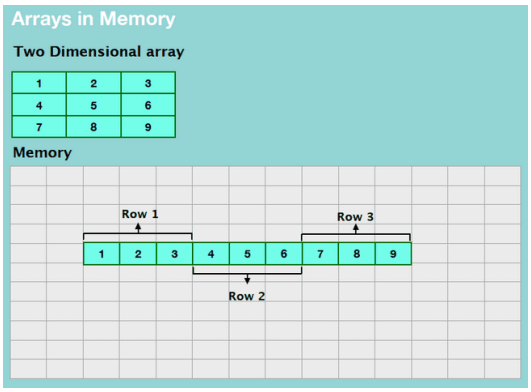
- Menyimpan data dalam tabel → data disusun dalam bentuk tabel yang memiliki baris dan kolom. Setiap baris berisi sejumlah elemen yang dapat diakses dengan indeks kolomnya.
- **Ukuran tetap**, seperti array satu dimensi, array dua dimensi juga memiliki ukuran tetap yang didefinisikan saat pembuatan array dan tidak dapat diubah.
- Data homogen, semua elemen dalam array dua dimensi biasanya memiliki **tipe data yang sama**, meskipun ini bisa berbeda jika menggunakan struktur data seperti list of lists di Python.

Array dua dimensi sering digunakan dalam berbagai aplikasi, seperti matriks dalam matematika dan algoritma pemrograman, gambar dalam pengolahan citra digital (diwakili sebagai grid piksel), dan tabulasi data dalam analisis data, seperti tabel database atau spreadsheet.



Gambar 4.6: Array Dua Dimensi (Karimov, 2022)

Gambar 4.6 menggambarkan sebuah array dua dimensi yang terdiri dari tiga baris dan sembilan kolom. Dalam array ini, setiap elemen diakses menggunakan dua indeks, yaitu indeks baris dan kolom. Indeks baris [i] berkisar dari 0 hingga 2, dan indeks kolom [j] berkisar dari 0 hingga 8. Sebagai contoh, elemen yang terletak di baris pertama dan kolom kelima diakses dengan notasi a[0][4], yang memiliki nilai 20. Begitu pula, elemen yang terletak di baris ketiga dan kolom keenam dapat diakses dengan notasi a[2][5], yang memiliki nilai 30. Dengan demikian, array dua dimensi ini memungkinkan penyimpanan dan pengaksesan data yang terstruktur dalam bentuk tabel, di mana elemen-elemen dalam array disusun dalam bentuk matriks. Array ini dapat digunakan untuk berbagai keperluan dalam pemrograman yang memerlukan pengolahan data yang terorganisir dalam dua dimensi, seperti pengolahan matriks atau penyimpanan data tabular dalam aplikasi komputer.



Gambar 4.7: Penyimpanan Array Dua Dimensi di Memori (Karimov, 2022)

Gambar 4.7 menunjukkan cara penyimpanan array dua dimensi dalam memori komputer. Array dua dimensi dengan dua baris dan tiga kolom, yang berisi nilai-nilai sebagai berikut: baris pertama berisi angka 1, 2, dan 3, baris kedua berisi angka 4, 5, dan 6, serta baris ketiga berisi angka 7, 8, dan 9. Array dua dimensi tersebut diwakili dalam bentuk memori, di mana elemen-elemen array disusun secara linear dalam memori. Penyusunan ini menggambarkan bahwa setiap elemen array, meskipun berada dalam baris yang berbeda, disimpan secara berurutan dalam memori (dari kiri ke kanan). Elemen-elemen dalam memori ini dikelompokkan berdasarkan baris, sehingga elemen-elemen yang berada dalam satu baris berdekatan satu sama lain. Baris pertama terdiri dari angka 1, 2, dan 3, baris kedua terdiri dari angka 4, 5,

dan 6, dan baris ketiga terdiri dari angka 7, 8, dan 9. Penyusunan array dua dimensi secara linear ini sering digunakan dalam berbagai bahasa pemrograman untuk mengelola data dalam format yang lebih efisien di dalam memori. Berikut cara membuat, mengakses elemen, menghapus, menukar, dan menambah elemen array dua dimensi di python.

program array dua dimensi dapat dilihat di https://colab.research.google.com/drive/1pcH5aJyNGUL3o0QYab1em3fpoR4FELnf?usp=sharing.

Latihan:
Soal 1: Array Dua Dimensi
Buatlah program Python untuk:
Membuat array dua dimensi yang berisi angka-angka sebagai berikut:
1 2 3
4 5 6
7 8 9

- Menampilkan elemen yang ada di baris ke-2 dan kolom ke-3.
- Menambah baris baru [10, 11, 12] ke dalam array.
- Menambah kolom baru [13, 14, 15, 16] ke dalam array.
- Menghapus baris ke-1 dari array.
- Menghapus kolom ke-2 dari array.

Soal 2: Array Dua Dimensi
Buatlah program Python untuk:
- Membuat array dua dimensi 3x3 dengan angka acak dari 1 hingga 10.
- Menampilkan seluruh elemen dari array dua dimensi.
- Menghitung jumlah dari setiap baris dalam array dua dimensi dan menampilkannya.
- Menukar elemen baris ke-1 dan baris ke-3.
- Menampilkan elemen yang ada di kolom ke-2.

Soal 3: Array Satu Dimensi dan Dua Dimensi
- Buatlah program Python untuk:
- Membuat array satu dimensi yang berisi 5 angka yang dimasukkan oleh pengguna.
- Membuat array dua dimensi 2x2 yang berisi nilai yang dimasukkan oleh pengguna.
- Tampilkan elemen baris pertama dan kolom kedua dari array dua dimensi.
- Cari dan tampilkan angka terbesar dari array satu dimensi.

C. Array Tiga Dimensi

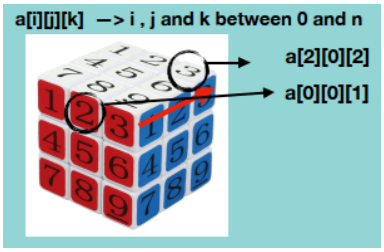
Array tiga dimensi adalah struktur data yang memungkinkan penyimpanan data dalam bentuk matriks yang memiliki lebih dari dua dimensi. Dalam array tiga dimensi, setiap elemen diorganisasi dalam "blok" yang berisi beberapa baris dan kolom. Dengan kata lain, array tiga dimensi dapat dianggap sebagai kumpulan dari beberapa array dua dimensi yang saling berhubungan. Array ini digunakan dalam berbagai bidang pemrosesan citra, analisis data multidimensi, dan simulasi fisik, yang memerlukan lebih dari dua tingkat data. Ciri-ciri array tiga dimensi yaitu:

- Array tiga dimensi memiliki tiga sumbu atau dimensi. Dimensi pertama biasanya mewakili "lapisan" atau "blok", dimensi kedua mewakili baris, dan dimensi ketiga mewakili kolom.
- Penggunaan indeks tiga untuk mengakses elemen dalam array tiga dimensi berupa satu untuk memilih lapisan (dimensi pertama), satu untuk memilih baris (dimensi kedua), dan satu lagi untuk memilih kolom (dimensi ketiga).

- Secara visual, array tiga dimensi dapat digambarkan sebagai tumpukan beberapa matriks dua dimensi yang disusun secara vertikal. Setiap elemen di dalamnya diidentifikasi dengan tiga koordinat yaitu [layer, row, column].
- Array tiga dimensi mendukung pelaksanaan operasi lanjutan terhadap dimensi yang lebih tinggi, seperti rotasi, pemotongan (*slicing*), dan penggabungan (*stacking*) antar lapisan.

Array tiga dimensi sangat berguna dalam banyak aplikasi praktis, seperti:

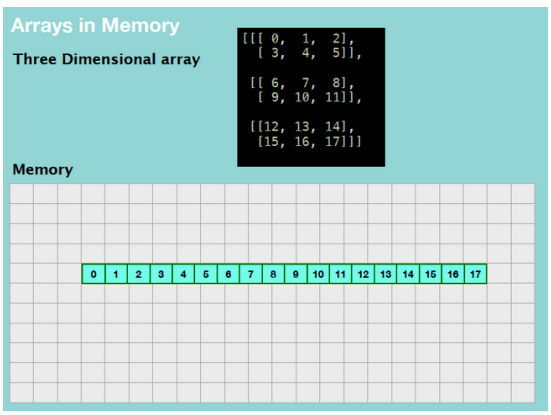
- Pemrosesan citra → citra digital disimpan dalam array tiga dimensi, di mana dimensi pertama mewakili pixel, dimensi kedua adalah tinggi citra, dan dimensi ketiga adalah warna (rgb).
- Data multidimensi → dalam analisis data ilmiah, array tiga dimensi digunakan untuk menyimpan data yang melibatkan beberapa variabel atau pengukuran dalam bentuk yang lebih terorganisir dan efisien.
- Simulasi fisika → dalam simulasi fisika atau grafik komputer, array tiga dimensi digunakan untuk merepresentasikan ruang tiga dimensi yang kompleks, seperti model objek tiga dimensi atau simulasi fluida.



Gambar 4.8: Array Tiga Dimensi (Karimov, 2022)

Gambar 4.8 menggambarkan representasi dari array tiga dimensi yang disusun dalam bentuk kubus, di mana setiap sisi kubus menunjukkan angka-angka yang mewakili elemen-elemen dalam array. Notasi a[i][j][k] digunakan untuk mengakses elemen dalam array tiga dimensi ini, di mana i, j, dan k masing-masing adalah indeks untuk dimensi pertama, kedua, dan ketiga array, yang berkisar antara 0 hingga nilai n.

Elemen-elemen array dapat diakses menggunakan tiga indeks. Sebagai contoh, a[0][0][1] merupakan elemen yang terletak di lapisan pertama, baris pertama, dan kolom kedua, yang memiliki nilai 2. Sedangkan a[2][0][2] merupakan elemen yang terletak di lapisan ketiga, baris pertama, dan kolom ketiga, yang memiliki nilai 3. Struktur ini menunjukkan bagaimana array tiga dimensi dapat dibayangkan sebagai tumpukan matriks dua dimensi yang saling berhubungan dan memungkinkan akses ke data yang lebih kompleks dan multidimensional.



Gambar 4.9: Penyimpanan Array Tiga Dimensi di Memori (Karimov, 2022)

Gambar 4.9 menggambarkan penyimpanan array tiga dimensi dalam memori komputer. Terlihat representasi array tiga dimensi dengan elemen-elemen yang tersusun dalam tiga lapisan. Array tersebut berisi tiga baris dan tiga kolom untuk setiap lapisan, yang masing-masing terdiri dari elemen-elemen bertipe data angka. Contohnya, lapisan pertama array memiliki elemen [0, 1, 2], lapisan kedua [3, 4, 5], dan seterusnya hingga lapisan ketiga.

Bagian bawah gambar menunjukkan bagaimana array tiga dimensi tersebut disimpan dalam memori komputer secara linier. Dalam memori, array tiga dimensi disusun sebagai satu rangkaian panjang elemen yang berurutan, yang dimulai dari elemen pertama hingga elemen terakhir. Meskipun array tersebut memiliki tiga dimensi, dalam memori, elemen-elemen tersebut disusun dalam satu dimensi dengan urutan tertentu. Dalam hal ini, urutan elemen disusun mulai dari lapisan pertama dan kolom pertama, hingga lapisan terakhir dan kolom terakhir. Gambar ini mengilustrasikan bagaimana array tiga dimensi dapat disimpan dalam memori dengan cara yang lebih efisien, meskipun secara konseptual array tersebut terdiri dari lebih dari dua dimensi.

Berikut cara membuat, mengakses elemen, menghapus, menukar, dan menambah elemen array tiga dimensi di python dengan link https://colab.research.google.com/drive/1W0TQwFJH3GWKQYfEv4ogmgw96LuAjInk?usp=sharing.

Latihan:
Soal 1: Membuat Array Tiga Dimensi
Buatlah program Python untuk:
- Membuat array tiga dimensi berukuran 3x3x3 yang berisi angka 1 hingga 27.
- Menampilkan array tiga dimensi yang telah dibuat.

Soal 2: Mengakses Elemen Array Tiga Dimensi
Buatlah program Python untuk:
- Membuat array tiga dimensi dengan ukuran 2x2x3 yang berisi nilai yang dimasukkan oleh pengguna.
- Menampilkan elemen yang ada di layer pertama, baris kedua, dan kolom ketiga.
- Menampilkan elemen yang ada di layer kedua, baris pertama, dan kolom kedua.

Soal 3: Menambah dan Menghapus Elemen Array Tiga Dimensi
Buatlah program Python untuk:
- Membuat array tiga dimensi dengan ukuran 2x3x3 yang berisi angka acak antara 1 hingga 50.
- Menambah satu layer baru yang berisi angka-angka dari 51 hingga 60.
- Menghapus layer pertama dari array tiga dimensi.

Soal 4: Menukar Elemen dalam Array Tiga Dimensi
Buatlah program Python untuk:
- Membuat array tiga dimensi berukuran 2x2x2 yang berisi nilai 1 hingga 8.
- Menukar elemen layer pertama dan layer kedua dari array tersebut.
- Menukar elemen dalam satu layer (misalnya menukar elemen pertama dan kedua di layer pertama).

Setelah membuat array menggunakan keyword numpy, selanjutnya array dapat dibuat menggunakan kata kunci array. Sebagai contoh, hanya dibuatkan array dengan satu dimensi. Berikut cara membuat, mengakses elemen, menghapus, menukar, dan menambah elemen array satu dimensi di python menggunakan keyword array. Program dapat dilihat di https://colab.research.google.com/drive/1MKOnx-eDjNn3nUn359hzRguy3U531Pys?usp=sharing.

Dalam array, mempunyai tipe data khusus ditunjukkan dengan kode tipe yang disebut typecode yang digunakan untuk mendefinisikan jenis elemen yang akan dimasukkan ke dalam array. Jenis typecode disajikan di Tabel 4.1.

Tabel 4.1: Jenis Typecode

TypeCode	Tipe Python	Ukuran Minim di Bytes	TypeCode	Tipe Python	Ukuran Minim di Bytes
'b'	int	1	'l'	int	4
'B'	int	1	'L'	int	4
'u'	Unicode character	2	'q'	int	8
'h'	int	2	'Q'	int	8
'H'	int	2	'f'	float	4
'i'	int	2	'd'	float	8
'I'	int	2			

Tujuan Penggunaan Tipe Data Khusus dalam Array:

❖ Efisiensi memori

Penggunaan tipe data khusus dalam array memberikan alokasi memori yang lebih efisien karena setiap elemen dalam array disimpan dengan jumlah byte yang tetap. Sebagai contoh, tipe 'i' untuk integer menyimpan nilai dengan ukuran 2, 4, atau 8 byte, tergantung ukuran tipe data integer yang dipilih. Pendekatan ini membuat array lebih hemat memori dibandingkan dengan list Python biasa, yang dapat menyimpan berbagai tipe data yang berbeda-beda.

❖ Kecepatan akses yang lebih cepat

Karena array menggunakan tipe data yang seragam, komputer dapat mengakses data dengan lebih cepat. Semua elemen dalam array disusun dalam blok memori yang berurutan dengan ukuran yang tetap, yang memungkinkan akses data lebih efisien dibandingkan dengan list Python yang memerlukan overhead lebih untuk menangani tipe data yang beragam.

❖ Pengolahan data yang terstruktur

Penggunaan tipe data khusus memungkinkan pengolahan data yang lebih terstruktur dan konsisten. Array yang hanya berisi satu jenis tipe data (seperti integer, float, atau karakter) memudahkan proses perhitungan, pemrosesan, serta penerapan operasi matematika atau fungsi lainnya, tanpa adanya kekhawatiran terhadap perbedaan tipe data antar elemen.

❖ Interoperabilitas dengan sistem eksternal

Array dengan tipe data tertentu, seperti 'i' untuk integer atau 'f' untuk float, memungkinkan Python untuk berinteraksi dengan bahasa pemrograman lain, seperti C atau Fortran, yang juga memerlukan format data tertentu. Hal ini mendukung pengolahan data yang lebih efisien dalam konteks komputasi numerik atau ilmiah, serta memungkinkan integrasi dengan sistem lain secara lebih mudah.

❖ Penggunaan dalam Komputasi Ilmiah dan Grafis

Tipe data khusus dalam array sering digunakan dalam pemrosesan gambar, analisis numerik, dan komputasi ilmiah, di mana tipe data seperti integer dan float digunakan untuk mengelola dan memproses data dalam jumlah besar dengan kecepatan tinggi serta penggunaan memori yang lebih efisien.

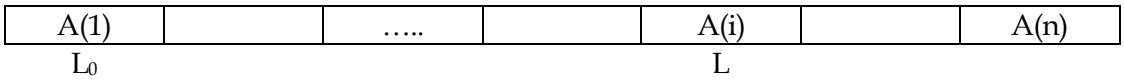
D. Perhitungan Array

Perhitungan Manual Alamat Array adalah proses yang digunakan untuk menentukan posisi atau alamat memori elemen array dalam memori komputer. Array dalam bahasa pemrograman biasanya disimpan dalam bentuk blok memori yang berurutan, sehingga perhitungan alamat elemen array dapat dilakukan secara matematis.

Vektor (Array 1 Dimensi)

Untuk sebuah vektor n elemen, yang tiap elemennya membutuhkan c byte, maka total memori yang dialokasikan sebesar c * n byte dengan struktur alokasi vektor. Rumus perhitungan alamat array satu dimensi adalah sebagai berikut: **L=Lo+c(i−1)**. Dimana:

L: Alamat elemen ke-i dalam array (alamat yang ingin dicari).
Lo: Alamat awal elemen pertama dalam array (base address).
c: Ukuran dalam byte untuk setiap elemen dalam array.
i: Indeks elemen yang dicari (indeks dimulai dari 1).
A[i]: Lokasi elemen ke-i dalam array.



Gambar 4.10: Ilustrasi Alamat Array Satu Dimensi

Gambar 4.10 menggambarkan ilustrasi alamat array satu dimensi, di mana array tersebut disusun secara linear dalam memori. L_0 adalah alamat awal (*base address*) dari array, yaitu lokasi tempat elemen pertama A(1) disimpan dalam memori. Kemudian, elemen-elemen lainnya seperti A(i) dan A(n) akan disusun secara berurutan setelahnya, berdasarkan ukuran elemen yang konsisten. Setiap elemen array disimpan di dalam blok memori yang berurutan dengan jarak antara elemen yang dihitung berdasarkan ukuran tiap elemen. L adalah alamat elemen ke-i dalam array, yang dihitung berdasarkan alamat awal (L0) ditambah dengan ukuran elemen yang dikali dengan indeks ke-i.

Contoh: Misalkan memiliki array dengan alamat awal (Lo) sebesar 1000, dan ukuran tiap elemen c adalah 4 byte (misalnya, untuk tipe data integer). Apabila ingin mencari alamat elemen ke-3 (misalnya A[3]), maka:

L=1000+4(3−1)=1000+8=1008

Dengan demikian, alamat elemen ke-3 adalah 1008.

Rumus perhitungan alamat array satu dimensi ini sangat penting untuk memahami bagaimana komputer mengelola dan mengakses data dalam array, serta untuk optimasi akses memori terhadap aplikasi yang membutuhkan manipulasi data dalam jumlah besar secara efisien.

Soal 1: Diberikan sebuah array satu dimensi sebagai berikut: A = [10, 20, 30, 40, 50]. Base Address (Lo) array ini adalah 1000. Ukuran elemen adalah 4 byte (misalnya, tipe data integer 32-bit). Hitunglah alamat elemen ke-3 (A[3]) dalam array tersebut!

Soal 2: Diberikan sebuah array satu dimensi dengan elemen: B = [15, 25, 35, 45, 55, 65]. Base Address (Lo) array ini adalah 2000. Ukuran elemen adalah 2 byte. Hitunglah alamat elemen ke-4 (A[4]) dalam array

Soal 3: Diberikan array berikut: C = [5, 10, 15, 20, 25]. Base Address (Lo) adalah 3000. Ukuran elemen adalah 2 byte. Tentukan alamat elemen ke-2 (A[2]) dalam array ini!

Soal 4: Diberikan array berikut: D = [1, 3, 5, 7, 9, 11]. Base Address (Lo) adalah 5000. Ukuran elemen adalah 4 byte. Hitunglah alamat elemen ke-5 (A[5]) dalam array ini!

Soal 5: Diberikan array E yang terdiri dari n elemen: E = [12, 24, 36, 48, 60, 72]. Base Address (Lo) adalah 7000. Ukuran elemen adalah 4 byte. Hitunglah alamat elemen ke-6 (A[6]) dalam array tersebut.

Array Dua Dimensi

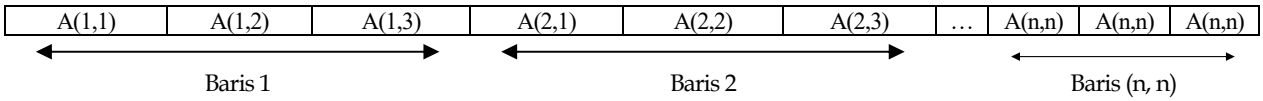
Array dua dimensi adalah struktur data yang memiliki dua indeks, yakni baris dan kolom, untuk mengakses elemen-elemen yang tersimpan dalam array tersebut. Dalam konteks penyimpanan array dua dimensi di memori komputer, ada dua cara utama yang digunakan, yaitu Row Major Order (RMO) dan Column Major Order (CMO).

1. Row Major Order (RMO)

Row Major Order adalah method penyimpanan array dua dimensi di mana elemen-elemen dalam **baris** yang sama disusun secara berurutan di memori. Dengan kata lain, seluruh elemen dalam baris pertama disimpan terlebih dahulu, kemudian diikuti oleh elemen-elemen dalam baris kedua, dan seterusnya. Kelebihan RMO yaitu method ini lebih cocok untuk aplikasi yang sering mengakses elemen-elemen dalam satu baris sekaligus, seperti dalam pengolahan citra atau matriks dalam komputasi ilmiah. Rumus yang digunakan untuk mencari alamat elemen ke-i dan ke-j dalam array dua dimensi adalah **L=Lo+{(i−1)×m+(j−1)}×c**

Dimana:

L = Alamat elemen A[i][j] yang dicari.
Lo = Alamat awal (base address) elemen pertama dalam array.
i = Indeks baris, yang menunjukkan baris ke-i (dimulai dari 1 hingga n).
j = Indeks kolom, yang menunjukkan kolom ke-j (dimulai dari 1 hingga n).
m = Jumlah kolom dalam array (dimensi kedua array).
c = Ukuran satu elemen array dalam byte.



Gambar 4.11: Ilustrasi Alamat Array Dua Dimensi secara Row Major Order (RMO)

Gambar 4.10 menunjukkan ilustrasi penyimpanan array dua dimensi menggunakan RMO. Dalam method RMO, elemen-elemen array disimpan dalam memori secara berurutan berdasarkan baris, di mana seluruh elemen dari baris pertama disimpan terlebih dahulu, diikuti oleh seluruh elemen dari baris kedua, dan seterusnya hingga baris terakhir. Array dua dimensi memiliki n baris dan n kolom, yang diwakili oleh A(i,j), dimana i adalah indeks baris dan j adalah indeks kolom. Elemen A(1,1), A(1,2), A(1,3), ... adalah elemen-elemen yang ada dalam baris pertama, dan setelah seluruh elemen baris pertama disimpan secara berurutan dalam memori, penyimpanan kemudian dilanjutkan dengan elemen-elemen baris kedua dan seterusnya. Proses penyimpanan data dalam RMO memudahkan pengaksesan data secara efisien apabila program sering mengakses elemen-elemen dalam baris yang sama. Dalam hal ini, akses ke elemen dalam satu baris akan lebih cepat karena elemen-elemen tersebut disusun secara berurutan dalam memori.

Contoh:

Diberikan array dua dimensi dengan ukuran 2x3 (2 baris dan 3 kolom) seperti berikut:

$$\begin{bmatrix} A(1,1) & A(1,2) & A(1,3) \\ A(2,1) & A(2,2) & A(2,3) \end{bmatrix}$$

Dengan informasi berikut:

Base Address (Lo) = 1000. Ukuran elemen (c) = 4 byte (misalnya integer 32-bit). Jumlah kolom (m) = 3. Hitunglah alamat elemen A(2,2) menggunakan Row Major Order (RMO).

Penyelesaian:

Untuk menghitung alamat elemen menggunakan Row Major Order, gunakan rumus berikut: $L=Lo+\{(i-1)\times m+(j-1)\}\times c$

Langkah-langkah Perhitungan:

Base Address (Lo) = 1000
Indeks baris (i) = 2 (elemen dalam baris ke-2).
Indeks kolom (j) = 2 (elemen dalam kolom ke-2).
Jumlah kolom (m) = 3.
Ukuran elemen (c) = 4 byte.

Sekarang substitusikan nilai-nilai tersebut ke dalam rumus:

$L=1000+\{(2-1)\times 3+(2-1)\}\times 4$
$L=1000+\{1\times 3+1\}\times 4$
$L=1000+\{3+1\}\times 4$
$L=1000+16=1016$

Hasil:

Alamat elemen A(2,2) adalah 1016.

RMO

Soal 1:
Diberikan array dua dimensi berikut dengan ukuran 2x3. Dengan informasi berikut:
Base Address (Lo) = 1000. Ukuran elemen (c) = 4 byte. Jumlah kolom (m) = 3.
Hitunglah alamat elemen A(1,3).

Soal 2:
Diberikan array dua dimensi dengan ukuran 3x2. Dengan informasi berikut:
Base Address (Lo) = 2000. Ukuran elemen (c) = 2 byte. Jumlah kolom (m) = 2.
Hitunglah alamat elemen B(3,1).

Soal 3:
Diberikan array dua dimensi dengan ukuran 4x4. Dengan informasi berikut:
Base Address (Lo) = 3000. Ukuran elemen (c) = 4 byte. Jumlah kolom (m) = 4.
Hitunglah alamat elemen C(3,4).

Soal 4:
Diberikan array dua dimensi dengan ukuran 2x5. Dengan informasi berikut:
Base Address (Lo) = 4000. Ukuran elemen (c) = 2 byte. Jumlah kolom (m) = 5.
Hitunglah alamat elemen D(2,4).

Soal 5:
Diberikan array dua dimensi dengan ukuran 3x3. Dengan informasi berikut:
Base Address (Lo) = 5000. Ukuran elemen (c) = 2 byte. Jumlah kolom (m) = 3.
Hitunglah alamat elemen E(1,3).

2. Column Major Order (CMO)

Column Major Order adalah method penyimpanan array dua dimensi di mana elemen-elemen dalam **kolom** yang sama disusun secara berurutan di memori. Dengan kata lain, seluruh elemen dalam kolom pertama disimpan terlebih dahulu, kemudian diikuti oleh elemen-elemen dalam kolom kedua, dan seterusnya. Kelebihan CMO adalah method ini lebih efisien untuk aplikasi yang sering mengakses elemen-elemen dalam satu kolom, seperti dalam algoritma numerik atau pemrograman yang berhubungan dengan data berbentuk matriks kolumnar.

Gambar 4.12: Ilustrasi Alamat Array Dua Dimensi secara Column Major Order (CMO)

Gambar 4.12 menggambarkan ilustrasi penyimpanan array dua dimensi menggunakan CMO. Dalam CMO, elemen-elemen array disimpan dalam memori secara berurutan berdasarkan kolom, di mana seluruh elemen dari kolom pertama disimpan terlebih dahulu, diikuti oleh seluruh elemen dari kolom kedua, dan seterusnya hingga kolom terakhir. Array dua dimensi memiliki n baris dan n kolom, yang diwakili oleh A(i,j), dimana i adalah indeks baris dan j adalah indeks kolom. Elemen A(1,1), A(2,1), A(3,1), ... adalah elemen-elemen dalam kolom pertama, dan setelah seluruh elemen kolom pertama disimpan secara berurutan dalam memori, penyimpanan kemudian dilanjutkan dengan elemen-elemen kolom kedua dan seterusnya. Proses penyimpanan data dalam CMO akan lebih efisien jika program sering mengakses elemen-elemen dalam kolom yang sama, karena elemen-elemen dalam kolom tersebut disusun secara berurutan dalam memori. Rumus yang digunakan untuk menghitung alamat elemen A[i][j] dalam array dua dimensi yang disusun menggunakan CMO adalah: **$L=Lo+\{(i-1)+(j-1)\times n\}\times c$**

Dimana:
L adalah alamat elemen A[i][j] yang dicari.
Lo adalah alamat awal (*base address*) dari elemen pertama dalam array.
i adalah indeks baris.
j adalah indeks kolom.
n adalah jumlah baris dalam array.
c adalah ukuran satu elemen dalam byte.
Lo adalah alamat elemen pertama dalam array (*base address*).

Soal:
Diberikan array dua dimensi dengan ukuran 3x3 (3 baris dan 3 kolom) sebagai berikut:

$$X = \begin{bmatrix} 3 & 5 & 7 \\ 8 & 2 & 9 \\ 4 & 6 & 1 \end{bmatrix}$$

Dengan informasi berikut:
Base Address (Lo) = 1000. Ukuran elemen (c) = 4 byte (misalnya integer 32-bit). Jumlah baris (n) = 3. Hitunglah alamat elemen A(2,3) menggunakan CMO!

Penyelesaian:
Untuk menghitung alamat elemen menggunakan CMO, gunakan rumus berikut:
$L=Lo+\{(i-1)+(j-1)\times n\}\times c$

Langkah-langkah Perhitungan:
Base Address (Lo) = 1000
Indeks baris (i) = 2 (elemen dalam baris ke-2).
Indeks kolom (j) = 3 (elemen dalam kolom ke-3).
Jumlah baris (n) = 3.
Ukuran elemen (c) = 4 byte.
Sekarang, substitusikan nilai-nilai tersebut ke dalam rumus:
$L=1000+\{(2-1)+(3-1)\times 3\}\times 4$

L=1000+{1+2×3}×4

L=1000+{1+6}×4=1000+7×4=1000+28=1028

Hasil: Alamat elemen A(2,3) adalah 1028.

Python memiliki jenis tipe data yang dapat direpresentasikan mirip seperti array, tetapi memiliki karakteristik dan keunikannya tersendiri antara lain list, tuple, set, range, dictionary, dan string.

Soal 1:
Diberikan array dua dimensi dengan ukuran 3x3
Dengan informasi berikut: Base Address (Lo) = 2000. Ukuran elemen (c) = 4 byte. Jumlah baris (n) = 3.
Hitunglah alamat elemen A(2,3) menggunakan CMO.

Soal 2:
Diberikan array dua dimensi dengan ukuran 4x2.
Dengan informasi berikut: Base Address (Lo) = 1500. Ukuran elemen (c) = 2 byte. Jumlah baris (n) = 4.
Hitunglah alamat elemen B(3,2) menggunakan CMO.

Soal 3:
Diberikan array dua dimensi dengan ukuran 3x3
Dengan informasi berikut: Base Address (Lo) = 3000. Ukuran elemen (c) = 4 byte. Jumlah baris (n) = 3.
Hitunglah alamat elemen C(2,2) menggunakan CMO.

Soal 4:
Diberikan array dua dimensi dengan ukuran 5x3. 4
Dengan informasi berikut: Base Address (Lo) = 4000. Ukuran elemen (c) = 2 byte. Jumlah baris (n) = 5.
Hitunglah alamat elemen D(4,3) menggunakan CMO.

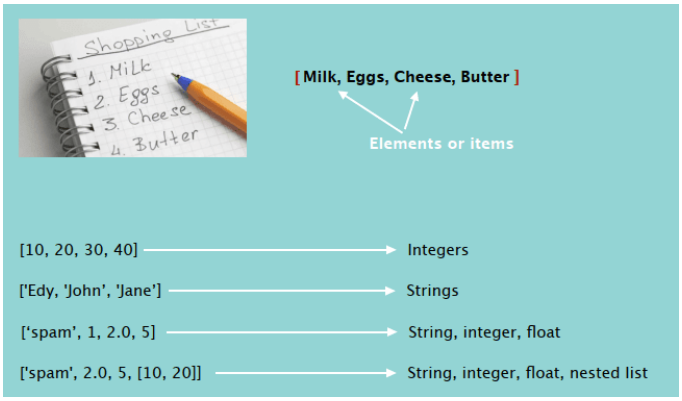
Soal 5:
Diberikan array dua dimensi dengan ukuran 2x4.
Dengan informasi berikut: Base Address (Lo) = 5000. Ukuran elemen (c) = 1 byte. Jumlah baris (n) = 2.
Hitunglah alamat elemen E(2,4) menggunakan CMO.

4.2 List

List adalah **koleksi yang terurut dan dapat diubah**. Memungkinkan elemen duplikat. **Dipisahkan koma (item) antara tanda kurung siku**. List merupakan struktur data berurutan yang paling sering digunakan di Python. List bisa menyimpan berbagai tipe data dalam satu struktur, seperti angka, string, objek, dan sebagainya. **List bersifat dinamis, artinya ukuran list dapat berubah sesuai kebutuhan**. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Gambar 4.13 menunjukkan contoh tipe data sequence dalam Python, menggunakan list sebagai ilustrasi. Terdapat contoh list belanjaan yang berisi empat item yaitu milk, eggs, cheese, dan butter. Daftar ini disimpan dalam sebuah list yang ditandai dengan tanda kurung siku [], dan setiap elemen dalam list disebut sebagai items atau elemen yang bisa berupa berbagai jenis data.

Terdapat beberapa contoh list yang menyimpan berbagai tipe data. List pertama berisi angka-angka bulat seperti 10, 20, 30, 40, yang semuanya termasuk dalam kategori integers. List kedua berisi elemen-elemen berupa string, seperti nama-nama 'Edy', 'John', 'Jane'. List ketiga mencakup elemen dengan berbagai tipe data, yaitu string ('spam'), integer (1), float (2.0), dan angka 5. Sedangkan list keempat menyimpan berbagai jenis data, termasuk string ('spam'), float (2.0, 5.0), dan sebuah nested list ([10, 20]), yang merupakan list di dalam list.



Gambar 4.13: Ilustrasi Penyimpanan Elemen List (Karimov, 2022)

Pembuatan List:

List dibuat menggunakan tanda kurung siku []. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Elemen List:

Elemen-elemen dalam list bersifat terurut (*ordered*), dapat diubah (*changeable*), dan memungkinkan adanya nilai duplikat. Setiap elemen dalam list memiliki indeks yang dimulai dari [0] untuk elemen pertama, [1] untuk elemen kedua, dan seterusnya. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Terurut (*Ordered*):

Ketika dikatakan bahwa list bersifat terurut, itu berarti elemen-elemen dalam list memiliki urutan yang sudah ditentukan, dan urutan tersebut tidak akan berubah. Apabila menambahkan elemen baru ke dalam list, elemen tersebut akan diletakkan di bagian akhir list. Catatan: Meskipun ada beberapa method list yang dapat mengubah urutan elemen, secara umum, urutan elemen dalam list tidak akan berubah. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Dapat Diubah (*Changeable*):

List bersifat dapat diubah, yang berarti dapat mengubah, menambah, atau menghapus elemen-elemen dalam list setelah list tersebut dibuat. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Memungkinkan Duplikasi (*Allow Duplicates*):

Karena elemen-elemen dalam list diindeks, list dapat memiliki elemen yang sama lebih dari satu kali. Hal ini memungkinkan adanya duplikasi nilai dalam list. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Panjang List (*List Length*):

Untuk mengetahui berapa banyak elemen yang ada dalam sebuah list, gunakan fungsi **len()**. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Tipe Data Elemen List:

Elemen-elemen dalam list dapat memiliki tipe data apa pun, yang memungkinkan list untuk menyimpan berbagai jenis data dalam satu struktur. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Konstruktor list():

Selain menggunakan tanda kurung siku, dapat membuat list baru dengan menggunakan konstruktor **list().** Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

(a) Mengakses Elemen List

Elemen-elemen dalam list diindeks, dapat diakses dengan merujuk nomor indeksnya. Catatan: elemen pertama memiliki indeks 0. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

♥ **Indeks Negatif**

Indeks negatif berarti memulai penghitungan dari ujung list. Catatatn: -1 merujuk elemen terakhir, -2 merujuk elemen kedua dari belakang, dan seterusnya. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

♥ **Rentang Indeks**

Rentang indeks dilakukan dengan cara menentukan indeks mulai dan indeks akhir dari rentang tersebut. Saat menentukan rentang, nilai yang dikembalikan akan berupa list baru dengan elemen-elemen yang ditentukan. Catatan: pencarian akan dimulai indeks 2 (termasuk) dan berakhir di indeks 5 (tidak termasuk). Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

♥ **Meninggalkan Nilai Mulai atau Akhir**

Apabila tidak menyebutkan nilai mulai, maka rentang akan dimulai dari elemen pertama. Jika tidak menyebutkan nilai akhir, rentang akan berlanjut hingga akhir list. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

♥ **Rentang Indeks Negatif**

List juga dapat menggunakan indeks negatif jika ingin memulai pencarian dari ujung list. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

♥ **Memeriksa Apakah Elemen Ada dalam List**

Untuk memeriksa apakah elemen yang ditentukan ada dalam sebuah list, dapat menggunakan kata kunci **in**. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

(b) Perubahan List Item

Untuk mengubah nilai dari elemen tertentu dalam list, dapat merujuk nomor indeks dari elemen tersebut. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

⌘ Mengubah Rentang Nilai Elemen

Untuk mengubah nilai elemen-elemen dalam rentang tertentu, dapat mendefinisikan list baru dengan nilai-nilai baru, dan merujuk rentang indeks tempat yang ingin disisipkan nilai baru tersebut. Jika jumlah elemen yang disisipkan lebih banyak daripada yang diganti, elemen-elemen baru akan disisipkan ke lokasi yang ditentukan, dan elemen yang tersisa akan bergerak sesuai dengan urutan tersebut. **Catatan**: panjang list akan berubah jika jumlah elemen yang disisipkan tidak sesuai dengan jumlah elemen yang diganti. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Jika jumlah elemen yang disisipkan lebih sedikit daripada yang diganti, elemen baru akan disisipkan ke posisi yang ditentukan, dan sisa elemen akan bergerak sesuai dengan urutan. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

⌘ Menyisipkan Elemen

Untuk menyisipkan elemen baru ke dalam list tanpa mengganti elemen yang sudah ada, dapat menggunakan method insert(). Method **insert()** menyisipkan elemen dalam indeks yang ditentukan.

```
# Menyisipkan elemen indeks tertentu
my_list = ['apel', 'jeruk', 'pisang']
my_list.insert(1, 'mangga')
print(my_list) # Output: ['apel', 'mangga', 'jeruk', 'pisang']
```

Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Catatan: Seperti yang terlihat dalam contoh di atas, setelah penyisipan, list berisi empat elemen.

(c) Menambah Elemen List

Menambahkan elemen list dapat menggunakan tifa fungsi yaitu method append(), insert(), dan extend(). Method append() digunakan untuk menambahkan elemen ke akhir list. Method insert() digunakan untuk menyisipkan elemen dalam indeks tertentu dalam list. Method extend() digunakan untuk menambahkan elemen dari list lain atau objek iterable lainnya ke akhir list yang ada.

♣ Menyisipkan Elemen dengan append()

Untuk menambahkan elemen ke akhir list, maka dapat menggunakan method **append()**. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

♣ Menyisipkan Elemen dengan insert()

Untuk menyisipkan elemen indeks yang ditentukan, dapat menggunakan method **insert()**. Method insert() menyisipkan elemen ke indeks yang ditentukan, dan elemen lainnya method untuk memberi ruang terhadap elemen yang baru disisipkan. Catatan: setelah penyisipan, list berisi empat elemen. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

♣ **Memperluas List dengan extend()**

Untuk menambahkan elemen-elemen dari list lain ke dalam list yang ada, dapat menggunakan method **extend()**. Elemen-elemen tersebut akan ditambahkan di akhir list. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

♣ **Menambahkan Iterable Apa Saja dengan extend()**

Method extend() tidak hanya digunakan untuk menambahkan elemen dari list lain, tetapi juga dapat menambahkan objek iterable lainnya, seperti tuple, set, atau dictionary. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

(d) Menghapus Elemen List

Menghapus elemen list dapat menggunakan tiga method yaitu method remove(), pop(), del, dan clear(). Method remove() digunakan untuk menghapus elemen berdasarkan nilai yang ditentukan. Method pop() digunakan untuk menghapus elemen berdasarkan indeks, dan jika indeks tidak ditentukan, elemen terakhir akan dihapus. Method del dapat menghapus elemen dalam indeks tertentu, atau menghapus seluruh list. Method clear() digunakan untuk mengosongkan list tanpa menghapus objek list itu sendiri. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

🖉 **Menghapus Elemen yang ditentukan dengan remove()**

Method remove() digunakan untuk menghapus elemen yang memiliki nilai tertentu dari dalam list. Jika terdapat lebih dari satu elemen dengan nilai yang sama, remove() akan menghapus kemunculan pertama elemen tersebut. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

🖉 **Menghapus Elemen Indeks yang Ditentukan dengan pop()**

Method pop() digunakan untuk menghapus elemen berdasarkan indeks yang ditentukan. Jika indeks tidak ditentukan, maka pop() akan menghapus elemen terakhir dalam list. Jika indeks tidak ditentukan, elemen terakhir akan dihapus.

Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

🖉 **Menghapus Elemen Indeks yang Ditentukan dengan del**

Kata kunci del juga dapat digunakan untuk menghapus elemen indeks tertentu dalam list. Selain itu, del dapat digunakan untuk menghapus seluruh list. Jika ingin menghapus seluruh list, dapat menggunakan del. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

🖉 **Mengosongkan List dengan clear()**

Method clear() digunakan untuk mengosongkan seluruh isi list. List tersebut tetap ada, namun tidak memiliki elemen lagi di dalamnya.

(e) Perulangan List

Perulangan dalam list dilakukan dengan beberapa cara yaitu 1)for loop adalah cara umum untuk melakukan perulangan melalui elemen dalam list. 2) Perulangan berdasarkan indeks dapat dilakukan menggunakan range() dan len(). 3) while loop juga dapat digunakan untuk melakukan perulangan dalam list dengan merujuk ke indeks dan memastikan indeks meningkat setelah setiap iterasi. 4) List comprehension menyediakan sintaks yang lebih singkat dan elegan untuk melakukan perulangan dan menghasilkan hasil dari perulangan tersebut.

Melakukan Perulangan terhadap Elemen List Menggunakan for Loop

Perulangan melalui elemen-elemen dalam list dapat dilakukan dengan menggunakan for loop. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Melakukan Perulangan Berdasarkan Nomor Indeks

Selain menggunakan elemen langsung, juga dapat melakukan perulangan melalui list dengan merujuk ke nomor indeks elemen. Gunakan fungsi range() dan len() untuk membuat iterable yang sesuai. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Melakukan Perulangan Menggunakan while Loop

Selain menggunakan for loop, juga dapat melakukan perulangan dalam list dengan menggunakan while loop. Gunakan fungsi len() untuk menentukan panjang list, kemudian mulai dari indeks 0 dan lakukan perulangan dengan merujuk ke indeks list. Pastikan untuk menambah indeks sebanyak 1 setelah setiap iterasi agar perulangan tidak berlangsung selamanya. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Melakukan Perulangan Menggunakan List Comprehension

List Comprehension menawarkan sintaksis yang lebih singkat untuk melakukan perulangan dalam list. Ini memungkinkan untuk membuat sebuah list baru dengan hasil yang berasal dari perulangan elemen-elemen dalam list. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

(f) ListComprehension

List comprehension menawarkan sintaksis yang lebih singkat ketika ingin membuat list baru berdasarkan nilai-nilai dari list yang sudah ada. Contoh: seseorang memiliki sebuah list berisi nama buah, dan ingin membuat list baru yang hanya berisi buah yang mengandung huruf "a" dalam namanya. Tanpa menggunakan list comprehension, maka harus menulis pernyataan for dengan tes kondisi di dalamnya. Namun, dengan **list comprehension**, semua itu dapat dilakukan hanya dalam satu baris kode. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Sintaks List Comprehension

Sintaksis untuk list comprehension adalah sebagai berikut:

```
newlist = [ekspresi for item in iterable if kondisi == True]
```

Sintaks ini menghasilkan list baru, tanpa mengubah list lama.

Kondisi (Condition)

Kondisi berfungsi sebagai filter yang hanya menerima elemen-elemen yang dievaluasi menjadi True. Sebagai contoh, jika ingin membuat list baru yang tidak mengandung elemen "apel", maka bisa menambahkan kondisi if x != "apple", yang akan mengembalikan True untuk semua elemen selain "apel", membuat list baru berisi semua buah kecuali "apel". Kondisi ini bersifat opsional dan dapat dihilangkan jika tidak diperlukan. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Iterable

Iterable bisa berupa objek iterable apa pun, seperti list, tuple, set, dan sebagainya. Sebuah **iterable** adalah objek yang dapat diiterasi elemen-elemennya, yang dalam hal ini adalah list yang sedang diproses oleh list comprehension. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Ekspresi (*Expression*)

Ekspresi adalah elemen yang sedang diproses dalam iterasi, namun juga merupakan hasil akhir yang dapat dimanipulasi sebelum menjadi elemen dalam list baru. Hasil tersebut sesuai keinginan. Ekspresi ini juga dapat mencakup kondisi yang tidak hanya berfungsi sebagai filter, tetapi sebagai cara untuk memanipulasi hasil yang diinginkan.

Contoh di atas, ekspresi tersebut mengubah elemen menjadi huruf kapital jika mengandung huruf "a", dan membiarkannya tetap seperti semula jika tidak. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

(g) Mengurutkan List

Method **sort()** digunakan untuk mengurutkan list dalam urutan menaik secara default. Untuk urutan menurun, gunakan argumen **reverse=True**. Fungsi pengurutan kustom dapat diterapkan dengan menggunakan **key=function** untuk mengurutkan berdasarkan kriteria tertentu, seperti panjang elemen. Pengurutan tidak sensitif huruf kapital dapat dilakukan dengan menggunakan fungsi **str.lower** sebagai fungsi kunci. Method **reverse()** digunakan untuk membalikkan urutan list tanpa mengubah urutan pengurutannya.

Mengurutkan List Secara Alfanumerik

Objek list di Python memiliki method sort() yang digunakan untuk mengurutkan list secara alfanumerik dalam urutan menaik (ascending) secara default. Pengurutan ini mencakup angka dan huruf, dengan angka yang diurutkan terlebih dahulu, diikuti oleh huruf sesuai urutannya. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Mengurutkan Secara Menurun (*Descending*)

Untuk mengurutkan list secara menurun, gunakan argumen kata kunci **reverse=True** dalam method **sort()**. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Kustomisasi Fungsi Pengurutan

Penyesuaian fungsi pengurutan dengan menggunakan argumen kata kunci **key=function**. Fungsi ini akan mengembalikan nilai yang akan digunakan untuk mengurutkan list (nilai terkecil pertama). Sebagai contoh, jika ingin mengurutkan jajanan pasar berdasarkan panjang namanya. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Pengurutan Tidak Sensitif Huruf Kapital

Secara default, method sort() bersifat sensitif terhadap huruf kapital, yang berarti huruf kapital akan diurutkan sebelum huruf kecil. Namun, bisa menggunakan fungsi **built-in** seperti **str.lower** untuk membuat pengurutan tidak sensitif terhadap huruf kapital. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Mengubah Urutan List

Jika ingin membalik urutan list tanpa mengubah urutannya, dapat menggunakan method **reverse()**, yang membalikkan urutan elemen-elemen yang ada dalam list. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

(h) Menyalin List

Dalam menyalin list digunakan beberapa method yaitu **copy()** adalah method yang dapat digunakan untuk menyalin list secara mandiri, tanpa merujuk ke list asli. **list()** juga dapat digunakan untuk menyalin list dengan cara yang sama. **Operator slicing [:]** adalah cara lain yang efektif untuk menyalin list. Dengan menggunakan method-method tersebut, perubahan list asli tidak akan memengaruhi salinan yang telah dibuat.

Menyalin sebuah list tidak dapat dilakukan dengan menuliskan list2 = list1, karena list2 akan menjadi referensi ke list1, yang berarti perubahan yang dilakukan dalam list1 akan otomatis berlaku juga terhadap list2.

Menggunakan Method copy()

Untuk menyalin list secara mandiri (bukan referensi), dapat menggunakan method **copy()** yang sudah disediakan oleh Python. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Menggunakan Method list()

Cara lain untuk menyalin list adalah dengan menggunakan method **list()**, yang menghasilkan salinan baru dari list yang ada. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Menggunakan Operator Slicing :

Menyalin list juga dapat dilakukan menggunakan **operator slicing (:)**. Cara ini membuat salinan shallow copy dari list. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

(i) Menggabungkan List

Ada beberapa cara untuk menggabungkan dua atau lebih list di Python. Salah satu cara yang paling mudah adalah dengan menggunakan operator +. Cara lainnya adalah dengan menambahkan elemen-elemen dari list2 ke dalam list1, satu per satu menggunakan method append(). Atau, juga bisa menggunakan method extend(), yang bertujuan untuk menambahkan elemen-elemen dari satu list ke dalam list lain.

Menggunakan Operator +

Operator + dapat digunakan untuk menggabungkan dua list menjadi satu list baru. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Menggunakan Append()

Method append() dapat digunakan untuk menambahkan elemen dari list2 ke dalam list1 satu per satu. Dengan cara ini, list1 akan diperbarui dengan elemen-elemen dari list2. Contoh dapat dilihat di https://colab.research.google.com/drive/18hD9hEWwJwjiNoca1I9AkGuF6FU-rwzx?usp=sharing.

Menggunakan Extend()

Method extend() digunakan untuk menambahkan elemen-elemen dari list2 ke dalam list1. Semua elemen dalam list2 akan ditambahkan ke akhir list1.

(j) Method List

Python memiliki serangkaian method bawaan (*built-in*) yang disajikan di Tabel 4.2.

Tabel 4.2: Method List *Built-In*

Method	Deskripsi
append()	Menambahkan elemen di akhir list
clear()	Menghapus semua elemen dalam list
copy()	Mengembalikan salinan dari list
count()	Mengembalikan jumlah elemen yang memiliki nilai tertentu
extend()	Menambahkan elemen dari list lain (atau iterable lainnya) ke akhir list
index()	Mengembalikan indeks dari elemen pertama yang memiliki nilai tertentu
insert()	Menambahkan elemen dalam posisi tertentu dalam list
pop()	Menghapus elemen dalam posisi tertentu dalam list
remove()	Menghapus elemen dengan nilai tertentu
reverse()	Membalikkan urutan elemen dalam list
sort()	Mengurutkan list

4.3 Tuple

Tuple adalah **koleksi yang terurut dan tidak dapat diubah (*immutable*).** Biasanya digunakan untuk data yang tidak perlu diubah setelah dibuat. Tupel menggunakan **tanda kurung ()**. Contoh: t = 'a', 'b', 'c', 'd', ‘e’ – bentuk array. t = ('a', 'b', 'c', 'd', 'e’) – bentuk list. Tuple digunakan untuk menyimpan beberapa elemen dalam satu variabel.

Pembuatan Tuple:

Tuple ditulis menggunakan tanda kurung bulat (). Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

Elemen Tuple:

Elemen-elemen dalam tuple bersifat terurut, tidak dapat diubah (*immutable*), dan memungkinkan adanya nilai duplikat. Elemen-elemen dalam tuple memiliki indeks, di mana elemen pertama memiliki indeks [0], elemen kedua memiliki indeks [1], dan seterusnya. Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

Terurut (*Ordered*)

Ketika mengatakan bahwa tuple bersifat **terurut**, artinya elemen-elemen dalam tuple memiliki urutan yang telah ditentukan, dan urutan tersebut tidak akan berubah. Jika menambahkan elemen baru, elemen tersebut akan diletakkan ke posisi yang sudah ditentukan, dan urutan elemen lainnya tidak akan terpengaruh. Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

Tidak Dapat Diubah (*Unchangeable*)

Tuple bersifat tidak dapat diubah, yang berarti tidak bisa mengubah, menambah, atau menghapus elemen dalam tuple setelah tuple tersebut dibuat. Hal ini berbeda dengan list, yang bersifat mutable dan memungkinkan perubahan. Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

Memungkinkan Duplikasi (*Allow Duplicates*)

Karena tuple bersifat terurut dan diindeks, tuple dapat menyimpan elemen yang memiliki nilai yang sama lebih dari sekali. Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

Panjang Tuple (*Tuple Length*)

Untuk mengetahui berapa banyak elemen yang ada dalam sebuah tuple, dapat menggunakan fungsi **len()**. Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

Tuple dengan Satu Elemen

Untuk membuat tuple yang hanya berisi satu elemen, harus menambahkan koma setelah elemen tersebut. Tanpa koma, Python tidak akan menganggapnya sebagai tuple. Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

Tuple dengan Berbagai Tipe Data

Elemen dalam tuple bisa berupa berbagai jenis tipe data, baik string, angka, boolean, atau bahkan tuple lainnya (*nested tuple*). Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

Menggunakan Konstruktor Tuple()

Selain menggunakan tanda kurung bulat, juga bisa membuat tuple dengan menggunakan konstruktor **tuple()**. Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

A. Mengakses Tuple

Mengakses elemen dalam tuple dengan merujuk ke nomor indeks yang ada di dalam **tanda kurung siku []**. Catatan: Elemen pertama dalam tuple memiliki indeks 0. **Tuple** untuk mengakses elemen berdasarkan indeks, baik menggunakan indeks **positif** maupun **negatif**. Rentang indeks digunakan untuk mengakses elemen-elemen tertentu dalam tuple, serta memeriksa apakah suatu elemen ada dengan menggunakan kata kunci **in**. **Indeks negatif** berguna untuk mengakses elemen dari akhir tuple, sementara **rentang indeks** memungkinkan akses beberapa elemen sekaligus dalam urutan yang ditentukan. Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

Indeks Negatif

Indeks negatif berarti mulai menghitung dari akhir tuple. Nilai -1 merujuk elemen terakhir, -2 merujuk elemen kedua dari akhir, dan seterusnya. Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

Rentang Indeks

Rentang indeks dapat ditentukan dengan menyebutkan posisi mulai dan posisi akhir rentang tersebut. Ketika menentukan rentang, nilai yang dikembalikan adalah tuple baru yang berisi elemen-elemen yang sesuai dengan rentang indeks yang ditentukan. Catatan: Pencarian akan dimulai dari indeks 2 (termasuk) dan berakhir dalam indeks 5 (tidak termasuk). Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

Meninggalkan Nilai Mulai atau Akhir

Apabila tidak menyebutkan nilai awal, maka rentang akan dimulai dari elemen pertama. Jika tidak menyebutkan nilai akhir, maka rentang akan berlanjut hingga akhir tuple. Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

Rentang Indeks Negatif

Jika ingin memulai pencarian dari akhir tuple, dapat menggunakan indeks negatif. Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

Memeriksa Keberadaan Elemen dalam Tuple

Untuk memeriksa apakah suatu elemen ada dalam tuple, dapat menggunakan kata kunci **in**. Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

B. Mengubah Tuple

Tuple bersifat tidak dapat diubah (*immutable*), yang berarti tidak dapat diubah, menambah, atau menghapus elemen setelah tuple dibuat. Namun, ada beberapa solusi alternatif yang dapat digunakan untuk mengatasi keterbatasan ini yaitu mengubah atau menambah elemen dalam tuple melibatkan pengonversian tuple menjadi list, melakukan perubahan, dan mengonversinya kembali menjadi tuple. Tuple dapat digabungkan dengan tuple lainnya untuk menambah elemen baru. Menghapus elemen dalam tuple tidak memungkinkan secara langsung, tetapi bisa menggunakan method yang sama dengan menambahkan elemen baru. Menghapus tuple sepenuhnya menggunakan del.

Mengubah Nilai Tuple

Setelah tuple dibuat, maka tidak dapat diubah nilainya. Tuple adalah struktur data *immutable*, yang berarti setelah tuple didefinisikan, elemen-elemennya tidak dapat diubah. Namun, ada solusi alternatif yaitu mengonversi tuple menjadi list, mengubah list tersebut, dan kemudian mengonversinya kembali menjadi tuple. Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

Menambahkan Elemen

Karena tuple bersifat *immutable*, tuple tidak memiliki method append() seperti yang dimiliki list. Namun, masih bisa menambahkan elemen ke dalam tuple dengan cara-cara berikut:

1. Mengonversi menjadi list → mengonversi tuple menjadi list, menambahkan elemen yang diinginkan, dan mengonversinya kembali menjadi tuple.

2. Menambahkan tuple ke tuple lain → python juga mengijinkan untuk menambahkan tuple ke tuple, sehingga dapat membuat tuple baru dengan elemen yang ingin ditambahkan, dan menggabungkannya dengan tuple yang sudah ada.

Catatan: **Ketika membuat tuple dengan hanya satu elemen, pastikan untuk menambahkan koma setelah elemen**, jika tidak, Python tidak akan mengenali itu sebagai tuple.

Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

Menghapus Elemen

Python tidak menijinkan menghapus elemen dalam tuple, karena tuple bersifat immutable. Namun, Anda bisa menggunakan solusi yang sama seperti yang digunakan untuk mengubah dan menambah elemen tuple, yaitu dengan mengonversinya menjadi list, menghapus elemen yang diinginkan, dan mengonversinya kembali menjadi tuple. Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

Menghapus Tuple Sepenuhnya

Python juga mengijinkan untuk menghapus tuple sepenuhnya dengan menggunakan kata kunci **del**. Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

C. Unpack Tuple

Ketika membuat sebuah tuple, menetapkan nilai-nilai untuknya. Proses ini disebut dengan "packing" tuple. Namun, di Python, juga diperbolehkan untuk mengekstrak nilai-nilai tersebut kembali ke dalam variabel-variabel. Proses ini disebut dengan "unpacking". Catatan: Jumlah variabel yang digunakan untuk unpacking harus sesuai dengan jumlah elemen dalam tuple. Jika jumlah variabel lebih sedikit, maka harus menggunakan tanda bintang (*) untuk mengumpulkan nilai yang tersisa dalam bentuk list. Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

Menggunakan Asterisk (*) untuk Unpacking

Jika jumlah variabel lebih sedikit daripada jumlah nilai dalam tuple, maka bisa menambahkan **tanda bintang (*)** di depan variabel untuk mengumpulkan nilai yang tersisa dalam bentuk list. Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

Menempatkan Asterisk di Variabel Lain

Jika tanda bintang (*) diletakkan dalam variabel selain yang terakhir, Python akan menetapkan nilai-nilai ke variabel tersebut hingga jumlah nilai tersisa sesuai dengan jumlah variabel tersisa. Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

D. Perulangan Tuple

Perulangan tuple menggunakan for loop untuk mengakses setiap elemen dalam tuple. Fungsi range() dan len() untuk merujuk indeks elemen dalam tuple. Selain itu, while loop dapat digunakan untuk meningkatkan indeks setelah setiap iterasi agar perulangan berjalan dengan benar.

Melakukan Perulangan Elemen Tuple Menggunakan for Loop

Perulangan elemen-elemen dalam tuple dengan menggunakan for loop adalah cara paling sederhana untuk mengakses setiap elemen dalam tuple secara berurutan. Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

Melakukan Perulangan Berdasarkan Indeks

Perulangan elemen-elemen dalam tuple dengan merujuk ke nomor indeks elemen. Untuk melakukannya, maka bisa menggunakan fungsi range() dan len() untuk membuat iterable yang sesuai. Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

Melakukan Perulangan Menggunakan while Loop

Selain menggunakan for loop, juga dapat melakukan perulangan dalam tuple menggunakan while loop. Gunakan fungsi len() untuk mengetahui panjang tuple, kemudian mulai dari indeks 0 dan lakukan perulangan dengan merujuk dalam indeks-elemen tuple. Pastikan untuk menambah indeks sebanyak 1 setelah setiap iterasi untuk menghindari perulangan yang tidak berakhir. Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

E. Menggabungkan Tuple

Menggabungkan tuple dapat dilakukan dengan menggunakan operator +, yang akan menggabungkan dua atau lebih tuple menjadi satu tuple baru. Selain itu, juga dapat dilakukan dengan cara mengalikan tuple dilakukan menggunakan operator *, yang akan mengulang elemen-elemen dalam tuple sebanyak jumlah yang ditentukan, menghasilkan tuple yang lebih panjang.

Menggabungkan Dua Tuple

Untuk menggabungkan dua atau lebih tuple, maka dapat menggunakan operator +. Operator ini menggabungkan elemen-elemen dari tuple yang satu dengan tuple yang lainnya menjadi satu tuple baru. Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

Mengalikan Tuple

Jika ingin mengalikan isi dari suatu tuple dengan jumlah tertentu, maka dapat menggunakan operator *. Operator ini akan mengulang elemen-elemen dalam tuple sesuai dengan jumlah yang ditentukan. Contoh dapat dilihat di https://colab.research.google.com/drive/17dcQDmy6V3Tcr-gS0wmZq1tzDoVSUpWa?usp=sharing.

F. Method Tuple

Python memiliki dua method *built-in* dalam tuple yang ditampilkan di Tabel 4.3.

Tabel 4.3: Method Tuple *Built-In*

method	Deskripsi
count()	Mengembalikan jumlah kemunculan nilai tertentu dalam tuple
index()	Mencari tuple untuk nilai tertentu dan mengembalikan posisi tempat nilai ditemukan

4.4 Set

Set adalah salah satu tipe data bawaan di Python yang digunakan untuk menyimpan beberapa elemen dalam satu variabel. **Set adalah koleksi yang tidak terurut, tidak dapat diubah (*immutable*), dan tidak terindeks**. Catatan: Meskipun elemen dalam set tidak dapat diubah, tetapi masih dapat menghapus elemen dan menambahkan elemen baru ke dalam set.

Pembuatan Set:

Set dituliskan menggunakan **tanda kurung kurawal** {}. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

Elemen Set:

Elemen-elemen dalam set bersifat tidak terurut, tidak dapat diubah, dan tidak memungkinkan adanya nilai duplikat. Tidak Terurut adalah elemen-elemen dalam set tidak memiliki urutan yang tetap. Artinya, ketika mengakses elemen-elemen dalam set, urutannya bisa berbeda setiap kali. Tidak Dapat Diubah ialah elemen dalam set tidak dapat diubah setelah set dibuat. Namun, masih bisa menghapus atau menambah elemen dalam set. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

Tidak Terurut (Unordered)

Ketika mengatakan bahwa set bersifat tidak terurut, itu berarti elemen-elemen dalam set tidak memiliki urutan yang tetap. Oleh karena itu, elemen dalam set dapat muncul dalam urutan yang berbeda setiap kali mengaksesnya. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

Tidak Dapat Diubah (Unchangeable)

Set bersifat *immutable*, yang berarti tidak dapat mengubah elemen dalam set setelah set tersebut dibuat. Namun, dapat menghapus elemen dan menambah elemen baru. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

Tidak Memungkinkan Duplikasi (Duplicates Not Allowed)

Set tidak memungkinkan dua elemen dengan nilai yang sama. Jika mencoba untuk menambahkan elemen duplikat ke dalam set, Python hanya akan menyimpan satu elemen tersebut. Catatan: Nilai True dan 1 dianggap sebagai nilai yang sama dalam set, serta False dan 0 dianggap sebagai nilai yang sama. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

Mendapatkan Panjang Set

Untuk mengetahui berapa banyak elemen yang ada dalam sebuah set, maka dapat menggunakan fungsi **len()**. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

Set dengan Berbagai Tipe Data

Elemen dalam set bisa berupa berbagai jenis tipe data, seperti angka, string, dan tuple. Set juga dapat menyimpan berbagai tipe data dalam satu set. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

Menggunakan Konstruktor set()

Selain menggunakan tanda kurung kurawal {}, Anda juga dapat membuat set menggunakan konstruktor set(). Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

a. Mengakses Elemen Set

Mengakses elemen dengan merujuk ke indeks atau kunci seperti terhadap list atau dictionary. Namun, dapat melakukan perulangan melalui elemen-elemen dalam set menggunakan **for loop**, atau memeriksa apakah suatu nilai tertentu ada dalam set dengan menggunakan kata kunci **in**. Untuk memeriksa apakah elemen tertentu ada dalam set, maka dapat menggunakan kata kunci **in**. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

Mengubah Elemen dalam Set

Set bersifat *immutable*, yang berarti tidak dapat mengubah elemen-elemen yang sudah ada dalam set setelah set tersebut dibuat. Namun, dapat menambah elemen baru ke dalam set. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

b. Menambah Elemen Set

Set bersifat *immutable*, yang berarti tidak dapat mengubah elemen-elemen yang sudah ada dalam set setelah set tersebut dibuat. Namun, tetap dapat menambahkan elemen baru ke dalam set. Menambahkan elemen baru ke dalam set menggunakan method add(). Method update() digunakan untuk menambahkan elemen-elemen dari set lain atau objek iterable lainnya ke dalam set. Method update() mendukung penambahan dari berbagai jenis objek iterable, seperti tuple, list, atau dictionary, ke dalam set yang ada. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

Menambahkan Satu Elemen dengan add()

Untuk menambahkan satu elemen ke dalam set, maka dapat menggunakan method add(). Method ini akan menambahkan elemen baru ke dalam set jika elemen tersebut belum ada di dalam set. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

Menambahkan Elemen dari Set Lain dengan update()

Untuk menambahkan elemen dari set lain ke dalam set yang sudah ada, maka dapat menggunakan method update(). Method ini akan menambahkan semua elemen dari set lain ke dalam set yang diproses. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

c. Menghapus Elemen Set

Untuk menghapus elemen dalam set, maka dapat menggunakan method **remove()** atau **discard()**. Method remove() digunakan untuk menghapus elemen dari set dan akan menghasilkan error jika elemen tidak ditemukan. Method discard() juga digunakan untuk menghapus elemen, namun tidak menghasilkan error jika elemen tidak ada dalam set. **Catatan**: Jika elemen yang ingin dihapus tidak ada dalam set, method remove() akan menghasilkan error. Sementara itu, method **discard()** tidak akan menghasilkan error meskipun elemen yang ingin dihapus tidak ada dalam set. Selain itu, method pop() untuk menghapus elemen secara acak dari set, dan mengembalikan elemen yang dihapus. Namun, method ini akan menghapus elemen secara acak, sehingga pengguna tidak dapat memastikan elemen mana yang akan dihapus. Nilai yang dikembalikan oleh method pop() adalah elemen yang telah dihapus. **Catatan**: Karena set tidak terurut, maka tidak dapat mengetahui elemen mana yang akan dihapus saat menggunakan method pop(). Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

Menggunakan Method remove()

Method remove() digunakan untuk menghapus elemen tertentu dari set. Namun, jika elemen yang ingin dihapus tidak ada dalam set, Python akan mengeluarkan error KeyError. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

Menggunakan Method discard()

Method discard() digunakan untuk menghapus elemen tertentu dari set. Jika elemen yang ingin dihapus tidak ada dalam set, method ini tidak akan menghasilkan error. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

Menggunakan Method pop()

Method pop() digunakan untuk menghapus elemen dari set. Method ini akan menghapus elemen secara acak karena set tidak terurut. Pengguna tidak dapat mengetahui elemen mana yang akan dihapus. Nilai yang dikembalikan oleh method pop() adalah elemen yang telah dihapus. **Catatan**: Hasil output dari method pop() dapat berbeda setiap kali dijalankan kode karena set tidak terurut. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

d. Perulangan Set

Perulangan elemen-elemen dalam set menggunakan **for loop**. Set adalah koleksi data yang tidak terurut, sehingga perulangan akan menghasilkan elemen-elemen dalam urutan yang tidak dapat dipastikan. **for loop** adalah method yang paling umum digunakan untuk mengakses elemen-elemen dalam set. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

Meskipun while loop tidak umum digunakan untuk melakukan perulangan terhadap set, while loop dapat digunakan perulangan terhadap set dengan cara tertentu. Untuk melakukannya, maka harus menggunakan iterator, karena set tidak mendukung pengaksesan elemen berdasarkan indeks. Berikut adalah cara untuk menggunakan while loop dengan iterator terhadap set. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

e. Penggabungan Set

Ada beberapa cara untuk menggabungkan dua atau lebih set di Python. Beberapa method yang dapat digunakan antara lain union(), update(), intersection(), difference(), dan symmetric_difference().

Setiap method memiliki fungsionalitas yang berbeda dalam menggabungkan elemen-elemen set. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

Union (Penggabungan)

method union() mengembalikan sebuah set baru yang berisi semua elemen dari kedua set. Selain itu, juga dapat menggunakan operator | untuk mencapai hasil yang sama. Dan juga bisa menggunakan operator | sebagai pengganti method union(). Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

Menggabungkan Banyak Set

Semua method dan operator penggabungan dapat digunakan untuk menggabungkan beberapa set. Untuk menggunakan method, maka cukup menambahkan set lainnya dalam tanda kurung yang dipisahkan dengan koma. Jika menggunakan operator |, pisahkan set dengan operator **|.**

Atau, jika menggunakan operator |. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

Menggabungkan Set dengan Tuple atau Tipe Data Lain

Method union() memungkinkan penggabungan set dengan tipe data lain, seperti tuple, list, atau dictionary. Hasilnya tetap berupa set. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

Update (Memperbarui Set)

Method update() digunakan untuk menambahkan semua elemen dari satu set ke set lainnya. Perbedaan utama dengan union() adalah **bahwa update() mengubah set yang sudah ada, sedangkan union() mengembalikan set baru**. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

Intersection (Penggabungan Elemen yang Sama)

Method intersection() mengembalikan set baru yang hanya berisi elemen-elemen yang ada dalam kedua set. Selain itu, juga bisa menggunakan operator & untuk mencapai hasil yang sama. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

Difference (Perbedaan Set)

Method difference() mengembalikan set baru yang berisi elemen-elemen yang hanya ada dalam set pertama dan tidak ada di set lainnya. Selain itu, juga bisa menggunakan operator - untuk mencapai hasil yang sama. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

Symmetric Difference (Perbedaan Simetris)

Mehod symmetric_difference() mengembalikan set baru yang hanya berisi elemen-elemen yang ada dalam salah satu set tetapi tidak ada dalam kedua set. Selain itu, juga bisa menggunakan operator ^ untuk mencapai hasil yang sama. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

f. Frozenset

Frozenset adalah versi *immutable* dari set. Meskipun mirip dengan set, **frozenset memiliki elemen yang unik, tidak terurut, dan tidak dapat diubah**. Berbeda dengan set, elemen dalam frozenset tidak dapat ditambahkan atau dihapus setelah frozenset dibuat.

Membuat Frozenset

Pembuatan frozenset menggunakan konstruktor **frozenset()**, yang dapat menerima objek iterable (seperti list, tuple, atau set) untuk membuat frozenset. Contoh dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

Method Frozenset

Karena frozenset bersifat *immutable*, maka tidak dapat menambahkan atau menghapus elemen dari frozenset. Namun, frozenset mendukung semua operasi *non-mutating* yang dapat dilakukan dalam set, seperti union, intersection, difference, dan symmetric_difference. Contoh Penggunaan method dalam Frozenset dapat dilihat di https://colab.research.google.com/drive/1hA_yA5JDobLyxDGb0FGrZ2P5vkkDGX80?usp=sharing.

Perbedaan dengan Set

Berbeda dengan set, yang bersifat *mutable* (dapat diubah), frozenset tidak dapat dimodifikasi setelah dibuat. Pengguna tidak dapat menambahkan atau menghapus elemen dalam frozenset. Namun, frozenset tetap mendukung operasi yang tidak mengubah elemen-elemen di dalamnya, seperti union, intersection, dan difference.

```
# Mencoba menambahkan elemen dalam frozenset (akan menghasilkan error)
# bukit_indonesia.add('Bukit Tiga Warna') # AttributeError: 'frozenset' object has no attribute 'add'
```

Tabel 4.4: Method Frozenset

Method	Shortcut	Deskripsi
copy()		Mengembalikan salinan dangkal dari frozenset
difference()	-	Mengembalikan frozenset baru yang berisi selisih antara dua frozenset
intersection()	&	Mengembalikan frozenset baru yang berisi irisan antara dua frozenset
isdisjoint()		Mengembalikan apakah dua frozenset tidak memiliki irisan
issubset()	<= / <	Mengembalikan True jika frozenset ini merupakan subset (sebagian) dari frozenset lain
issuperset()	>= / >	Mengembalikan True jika frozenset ini merupakan superset (superior) dari frozenset lain
symmetric_difference()	^	Mengembalikan frozenset baru yang berisi perbedaan simetris antara dua frozenset
union()	|	Mengembalikan frozenset baru yang berisi gabungan tersebut.

Bersifat immutable berarti tidak dapat menambah atau menghapus elemen. Meskipun demikian, frozenset mendukung semua operasi yang tidak mengubah elemen-elemen dalam himpunan. Terdapat beberapa method Frozenset yang disajikan di Tabel 4.4.

Tabel 4.5: Method Set *Built-In*

Method	Shortcut	Deskripsi
add()		Menambahkan elemen ke dalam set
clear()		Menghapus semua elemen dari set
copy()		Mengembalikan salinan dari set
difference()	-	Mengembalikan set yang berisi selisih antara dua atau lebih set
difference_update()	-=	Menghapus elemen yang ada dalam set ini tetapi juga termasuk dalam set lainnya
discard()		Menghapus elemen yang ditentukan dari set
intersection()	&	Mengembalikan set yang berisi irisan dari dua set lainnya
intersection_update()	&=	Menghapus elemen yang ada dalam set ini yang tidak ada dalam set lainnya
isdisjoint()		Mengembalikan apakah dua set memiliki irisan atau tidak
issubset()	<=	Mengembalikan True jika semua elemen dari set ini ada dalam set lainnya
issuperset()	>=	Mengembalikan True jika semua elemen dari set lainnya ada dalam set ini
pop()		Menghapus elemen dari set
remove()		Menghapus elemen yang ditentukan dari set
symmetric_difference()	^	Mengembalikan set yang berisi perbedaan simetris antara dua set
symmetric_difference_update()	^=	Menyisipkan perbedaan simetris antara set ini dengan set lainnya
union()	|	Mengembalikan set yang berisi gabungan dari set tersebut.
update()	|=	Mengupdate set dengan gabungan set ini dan set lainnya.

g. Method Set

Python memiliki serangkaian method built-in yang dapat digunakan dalam set, ditampilkan di Tabel 4.5.

4.5 Range

Range adalah tipe data sequence yang digunakan untuk menghasilkan urutan angka. Range sering digunakan dalam looping dan iterasi. Fungsi **range()** menghasilkan objek range yang berisi urutan angka dalam rentang tertentu. Range sering digunakan dalam operasi looping dan iterasi, seperti dalam for loop, untuk mengulang sejumlah langkah atau melakukan iterasi berdasarkan urutan angka yang ditentukan.

```
my_range = range(1, 5) # menghasilkan 1, 2, 3, 4
```

Membuat Range

Fungsi range() dapat digunakan untuk menghasilkan urutan angka dengan tiga cara utama:

1. range(stop)

Menghasilkan urutan angka dari 0 hingga stop-1. Contoh: range(5) akan menghasilkan angka 0, 1, 2, 3, 4. Contoh dapat dilihat di https://colab.research.google.com/drive/1r_0DNzCHwdkUFQXsBXnCaXdUpRj48Rl8?usp=sharing.

2. range(start, stop)

Menghasilkan urutan angka dari start hingga stop-1. Contoh: range(1, 5) akan menghasilkan angka 1, 2, 3, 4. Contoh dapat dilihat di https://colab.research.google.com/drive/1r_0DNzCHwdkUFQXsBXnCaXdUpRj48Rl8?usp=sharing.

3. range(start, stop, step)

Menghasilkan urutan angka dari start hingga stop-1, dengan langkah step. Contoh: range(1, 10, 2) akan menghasilkan angka 1, 3, 5, 7, 9. Contoh dapat dilihat di https://colab.research.google.com/drive/1r_0DNzCHwdkUFQXsBXnCaXdUpRj48Rl8?usp=sharing.

Mengubah Range menjadi List atau Tipe Data Lainnya

Walaupun objek range adalah iterable, namun tetap bisa mengubahnya menjadi list, tuple, atau set menggunakan fungsi pembuat tipe data tersebut.

Fungsi bawaan range() dalam Python menghasilkan suatu *sequence* (urutan) bilangan yang bersifat **immutable**. Sequence ini umumnya digunakan untuk melakukan perulangan sejumlah tertentu. Urutan bilangan yang dihasilkan memiliki tipe data tersendiri, yaitu range. **Immutable** berarti objek tersebut tidak dapat diubah setelah dibuat. Contoh dapat dilihat di https://colab.research.google.com/drive/1r_0DNzCHwdkUFQXsBXnCaXdUpRj48Rl8?usp=sharing.

Memanggil range() dengan Satu Argumen

Jika fungsi range() dipanggil dengan satu argumen saja, maka argumen tersebut merepresentasikan nilai stop. Nilai start bersifat opsional dan secara default bernilai 0. range(10) menghasilkan urutan angka dari 0 sampai 9 (0 inklusif dan 10 eksklusif). Contoh dapat dilihat di https://colab.research.google.com/drive/1r_0DNzCHwdkUFQXsBXnCaXdUpRj48Rl8?usp=sharing.

Memanggil range() dengan Dua Argumen

Jika dipanggil dengan dua argumen, argumen pertama adalah start, dan argumen kedua adalah stop. Range(3, 10) menghasilkan urutan angka dari 3 sampai 9. Contoh dapat dilihat di https://colab.research.google.com/drive/1r_0DNzCHwdkUFQXsBXnCaXdUpRj48Rl8?usp=sharing.

Memanggil range() dengan Tiga Argumen

Jika dipanggil dengan tiga argumen, argumen ketiga adalah step (selisih antarangka). Nilai default step adalah 1. Range(3, 10, 2) menghasilkan urutan angka dari 3 sampai 9 dengan selisih 2. Contoh dapat dilihat di https://colab.research.google.com/drive/1r_0DNzCHwdkUFQXsBXnCaXdUpRj48Rl8?usp=sharing.

Penggunaan Range dalam Perulangan

Objek range sering digunakan dalam struktur perulangan for. Contoh dapat dilihat di https://colab.research.google.com/drive/1r_0DNzCHwdkUFQXsBXnCaXdUpRj48Rl8?usp=sharing.

Objek range adalah sequence iterable, sehingga dapat digunakan dalam berbagai bentuk iterasi, termasuk for maupun while. Perbedaannya adalah, for → langsung memanfaatkan sifat iterable dari range. Sedangkan while → biasanya menggunakan indeks atau iterator secara manual. Karena range adalah sequence yang bisa diindeks, maka dapat mengakses elemennya menggunakan indeks dalam while.

Cara 1: Menggunakan indeks. Contoh dapat dilihat di https://colab.research.google.com/drive/1r_0DNzCHwdkUFQXsBXnCaXdUpRj48Rl8?usp=sharing.

Cara 2: Menggunakan iterator (iter() dan next()). Contoh dapat dilihat di https://colab.research.google.com/drive/1r_0DNzCHwdkUFQXsBXnCaXdUpRj48Rl8?usp=sharing.

Menampilkan Range dengan List

Objek range tidak langsung menampilkan seluruh elemen ketika dicetak. Oleh karena itu, biasanya dikonversi menjadi list untuk ditampilkan. Contoh dapat dilihat di https://colab.research.google.com/drive/1r_0DNzCHwdkUFQXsBXnCaXdUpRj48Rl8?usp=sharing.

Slicing dalam Range

Sebagaimana tipe data sequence lainnya, range mendukung operasi slicing. Contoh dapat dilihat di https://colab.research.google.com/drive/1r_0DNzCHwdkUFQXsBXnCaXdUpRj48Rl8?usp=sharing.

Keterangan: Pernyataan pertama mengembalikan nilai indeks ke-2. Pernyataan kedua menghasilkan objek range baru dari indeks 0 hingga 3.

Pengujian Keanggotaan (Membership Testing)

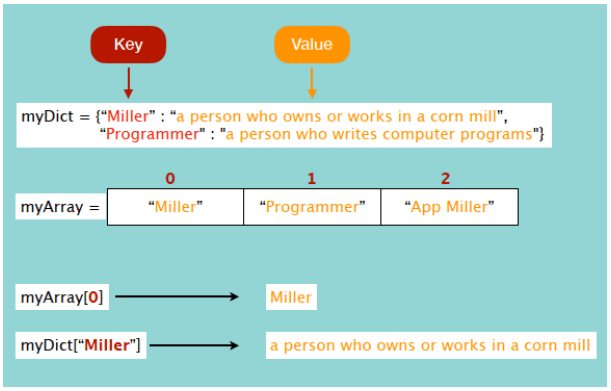
Objek range mendukung operator in untuk mengecek apakah suatu nilai termasuk dalam rentang. Nilai True menunjukkan angka tersebut terdapat dalam range, sedangkan False menunjukkan sebaliknya. Contoh dapat dilihat di https://colab.research.google.com/drive/1r_0DNzCHwdkUFQXsBXnCaXdUpRj48Rl8?usp=sharing.

Menentukan Panjang Range

Objek range mendukung fungsi len() untuk mengetahui jumlah elemen di dalamnya. Contoh dapat dilihat di https://colab.research.google.com/drive/1r_0DNzCHwdkUFQXsBXnCaXdUpRj48Rl8?usp=sharing.

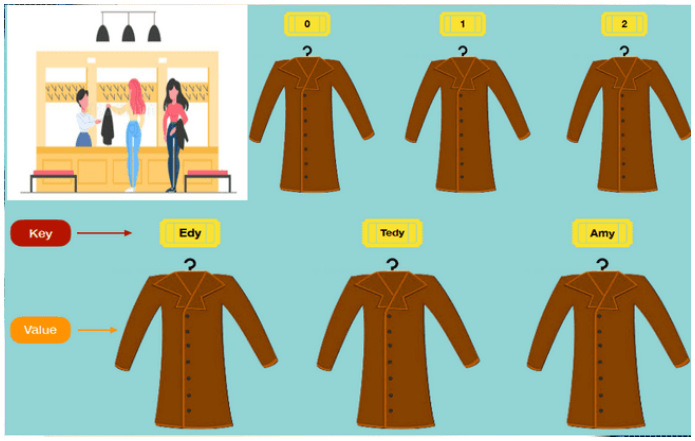
4.6 Dictionary

Dictionary adalah koleksi yang tidak terurut, dapat diubah, dan diindeks. Dictionary Python berbeda dengan list ataupun tuple, karena setiap urutanya berisi key dan value. **Setiap key dipisahkan dari value-nya** oleh **titik dua (:)**, **item dipisahkan oleh koma**, dan **semuanya tertutup dalam kurung kurawal {}**.



Gambar 4.14: Perbedaan Dictionary dan Array (Karimov, 2022)

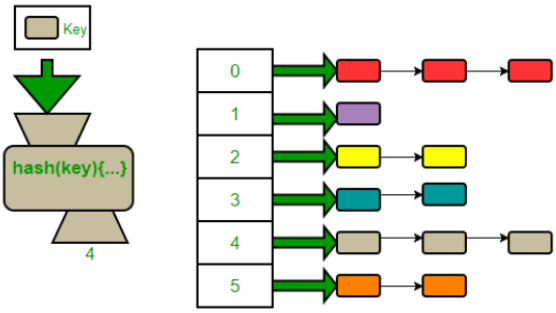
Dictionary kosong tanpa barang **ditulis hanya dengan dua kurung kurawal {}**. Dictionary merupakan struktur data yang digunakan untuk menyimpan nilai dalam bentuk pasangan **key:value**. **Setiap key berfungsi sebagai identitas unik yang mengacu ke suatu value tertentu**.



Gambar 4.15: Ilustrasi Dictionary (Karimov, 2022)

Gambar 4.14 mengilustrasikan perbedaan konseptual antara dictionary dan array (list) dalam pemrograman Python. Sebuah dictionary bernama myDict yang terdiri atas pasangan key–value. Setiap key berfungsi sebagai penanda unik yang digunakan untuk mengakses suatu value. Sebagai contoh, key "Miller" dipetakan ke value berupa definisi teks "a person who owns or works in a corn mill", sedangkan key "Programmer" dipetakan ke definisi "a person who writes computer programs". Akses terhadap nilai dictionary dilakukan menggunakan key, misalnya myDict["Miller"], yang secara langsung mengembalikan nilai yang terkait dengan key tersebut. Sebaliknya, bagian tengah gambar memperlihatkan struktur array bernama myArray yang berisi elemen-elemen berurutan, yaitu "Miller", "Programmer", dan "App Miller", dengan indeks masing-masing 0, 1, dan 2. Array bersifat terindeks secara numerik dan mempertahankan urutan elemen. Akses terhadap elemen array dilakukan menggunakan indeks, seperti myArray[0], yang mengembalikan elemen pertama yaitu "Miller". Dengan demikian, **perbedaan utama adalah array mengakses data berdasarkan posisi indeks numerik, sedangkan dictionary mengakses data berdasarkan key yang bersifat unik dan tidak bergantung urutan posisi**.

Gambar 4.15 menampilkan ilustrasi konseptual mengenai dictionary dalam pemrograman, dengan menggunakan analogi penyimpanan barang di sebuah toko pakaian. Ditunjukkan tiga objek pakaian yang diberi label indeks numerik 0, 1, dan 2. Representasi ini menyerupai struktur array atau list, di mana setiap elemen diakses berdasarkan posisi atau indeks tertentu. Dalam struktur berbasis indeks, identifikasi data dilakukan melalui urutan numerik, sehingga akses terhadap elemen bergantung posisi relatifnya dalam koleksi. Selain itu, juga diperlihatkan pendekatan berbasis dictionary, yang menggunakan konsep pasangan key–value. Dalam ilustrasi tersebut, label seperti “Edy”, “Tedy”, dan “Amy” berfungsi sebagai key, sedangkan pakaian yang bersesuaian berperan sebagai value. Artinya, setiap nama secara unik dipetakan ke suatu objek pakaian tertentu. Mekanisme ini menunjukkan bahwa dictionary tidak



Gambar 4.16: Dictionary dalam Memori (Karimov, 2022)

mengandalkan indeks numerik, melainkan menggunakan key sebagai identifikasi langsung terhadap data yang tersimpan. Dengan demikian, dictionary merepresentasikan struktur data berbasis pemetaan (mapping), di mana setiap key bersifat unik dan digunakan untuk mengakses nilai yang terkait secara langsung, tanpa mempertimbangkan urutan posisi dalam memori.

Gambar 4.16 menjelaskan konsep dictionary dalam memori yang diimplementasikan menggunakan struktur hash table. Terlihat bahwa mekanisme penyimpanan dan pencarian data dalam dictionary tidak dilakukan berdasarkan urutan indeks tetap seperti array biasa, melainkan menggunakan pendekatan key–value lookup. Setiap key yang dimasukkan akan diproses oleh suatu fungsi hash (ditunjukkan sebagai hash(key)), yang berfungsi untuk mengubah key menjadi sebuah nilai numerik tertentu. Nilai numerik tersebut kemudian digunakan sebagai indeks untuk menentukan posisi penyimpanan data di dalam struktur array internal. Diagram memperlihatkan array dengan indeks 0 hingga 5 yang berperan sebagai wadah utama penyimpanan. Setelah fungsi hash menghasilkan suatu indeks, data yang terkait dengan key tersebut ditempatkan dalam slot yang sesuai. Beberapa indeks terlihat lebih dari satu elemen yang terhubung dalam bentuk rantai (linked nodes), yang merepresentasikan mekanisme penanganan collision (tabrakan hash), yaitu kondisi ketika dua key berbeda menghasilkan indeks yang sama. Dalam situasi ini, sistem menyimpan elemen-elemen tersebut secara berantai terhadap indeks yang sama.

⌘ Dictionary Items

Item dalam dictionary disajikan dalam bentuk pasangan **key:value**, **bersifat terurut, dapat diubah, dan tidak mengizinkan key yang sama. Nilai dapat diakses menggunakan nama key**. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

⌘ Ordered atau Unordered

Python versi 3.7 dan seterusnya, dictionary bersifat ordered, artinya urutan item mengikuti urutan saat dimasukkan dan tidak berubah. Dictionary tidak dapat diakses menggunakan indeks numerik.

```
# print(hewan_tropis[0])
# Akan menghasilkan error: TypeError: 'dict' object is not subscriptable
```

⌘ Changeable (Dapat Diubah)

Dictionary bersifat mutable, sehingga nilai dapat diubah, ditambahkan, atau dihapus setelah dibuat. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

⌘ Duplicates Not Allowed

Dictionary tidak mengizinkan dua key yang sama. Jika key yang sama dimasukkan kembali, maka nilai sebelumnya akan ditimpa. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

⌘ Dictionary Length

Jumlah item dalam dictionary dapat diketahui menggunakan fungsi **len()**. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

⌘ Dictionary Items – Data Types

Value dalam dictionary dapat berupa tipe data apa pun, seperti string, integer, list, tuple, atau dictionary lain. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

⌘ type() Dictionary

Secara konseptual dalam Python, dictionary memiliki tipe data **dict**. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

⌘ Menggunakan Constructor dict()

Dictionary juga dapat dibuat menggunakan **konstruktor dict()**. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

a. Mengakses Elemen Dictionary

Dictionary memungkinkan akses data berdasarkan nama key yang dituliskan di dalam tanda kurung siku []. Selain itu, method get() juga dapat digunakan untuk memperoleh nilai yang sama.

⌘ Mengakses Item Menggunakan Nama Key

Nilai dictionary dapat diakses dengan menyebutkan key di dalam tanda kurung siku. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

⌘ Mengakses Item Menggunakan Method get()

Method **get()** menghasilkan nilai yang sama dengan akses menggunakan tanda kurung siku. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

⌘ Mendapatkan Semua Key (Get Keys)

Method **keys()** mengembalikan seluruh key dalam dictionary dalam bentuk view object. View ini bersifat dinamis, artinya perubahan dictionary akan tercermin dalam hasil keys(). Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

⌘ Mendapatkan Semua Value (Get Values)

Method **values()** mengembalikan seluruh value dalam dictionary dalam bentuk view object yang dinamis. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

⌘ Mendapatkan Seluruh Item (Get Items)

Method **items()** mengembalikan seluruh pasangan key:value dalam bentuk tuple yang tergabung dalam view object. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

⌘ Memeriksa Keberadaan Key (Check if Key Exists)

Untuk memeriksa apakah suatu key terdapat dalam dictionary, digunakan kata kunci **in**. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

b. Merubah Elemen Dictionary

Dictionary bersifat mutable, sehingga nilai dari suatu item dapat diubah dengan merujuk langsung ke nama key yang bersangkutan.

⌘ Mengubah Nilai Berdasarkan Key

Nilai dalam pasangan key:value dapat diperbarui dengan menetapkan nilai baru dalam key tertentu. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

⌘ Memperbarui Dictionary (Update Dictionary)

Method **update()** digunakan untuk memperbarui dictionary dengan pasangan **key:value** dari argumen yang diberikan.

Menggunakan update() dengan Dictionary

Argumen tersebut harus berupa dictionary lain atau iterable yang berisi pasangan key:value. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

Menggunakan update() dengan Iterable Berisi Pasangan Key:Value

Method update() juga dapat menerima iterable seperti list yang berisi tuple pasangan key:value. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

c. Menambah Elemen Dictionary

Penambahan item baru dictionary dilakukan dengan mendefinisikan key baru dan menetapkan nilai (value) yang sesuai. Jika key tersebut belum ada sebelumnya, maka pasangan key:value akan ditambahkan sebagai elemen baru dalam dictionary.

⌘ Menambahkan Item Menggunakan Penugasan Langsung

Item baru dapat ditambahkan dengan menuliskan key baru dan memberikan nilai yang diinginkan. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

⌘ Memperbarui Dictionary Menggunakan update()

Method update() digunakan untuk memperbarui dictionary dengan pasangan key:value dari argumen yang diberikan. Jika key belum ada, maka item tersebut akan ditambahkan. Jika key sudah ada, maka nilainya akan diperbarui. Argumen yang diberikan harus berupa dictionary lain atau iterable yang berisi pasangan key:value.

Menambahkan Item Menggunakan update() dengan Dictionary

Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

Menambahkan Item Menggunakan update() dengan Iterable Pasangan Key:Value

Method update() juga dapat menerima iterable seperti list berisi tuple pasangan key:value. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

d. Menghapus Elemen Dictionary

Dictionary sebagai struktur data yang bersifat mutable memungkinkan penghapusan item setelah dictionary dibuat. Terdapat beberapa method yang dapat digunakan untuk menghapus pasangan key:value dalam dictionary, baik secara selektif maupun secara keseluruhan.

1. Menghapus Item Menggunakan del

Kata kunci del digunakan untuk menghapus pasangan key:value berdasarkan key tertentu. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

2. Menghapus Item Menggunakan pop()

Method pop() menghapus item berdasarkan key dan mengembalikan nilai yang dihapus. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

3. Menghapus Item Terakhir Menggunakan popitem()

Method **popitem()** menghapus dan mengembalikan pasangan key:value terakhir yang dimasukkan. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

4. Menghapus Seluruh Item Menggunakan clear()

Method clear() menghapus seluruh isi dictionary, tetapi objek dictionary tetap ada dalam keadaan kosong.Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

5. Menghapus Dictionary Secara Keseluruhan

Untuk menghapus objek dictionary sepenuhnya dari memori, digunakan kata kunci **del**. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

e. Perulangan dalam Dictionary

Dictionary dapat diiterasi menggunakan struktur perulangan. Secara umum, perulangan dilakukan menggunakan for, karena dictionary merupakan objek iterable. Ketika dilakukan iterasi langsung terhadap dictionary, nilai yang dikembalikan adalah key-nya. Namun, tersedia method untuk mengakses value maupun pasangan key:value. Selain for, perulangan juga dapat dilakukan menggunakan while, dengan memanfaatkan daftar key atau item sebagai perantara.

1. Perulangan Menggunakan for (Menghasilkan Key)

Ketika dictionary diiterasi secara langsung, Python akan mengembalikan key. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

2. Mengakses Value Menggunakan values()

Untuk memperoleh nilai (value), digunakan method **values()**. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

3. Mengakses Key dan Value Menggunakan items()

Method items() mengembalikan pasangan key:value dalam bentuk tuple. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

Perulangan Dictionary Menggunakan while

Dictionary tidak secara langsung mendukung iterasi berbasis indeks numerik seperti list. Oleh karena itu, untuk menggunakan while, biasanya key dikonversi terlebih dahulu menjadi list. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

4. Perulangan while Menggunakan Daftar Key

Berikut perulangan menggunakan while. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

5. Perulangan while Menggunakan Daftar Item

Alternatif lain adalah menggunakan items() dan mengubahnya menjadi list. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

f. Menyalin Dictionary

Dalam Python, penyalinan dictionary tidak dapat dilakukan hanya dengan pernyataan penugasan sederhana seperti dict2 = dict1. Pernyataan tersebut tidak membuat salinan baru, melainkan hanya membuat referensi terhadap objek yang sama di memori. Akibatnya, perubahan yang dilakukan dict1 akan turut memengaruhi dict2, karena keduanya menunjuk objek yang identik. Untuk membuat salinan yang **terpisah (shallow copy)**, dapat digunakan method bawaan **copy()** atau fungsi **konstruktor dict()**.

1. Penugasan Langsung (Hanya Referensi, Bukan Salinan)

Contoh berikut menunjukkan bahwa perubahan dictionary asli akan memengaruhi variabel referensinya. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

2. Menggunakan Method copy()

Method copy() digunakan untuk membuat salinan dictionary yang terpisah dari objek asli. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

3. Menggunakan Konstruktor dict()

Fungsi bawaan dict() juga dapat digunakan untuk membuat salinan dictionary. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

g. Nested Dictionary

Dictionary dalam Python dapat memuat dictionary lain sebagai nilai (value). Struktur ini dikenal sebagai nested dictionary atau dictionary bersarang. Nested dictionary memungkinkan pengorganisasian data yang lebih kompleks dan hierarkis, karena setiap key dapat memetakan ke struktur data lain yang juga berupa dictionary.

1. Membuat Nested Dictionary

Dictionary dapat berisi beberapa dictionary di dalamnya sebagai value. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

2. Menggabungkan Beberapa Dictionary ke Dictionary Baru

Beberapa dictionary terpisah dapat dimasukkan ke dalam satu dictionary utama. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

3. Mengakses Elemen dalam Nested Dictionary

Untuk mengakses item dalam nested dictionary, digunakan nama dictionary luar terlebih dahulu, kemudian diikuti dengan key dalam dictionary dalam. Program dapat dilihat di https://colab.research.google.com/drive/1EgcnJBOHtOSIY4vn7fauQFhMN9fm_ju8?usp=sharing .

Tabel 4.6: Method Dictionary *Built-In*

Method	Deskripsi
clear()	Menghapus seluruh elemen dari dictionary
copy()	Mengembalikan salinan dari dictionary
fromkeys()	Mengembalikan dictionary dengan key yang ditentukan dan nilai tertentu
get()	Mengembalikan nilai dari key yang ditentukan
items()	Mengembalikan daftar yang berisi tuple untuk setiap pasangan key dan value
keys()	Mengembalikan daftar yang berisi seluruh key dalam dictionary
pop()	Menghapus elemen dengan key yang ditentukan
popitem()	Menghapus pasangan key-value terakhir yang dimasukkan
setdefault()	Mengembalikan nilai dari key yang ditentukan. Jika key tidak ada, maka key akan ditambahkan dengan nilai yang ditentukan
update()	Memperbarui dictionary dengan pasangan key-value yang ditentukan
values()	Mengembalikan daftar seluruh nilai dalam dictionary

h. Method Dictionary

Python memiliki serangkaian method built-in yang dapat digunakan dalam dictionary, ditampilkan di Tabel 4.6.

4.7 String

String di Python dikelilingi oleh tanda kutip tunggal ‘ ‘ atau tanda kutip ganda “ “. Hasil string dapat ditampilkan menggunakan fungsi **print()**. Python mengijinkan tanda kutip di dalam sebuah string, **asalkan tanda kutip tersebut tidak sama dengan tanda kutip yang digunakan untuk mengelilingi string**. Contoh: `print("It's fine")`. Menetapkan string ke variabel dilakukan dengan menuliskan nama variabel diikuti dengan tanda sama dengan dan string tersebut. Program dapat dilihat di https://colab.research.google.com/drive/1lGTrAq5zExMXR5LlPagS1Oyd9zXp0Aub?usp=sharing .

String dapat ditulis secara multibaris dan dimasukkan ke sebuah variabel dengan tiga tanda kutip ganda atau dengan tanda kutip tunggal. Program dapat dilihat di https://colab.research.google.com/drive/1lGTrAq5zExMXR5LlPagS1Oyd9zXp0Aub?usp=sharing .

String adalah array. Seperti banyak bahasa pemrograman populer lainnya, string di Python adalah array dari karakter-karakter Unicode. Namun, Python tidak memiliki tipe data karakter, karena satu karakter dianggap sebagai string dengan panjang 1. Gunakan tanda kurung siku [ ] untuk mengakses elemen-elemen dalam string. Program dapat dilihat di https://colab.research.google.com/drive/1lGTrAq5zExMXR5LlPagS1Oyd9zXp0Aub?usp=sharing .

String dapat dilakukan operasi perulangan. Karena string adalah array, dapat dilakukan perulangan untuk mengakses setiap karakter dalam string menggunakan perulangan **for**. Program dapat dilihat di https://colab.research.google.com/drive/1lGTrAq5zExMXR5LlPagS1Oyd9zXp0Aub?usp=sharing .

Gunakan fungsi **len()** untuk mengetahui panjang sebuah string. Program dapat dilihat di https://colab.research.google.com/drive/1lGTrAq5zExMXR5LlPagS1Oyd9zXp0Aub?usp=sharing .

Gunakan *keyword* **in** untuk memeriksa apakah sebuah frase atau karakter ada dalam string. Program dapat dilihat di https://colab.research.google.com/drive/1lGTrAq5zExMXR5LlPagS1Oyd9zXp0Aub?usp=sharing .

Gunakan *keyword* **not in** untuk memeriksa sebuah frase atau karakter TIDAK ADA dalam string. Program dapat dilihat di https://colab.research.google.com/drive/1lGTrAq5zExMXR5LlPagS1Oyd9zXp0Aub?usp=sharing .

Pemotongan String (Slicing)

Suatu kata dapat diambil rentang karakter dalam string dengan menggunakan sintaksis pemotongan (*slice*). Tentukan indeks awal dan indeks akhir, yang dipisahkan oleh tanda titik dua (:), untuk mengembalikan sebagian dari string tersebut. Berikut beberapa cara pemotongan string:

- **Memotong dari Awal**

  Dengan mengabaikan indeks awal, rentang akan dimulai dari karakter pertama. Program dapat dilihat di https://colab.research.google.com/drive/1lGTrAq5zExMXR5LlPagS1Oyd9zXp0Aub?usp=sharing .

- **Memotong hingga Akhir**

  Dengan mengabaikan indeks akhir, rentang akan berlanjut hingga akhir string. Program dapat dilihat di https://colab.research.google.com/drive/1lGTrAq5zExMXR5LlPagS1Oyd9zXp0Aub?usp=sharing .

- **Indeks Negatif**

  Gunakan indeks negatif untuk memulai pemotongan dari akhir string. Program dapat dilihat di https://colab.research.google.com/drive/1lGTrAq5zExMXR5LlPagS1Oyd9zXp0Aub?usp=sharing .

Python - Mengubah String

Python menyediakan sejumlah method bawaan yang dapat digunakan dalam string. berikut beberapa method bawaan string yaitu:

- *Upper Case* yaitu method ini digunakan untuk mengubah semua karakter dalam string menjadi huruf kapital. *Keyword*: **upper()**.

- *Lower Case* merupakan method ini digunakan untuk mengubah semua karakter dalam string menjadi huruf kecil. *Keyword*: **lower()**.
- Spasi kosong adalah ruang kosong yang ada sebelum dan/atau setelah teks yang sebenarnya, dan sering kali ingin menghapus ruang kosong ini. *Keyword*: **strip()**.
- Mengganti string adalah mengganti bagian dari string dengan teks lain. *Keyword*: **replace()**.
- Memisahkan String menggunakan *keyword* **split()** mengembalikan sebuah list, di mana teks yang berada di antara pemisah yang ditentukan akan menjadi item dalam list tersebut.
- Penggabungan string untuk menggabungkan, atau menyatukan, dua string, Anda dapat menggunakan operator +.

Program dapat dilihat di https://colab.research.google.com/drive/1lGTrAq5zExMXR5LlPagS1Oyd9zXp0Aub?usp=sharing .

Format – String:

Beberapa format untuk mengatur string yaitu:

- Format String
  Python tidak bisa menggabungkan string dan angka secara langsung. Namun, dapat digabungkan string dan angka dengan menggunakan f-string atau method **format()**.
- F-String
  F-String diperkenalkan Python 3.6 dan kini menjadi cara yang lebih disarankan untuk memformat string. Untuk menggunakan f-string, cukup tambahkan huruf **f** di depan literal string dan gunakan tanda kurung kurawal {} sebagai tempat penampung (*placeholder*) untuk variabel dan operasi lainnya.
- Tempat Penampung (*placeholder*) dan Modifikasi
  Tempat penampung (*placeholder*) dapat berisi variabel, operasi, fungsi, dan modifikasi untuk memformat nilai. Tempat penampung (*placeholder*) bisa menyertakan modifikasi untuk memformat nilai. Modifikasi dilakukan dengan menambahkan tanda titik dua **:** diikuti dengan tipe format yang sah, seperti .2f yang berarti angka dengan dua tempat desimal. Tempat penampung (*placeholder*) juga bisa berisi kode Python, seperti operasi matematika.
- Karakter Escape
  Untuk menyisipkan karakter yang tidak diperbolehkan dalam sebuah string, dapat menggunakan karakter escape. **Karakter escape** adalah tanda backslash (\) yang diikuti dengan karakter yang ingin dimasukkan. **Contoh karakter yang tidak diperbolehkan adalah tanda kutip ganda di dalam string yang juga dikelilingi oleh tanda kutip ganda**.

Program dapat dilihat di https://colab.research.google.com/drive/1lGTrAq5zExMXR5LlPagS1Oyd9zXp0Aub?usp=sharing .

Tabel 4.7: Karakter Escape dalam Python

Code	Result	Code	Result	Code	Result
\'	Single Quote	\b	Backspace	\t	Tab
\\	Backslash	\f	Form Feed	\r	Carriage Return
\n	New Line	\ooo	Octal value	\xxh	Hex value

Karakter escape lain dalam Python yang disajikan di Tabel 4.7.

***Method*() String**

Python memiliki serangkaian method bawaan yang dapat digunakan dalam string. Semua method string mengembalikan nilai baru. Method tersebut tidak mengubah string asli. Macam-macam method() string dalam Python ditampilkan di Tabel 4.8.

Tabel 4.8: Method() String dalam Python

Method()	Deskripsi
capitalize()	Mengubah karakter pertama menjadi huruf kapital
casefold()	Mengubah string menjadi huruf kecil
center()	Mengembalikan string yang terpusat
count()	Mengembalikan jumlah kemunculan nilai yang ditentukan dalam string
encode()	Mengembalikan versi terkode dari string
endswith()	Mengembalikan True jika string berakhir dengan nilai yang ditentukan
expandtabs()	Mengatur ukuran tab dalam string
find()	Mencari nilai tertentu dalam string dan mengembalikan posisi ditemukannya
format()	Memformat nilai yang ditentukan dalam string
format_map()	Memformat nilai yang ditentukan dalam string
index()	Mencari nilai dalam string dan mengembalikan posisi ditemukannya
isalnum()	Mengembalikan *True* jika semua karakter dalam string adalah alfanumerik
isalpha()	Mengembalikan *True* jika semua karakter dalam string adalah alfabet
isascii()	Mengembalikan *True* jika semua karakter dalam string adalah karakter ASCII
isdecimal()	Mengembalikan *True* jika semua karakter dalam string adalah angka desimal
isdigit()	Mengembalikan *True* jika semua karakter dalam string adalah digit
isidentifier()	Mengembalikan *True* jika string adalah sebuah pengenal (identifier)
islower()	Mengembalikan *True* jika semua karakter dalam string adalah huruf kecil
isnumeric()	Mengembalikan *True* jika semua karakter dalam string adalah numerik
isprintable()	Mengembalikan *True* jika semua karakter dalam string dapat dicetak
isspace()	Mengembalikan *True* jika semua karakter dalam string adalah spasi
istitle()	Mengembalikan *True* jika string mengikuti aturan judul
isupper()	Mengembalikan *True* jika semua karakter dalam string adalah huruf kapital
join()	Menggabungkan elemen-elemen dari iterable ke dalam string
ljust()	Mengembalikan versi string yang dijustifikasi ke kiri
lower()	Mengubah string menjadi huruf kecil
lstrip()	Mengembalikan string yang dipangkas dari kiri
maketrans()	Mengembalikan tabel terjemahan yang digunakan dalam terjemahan
partition()	Mengembalikan tuple yang membagi string menjadi tiga bagian
replace()	Mengembalikan string dengan nilai yang ditentukan diganti dengan nilai baru
rfind()	Mencari nilai dalam string dan mengembalikan posisi terakhir ditemukannya
rindex()	Mencari nilai dalam string dan mengembalikan posisi terakhir ditemukannya
rjust()	Mengembalikan versi string yang dijustifikasi ke kanan
rpartition()	Mengembalikan tuple yang membagi string menjadi tiga bagian
rsplit()	Memisahkan string berdasarkan pemisah yang ditentukan dan mengembalikan list
rstrip()	Mengembalikan string yang dipangkas dari kanan
split()	Memisahkan string berdasarkan pemisah yang ditentukan dan mengembalikan list
splitlines()	Memisahkan string di setiap baris dan mengembalikan list
startswith()	Mengembalikan True jika string dimulai dengan nilai yang ditentukan
strip()	Mengembalikan versi string yang dipangkas dari kiri dan kanan

Method()	Deskripsi
swapcase()	Menukar huruf besar dan kecil dalam string
title()	Mengubah huruf pertama setiap kata menjadi kapital
translate()	Mengembalikan string yang telah diterjemahkan
upper()	Mengubah string menjadi huruf kapital
zfill()	Mengisi string dengan sejumlah nilai 0 di bagian depan

Python mempunyai berbagai tipe format string yang ditampilkan di Tabel 4.9.

Tabel 4.9: Tipe Format String

Format	Deskripsi
:<	Menyusun hasil ke kiri (dalam ruang yang tersedia)
:>	Menyusun hasil ke kanan (dalam ruang yang tersedia)
:^	Menyusun hasil ke tengah (dalam ruang yang tersedia)
:=	Menempatkan tanda di posisi paling kiri
:+	Menggunakan tanda plus untuk menunjukkan apakah hasilnya positif atau negatif
:-	Menggunakan tanda minus hanya untuk nilai negatif
:	Menggunakan ruang untuk menyisipkan ruang ekstra sebelum angka positif (tanda minus sebelum angka negatif)
:,	Menggunakan koma sebagai pemisah ribuan
:_	Menggunakan garis bawah sebagai pemisah ribuan
:b	Format biner
:c	Mengubah nilai menjadi karakter Unicode yang sesuai
:d	Format desimal
:e	Format ilmiah dengan huruf e kecil
:E	Format ilmiah dengan huruf E besar
:f	Format titik tetap
:F	Format titik tetap, dalam format huruf besar (tampilkan inf dan nan sebagai INF dan NAN)
:g	Format umum
:G	Format umum (menggunakan huruf besar E untuk notasi ilmiah)
:o	Format oktal
:x	Format hex, huruf kecil
:X	Format hex, huruf besar
:n	Format angka
%	Format persentase

Latihan:
1) SOAL ARRAY
Tugas: Buat program yang menerima array berisi bilangan bulat, lalu:
Membalik urutan elemen (reverse), menampilkan elemen pertama dan terakhir setelah dibalik.
Contoh input: [10, 5, 8, 2, 9]. Contoh output: array_baru = [9, 2, 8, 5, 10], first = 9, last = 10

2) SOAL LIST
Tugas: Buat program yang menerima list nama mahasiswa, lalu:
Menghapus semua nama yang sama (duplikat) tetap mempertahankan urutan kemunculan pertama, menampilkan list hasil dan jumlah data akhir.
Contoh input: ["Ayu", "Bima", "Ayu", "Dina", "Bima", "Eko"]. Contoh output: hasil = ["Ayu", "Bima", "Dina", "Eko"], jumlah = 4

3) SOAL TUPLE
Tugas: Diberi tuple berisi data produk: (kode, nama, stok). Buat program yang:
Melakukan unpacking tuple, menampilkan kalimat: "Produk <nama> (<kode>) stok = <stok>", jika stok = 0 tampilkan tambahan teks: "Status: Habis".
Contoh input: ("P01", "Pulpen", 0). Contoh output: Produk Pulpen (P01) stok = 0. Status: Habis.

4) SOAL SET
Tugas: Buat program yang menerima dua set angka, lalu:
Menampilkan gabungan (union), menampilkan irisan (intersection), menampilkan angka yang hanya ada di set pertama (difference).
Contoh input: A = {1, 2, 3, 5} B = {3, 4, 5, 6}
Contoh output: union = {1, 2, 3, 4, 5, 6} intersection = {3, 5} difference_A = {1, 2}

5) SOAL DICTIONARY
Tugas: Buat program untuk mengelola dictionary nilai mahasiswa (key = nama, value = nilai), dengan fitur: tambah data baru (nama dan nilai), update nilai jika nama sudah ada, hapus data berdasarkan nama, tampilkan seluruh data dalam format rapi: "Nama: nilai" urut berdasarkan nama.
Contoh input awal:
{"Ayu": 80, "Bima": 75, "Dina": 90}
Operasi: tambah "Eko": 70, update "Bima": 85, hapus "Ayu"
Contoh output akhir:
Bima: 85
Dina: 90
Eko: 70

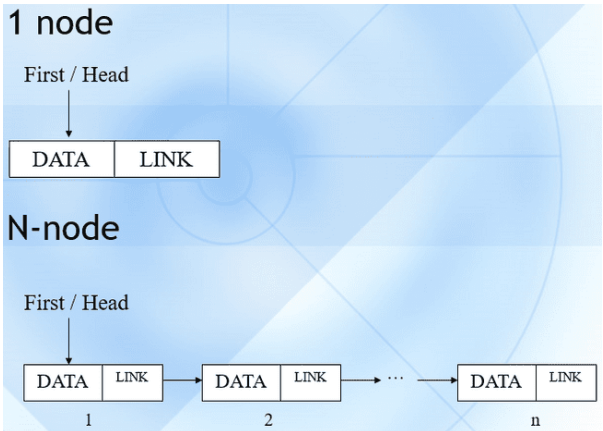
BAB 5
Linked List

Linked list merupakan struktur data linier dan dinamis yang menyimpan sekumpulan elemen secara berurutan melalui node-node yang bersifat independen (Thareja, 2014). **Setiap node** umumnya tersusun atas **dua komponen** utama, yaitu **data** (informasi yang disimpan) serta **link** (penunjuk atau alamat memori) yang menghubungkan node tersebut dengan node berikutnya (dan dalam beberapa varian juga dengan node sebelumnya). Mekanisme tautan berbasis penunjuk ini menjadikan linked list fleksibel karena operasi penyisipan (*insertion*) dan penghapusan (*deletion*) elemen dapat dilakukan di berbagai posisi tanpa perlu menggeser elemen lain sebagaimana terjadi dalam struktur berbasis array, sehingga mendukung pengelolaan data yang efisien dalam representasi urutan linier.

Linked list dapat menyimpan data seperti array. Perbedaan utama antara array dan linked list berkaitan dengan manajemen memori serta cara merepresentasikan elemen. Array menyimpan seluruh elemen secara kontigu di satu blok memori, sehingga alamat tiap elemen tersusun berurutan dan penyimpanannya bersifat terpusat. Sebaliknya, linked list menyimpan elemen sebagai node-node yang dialokasikan secara bertahap selama program berjalan, sehingga posisi node dapat tersebar di berbagai lokasi memori. Setiap node memuat data sekaligus pointer menuju node berikutnya (dan juga menuju node sebelumnya), sehingga urutan elemen terbentuk melalui hubungan antar-node, bukan karena kedekatan letaknya di memori. Oleh karena itu, array menunjukkan representasi terpusat dalam satu blok memori besar, sedangkan linked list menunjukkan representasi terdistribusi melalui rangkaian node yang saling terhubung untuk membentuk urutan linier (Agarwal, 2022).

Linked list mempunyai karakteristik utama sebagai berikut:

1. **Elemen data**, disimpan dalam lokasi memori yang tidak berurutan (non-kontigu) dan saling dihubungkan melalui pointer.
   Pointer merupakan **entitas yang menyimpan alamat memori suatu variabel**, sehingga setiap elemen (node) memiliki referensi yang mengarah ke elemen berikutnya secara berantai hingga mencapai elemen terakhir, yang referensinya bernilai null sebagai penanda akhir struktur.
2. Ukuran atau panjang linked list bersifat dinamis, artinya dapat bertambah maupun berkurang selama program dieksekusi sesuai kebutuhan operasi penyisipan dan penghapusan elemen.
   Berbeda dari array yang menyimpan elemen dalam blok memori kontigu, linked list menyimpan setiap item data secara terpisah dalam lokasi memori yang berbeda, kemudian membentuk urutan logis melalui pointer.

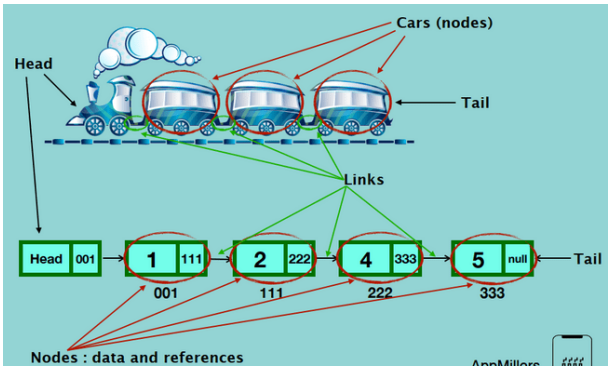


Gambar 5.1: Linked List

Setiap elemen linked list disebut node, yang secara umum memuat dua komponen, yakni **data aktual dan pointer** yang menghubungkannya dengan node lain.

Gambar 5.1 memvisualisasikan konsep linked list melalui analogi rangkaian kereta. **Lokomotif** merepresentasikan **head**, yaitu referensi awal yang menunjuk ke node pertama. Setiap **gerbong** merepresentasikan **node**, sedangkan sambungan **antar-gerbong** menggambarkan **link** atau **pointer** yang menghubungkan satu node ke node berikutnya. Bagian ilustrasi bawah memperjelas **struktur node yaitu tiap node terdiri dari dua komponen, yakni data** (misalnya nilai 1, 2, 4, 5) dan **pointer ke node berikutnya** (ditunjukkan sebagai alamat seperti 111, 222, 333). Head menyimpan alamat node pertama (misalnya 001), kemudian node pertama menunjuk ke node kedua, dan seterusnya hingga node terakhir (tail) yang referensinya bernilai null sebagai penanda akhir linked list. Node-node dapat tersimpan dalam lokasi memori yang berbeda (tidak harus berdampingan), namun urutan elemen tetap terbentuk secara logis melalui pointer, sehingga penelusuran data dilakukan dengan mengikuti pointer dari head sampai mencapai null.

Gambar 5.2 menunjukkan visualisasi linked list dalam dua kondisi, yaitu 1 node dan N-node. Bagian 1

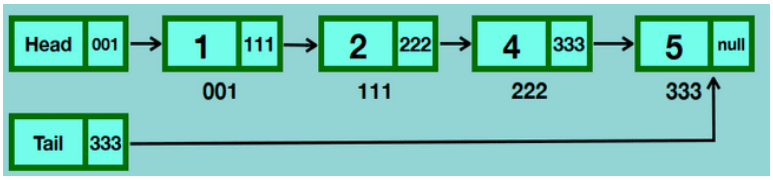


Gambar 5.2: Ilustrasi Linked List (Karimov, 2022)

node, struktur linked list direpresentasikan sebagai satu node yang terdiri atas dua bidang, yakni DATA untuk menyimpan nilai/informasi dan LINK untuk menyimpan alamat menuju node berikutnya. karena hanya terdapat satu node, maka Head berperan sebagai penunjuk awal yang langsung mengarah ke node tersebut, sedangkan nilai link secara konseptual mengarah ke null karena tidak ada node lanjutan. Bagian N-node, ditunjukkan bahwa Head menunjuk ke node pertama, kemudian setiap node terhubung secara berurutan melalui LINK yang menunjuk ke node berikutnya (node 2, dan seterusnya) hingga mencapai node terakhir ke-n, sehingga urutan elemen terbentuk melalui keterkaitan referensi antar-node.

Terdapat empat jenis linked list yaitu:

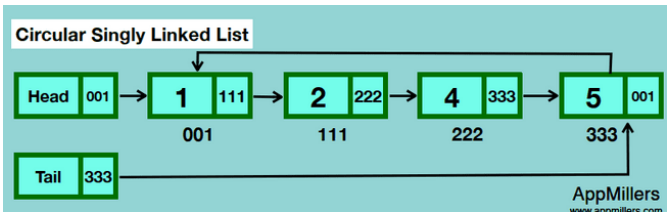
1. Singly linked linear list yaitu satu node, berisi satu link (untuk menyimpan informasi sebelumnya).



Gambar 5.3: Singly Linked Linear List (Karimov, 2022)

Gambar 5.3 menampilkan singly linked list dengan penunjuk Head dan Tail. Head menyimpan alamat node pertama (001), kemudian setiap node berisi data (1, 2, 4, 5) dan link yang menunjuk ke alamat node berikutnya (111, 222, 333). Node terakhir (data 5) memiliki link null sebagai penanda akhir, sedangkan Tail menyimpan alamat node terakhir (333) sehingga akses ke elemen terakhir dapat dilakukan lebih cepat.

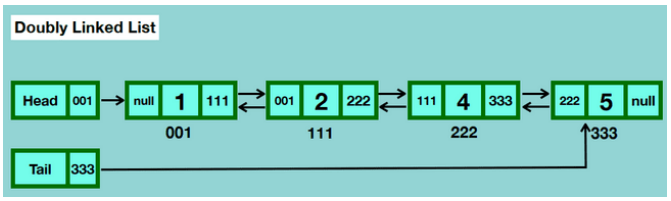
2. Circularly singly linked linear list yaitu satu node, berisi satu link (untuk menyimpan informasi sebelumnya), dan node yang terakhir menuju ke Head.



Gambar 5.4: Linked List (Karimov, 2022)

Gambar 5.4 menggambarkan struktur circular singly linked list, yaitu daftar berantai satu arah yang membentuk siklus. Head menyimpan alamat node pertama (001) dan setiap node tersusun atas dua bagian: data (1, 2, 4, 5) serta link yang menunjuk alamat node berikutnya (111, 222, 333). **Ciri utama sifat circular** tampak dalam node terakhir (data 5) yang link-nya tidak bernilai null, melainkan menunjuk kembali ke alamat node pertama (001), sehingga rangkaian node menjadi melingkar dan traversal dapat berlanjut kembali ke awal. Tail menunjuk node terakhir (beralamat 333), yang memudahkan akses langsung ke elemen akhir.

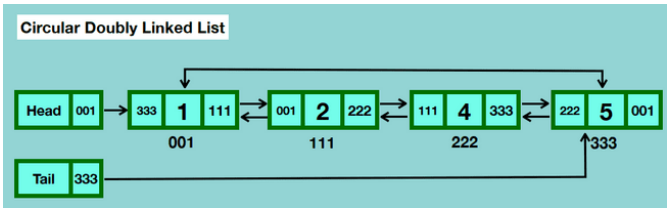
3. Doubly linked linear list yaitu satu node, berisi dua link (untuk menyimpan informasi sebelumnya dan setelahnya).



Gambar 5.5: Linked List (Karimov, 2022)

Gambar 5.5 merepresentasikan doubly linked list, yaitu struktur daftar berantai yang setiap node-nya memiliki dua referensi sehingga dapat ditelusuri dua arah. Head menyimpan alamat node pertama (001), sedangkan Tail menyimpan alamat node terakhir (333). Setiap node terdiri atas tiga bagian yaitu pointer ke node sebelumnya (prev), data (1, 2, 4, 5), dan pointer ke node berikutnya (next). Node pertama memiliki nilai prev = null karena tidak memiliki pendahulu, sementara node terakhir memiliki next = null karena tidak memiliki penerus. Keterkaitan dua arah ini ditunjukkan oleh panah bolak-balik antar-node, yang memungkinkan proses traversal maupun operasi penyisipan atau penghapusan dilakukan dengan akses yang lebih fleksibel dibandingkan singly linked list.

4. Circularly doubly linked linear list → satu node, berisi dua link (untuk menyimpan informasi sebelumnya dan setelahnya). Link kanan dari node terakhir menunjuk ke head, link kiri dari head-node menunjuk ke node terakhir.

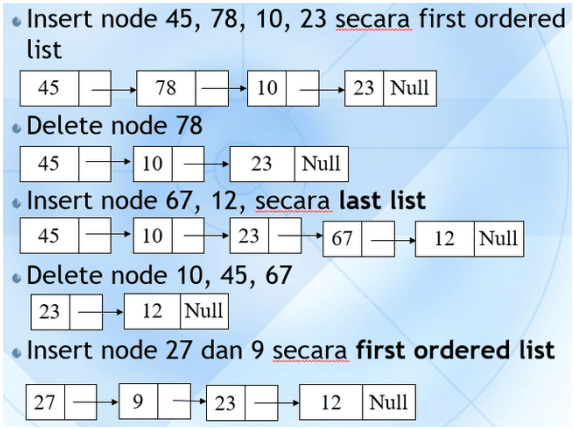


Gambar 5.6: Linked List (Karimov, 2022)

Gambar 5.6 menggambarkan circular doubly linked list, yakni struktur daftar berantai dua arah yang membentuk siklus. Head menyimpan alamat node pertama (001) dan setiap node terdiri atas tiga komponen, yaitu pointer ke node sebelumnya (prev), data (1, 2, 4, 5), serta pointer ke node berikutnya (next). Keterhubungan dua arah ditunjukkan melalui panah bolak-balik antar-node, sehingga penelusuran dapat dilakukan dari head ke tail maupun sebaliknya. Sifat circular tampak karena node terakhir (beralamat 333) memiliki next yang menunjuk kembali ke node pertama (001), dan node pertama memiliki prev yang menunjuk ke node terakhir (333), sehingga tidak ada pointer bernilai null. Tail menunjuk alamat node terakhir, yang memudahkan akses langsung ke elemen akhir sekaligus mempertahankan struktur melingkar.

Terdapat beberapa operasi umum dalam linked list diantaranya adalah:

1. Insert: operasi umum untuk menambahkan node baru ke dalam linked list. Implementasinya bisa dilakukan di awal, di akhir, atau di posisi tertentu.
2. Insert at end: perasi penambahan node di bagian akhir (tail). Prosesnya, telusuri node sampai node terakhir, lalu ubah next node terakhir agar menunjuk node baru (node baru menjadi elemen terakhir).
3. Insert at beginning: operasi penambahan node di bagian awal (head). Prosesnya, next node baru diarahkan ke head lama, kemudian head diperbarui agar menunjuk node baru.
4. Insert between atau insertion: operasi penyisipan node di antara dua node (posisi tengah). Prosesnya, temukan node sebelum posisi sisip, lalu atur next node baru agar menunjuk node sesudahnya, kemudian next node sebelumnya diubah agar menunjuk node baru.
5. Delete: operasi menghapus node dari linked list. Prosesnya, temukan node yang akan dihapus beserta node sebelumnya, lalu ubah next node sebelumnya agar "melewati" node yang dihapus, sehingga rantai tetap utuh.
6. Search: operasi mencari elemen berdasarkan nilai data. Prosesnya, traversal dari head, membandingkan nilai setiap node sampai ketemu atau sampai list berakhir.
7. Indexing: operasi mengakses elemen berdasarkan indeks (misalnya elemen ke-0, ke-1, dan seterusnya). Berbeda dengan array, indexing dalam linked list dilakukan dengan traversal dari head hingga indeks tujuan (kompleksitas umumnya O(n)).



Gambar 5.7: Perhitungan Manual Linked List

Gambar 5.7 menunjukkan perhitungan manual operasi dalam singly linked list dengan menekankan perubahan urutan node dan pembaruan pointer setiap kali terjadi insert atau delete. Proses dimulai dengan operasi insert node 45, 78, 10, 23 secara first ordered list, yang dibentuk menjadi rangkaian 45 → 78 → 10 → 23 → Null, artinya setiap node menunjuk ke node berikutnya dan node terakhir menunjuk ke Null sebagai penanda akhir list. Selanjutnya dilakukan delete node 78, sehingga pointer dari node 45 diperbarui untuk melewati node 78 dan langsung menunjuk ke node 10, menghasilkan 45 → 10 → 23 → Null. Setelah itu, dilakukan insert node 67 dan 12 secara last list (penyisipan di bagian akhir), sehingga node 23 yang semula terakhir kini menunjuk ke 67, dan

67 menunjuk ke 12, membentuk 45 → 10 → 23 → 67 → 12 → Null. Tahap berikutnya adalah delete node 10, 45, dan 67, sehingga node-node tersebut dihapus satu per satu dengan cara mengatur ulang pointer node sebelumnya agar melewati node yang dihapus; hasil akhirnya tersisa 23 → 12 → Null. Terakhir, dilakukan insert node 27 dan 9 secara first ordered list (penyisipan di bagian awal), sehingga head berpindah ke 27 lalu ke 9, membentuk urutan akhir 27 → 9 → 23 → 12 → Null.

Latihan:
Gambarkan setiap perubahan yang terjadi bila dilakukan proses insert & delete berikut:

- Insert node 45, 40, 15, 20, 100, 80, 70, 1000, 2000 secara first ordered list
- Delete node 100
- Insert node 60 dan 50 secara first ordered list
- Delete node 15, 100
- Insert node 55, 22, secara last list
- Delete node 15, 70, 22

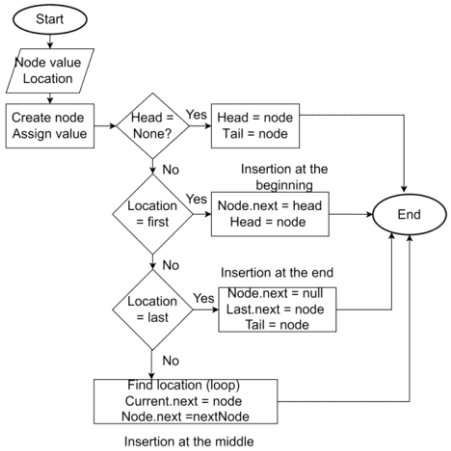
Latihan:
Buatkan program circularly single linked list, double linked list, dan circularly double linked list. Gunakan data string (nama hewan). Pastikan setiap program memuat **operasi-operasi umum dalam linked list** operasi umum linked list.

5.2 Single Linked List

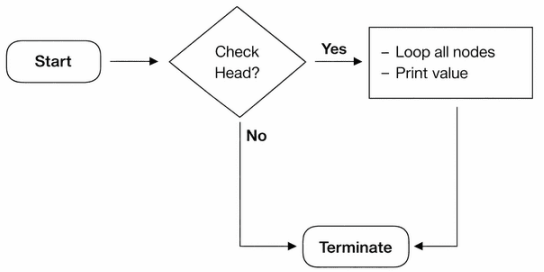
Berikut beberapa flowchart dari single linked list:

♥ Flowchart Penyisipan Node dalam Single Linked List

Gambar 5.8 menunjukkan penyisipan node dalam single linked list. Pengguna menentukan nilai node dan lokasi penyisipan. Selanjutnya dibuat node baru dan nilainya diinisialisasi. Algoritma memeriksa apakah head bernilai None (list kosong). Jika ya, maka node baru ditetapkan sebagai head dan tail. Jika tidak, sistem memeriksa lokasi penyisipan. Apabila lokasi berada di awal (first), maka node baru diarahkan ke head lama dan head diperbarui. Jika lokasi di akhir (last), maka node baru dijadikan elemen terakhir dengan memperbarui pointer tail. Jika bukan di awal maupun akhir, maka dilakukan pencarian posisi menggunakan proses perulangan (loop), kemudian pointer diatur sehingga node baru tersisip di tengah daftar (Karimov, 2022).



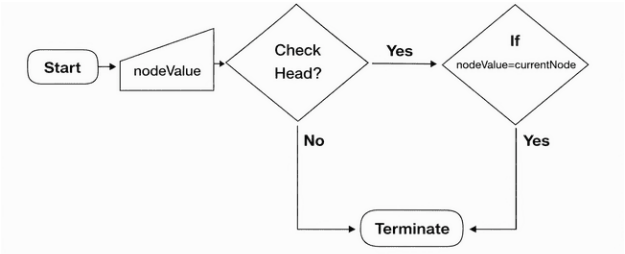
Gambar 5.8: Algoritma Insert dalam Single Linked List



Gambar 5.9: Algoritma Penelusuran Node dalam Single Linked List (Karimov, 2022)

♥ Flowchart Penelusuran dalam Single Linked List

Gambar 5.9 menunjukkan flowchart penelusuran node dalam linked list. Proses diawali dengan pemeriksaan kondisi melalui keputusan "Check Head?". Keputusan ini bertujuan untuk menentukan apakah pointer head bernilai kosong (null) atau tidak. Jika hasilnya "Yes" (artinya head tidak kosong dan terdapat node dalam linked list), maka algoritma melanjutkan ke proses perulangan untuk mengunjungi seluruh node (loop all nodes) dan menampilkan nilai yang tersimpan dalam setiap node (print value). Setelah proses penelusuran selesai, alur diarahkan menuju Terminate. Sebaliknya, jika hasil pemeriksaan adalah "No" (linked list kosong), maka proses langsung diakhiri tanpa melakukan penelusuran.



Gambar 5.10: Algoritma Pencarian Node dalam Single Linked List (Karimov, 2022)

♥ Flowchart Pencarian dalam Single Linked List

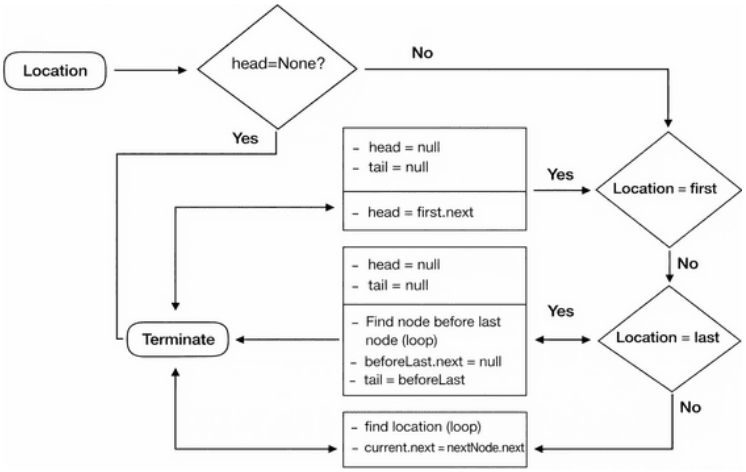
Gambar 5.10 menampilkan flowchart yang menggambarkan pencarian node dalam single linked list. Sistem menerima nilai yang akan dicari (nodeValue). Selanjutnya dilakukan pemeriksaan Check Head? untuk memastikan bahwa linked list tidak kosong. Jika head tidak kosong (Yes), algoritma membandingkan nilai yang dicari dengan nilai dalam node saat ini (If nodeValue = currentNode). Jika sesuai (Yes), proses dihentikan (Terminate) karena data telah ditemukan. Jika tidak sesuai (No), proses berlanjut ke node berikutnya dan perbandingan diulang hingga data ditemukan atau daftar berakhir. Jika head kosong sejak awal (No), proses langsung dihentikan.

♥ Flowchart Penghapusan dalam Single Linked List

Gambar 5.11 menampilkan diagram alir penghapusan node dalam singly linked list. Proses dimulai dengan menentukan lokasi node yang akan dihapus, kemudian dilakukan pemeriksaan apakah head bernilai kosong (head = None?). Jika daftar kosong, proses langsung dihentikan (*Terminate*). Jika tidak kosong, algoritma memeriksa apakah node yang dihapus berada dalam posisi pertama (location = first). Jika benar, maka head diperbarui menjadi node berikutnya (head = first.next), atau jika hanya terdapat satu node, maka head dan tail diatur menjadi null. Apabila node yang dihapus berada di posisi terakhir (location = last), algoritma mencari node sebelum node terakhir melalui proses perulangan, kemudian memperbarui tail dan memutuskan relasi ke node terakhir. Jika node berada di posisi tengah, algoritma melakukan penelusuran untuk menemukan lokasi node tersebut, lalu menghubungkan kembali node sebelumnya ke node setelahnya (current.next = nextNode.next). Diagram ini

secara sistematis menggambarkan percabangan logika dalam proses penghapusan node terhadap singly linked list.

Contoh program singly linked list dapat dilihat di link https://colab.research.google.com/drive/1nf5LW0Wv3AMBbSVsmfO-3Hzed--xZh0J?usp=sharing.

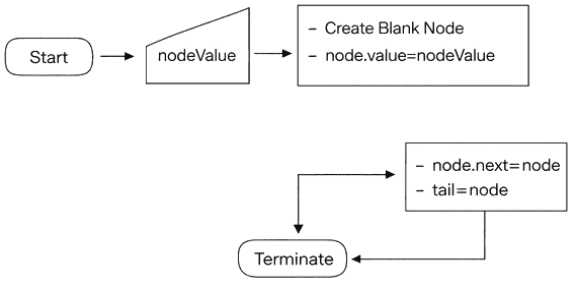


Gambar 5.11: Algoritma Penghapusan Node dalam Single Linked List (Karimov, 2022)

5.3 Circularly Single Linked List

Berikut beberapa flowchart dari circularly single linked list:

♥ Flowchart Pembuatan Node dalam Circularly Single Linked List



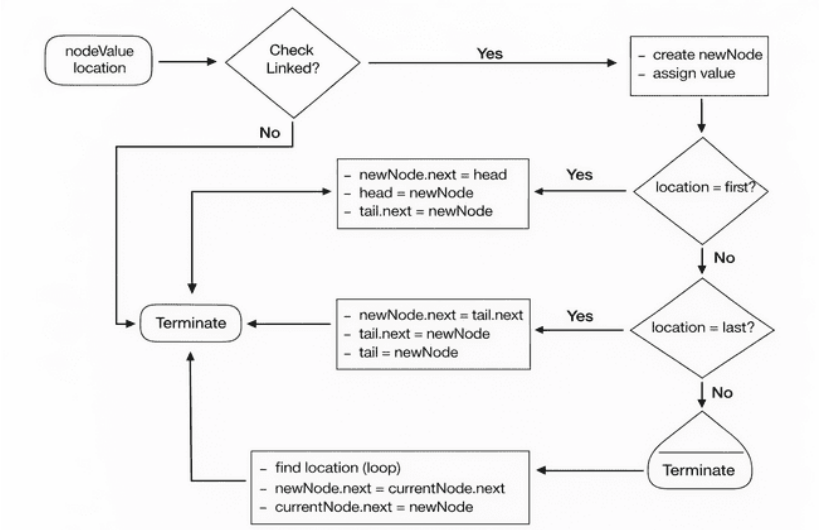
Gambar 5.12: Algoritma Pembuatan Node dalam Circularly Single Linked List (Karimov, 2022)

Gambar 5.12 menampilkan diagram alir yang menggambarkan proses pembuatan node pertama dalam singly linked list. Proses dimulai dengan sistem menerima nilai data yang akan dimasukkan (nodeValue). Selanjutnya dibuat sebuah node baru (Create Blank Node) dan nilai node tersebut diisi sesuai dengan nilai masukan (node.value = nodeValue). Setelah itu, pointer next dari node diarahkan ke dirinya sendiri atau disesuaikan sesuai kebutuhan inisialisasi awal (node.next = node). Tahap akhir, variabel head dan tail diperbarui agar menunjuk ke node yang baru dibuat (head = node dan tail = node). Proses kemudian diakhiri di tahap Terminate.

♥ Flowchart Penyisipan Node dalam Circularly Single Linked List

Gambar 5.13 menampilkan diagram alir penyisipan node dalam singly linked list. Proses dimulai dengan menerima dua masukan, yaitu nilai node (nodeValue) dan lokasi penyisipan (location). Selanjutnya dilakukan pemeriksaan apakah linked list telah terhubung atau tidak (Check Linked?). Jika tidak terhubung (kosong), proses dihentikan (Terminate). Jika terhubung, maka dibuat node baru (create newNode) dan nilainya diisi sesuai input (assign value). Tahap berikutnya adalah menentukan posisi penyisipan. Jika lokasi adalah posisi pertama (location

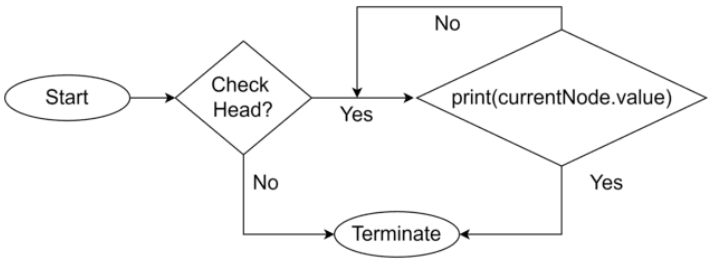
= first), maka node baru diarahkan ke head sebelumnya (newNode.next = head), kemudian head diperbarui menjadi node baru, dan relasi tail.next disesuaikan. Jika lokasi adalah posisi terakhir (location = last), maka node baru dihubungkan setelah tail (newNode.next = tail.next), relasi tail diperbarui (tail.next = newNode), dan tail menunjuk ke node baru. Apabila lokasi berada di tengah, algoritma melakukan penelusuran (loop) untuk menemukan posisi yang sesuai, kemudian mengatur ulang pointer dengan menghubungkan node baru di antara node sebelumnya dan node berikutnya (newNode.next = currentNode.next dan currentNode.next = newNode). Setelah proses selesai, algoritma berakhir di tahap Terminate. Diagram ini secara sistematis menggambarkan percabangan logika dalam proses penyisipan dalam singly linked list.



Gambar 5.13: Algoritma Penyisipan Node dalam Circularly Single Linked List (Karimov, 2022)

♥ Flowchart Penelusuran Node dalam Circularly Single Linked List

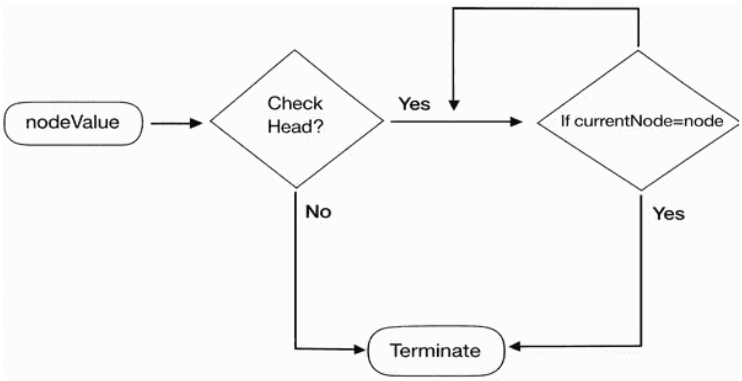
Gambar 5.14 menunjukkan algoritma traversal dalam single linked list. Proses dimulai dengan dilakukan pemeriksaan terhadap kondisi Check Head? untuk memastikan apakah node awal (head) tersedia atau tidak. Jika kondisi bernilai No (misalnya head bernilai null), maka proses langsung dihentikan dalam tahap Terminate karena tidak ada elemen yang dapat ditelusuri. Apabila kondisi Check Head? bernilai Yes, maka algoritma melanjutkan ke proses print(currentNode.value), yaitu menampilkan nilai dari node yang sedang diakses. Setelah nilai dicetak, alur kembali melakukan pemeriksaan untuk menentukan apakah masih terdapat node berikutnya yang perlu ditelusuri. Proses ini berlangsung secara berulang hingga seluruh node selesai diproses, kemudian algoritma



Gambar 5.14: Algoritma Traversal dalam Circularly Single Linked List (Karimov, 2022)

diakhiri di simbol Terminate. Diagram ini secara sederhana menggambarkan mekanisme penelusuran elemen secara berurutan dari node pertama hingga node terakhir dalam sebuah singly linked list.

♥ Flowchart Pencarian dalam Circularly Single Linked List

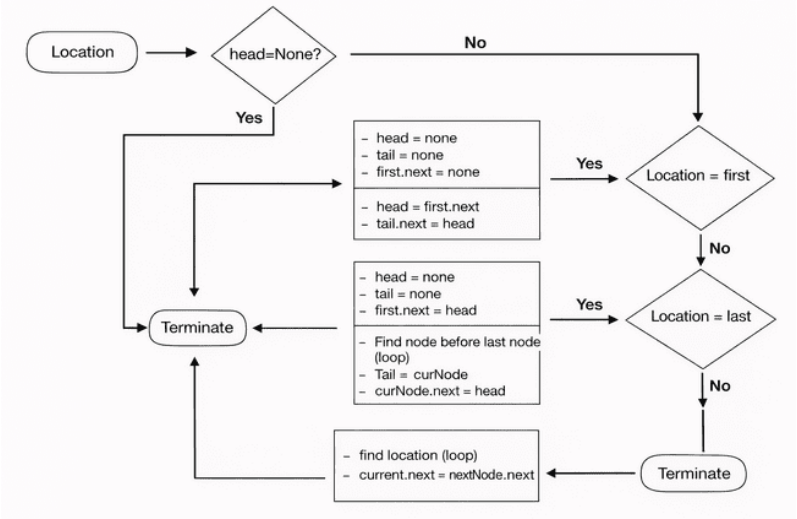


Gambar 5.15: Algoritma Pencarian Node dalam Circularly Single Linked List (Karimov, 2022)

Gambar 5.15 menampilkan diagram alir algoritma pencarian dalam circularly singly linked list. Proses dimulai dengan menerima nilai yang akan dicari (nodeValue), kemudian dilakukan pemeriksaan terhadap kondisi Check Head? untuk memastikan apakah node awal (head) tersedia. Jika head bernilai kosong (No), maka proses langsung dihentikan di tahap Terminate karena tidak ada elemen yang dapat diperiksa. Apabila head tersedia (Yes), algoritma melanjutkan dengan membandingkan apakah node yang sedang diperiksa (currentNode) sama dengan nilai yang dicari (If currentNode = node). Jika kondisi tersebut bernilai Yes, maka pencarian dinyatakan berhasil dan proses diakhiri. Jika bernilai No, karena struktur bersifat sirkular, penelusuran dilanjutkan ke node berikutnya dan akan kembali ke node awal apabila belum ditemukan. Proses ini berulang hingga elemen ditemukan atau seluruh node dalam siklus telah diperiksa, kemudian algoritma berakhir dalam tahap Terminate.

♥ Flowchart Penghapusan Node dalam Circularly Single Linked List

Gambar 5.16 menunjukkan diagram alir penghapusan node dalam circularly singly linked list. Proses



Gambar 5.16: Algoritma Penghapusan Node dalam Circularly Single Linked List (Karimov, 2022)

dimulai dengan menerima parameter location, kemudian dilakukan pemeriksaan apakah head bernilai kosong (head = None?). Jika kondisi bernilai Yes, artinya list tidak memiliki elemen, sehingga proses langsung dihentikan di tahap Terminate. Apabila head tidak kosong (No), algoritma memeriksa posisi penghapusan. Jika location = first, maka node pertama dihapus dengan memperbarui head menjadi first.next dan mengatur ulang tail.next agar tetap menunjuk ke head, sehingga sifat sirkular tetap terjaga. Jika location = last, algoritma melakukan penelusuran untuk menemukan node sebelum node terakhir (find node before last node), kemudian memperbarui tail dan mengatur curNode.next = head. Jika lokasi berada di tengah, dilakukan pencarian posisi menggunakan

proses loop, lalu pointer diperbarui dengan menghubungkan current.next ke nextNode.next. Setelah seluruh proses penghapusan dan pembaruan pointer selesai dilakukan, algoritma diakhiri di tahap Terminate.

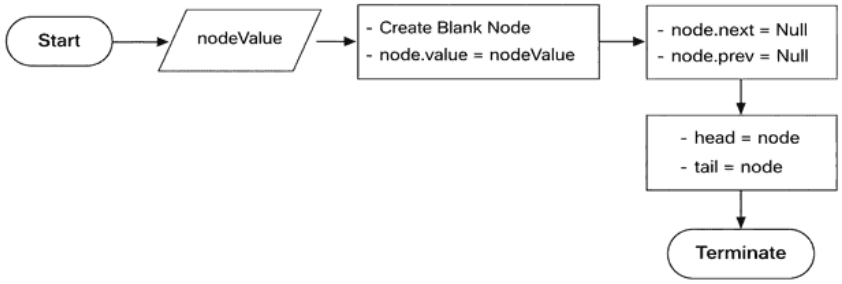
Contoh program singly linked list dapat diluhat di link https://colab.research.google.com/drive/1Ur2WoFiKq7GtatGUJi9Zkwl1Jo-r6hjz?usp=sharing.

5.4 Double Linked List

Berikut beberapa flowchart dari double linked list:

♥ Flowchart Pembuatan Node dalam Double Linked List

Gambar 5.17 menunjukkan diagram alir algoritma pembuatan node dalam double linked list. Proses

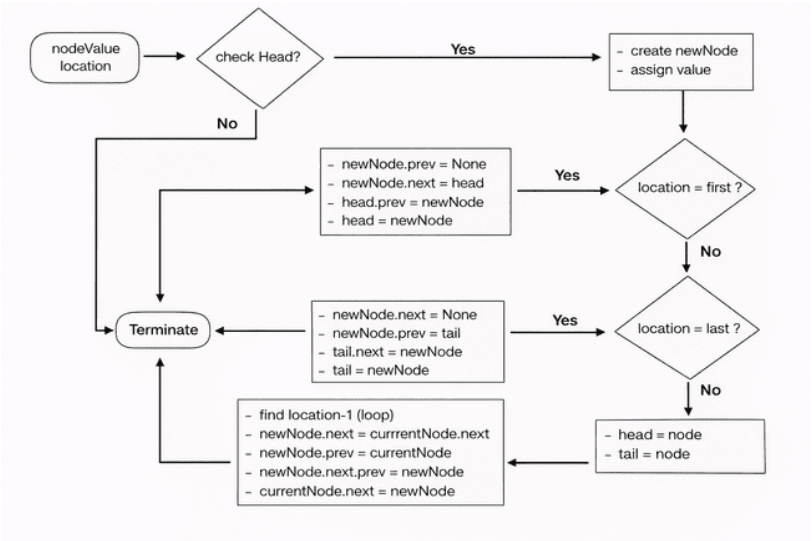


Gambar 5.17: Algoritma Pembuatan Node dalam Double Linked List (Karimov, 2022)

dimulai dengan sistem menerima masukan berupa nodeValue sebagai nilai yang akan disimpan dalam node baru. Setelah itu dilakukan proses pembuatan node kosong (Create Blank Node) dan pengisian nilai ke atribut node (node.value = nodeValue). Langkah berikutnya adalah inisialisasi pointer dengan mengatur node.next = Null dan node.prev = Null, yang menandakan bahwa node tersebut belum terhubung dengan node lain. Karena node yang dibuat merupakan elemen pertama dalam list, maka pointer head dan tail diarahkan ke node tersebut (head = node dan tail = node). Setelah seluruh proses inisialisasi selesai, algoritma diakhiri di tahap Terminate.

♥ Flowchart Penyisipan Node dalam Double Linked List

Gambar 5.18 menampilkan diagram alir algoritma penyisipan node dalam Double Linked List. Proses



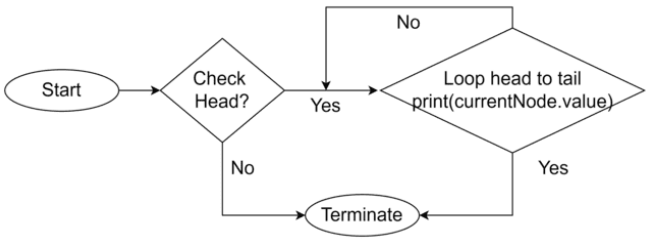
Gambar 5.18: Algoritma Penyisipan Node dalam Double Linked List (Karimov, 2022)

dimulai dengan menerima dua parameter, yaitu nodeValue dan location. Selanjutnya dilakukan pemeriksaan terhadap kondisi check Head? untuk memastikan apakah list sudah memiliki elemen. Jika tidak memiliki head

(kosong), proses dapat langsung diarahkan ke Terminate. Apabila head tersedia, maka membuat node baru (create newNode) dan mengisi nilainya (assign value). Setelah itu, sistem menentukan lokasi penyisipan. Jika lokasi adalah posisi pertama (location = first), maka pointer diatur dengan newNode.prev = None, newNode.next = head, kemudian head.prev = newNode dan head = newNode. Jika lokasi adalah posisi terakhir (location = last), maka pointer diperbarui dengan newNode.next = None, newNode.prev = tail, tail.next = newNode, dan tail = newNode. Apabila penyisipan dilakukan di tengah, algoritma melakukan pencarian posisi menggunakan proses loop, lalu memperbarui pointer secara dua arah, yaitu newNode.next = currentNode.next, newNode.prev = currentNode, newNode.next.prev = newNode, dan currentNode.next = newNode. Setelah seluruh pembaruan pointer selesai dilakukan, proses diakhiri di tahap Terminate.

♥ Flowchart Penelusuran Node dalam Double Linked List

Gambar 5.19 menampilkan algoritma penelusuran node dalam double linked list. Proses dimulai dari

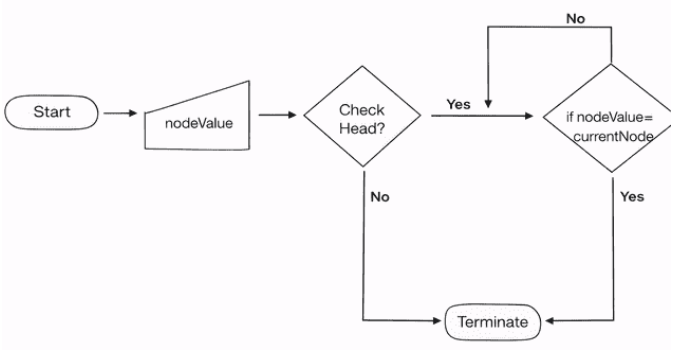


Gambar 5.19: Algoritma Penelusuran Node dalam Double Linked List (Karimov, 2022)

simbol Start, kemudian dilakukan pemeriksaan terhadap kondisi Check Head? untuk memastikan bahwa list memiliki node awal (head). Jika head tidak tersedia (No), maka proses langsung dihentikan di tahap Terminate karena tidak ada elemen yang dapat ditelusuri. Apabila head tersedia (Yes), algoritma melanjutkan ke proses perulangan dari head hingga tail (Loop head to tail). Setiap iterasi, nilai node yang sedang ditunjuk oleh currentNode ditampilkan melalui proses print(currentNode.value). Penelusuran berlangsung secara berurutan dengan mengikuti pointer next hingga mencapai node terakhir (tail). Setelah seluruh node selesai diproses, algoritma diakhiri dalam tahap Terminate.

♥ Flowchart Pencarian Node dalam Double Linked List

Gambar 5.20 menunjukkan algoritma pencarian node dalam Double Linked List. Proses dimulai dari



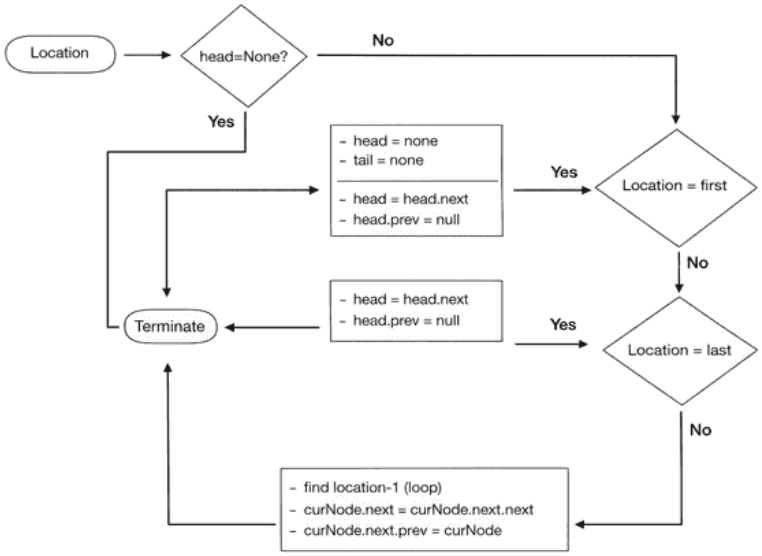
Gambar 5.20: Algoritma Pencarian Node dalam Double Linked List (Karimov, 2022)

simbol Start, kemudian sistem menerima masukan berupa nodeValue yang akan dicari dalam daftar berantai ganda. Setelah itu dilakukan pemeriksaan kondisi awal melalui keputusan "Check Head?", yang bertujuan memastikan apakah list memiliki node (tidak kosong). Jika list kosong (No), maka proses langsung dihentikan (Terminate). Apabila list tidak kosong (Yes), algoritma melanjutkan pemeriksaan dengan membandingkan nilai

yang dicari (nodeValue) dengan nilai dalam node saat ini (currentNode). Jika keduanya sama (Yes), maka node yang dicari ditemukan dan proses dihentikan. Namun, jika tidak sama (No), proses akan bergerak (melalui pointer next) ke node berikutnya dan melakukan pemeriksaan ulang secara iteratif hingga node ditemukan atau mencapai kondisi akhir list. Jika seluruh node telah diperiksa dan nilai tidak ditemukan, maka algoritma berakhir dalam kondisi Terminate.

♥ Flowchart Penghapusan Node dalam Double Linked List

Gambar 5.21 menggambarkan algoritma penghapusan node dalam struktur Double Linked List



Gambar 5.21: Algoritma Penghapusan Node dalam Double Linked List (Karimov, 2022)

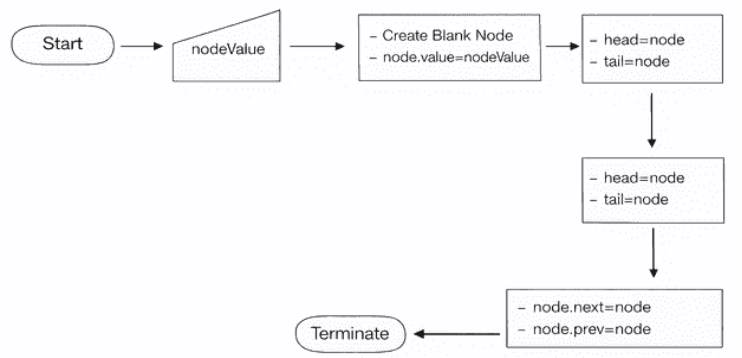
berdasarkan posisi (location) tertentu. Proses dimulai dari masukan Location, kemudian dilakukan pemeriksaan awal terhadap kondisi list melalui keputusan "head = None?". Jika kondisi bernilai Yes, berarti list kosong sehingga tidak ada node yang dapat dihapus dan proses langsung berakhir (Terminate). Apabila list tidak kosong (No), algoritma melanjutkan dengan memeriksa apakah posisi yang akan dihapus adalah node pertama (Location = first). Jika benar, maka penghapusan dilakukan dengan memindahkan pointer head ke head.next dan mengatur head.prev = null agar referensi ke node lama terputus. Jika posisi yang dihapus adalah node terakhir (Location = last), maka pointer tail diperbarui ke tail.prev dan tail.next = null untuk memutus hubungan dengan node terakhir sebelumnya. Jika node yang dihapus berada di posisi tengah, algoritma melakukan proses pencarian hingga posisi ke-(location–1). Setelah ditemukan, pointer diperbarui dengan menghubungkan curNode.next langsung ke curNode.next.next, dan mengatur kembali prev dalam node setelahnya agar tetap menunjuk ke node sebelumnya. Setelah seluruh penyesuaian pointer selesai dilakukan, proses berakhir di kondisi Terminate. Program dan output dari doubly linked list dapat dilihat di link https://colab.research.google.com/drive/1deB6NieVMSUORZ4zD6Rk4LpmLoFpdfOg?usp=sharing.

5.5 Circularly Double Linked List

Berikut beberapa flowchart dari circularly double linked list:

♥ Flowchart Pembuatan Node dalam Circularly Double Linked List

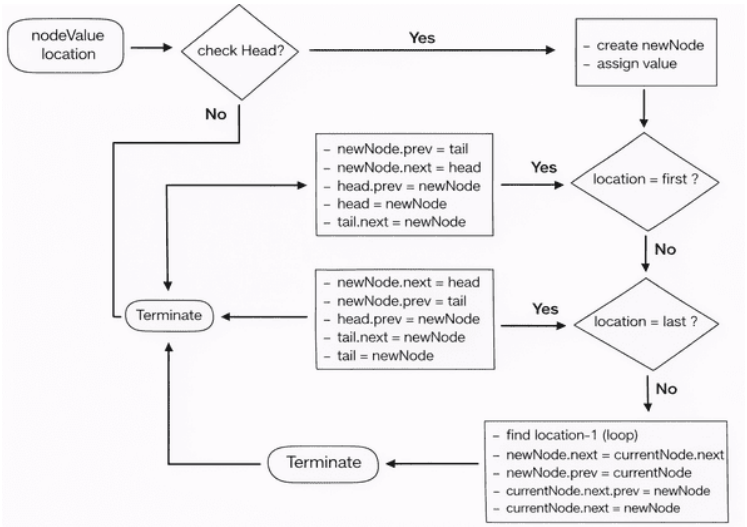
Gambar 5.22 menggambarkan algoritma pembuatan node dalam Circularly Double Linked List. Proses diawali dari simbol Start, kemudian sistem menerima masukan berupa nodeValue yang akan disimpan dalam node baru. Selanjutnya dilakukan proses pembuatan node kosong (Create Blank Node) dan pengisian nilai dengan pernyataan node.value = nodeValue. Setelah node berhasil dibuat, algoritma menginisialisasi struktur list dengan menetapkan head = node dan tail = node, yang berarti node pertama tersebut berperan sekaligus sebagai awal dan akhir daftar. Karena struktur yang dibangun adalah circular, langkah berikutnya adalah mengatur referensi pointer sehingga node.next = node dan node.prev = node. Pengaturan ini memastikan bahwa node menunjuk ke dirinya sendiri baik melalui pointer berikutnya (next) maupun sebelumnya (prev), sehingga membentuk siklus tertutup.



Gambar 5.22: Algoritma Pembuatan dalam Circularly Double Linked List (Karimov, 2022)

♥ Flowchart Penyisipan Node dalam Circularly Double Linked List

Gambar 5.23 menjelaskan algoritma penyisipan node dalam Circularly Double Linked List berdasarkan posisi tertentu (location). Proses dimulai dari masukan nodeValue dan location, kemudian dilakukan pemeriksaan awal melalui keputusan "check Head?" untuk memastikan apakah list dalam kondisi kosong atau tidak. Jika list kosong, maka node baru dibuat melalui langkah create newNode dan assign value, kemudian node tersebut diinisialisasi sebagai satu-satunya elemen dalam list. Karena struktur bersifat sirkular, pointer diatur sehingga newNode.prev = newNode dan newNode.next = newNode, serta head dan tail menunjuk ke node tersebut. Setelah itu proses berakhir (Terminate). Apabila list tidak kosong, algoritma memeriksa posisi penyisipan. Jika location = first, maka node baru ditempatkan di awal dengan pengaturan pointer: newNode.prev = tail, newNode.next = head, head.prev = newNode, dan tail.next = newNode, lalu head diperbarui menjadi newNode. Jika location =

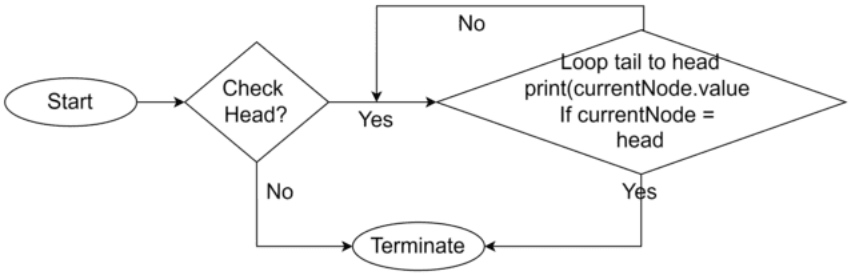


Gambar 5.23: Algoritma Penyisipan Node dalam Circularly Double Linked List (Karimov, 2022)

last, maka node baru disisipkan di akhir dengan mengatur newNode.next = head, newNode.prev = tail, tail.next = newNode, head.prev = newNode, kemudian tail diperbarui menjadi newNode. Jika penyisipan dilakukan di posisi tengah, algoritma melakukan pencarian hingga posisi ke-(location–1). Setelah node sebelumnya ditemukan, pointer diperbarui dengan menghubungkan newNode di antara dua node yang berdekatan, yaitu mengatur newNode.next = currentNode.next, newNode.prev = currentNode, kemudian menyesuaikan pointer node di sekitarnya agar tetap konsisten.

♥ Flowchart Penelusuran Node dalam Circularly Double Linked List

Gambar 5.24 menggambarkan algoritma penelusuran dalam Circularly Double Linked List. Proses

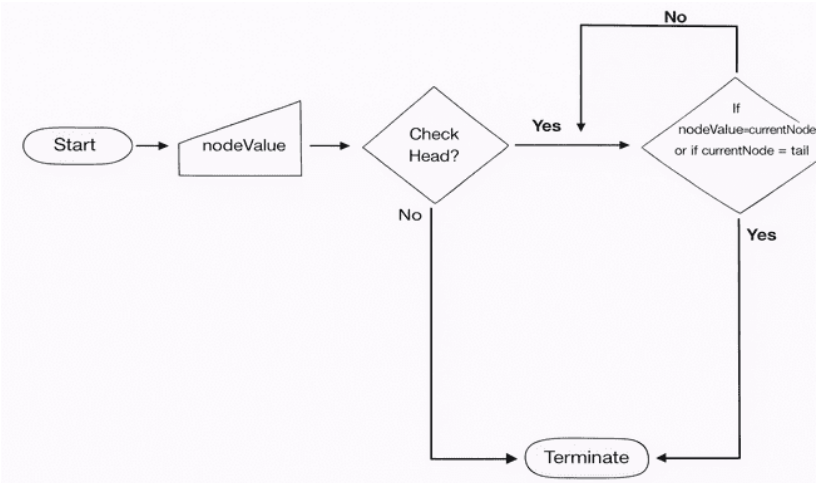


Gambar 5.24: Algoritma Penelusuran Node dalam Circularly Double Linked List (Karimov, 2022)

dimulai dari simbol Start, kemudian dilakukan pemeriksaan awal melalui keputusan "Check Head?" untuk memastikan apakah struktur list memiliki node (tidak kosong). Jika kondisi bernilai No (head kosong), maka proses langsung dihentikan di tahap Terminate. Apabila list tidak kosong (Yes), algoritma melanjutkan dengan melakukan proses perulangan dari tail menuju head (loop tail to head). Setiap iterasi, nilai node saat ini dicetak menggunakan perintah print(currentNode.value). Proses ini terus berulang hingga kondisi currentNode = head terpenuhi, yang menandakan bahwa seluruh node dalam struktur sirkular telah dilalui satu putaran penuh.

♥ Flowchart Pencarian Node dalam Circularly Double Linked List

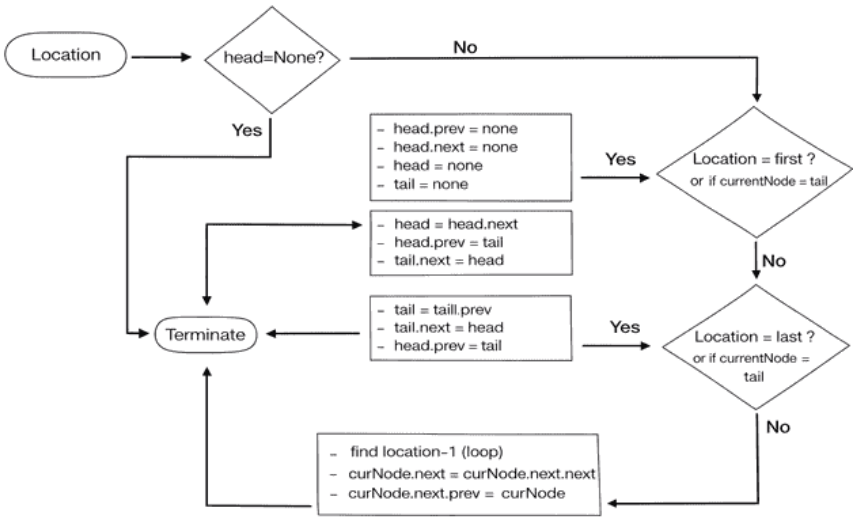
Gambar 5.25 menggambarkan algoritma pencarian node dalam Circularly Double Linked List. Proses dimulai dari simbol Start, kemudian sistem menerima masukan berupa nodeValue yang akan dicari dalam struktur list. Setelah itu dilakukan pemeriksaan awal melalui keputusan "Check Head?" untuk memastikan apakah list memiliki elemen (tidak kosong). Jika kondisi No (head kosong), maka proses langsung dihentikan di tahap Terminate. Apabila list tidak kosong (Yes), algoritma melakukan proses penelusuran dengan membandingkan nilai yang dicari (nodeValue) terhadap nilai dalam node saat ini (currentNode). Setiap iterasi, dilakukan pengecekan kondisi: apakah nodeValue = currentNode atau apakah currentNode = tail. Jika nodeValue sama dengan nilai dalam node saat ini, maka elemen berhasil ditemukan dan proses dihentikan. Jika belum ditemukan dan node saat ini belum mencapai tail, maka proses berlanjut ke node berikutnya. Karena struktur bersifat sirkular, tidak terdapat penanda null sebagai akhir list. Oleh sebab itu, penghentian pencarian didasarkan dalam dua kondisi yaitu nilai ditemukan atau telah mencapai node terakhir (tail). Jika seluruh node telah diperiksa hingga tail dan nilai tidak ditemukan, maka proses berakhir di kondisi Terminate.



Gambar 5.25: Algoritma Pencarian Node dalam Circularly Double Linked List (Karimov, 2022)

♥ Flowchart Penghapusan Node dalam Circularly Double Linked List

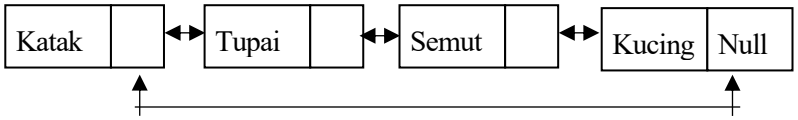
Gambar 5.26 menggambarkan algoritma penghapusan node dalam struktur data Circularly Doubly Linked List. Proses dimulai dengan menerima input berupa location (posisi node yang akan dihapus), kemudian dilakukan pemeriksaan apakah list dalam kondisi kosong melalui keputusan head = None?. Jika list kosong, maka tidak ada node yang dapat dihapus dan proses langsung dihentikan (terminate). Apabila list tidak kosong, algoritma memeriksa apakah node yang akan dihapus berada dalam posisi pertama (location = first). Jika benar, maka pointer head dipindahkan ke node berikutnya, serta hubungan sirkular diperbarui dengan mengatur head.prev = tail dan tail.next = head. Jika node yang dihapus berada dalam posisi terakhir (location = last), maka pointer tail dipindahkan ke node sebelumnya, kemudian tail.next diarahkan ke head dan head.prev diarahkan ke tail untuk mempertahankan sifat melingkar. Jika node yang dihapus berada di posisi tengah, algoritma melakukan pencarian node sebelumnya melalui proses perulangan (find location-1). Setelah ditemukan, pointer next dan prev diperbarui sehingga node sebelum dan sesudahnya saling terhubung kembali, dan node yang ditargetkan terlepas dari struktur list. Secara keseluruhan, algoritma ini memastikan bahwa setelah proses penghapusan, struktur circular doubly linked list tetap konsisten, terhubung dua arah, dan mempertahankan sifat sirkularnya.



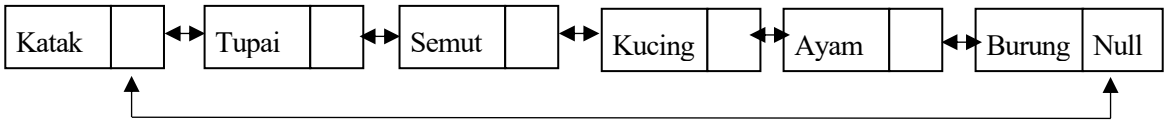
Gambar 5.26: Algoritma Penghapusan Node dalam Circularly Double Linked List (Karimov, 2022)

Contoh perhitungan double linked list:

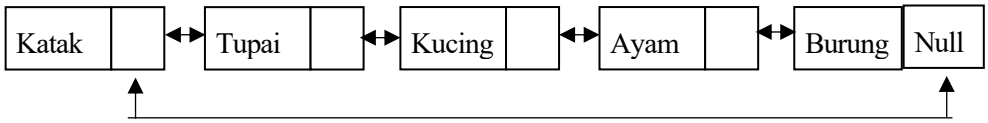
- Insert node Katak, Tupai, Semut, Kucing secara **first ordered list**



- Insert node Ayam, Burung secara **last list**



- Delete node Semut

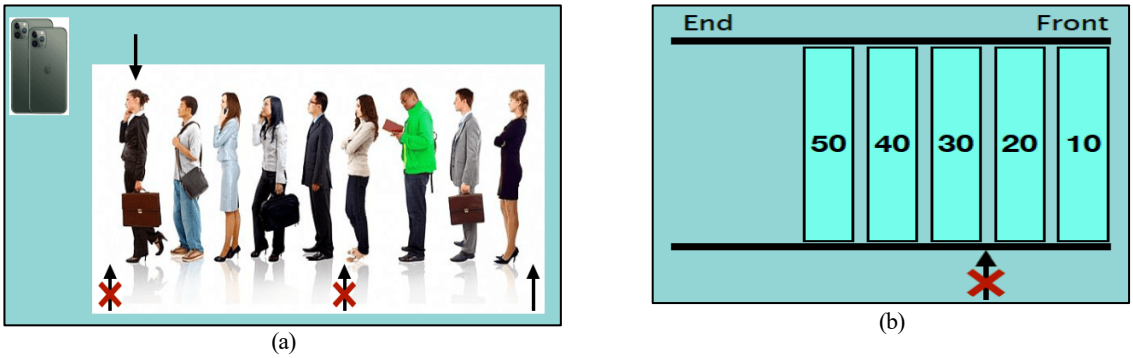


Contoh program circular doubly linked list dapat dilihat di link https://colab.research.google.com/drive/1snOR-pozEPoefOzukdKF9qY1zW76GQwp?usp=sharing.

BAB 6
Queue (Antrian)

Queue (antrian) merupakan struktur data linear yang bekerja berdasarkan prinsip First In First Out (FIFO), yaitu elemen yang pertama kali masuk akan menjadi elemen pertama yang keluar. Konsep ini merepresentasikan mekanisme pelayanan dalam kehidupan sehari-hari, seperti barisan pelanggan di toko atau sistem pusat panggilan, di mana layanan diberikan sesuai urutan kedatangan. Dalam struktur ini, **penambahan elemen (*insert*) dilakukan di bagian belakang (*rear*)**, sedangkan penghapusan elemen (*delete*) dilakukan di ujung depan (*front*) (Agarwal, 2022). Elemen pertama dalam antrian disebut sebagai *front* (*head*), yang menunjukkan elemen pertama yang telah masuk dan siap untuk diambil. Sebaliknya, *rear* (*tail*) merujuk ke elemen terakhir dalam antrian, yaitu elemen yang baru saja ditambahkan dan siap untuk dikeluarkan setelah elemen-elemen sebelumnya diproses. Queue umumnya digunakan dalam berbagai aplikasi, seperti penjadwalan tugas dalam sistem komputer, antrian di layanan pelanggan, serta dalam pemrograman yang melibatkan pengelolaan sumber daya secara teratur dan berurutan. **Queue** ini juga sering disebut sebagai **linier queue**.

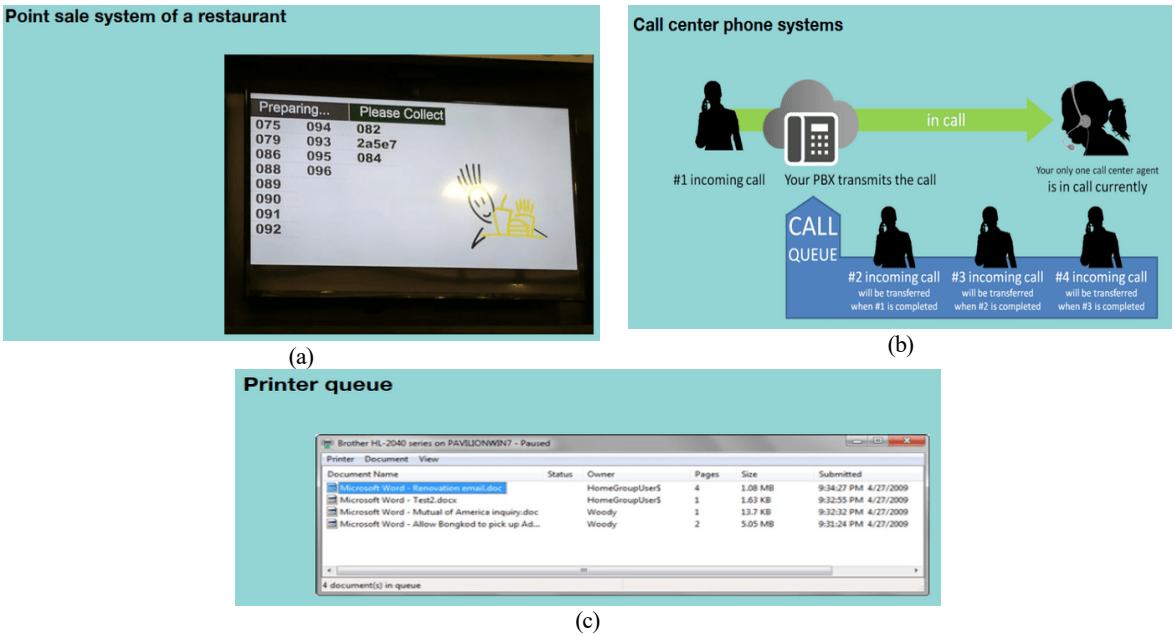
Secara implementatif, antrian dapat direpresentasikan menggunakan array linier maupun struktur daftar tertaut, dengan dua penunjuk utama, yaitu front sebagai penanda posisi penghapusan dan rear sebagai penanda posisi penyisipan. Karakteristik operasional ini menegaskan bahwa elemen hanya dapat keluar melalui bagian depan setelah dilayani, sementara elemen baru selalu ditempatkan di bagian belakang. Sebagai salah satu struktur data fundamental, antrian memiliki peran penting dalam berbagai sistem komputasi dan menjadi dasar bagi pengembangan struktur data serta mekanisme pemrosesan lainnya yang memerlukan pengelolaan urutan secara terstruktur dan konsisten.



(a) (b)

Gambar 6.1: Ilustrasi Queue (Karimov, 2022)

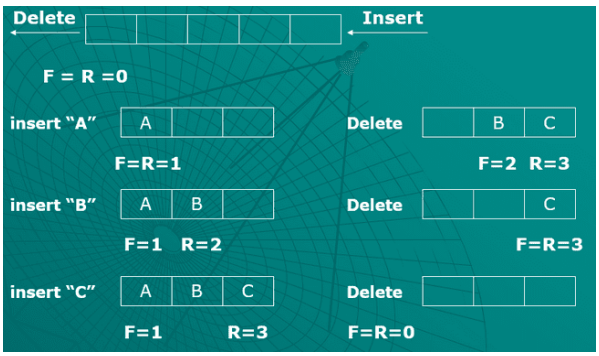
Gambar 6.1 menampilkan ilustrasi konsep queue (antrian) melalui dua representasi, yaitu analogi kehidupan nyata dan model struktur data. Bagian (a), antrian digambarkan sebagai barisan individu yang menunggu layanan secara berurutan. Ilustrasi tersebut menunjukkan bahwa individu yang berada di posisi paling depan akan dilayani terlebih dahulu, sedangkan individu yang baru datang akan bergabung di bagian belakang barisan. Tanda silang (X) menegaskan bahwa penghapusan elemen hanya dapat dilakukan bagian depan antrian, sedangkan penambahan elemen terjadi di bagian belakang. Bagian (b), konsep yang sama direpresentasikan dalam bentuk struktur data linear. Elemen-elemen dengan nilai 50, 40, 30, 20, dan 10 tersusun dalam satu baris, dengan penanda Front dan End yang menunjukkan posisi penghapusan dan penyisipan elemen. Elemen yang berada dalam posisi Front merupakan elemen pertama yang akan dikeluarkan, sesuai dengan prinsip FIFO. Secara umum, ilustrasi tersebut menegaskan bahwa queue merupakan struktur data linear yang mengatur elemen sesuai urutan kedatangannya. Proses penambahan elemen (enqueue) berlangsung di bagian belakang, sedangkan proses pengeluaran elemen (dequeue) dilakukan melalui bagian depan secara sistematis dan konsisten.



Gambar 6.2: Penerapan Queue (Karimov, 2022)

Gambar 6.2 menyajikan beberapa contoh penerapan konsep queue dalam konteks nyata dan sistem komputasi. Bagian (a) memperlihatkan sistem point of sale di restoran, di mana pesanan pelanggan diproses sesuai urutan kedatangan. Pesanan yang masuk lebih dahulu akan disiapkan dan diselesaikan lebih awal dibandingkan pesanan yang datang kemudian, sehingga mencerminkan prinsip FIFO. Bagian (b) menggambarkan sistem pusat panggilan (call center), di mana panggilan masuk ditangani berdasarkan urutan penerimaan. Ketika satu agen masih melayani panggilan, panggilan berikutnya akan menunggu dalam antrean hingga agen tersedia. Mekanisme ini menunjukkan pengelolaan layanan secara berurutan sesuai prinsip FIFO. Bagian (c) menampilkan antrean pencetakan (printer queue), di mana dokumen yang dikirim ke printer akan dicetak sesuai urutan pengiriman. Dokumen pertama yang masuk ke sistem pencetakan akan diproses lebih dahulu sebelum dokumen berikutnya. Secara keseluruhan, ilustrasi tersebut menegaskan bahwa struktur data queue banyak digunakan dalam berbagai sistem layanan dan pemrosesan yang memerlukan pengelolaan urutan secara teratur dan konsisten.

Kondisi awal queue dimulai dengan F = R = 0, yang mengindikasikan bahwa antrian tersebut kosong (*empty queue*). Ini berarti tidak ada elemen dalam antrian yang dapat diproses atau diakses. Ketika data baru dimasukkan ke dalam antrian, elemen tersebut akan masuk melalui ujung belakang (REAR). Proses penambahan data ini terjadi dengan menambahkannya ke akhir antrian. Di sisi lain, ketika data dihapus, elemen yang dihapus akan diambil dari ujung depan (FRONT) antrian. Dengan kata lain, elemen pertama yang dimasukkan akan menjadi elemen pertama yang dikeluarkan, sesuai dengan prinsip FIFO.



Gambar 6.3: Contoh Operasi Queue

Gambar 6.3 menunjukkan contoh operasi dalam queue, yang menggunakan prinsip FIFO. Contoh pertama, antrian dimulai dengan keadaan kosong, yaitu F = R = 0. Setelah itu, data pertama ("A") dimasukkan ke dalam antrian, dan nilai F dan R berubah menjadi 1, menandakan bahwa antrian berisi satu elemen. Selanjutnya, data kedua ("B") dimasukkan ke antrian, dengan nilai F tetap 1 dan R menjadi 2, menunjukkan ada dua elemen di antrian. Operasi berikutnya, data "A" dihapus dari antrian melalui posisi depan (F), yang mengubah antrian menjadi hanya berisi elemen "B". Nilai F kemudian diubah menjadi 2, dan R tetap 3. Setelah itu, data ketiga ("C") dimasukkan ke dalam antrian, mengubah antrian menjadi berisi elemen "B" dan "C", dengan F tetap 2 dan R menjadi 3. Proses penghapusan dilakukan lagi, kali ini menghapus elemen "B" dari depan, dengan F menjadi 3 dan R tetap 3. Terakhir, elemen "C" dihapus, mengembalikan antrian ke kondisi kosong dengan F = R = 0.

Kelemahan queue, khususnya terkait dengan kondisi penuh dalam antrian. Meskipun ada ruang kosong dalam antrian, jika kondisi queue sudah penuh dengan nilai R (Rear) dan F (Front) sama dengan nilai maksimum (Max) queue, maka penambahan elemen baru tidak dapat dilakukan. Ini terjadi karena queue menggunakan batasan kapasitas, dan ketika posisi Rear dan Front bertemu, meskipun ada ruang kosong, tidak bisa menambah elemen baru sampai ada penghapusan elemen dari antrian. Kondisi ini sering disebut sebagai *queue overflow*, yang menghambat aliran data yang masuk ke dalam antrian. Sebagai contoh, jika antrian memiliki kapasitas maksimum dan posisi rear berada di akhir kapasitas tersebut (R=Max), maka meskipun ada ruang kosong setelah elemen dihapus dari bagian depan, sistem tidak akan membiarkan elemen baru ditambahkan kecuali ada operasi penghapusan yang benar-benar mengosongkan ruang di antrian. Hal ini mempengaruhi efisiensi dari penggunaan antrian dalam aplikasi yang memerlukan penambahan elemen secara terus menerus. Contoh 1 perhitungan manual linier queue.

Queue dengan kapasitas 6.
Langkah 1: Inisialisasi
Terdapat antrian dengan kapasitas maxQueue = 6, dan dimulai dengan antrian kosong.
Queue: []
Front: 0
Rear: -1

Langkah 2: Insert Elemen
Insert A: Menambahkan elemen pertama "A".
Queue: [A]
Front: 0
Rear: 0

Insert B: Menambahkan elemen kedua "B".
Queue: [A, B]
Front: 0
Rear: 1

Insert C: Menambahkan elemen ketiga "C".
Queue: [A, B, C]
Front: 0
Rear: 2

Insert D: Menambahkan elemen keempat "D".
Queue: [A, B, C, D]
Front: 0
Rear: 3

Insert E: Menambahkan elemen kelima "E".
Queue: [A, B, C, D, E]
Front: 0
Rear: 4

Insert F: Menambahkan elemen keenam "F".
Queue: [A, B, C, D, E, F]
Front: 0
Rear: 5 (antrian penuh, kapasitas 6 tercapai)

Langkah 3: Antrian Penuh
Antrian telah penuh, dan Rear berada di posisi 5 (maksimum kapasitas). Jika mencoba menambahkan elemen lagi, misalnya "G", maka penambahan gagal karena Rear sudah mencapai kapasitas maksimum.

Langkah 4: Dequeue (Penghapusan Elemen)
Delete: Menghapus elemen pertama "A".
Queue: [B, C, D, E, F]
Front: 1
Rear: 5

Meskipun telah menghapus elemen pertama, Rear tetap di posisi 5, maka tidak ada perubahan dalam kemampuan untuk menambahkan elemen baru.

Langkah 5: Insert Lagi
Insert G: Coba menambahkan elemen baru "G" ke dalam antrian.
Queue: Gagal, karena Rear = 5 dan antrian penuh meskipun ada ruang kosong setelah penghapusan elemen.

Contoh 2 perhitungan manual linier queue.

Jawaban dari contoh 2 tersebut:

Kondisi Awal: MaxQueue = 11
Antrian dimulai dalam keadaan kosong.

Langkah-langkah Proses:
1. InitQueue (Antrian Kosong)
Queue: []
Front = -1, Rear = -1
(Queue kosong, tidak ada elemen)

2. Insert I
Queue: [I]
Front = 0, Rear = 0
(Elemen pertama dimasukkan di posisi 0)

3. Insert N
Queue: [I, N]
Front = 0, Rear = 1
(Elemen kedua dimasukkan di posisi 1)

4. Insert F
Queue: [I, N, F]
Front = 0, Rear = 2
(Elemen ketiga dimasukkan di posisi 2)

5. Insert O
Queue: [I, N, F, O]
Front = 0, Rear = 3
(Elemen keempat dimasukkan di posisi 3)

6. Insert R
Queue: [I, N, F, O, R]
Front = 0, Rear = 4
(Elemen kelima dimasukkan di posisi 4)

7. Insert M
Queue: [I, N, F, O, R, M]
Front = 0, Rear = 5
(Elemen keenam dimasukkan di posisi 5)

8. Insert A
Queue: [I, N, F, O, R, M, A]
Front = 0, Rear = 6
(Elemen ketujuh dimasukkan di posisi 6)

9. Insert T
Queue: [I, N, F, O, R, M, A, T]
Front = 0, Rear = 7
(Elemen kedelapan dimasukkan di posisi 7)

10. Insert I
Queue: [I, N, F, O, R, M, A, T, I]
Front = 0, Rear = 8
(Elemen kesembilan dimasukkan di posisi 8)

11. Insert K
Queue: [I, N, F, O, R, M, A, T, I, K]
Front = 0, Rear = 9
(Elemen kesepuluh dimasukkan di posisi 9)

12. Insert A
Queue: [I, N, F, O, R, M, A, T, I, K, A]
Front = 0, Rear = 10
(Elemen kesebelas dimasukkan di posisi 10)
Queue Penuh
Antrian penuh dengan kapasitas MaxQueue = 11
Front = 0, Rear = 10

Langkah Deletion:
13. Delete (Dequeue)
Queue setelah Dequeue: [N, F, O, R, M, A, T, I, K, A]
Front = 1, **Rear = 10
Elemen pertama "I" dihapus, Front bergerak ke posisi 1.

14. Delete (Dequeue)
Queue setelah Dequeue: [F, O, R, M, A, T, I, K, A]
Front = 2, **Rear = 10
Elemen kedua "N" dihapus, Front bergerak ke posisi 2.

Langkah Insert:
15. Insert M
Queue setelah Insert M: [F, O, R, M, A, T, I, K, A]
Front = 2, Rear = 10
Queue: Gagal, karena Rear = 10 dan antrian penuh, meskipun ada ruang kosong setelah penghapusan elemen.

16. Insert P
Queue setelah Insert M: [F, O, R, M, A, T, I, K, A]
Front = 2, Rear = 10
Queue: Gagal, karena Rear = 10 dan antrian penuh, meskipun ada ruang kosong setelah penghapusan elemen.

17. Delete (Dequeue)
Queue setelah Dequeue: [O, R, M, A, T, I, K, A]
Front = 3, **Rear = 10
Elemen ketiga "F" dihapus, Front bergerak ke posisi 3.

18. Delete (Dequeue)
Queue setelah Dequeue: [R, M, A, T, I, K, A]
Front = 4, **Rear = 10
Elemen keempat "O" dihapus, Front bergerak ke posisi 4.

19. Delete (Dequeue)
Queue setelah Dequeue: [M, A, T, I, K, A]
Front = 5 **Rear = 10
Elemen kelima "R" dihapus, Front bergerak ke posisi 5.

20. Insert U
Queue setelah Dequeue: [M, A, T, I, K, A]
Front = 3, **Rear = 10
Queue: Gagal, karena Rear = 10 dan antrian penuh, meskipun ada ruang kosong setelah penghapusan elemen.

21. Delete (Dequeue)
Queue setelah Dequeue: [A, T, I, K, A]
Front = 6, **Rear = 10
Elemen keenam "M" dihapus, Front bergerak ke posisi 6.

22. Insert S (Dequeue)
Queue setelah Dequeue: [A, T, I, K, A]
Front = 6, **Rear = 10
Queue: Gagal, karena Rear = 10 dan antrian penuh, meskipun ada ruang kosong setelah penghapusan elemen.

23. Delete (Dequeue)
Queue setelah Dequeue: [T, I, K, A]
Front = 7, **Rear = 10
Elemen ketujuh "A" dihapus, Front bergerak ke posisi 7.

6.1 Operasi-operasi Queue

Operasi-operasi dalam queue antara lain:

- Create Queue

Operasi ini digunakan untuk membuat antrian baru yang kosong. Antrian ini siap untuk menerima elemen-elemen yang akan dimasukkan. Operasi ini melihat antrian baru yang kosong dengan nilai FRONT dan REAR diatur dengan -1 sebagai penanda bahwa antrian belum memiliki elemen. Ukuran maksimal antrian juga ditentukan MAX_SIZE, seperti yang terlihat dalam algoritma create queue.

Algoritma create queue:
Step 1: SET FRONT = -1
Step 2: SET REAR = -1
Step 3: SET MAX = MAX_SIZE (Ukuran maksimal antrian)
Step 4: SET QUEUE[MAX] (Inisialisasi antrian kosong)
Step 5: END

- Enqueue

Proses ini melibatkan penambahan elemen baru ke dalam antrian, yang dilakukan di ujung belakang antrian (*rear*). Setiap elemen yang dimasukkan akan ditempatkan di bagian belakang. Operasi ini menambahkan elemen baru ke antrian di ujung belakang (REAR). Jika antrian sudah penuh, maka terjadi overflow. Jika antrian kosong (REAR = -1), maka FRONT diatur ke 0. Kemudian elemen baru dimasukkan ke dalam antrian, seperti yang terlihat dalam algoritma enqueue.

Algoritma enqueue:
Step 1: IF REAR = MAX-1
  PRINT "Overflow"
  Goto Step 4
Step 2: IF FRONT = -1
  SET FRONT = 0
Step 3: SET REAR = REAR + 1
Step 4: SET QUEUE[REAR] = VALUE
Step 5: END

- Dequeue

  Operasi ini digunakan untuk menghapus elemen dari antrian, yang dilakukan di ujung depan (*front*). Elemen pertama yang masuk adalah elemen pertama yang keluar, sesuai dengan prinsip FIFO. Operasi ini menghapus elemen dari ujung depan antrian (FRONT). Jika antrian kosong, terjadi underflow. Setelah elemen dihapus, jika hanya ada satu elemen, FRONT dan REAR akan diatur ke -1, menandakan bahwa antrian kosong, seperti yang terlihat dalam algoritma dequeue.

```
Algoritma dequeue:
Step 1: IF FRONT = -1
   PRINT "Underflow"
   Goto Step 4
Step 2: SET REMOVED = QUEUE[FRONT]
Step 3: IF FRONT = REAR
   SET FRONT = -1
   SET REAR = -1
Step 4: ELSE
   SET FRONT = FRONT + 1
Step 5: END
```

- Peek

  Operasi ini untuk melihat elemen yang berada di depan antrian tanpa menghapusnya. Biasanya digunakan untuk memeriksa elemen yang akan diproses selanjutnya. Apabila antrian kosong, pesan "Queue is empty" ditampilkan, seperti yang terlihat dalam algoritma peek.

```
Algoritma peek:
Step 1: IF FRONT = -1
   PRINT "Queue is empty"
Step 2: ELSE
   PRINT QUEUE[FRONT]
Step 3: END
```

- isEmpty

  Fungsi ini digunakan untuk memeriksa apakah antrian kosong. Jika antrian kosong, maka fungsi ini akan mengembalikan nilai true. Operasi ini digunakan untuk melihat elemen yang ada di depan antrian tanpa menghapusnya. Fungsi ini memeriksa apakah antrian kosong dengan mengecek apakah FRONT bernilai -1. Jika ya, antrian kosong (mengembalikan True), jika tidak maka antrian tidak kosong (mengembalikan False), seperti yang terlihat dalam algoritma isEmpty.

```
Algoritma isEmpty:
Step 1: IF FRONT = -1
   RETURN True
Step 2: ELSE
   RETURN False
Step 3: END
```

- isFull

  Fungsi ini digunakan untuk memeriksa apakah antrian sudah penuh, biasanya digunakan jika antrian memiliki kapasitas terbatas. Fungsi ini memeriksa apakah antrian sudah penuh dengan mengecek apakah REAR telah mencapai kapasitas maksimum antrian (MAX-1). Jika ya, maka antrian penuh (mengembalikan True), jika tidak (mengembalikan False), seperti yang terlihat dalam algoritma isFull.

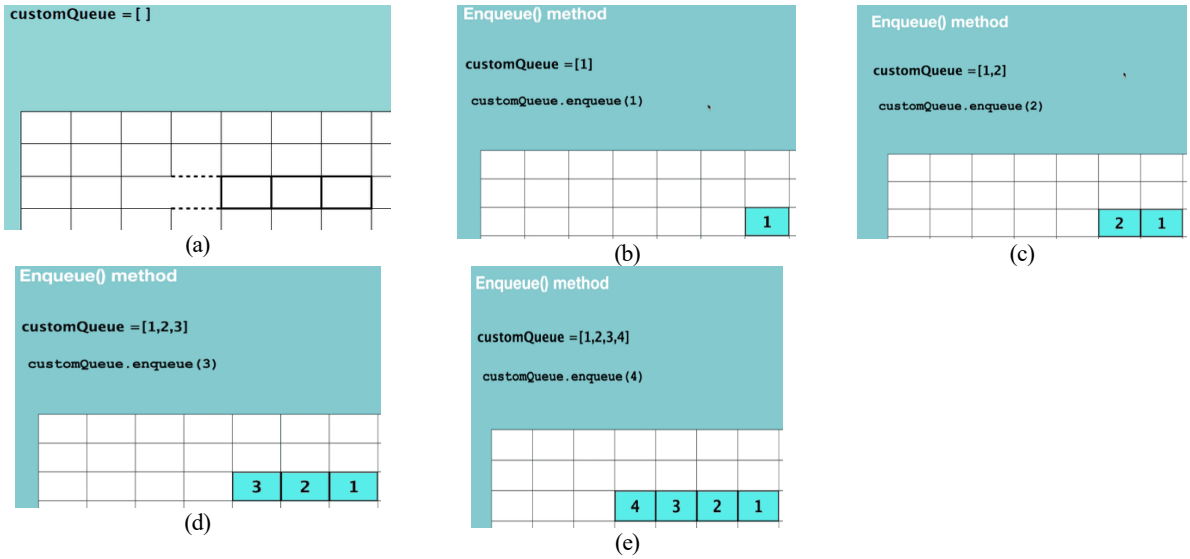
```
Algoritma isFull:
Step 1: IF REAR = MAX-1
   RETURN True
Step 2: ELSE
   RETURN False
Step 3: END
```

- deleteQueue

  Operasi ini digunakan untuk menghapus atau mengosongkan seluruh antrian, menghapus semua elemen yang ada di dalamnya. Langkah pertama dalam algoritma ini adalah mengatur nilai FRONT menjadi -1, yang menandakan bahwa antrian tidak memiliki elemen yang tersisa di bagian depan. Kemudian, nilai REAR juga diatur menjadi -1, yang berarti tidak ada elemen yang tersisa di bagian belakang antrian. Setelah itu, algoritma menampilkan pesan "Queue deleted" untuk memberi tahu bahwa antrian telah berhasil dikosongkan, seperti yang terlihat dalam algoritma deleteQueue.

```
Algoritma deleteQueue:
Step 1: SET FRONT = -1
Step 2: SET REAR = -1
Step 3: PRINT "Queue deleted"
Step 4: END
```

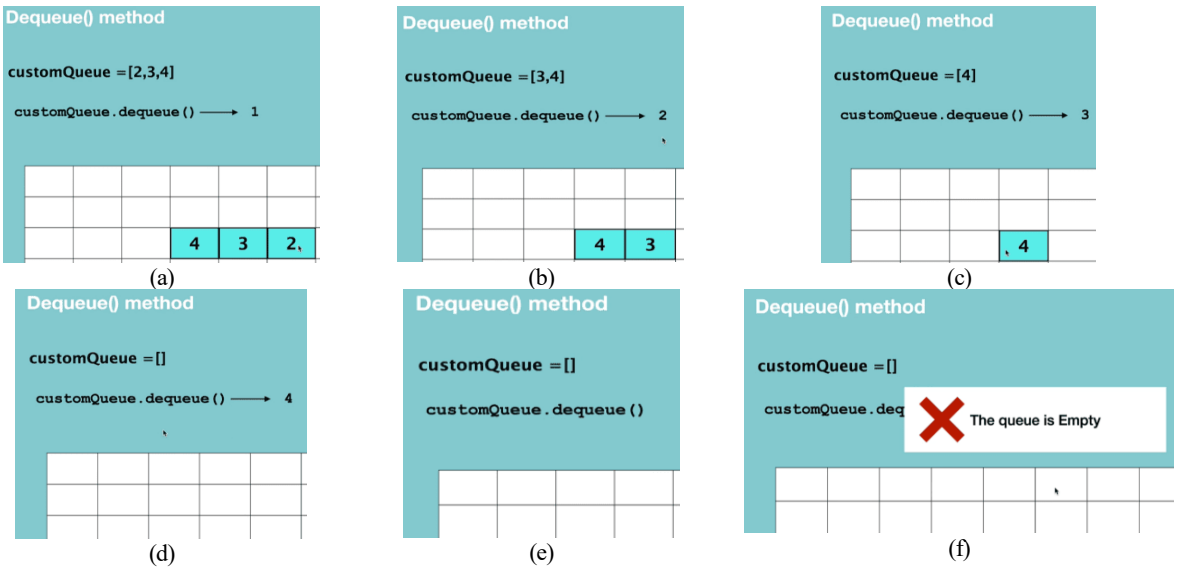
Gambar 6.4 menunjukkan ilustrasi proses pembuatan dan penambahan elemen-elemen ke dalam Queue berbentuk list menggunakan metode enqueue(). Proses dimulai dengan antrian kosong, kemudian elemen pertama, yaitu angka 1, dimasukkan menggunakan metode enqueue(), sehingga antrian berisi satu elemen: [1].



Gambar 6.4: Pembuatan dan Penambahan Elemen dalam Queue Berbentuk List (Karimov, 2022)

Elemen kedua, angka 2, kemudian ditambahkan, mengubah antrian menjadi [2, 1]. Setelah itu, elemen 3 ditambahkan ke antrian, sehingga antrian berisi [3, 2, 1]. Akhirnya, elemen 4 ditambahkan, menjadikan antrian [4, 3, 2, 1]. Proses ini menggambarkan cara kerja antrian yang mengikuti prinsip FIFO, dimana elemen baru selalu dimasukkan ke bagian belakang antrian.

Gambar 6.5 menggambarkan ilustrasi proses penghapusan elemen dalam Queue berbentuk list dengan menggunakan metode dequeue(). Proses dimulai dengan antrian yang berisi elemen [2, 3, 4] di langkah (a), di mana elemen pertama, yaitu angka 2, dihapus menggunakan metode dequeue(), yang menyebabkan antrian

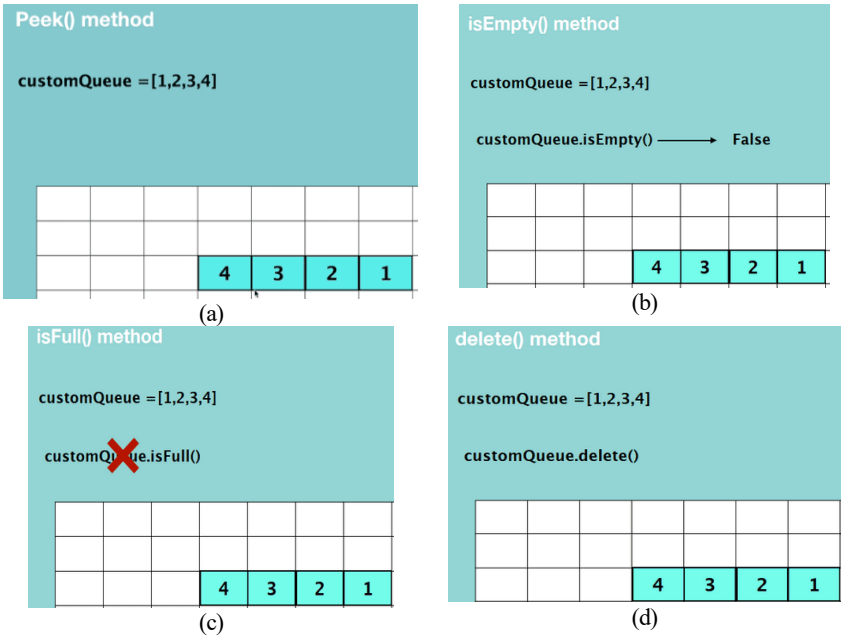


Gambar 6.5: Penghapusan Elemen dalam Queue Berbentuk List (Karimov, 2022)

berubah menjadi [3, 4] di langkah (b). Kemudian, di langkah (c), elemen berikutnya, yaitu angka 3, dihapus, sehingga antrian hanya menyisakan elemen [4]. Selanjutnya, di langkah (d), elemen 4 dihapus, menjadikan antrian kosong. Langkah (e) menunjukkan bahwa antrian telah kosong, dan jika ada upaya untuk menghapus

elemen lagi, seperti yang ditunjukkan di langkah (f), akan muncul pesan bahwa antrian kosong. Proses ini menggambarkan cara kerja queue berdasarkan prinsip FIFO, dimana elemen pertama yang masuk akan menjadi elemen pertama yang keluar.

Gambar 6.6 menunjukkan beberapa metode yang digunakan untuk memeriksa status elemen dalam Queue berbentuk list. Tahap (a), metode Peek() berfungsi untuk menampilkan elemen paling depan dalam antrian tanpa menghapusnya, yaitu angka 4. Tahap (b), metode isEmpty() digunakan untuk memeriksa apakah antrian kosong



Gambar 6.6: Status Elemen dalam Queue Berbentuk List (Karimov, 2022)

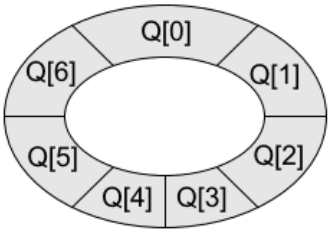
atau tidak. Hasil pemeriksaan menunjukkan bahwa antrian tidak kosong karena masih ada elemen di dalamnya, sehingga metode ini mengembalikan nilai False. Tahap (c), metode isFull() digunakan untuk memeriksa apakah antrian sudah penuh. Namun, meskipun ada ruang kosong di antrian, metode ini menunjukkan bahwa antrian belum penuh. Tahap (d), metode delete() digunakan untuk menghapus elemen pertama dalam antrian, yaitu angka 1, sehingga antrian terupdate menjadi [4, 3, 2]. Proses-proses ini menggambarkan status elemen dalam Queue dapat diperiksa dan dimanipulasi menggunakan berbagai metode.

Contoh program queue dapat dilihat di link https://colab.research.google.com/drive/10ssH1E3lHQM1PjXZQnWmloo7sUl51eNV?usp=sharing.

Ada beberapa **jenis antrian** yang sering digunakan dalam berbagai aplikasi dan sistem komputasi yaitu **circular queue, deque, priority queue, dan multiple queue** (Thareja, 2014).

6.2 Circular Queue

Circular queue adalah jenis antrian yang menghubungkan elemen terakhir kembali ke elemen pertama jika terdapat ruang kosong, gambar 6.7**.** Artinya, setelah mencapai batas kapasitas antrian, elemen baru dapat masuk ke bagian depan antrian, apabila ada elemen yang dihapus. Ini menghindari pemborosan ruang di array atau struktur antrian berbasis array biasa. Circular queue sangat berguna dalam aplikasi seperti pemrograman sistem atau jaringan, di mana aliran data terus menerus.



Gambar 6.7: Ilustrasi Circular Queue (Thareja, 2014)

Circular queues merupakan solusi untuk masalah overflow yang terjadi dalam linear queue, yang memiliki kelemahan dalam penggunaan ruang ketika elemen-elemen telah dikeluarkan dari antrian tetapi ruang di ujung depan antrian tidak dapat digunakan kembali. Linear queue memiliki operasi *enqueue* (penambahan elemen) hanya dapat dilakukan di bagian belakang (*REAR*), dan *dequeue* (penghapusan elemen) hanya dapat dilakukan di bagian depan (*FRONT*). Ketika antrian penuh, meskipun terdapat ruang kosong di depan setelah beberapa elemen dikeluarkan, penambahan elemen baru tidak dapat dilakukan karena kondisi REAR = MAX – 1, yang berarti posisi terakhir di antrian telah tercapai. Untuk mengatasi hal ini, salah satu solusi adalah dengan menggunakan circular queue. Dalam circular queue, struktur data ini memanfaatkan ruang kosong dengan cara membuat hubungan antara ujung belakang dan ujung depan antrian secara melingkar. Ketika elemen-elemen dihapus, ruang yang kosong dapat digunakan untuk menambah elemen baru. Misalnya, jika FRONT = 0 dan REAR = MAX – 1, dan terdapat ruang kosong sebelum FRONT, maka dapat memanfaatkan ruang tersebut dengan menyisipkan elemen di posisi yang telah tersedia.

```
Algoritma memasukkan elemen ke dalam circular queue:
Step 1: IF FRONT = and Rear = MAX - 1
    Write "OVERFLOW"
    Goto step 4
    [End OF IF]
Step 2: IF FRONT = -1 and REAR = -1
          SET FRONT = REAR =
        ELSE IF REAR = MAX - 1 and FRONT !=
          SET REAR =
        ELSE
          SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
```

Penerapan circular queue dilakukan dengan cara yang mirip dengan linear queue, tetapi dengan beberapa perbedaan dalam kode untuk melakukan penyisipan (*enqueue*) dan penghapusan (*dequeue*). Untuk **penyisipan**, ada tiga kondisi yang harus diperiksa:

1. Jika FRONT = 0 dan REAR = MAX – 1, maka antrian sudah penuh.
2. Jika REAR ≠ MAX – 1, maka REAR akan ditingkatkan dan elemen baru dapat dimasukkan di ujung belakang.
3. Jika FRONT ≠ 0 dan REAR = MAX – 1, maka antrian tidak penuh, dan REAR diatur ke 0, sehingga elemen baru dapat dimasukkan ke posisi tersebut.

Untuk **penghapusan**, terdapat tiga kondisi yang harus dipertimbangkan:

1. Jika FRONT = –1, maka antrian kosong dan kondisi underflow akan terjadi.
2. Jika FRONT = REAR, setelah elemen dihapus, antrian akan kosong, sehingga FRONT dan REAR diatur menjadi –1.
3. Jika FRONT = MAX – 1, setelah penghapusan, FRONT akan diatur kembali ke 0, karena antrian menggunakan struktur melingkar.

```
Algoritma penghapusan elemen ke dalam circular queue:
Step 1: IF FRONT = -1
    Write "ONDERFLOW"
    Goto step 4
    [End OF IF]
Step 2: ET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
            SET FRONT = REAR = -1
        ELSE
          IF FRONT = MAX -1
              SET FRONT =0
        ELSE
          SET FRONT = FRONT + 1
          [END of IF]
        [END OF IF]
Step 4: EXIT
```

Contoh perhitungan manual circular queue, menggunakan contoh 2.

Inisialisasi antrian (InitQueue):
Queue: []
Front = -1, Rear = -1

Langkah 1: Insert I
Queue: [I]
Front = 0, Rear = 0

Langkah 2: Insert N
Queue: [I, N]
Front = 0, Rear = 1

Langkah 3: Insert F
Queue: [I, N, F]
Front = 0, Rear = 2

Langkah 4: Insert O
Queue: [I, N, F, O]
Front = 0, Rear = 3

Langkah 5: Insert R
Queue: [I, N, F, O, R]
Front = 0, Rear = 4

Langkah 6: Insert M
Queue: [I, N, F, O, R, M]
Front = 0, Rear = 5

Langkah 7: Insert A
Queue: [I, N, F, O, R, M, A]
Front = 0, Rear = 6

Langkah 8: Insert T
Queue: [I, N, F, O, R, M, A, T]
Front = 0, Rear = 7

Langkah 9: Insert I
Queue: [I, N, F, O, R, M, A, T, I]
Front = 0, Rear = 8

Langkah 10: Insert K
Queue: [I, N, F, O, R, M, A, T, I, K]
Front = 0, Rear = 9

Langkah 11: Insert A
Queue: [I, N, F, O, R, M, A, T, I, K, A]
Front = 0, Rear = 10

Langkah 12: Delete
Queue: [N, F, O, R, M, A, T, I, K, A]
Front = 1, Rear = 10

Langkah 13: Delete
Queue: [F, O, R, M, A, T, I, K, A]
Front = 2, Rear = 10

Langkah 14: Insert M
Queue: [F, O, R, M, A, T, I, K, A, M]
Front = 2, Rear = 0

Langkah 15: Insert P
Queue: [F, O, R, M, A, T, I, K, A, M, P]
Front = 2, Rear = 1

Langkah 16: Delete
Queue: [O, R, M, A, T, I, K, A, M, P]
Front = 3, Rear = 1

Langkah 17: Delete
Queue: [R, M, A, T, I, K, A, M, P]
Front = 4, Rear = 1

Langkah 18: Delete
Queue: [M, A, T, I, K, A, M, P]
Front = 5, Rear = 1

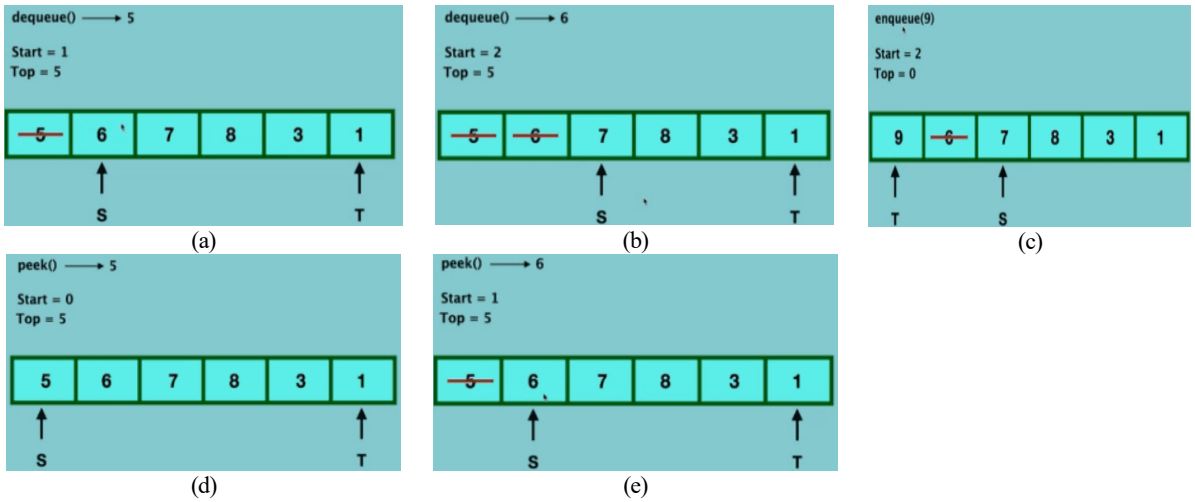
Langkah 19: Insert U
Queue: [M, A, T, I, K, A, M, P, U]
Front = 5, Rear = 2

Langkah 20: Delete
Queue: [A, T, I, K, A, M, P, U]
Front = 6, Rear = 2

Langkah 21: Insert S
Queue: [A, T, I, K, A, M, P, U, S]
Front = 6, Rear = 3

Langkah 22: Delete
Queue: [T, I, K, A, M, P, U, S]
Front = 7, Rear = 3

Gambar 6.8 menggambarkan langkah-langkah dalam proses pembuatan dan penambahan elemen ke dalam circular queue yang berbentuk list dengan ukuran maksimal 6. Langkah pertama adalah inisialisasi queue yang dimulai dengan kondisi kosong, yang ditunjukkan di Gambar (a), dengan Start = -1 dan Top = -1, menandakan tidak ada elemen di dalam queue. Gambar (b), proses enqueue(5) dilakukan, menambahkan elemen pertama yaitu angka 5 ke dalam queue. Dengan demikian, Start tetap di -1, tetapi Top berubah menjadi 0, menunjukkan posisi pertama elemen yang baru dimasukkan. Gambar (c), enqueue(6) dilakukan, memasukkan elemen kedua yaitu angka 6. Posisi Start tetap di

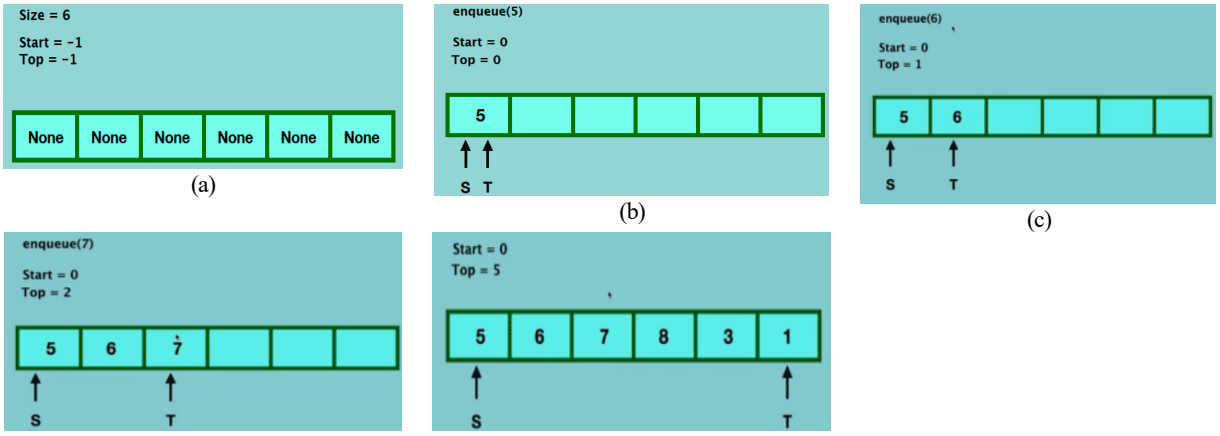


Gambar 6.8: Ilustrasi Operasi Dequeue dan Peek dalam Circular Queue Berbentuk List (Karimov, 2022)

0, sementara Top berpindah ke posisi 1, menandakan bahwa elemen kedua berhasil dimasukkan ke dalam antrian. Gambar (d), enqueue(7) dilakukan, menambahkan angka 7 ke dalam queue. Posisi Start tetap di 0, sementara Top bergerak ke posisi 2, menandakan bahwa antrian berisi tiga elemen, yaitu 5, 6, dan 7. Gambar (e), penambahan elemen berikutnya, yaitu angka 3. Elemen baru ini dimasukkan di posisi yang tersedia dalam queue, dan Top berpindah ke posisi 5.

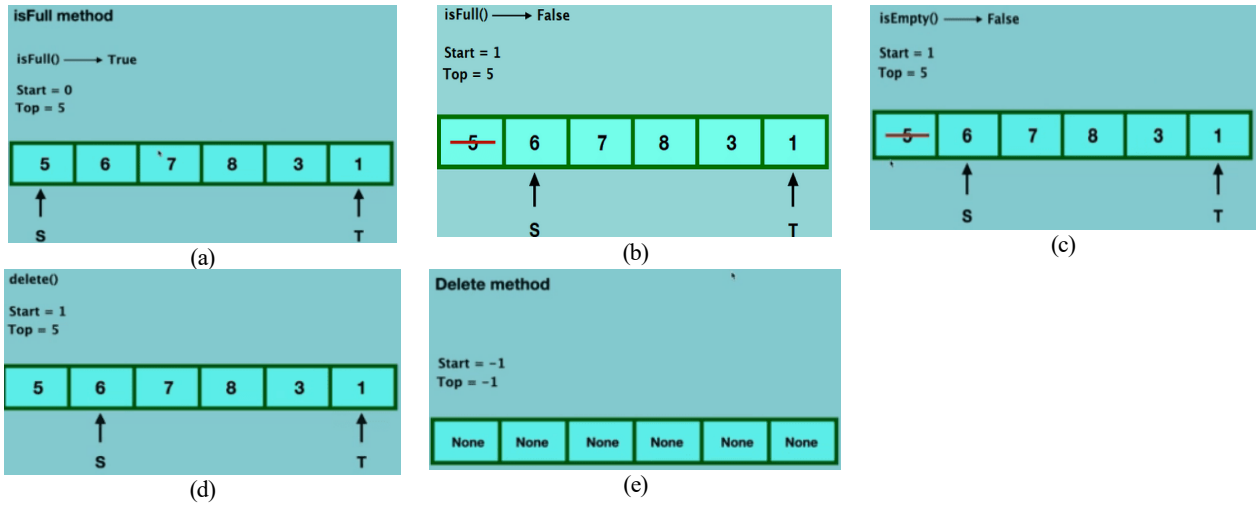
Gambar 6.9 menggambarkan operasi dequeue dan peek dalam circular queue. Gambar (a), operasi dequeue() dilakukan untuk menghapus elemen pertama antrian, yaitu angka 5. Setelah penghapusan, elemen yang tersisa dalam queue adalah [6, 7, 8, 3, 1], dengan Start bergerak maju ke posisi 1 dan Top tetap di posisi 5. Gambar (b), operasi dequeue() berikutnya dilakukan untuk menghapus elemen di posisi Start, yaitu angka 6. Setelah penghapusan, antrian berubah menjadi [7, 8, 3, 1], dan posisi Start berpindah ke 2, sementara Top tetap di posisi 5. Proses ini memperlihatkan bagaimana elemen-elemen dalam circular queue dipindahkan ke depan setelah dilakukan penghapusan. Gambar (c), operasi enqueue(9) dilakukan untuk menambahkan angka 9 ke dalam antrian. Setelah elemen baru dimasukkan, antrian berubah menjadi [9, 7, 8, 3, 1], dengan posisi Start tetap di 2 dan Top bergerak kembali ke 0, menunjukkan bahwa antrian sekarang mengelola data secara sirkular. Gambar (d), operasi peek() dilakukan untuk melihat elemen di posisi terdepan antrian tanpa menghapusnya. Elemen pertama dalam antrian, yaitu angka 5, ditampilkan, tetapi tidak ada perubahan di posisi Start atau Top. Gambar (e), operasi peek() dilakukan lagi, kali ini untuk melihat elemen terdepan setelah operasi dequeue sebelumnya. Elemen yang ditampilkan adalah angka 6, yang terletak di posisi pertama antrian saat ini.

Gambar 6.10 menggambarkan beberapa operasi dalam circular queue, yaitu operasi isFull, isEmpty, dan



Gambar 6.9: Pembuatan dan Penambahan Elemen dalam Circular Queue Berbentuk List (Karimov, 2022)

delete(). Gambar (a), operasi isFull() dilakukan untuk memeriksa apakah antrian penuh. Hasil dari operasi ini menunjukkan bahwa antrian penuh (isFull() = True), dengan antrian berisi [5, 6, 7, 8, 3, 1], dan posisi Start = 0 serta Top = 5, menandakan antrian telah mencapai kapasitas maksimum. Gambar (b), operasi isEmpty() dilakukan untuk memeriksa apakah antrian kosong. Hasilnya adalah False, yang berarti antrian tidak kosong karena masih ada elemen yang tersisa. Antrian saat ini berisi [6, 7, 8, 3, 1] dengan posisi Start di 1 dan Top di 5. Gambar (c), operasi isEmpty() kembali dilakukan setelah operasi sebelumnya. Hasil yang sama, yaitu False, menunjukkan bahwa antrian masih memiliki elemen. Proses ini mengilustrasikan bagaimana status antrian dapat diperiksa menggunakan metode isEmpty(). Gambar (d), operasi delete() dilakukan untuk menghapus elemen pertama dari antrian. Elemen yang dihapus adalah angka 5, sehingga antrian berubah menjadi [6, 7, 8, 3, 1]. Posisi Start bergeser ke 1, sementara Top tetap 5. Hal ini menunjukkan bahwa elemen pertama telah dikeluarkan dari antrian. Gambar (e), setelah operasi penghapusan, antrian terlihat kosong, dengan semua elemen dihapus dan Start serta Top diset ke -1, yang menandakan bahwa antrian telah benar-benar kosong.



Gambar 6.10: Ilustrasi Operasi IsFull, IsEmpty, dan Delete dalam Circular Queue Berbentuk List (Karimov, 2022)

Tabel 6.1 menunjukkan kompleksitas waktu dan ruang dari berbagai operasi Circular Queue. Create queue memiliki kompleksitas waktu O(1), yang berarti proses pembuatan antrian berlangsung dalam waktu konstan, terlepas dari ukuran antrian. Enqueue memiliki kompleksitas waktu O(1), menunjukkan bahwa waktu yang dibutuhkan untuk menambahkan elemen tidak tergantung jumlah elemen dalam antrian. Dequeue memiliki kompleksitas waktu O(1), yang berarti operasi ini dapat dilakukan dalam waktu konstan. Peek memerlukan waktu O(1), karena tidak ada pengubahan atau perhitungan yang memerlukan iterasi atau pemeriksaan lebih lanjut. isEmpty dilakukan dalam waktu konstan, O(1), yang berarti operasi ini dijalankan dalam waktu konstan. Pemeriksaan ini hanya memerlukan langkah sederhana, tanpa perlu melihat atau menghitung elemen lainnya. isFull memiliki kompleksitas waktu O(1). Pemeriksaan ini tidak memerlukan iterasi atau proses bergantung jumlah elemen dalam antrian, sehingga dapat dilakukan dalam waktu konstan. Delete entire queue memiliki kompleksitas waktu O(1), karena penghapusan seluruh elemen dapat dilakukan dalam satu langkah.

Tabel 6.1: Kompleksitas Waktu dan Ruang untuk Operasi Queue

Operasi	Kompleksitas Waktu	Kompleksitas Ruang
Create Queue	O(1)	O(n)
Enqueue	O(1)	O(1)
Dequeue	O(1)	O(1)
Peek	O(1)	O(1)
isEmpty	O(1)	O(1)
isFull	O(1)	O(1)
Delete Entire Queue	O(1)	O(1)

Create queue membutuhkan ruang O(n), di mana n adalah jumlah elemen yang dapat ditampung oleh antrian. Enqueue, dequeue, peek, isEmpty, isFull, delete entire queue memerlukan ruang O(1), karena tidak mengubah jumlah ruang yang dibutuhkan oleh antrian secara keseluruhan selama operasi. Hanya ruang untuk menyimpan antrian itu sendiri yang diperlukan, tidak ada alokasi ruang tambahan yang signifikan.

Contoh program circular queue terdapat di link https://colab.research.google.com/drive/1b6u5eD03eyQeL06KSZQ9o4K76pyMoZOh?usp=sharing.

6.3 Deque (Double-Ended Queue)

Deque (*double-ended queue*) adalah struktur data yang memungkinkan elemen-elemen untuk disisipkan atau dihapus dari kedua ujungnya, baik di depan (*head* atau *front*) maupun di belakang (*tail* atau *rear*). Dalam implementasinya, deque berfungsi mirip dengan linked list berjenis head-tail, karena elemen dapat ditambahkan

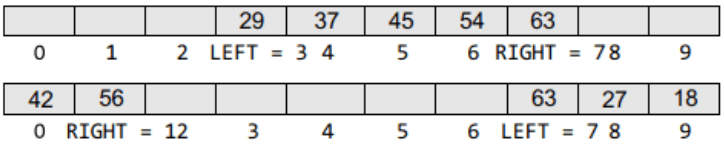
atau dihapus dari dua sisi antrian. Namun, dalam deque, tidak ada elemen yang dapat ditambahkan atau dihapus dari bagian tengah. Deque sering digunakan dalam implementasi algoritma seperti pencarian dalam graf atau penjadwalan dalam sistem operasi.

Untuk implementasinya dalam komputer, deque bisa menggunakan array melingkar (*circular array*) atau linked list dua arah melingkar (*circular doubly linked list*). Deque mempunyai dua pointer digunakan, yaitu *LEFT* dan *RIGHT*, yang masing-masing menunjuk ke ujung kiri dan kanan dari deque. Elemen dalam deque dapat diakses melalui pointer tersebut, dan karena menggunakan struktur melingkar, *Dequeue*[N-1] diikuti oleh *Dequeue*[0], artinya setelah mencapai ujung terakhir, sistem akan kembali ke awal.

Terdapat dua varian utama dalam *deque* yaitu:

1. *Input restricted deque* → penyisipan hanya dapat dilakukan di salah satu ujung, sedangkan penghapusan dapat dilakukan di kedua ujung. Hal ini membatasi sisi mana yang bisa menerima elemen baru.
2. *Output restricted deque* → penghapusan hanya dapat dilakukan di satu ujung saja, sementara penyisipan bisa dilakukan di kedua ujung deque. Penambahan elemen dalam queue dilakukan di satu ujung, sedangkan ujung lainnya untuk menghapus elemen.

Gambar 6.11 menggambarkan dua contoh *deque* (*double-ended queue*) menggunakan array untuk



Gambar 6.11: *Deque* (*Double-Ended Queue*) (Thareja, 2014)

mewakili struktur data antrian. Dalam deque, elemen dapat ditambahkan atau dihapus baik dari ujung depan (*LEFT*) maupun ujung belakang (*RIGHT*), yang membedakannya dari antrian linier biasa. Gambar pertama, terlihat bahwa elemen-elemen 29, 37, 45, 54, dan 63 berada di antrian dengan posisi LEFT berada dalam indeks 3 dan RIGHT dengan indeks 8. Hal ini menunjukkan posisi dua pointer yang mengarah ke ujung depan dan belakang deque. Gambar kedua, terlihat perubahan yang terjadi setelah operasi dequeue yang menghapus elemen dari posisi LEFT dan RIGHT. Misalnya, elemen 42 dan 56 berada di posisi LEFT dan elemen lainnya menempati posisi yang berbeda sesuai dengan operasi yang dilakukan.

Contoh soal *double-ended queue*. Diberikan deque kosong dengan kapasitas 7. Lakukan serangkaian operasi berikut menggunakan nama-nama bunga Indonesia:

a. InsertFirst: Menambahkan bunga "Melati".
b. InsertLast: Menambahkan bunga "Anggrek".
c. InsertFirst: Menambahkan bunga "Cempaka".
d. InsertLast: Menambahkan bunga "Bunga Wijaya Kusuma".
e. DeleteFirst: Menghapus bunga dari depan.
f. InsertLast: Menambahkan bunga "Teratai".
g. DeleteLast: Menghapus bunga dari belakang.
h. InsertFirst: Menambahkan bunga "Kamboja".
i. InsertLast: Menambahkan bunga "Bougenville".

Tampilkan keadaan deque setelah setiap operasi dengan posisi LEFT dan RIGHT, serta elemen yang ada di dalam *deque*. Penyelesaian:

```
Deque (Kosong)
Queue: []
LEFT = -1, RIGHT = -1

1. InsertFirst: Menambahkan bunga "Melati".
Queue: [Melati]
LEFT = 0, RIGHT = 0

2. InsertLast: Menambahkan bunga "Anggrek".
Queue: [Melati, Anggrek]
LEFT = 0, RIGHT = 1

3. InsertFirst: Menambahkan bunga "Cempaka".
Queue: [Cempaka, Melati, Anggrek]
LEFT = 0, RIGHT = 2

4. InsertLast: Menambahkan bunga "Bunga Wijaya Kusuma".
Queue: [Cempaka, Melati, Anggrek, Bunga Wijaya Kusuma]
LEFT = 0, RIGHT = 3
```

```
5. DeleteFirst: Menghapus bunga dari depan (Cempaka).
Queue: [Melati, Anggrek, Bunga Wijaya Kusuma]
LEFT = 1, RIGHT = 3

6. InsertLast: Menambahkan bunga "Teratai".
Queue: [Melati, Anggrek, Bunga Wijaya Kusuma, Teratai]
LEFT = 1, RIGHT = 4

7. DeleteLast: Menghapus bunga dari belakang (Teratai).
Queue: [Melati, Anggrek, Bunga Wijaya Kusuma]
LEFT = 1, RIGHT = 3

8. InsertFirst: Menambahkan bunga "Kamboja".
Queue: [Kamboja, Melati, Anggrek, Bunga Wijaya Kusuma]
LEFT = 0, RIGHT = 3
9. InsertLast: Menambahkan bunga "Bougenville".
Queue: [Kamboja, Melati, Anggrek, Bunga Wijaya Kusuma, Bougenville]
LEFT = 0, RIGHT = 4
```

Soal ini, LEFT selalu menunjuk posisi depan antrian dan RIGHT menunjuk posisi belakang antrian. Setelah melakukan operasi insertFirst dan insertLast, queue dapat dimanipulasi posisi depan dan belakang deque untuk menambahkan atau menghapus elemen dari kedua ujung. Saat operasi deleteFirst atau deleteLast dilakukan, elemen akan dihapus dari ujung yang sesuai dan posisi LEFT atau RIGHT akan disesuaikan.

Contoh program *double-ended queue* di link https://colab.research.google.com/drive/1930LBA-OK_HjtuISzjjfhiUfrSNBKmM4?usp=sharing.

6.4 Priority Queue

Queue priority adalah struktur data yang setiap elemen diberi prioritas tertentu. Prioritas elemen ini digunakan untuk menentukan urutan pemrosesan elemen-elemen tersebut. Secara umum, aturan pemrosesan elemen dalam antrian prioritas adalah:

- Elemen dengan prioritas lebih tinggi akan diproses terlebih dahulu dibandingkan dengan elemen dengan prioritas lebih rendah.
- Jika terdapat dua elemen dengan prioritas yang sama, maka elemen-elemen tersebut akan diproses berdasarkan urutan kedatangan (*first-come-first-served* / FCFS).

Queue priority merupakan antrian yang dimodifikasi, ketika elemen harus dihapus dari antrian, elemen dengan prioritas tertinggi akan diambil terlebih dahulu. Prioritas suatu elemen dapat ditentukan berdasarkan berbagai faktor. Antrian prioritas banyak digunakan dalam sistem operasi untuk mengeksekusi proses dengan prioritas tertinggi terlebih dahulu. Prioritas proses biasanya ditentukan berdasarkan waktu CPU yang dibutuhkan untuk menyelesaikan eksekusinya. Sebagai contoh, jika terdapat tiga proses, di mana proses pertama membutuhkan 5 ns untuk selesai, proses kedua membutuhkan 4 ns, dan proses ketiga membutuhkan 7 ns, maka proses kedua akan memiliki prioritas tertinggi dan akan dieksekusi terlebih dahulu. Namun, waktu CPU bukan satu-satunya faktor yang menentukan prioritas, karena ada faktor lain yang mempengaruhi, seperti pentingnya satu proses dibandingkan dengan yang lainnya. Misalnya, jika ada dua proses yang harus dijalankan secara bersamaan, di mana satu proses terkait dengan pemesanan pesanan online dan proses lainnya terkait dengan pencetakan detail stok, jelas bahwa pemesanan online lebih penting dan harus dieksekusi terlebih dahulu. Selain itu, *queue* jenis ini banyak digunakan dalam sistem yang membutuhkan pemrosesan tugas berdasarkan urgensi, seperti dalam algoritma pemrograman seperti Dijkstra atau manajemen antrean layanan pelanggan dengan prioritas.

⌘ Implementasi *Priority Queue*

Terdapat dua cara untuk mengimplementasikan *queue priority* yaitu:

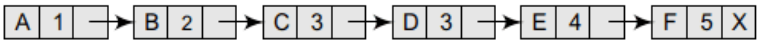
1. Menggunakan daftar yang sudah diurutkan untuk menyimpan elemen-elemen, sehingga ketika elemen perlu diambil, antrian tidak perlu mencari elemen dengan prioritas tertinggi.
   Penggunaan daftar yang terurut memerlukan waktu O(n) untuk menyisipkan elemen, tetapi hanya membutuhkan waktu O(1) untuk menghapus elemen.
2. Menggunakan daftar yang tidak diurutkan, setiap penyisipan dilakukan di akhir daftar, dan setiap kali elemen harus dihapus, elemen dengan prioritas tertinggi akan dicari dan dihapus. Sebaliknya, daftar yang tidak terurut membutuhkan waktu O(1) untuk menyisipkan elemen, tetapi membutuhkan waktu O(n) untuk menghapus elemen.

Secara praktis, kedua teknik ini tidak efisien, dan kedua pendekatan memerlukan waktu sekitar O(log n) atau lebih sedikit.

⌘ Representasi Linked List dalam *Priority Queue*

Dalam memori komputer, antrian prioritas dapat diwakili menggunakan array atau linked list. Ketika *queue priority* diimplementasikan menggunakan linked list, setiap node dalam daftar akan memiliki tiga bagian, yaitu **(a) bagian informasi atau data, (b) nomor prioritas elemen,** dan **(c) alamat elemen berikutnya**. Jika menggunakan linked list yang terurut, elemen dengan prioritas lebih tinggi akan mendahului elemen dengan prioritas lebih rendah. Semakin rendah nomor prioritas, semakin tinggi prioritasnya. Sebagai contoh, jika terdapat dua elemen A dan B, A memiliki nomor prioritas 1 dan B memiliki nomor prioritas 5, maka A akan diproses terlebih dahulu karena prioritasnya lebih tinggi daripada B.

Priority queue dalam Gambar 6.12 adalah antrian prioritas terurut yang memiliki enam elemen. Dari antrian tersebut, tidak dapat mengetahui apakah A dimasukkan sebelum E atau apakah E bergabung

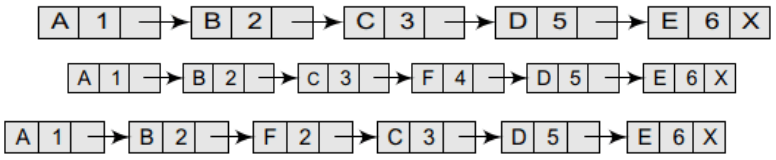


Gambar 6.12: *Priority Queue* (Thareja, 2014)

dengan antrian sebelum A, karena daftar tersebut tidak diurutkan berdasarkan prinsip FCFS (*first-come-first-served*). Di sini, elemen dengan prioritas lebih tinggi akan mendahului elemen dengan prioritas lebih rendah. Namun, dapat dipastikan bahwa C dimasukkan sebelum D, karena ketika dua elemen memiliki prioritas yang sama, elemen-elemen tersebut diatur dan diproses berdasarkan prinsip FCFS.

Penyisipan

Ketika elemen baru perlu disisipkan dalam *priority queue*, maka harus menjelajahi seluruh daftar hingga menemukan node yang memiliki prioritas lebih rendah daripada elemen baru. Node baru tersebut akan disisipkan sebelum node dengan prioritas lebih rendah. Namun, jika ada elemen yang memiliki prioritas yang sama dengan elemen baru, elemen baru akan disisipkan setelah elemen tersebut. Sebagai contoh, jika menyisipkan elemen baru dengan data = F dan nomor prioritas = 4, maka elemen tersebut akan disisipkan sebelum D yang memiliki nomor prioritas 5, yang lebih rendah dari prioritas elemen baru. Namun, jika memiliki elemen baru dengan data = F dan nomor prioritas = 2, maka elemen tersebut akan disisipkan setelah B, karena kedua elemen tersebut memiliki prioritas yang sama, tetapi penyisipannya dilakukan berdasarkan prinsip FCFS, seperti yang ditunjukkan dalam Gambar 6.13.



Gambar 6.13: *Priority Queue* dengan Penyisipan Node (Thareja, 2014)

Penghapusan

Proses penghapusan dalam hal ini sangat sederhana. Node pertama dari daftar akan dihapus, dan data dari node tersebut akan diproses terlebih dahulu.

✂ Representasi Array dari Antrian Prioritas

Ketika array digunakan untuk mengimplementasikan *priority queue*, maka akan dibuat antrian terpisah untuk setiap nomor prioritas. Setiap antrian ini akan diimplementasikan menggunakan array melingkar atau circular queue. Setiap antrian individu akan memiliki pointer FRONT dan REAR masing-masing. Untuk tujuan ini, digunakan array dua dimensi, di mana setiap antrian akan diberikan ruang yang sama. Gambar 6.14 merupakan representasi dua dimensi dari *priority queue*. Nilai FRONT[K] dan REAR[K] berisi nilai depan dan belakang dari baris ke-K, di mana K adalah nomor prioritas. Perlu dicatat bahwa dalam representasi ini, mengasumsikan bahwa indeks baris dan kolom dimulai dari 1, bukan 0.

FRONT	REAR
3	3
1	3
4	5
4	1

	1	2	3	4	5
1			A		
2	B	C	D		
3				E	F
4	I			G	H

Gambar 6.14: Matriks *Priority Queue* (Thareja, 2014)

Matriks dalam Gambar 6.14 terdiri dari dua bagian yaitu bagian kiri (tabel front dan rear) dan bagian kanan (matriks antrian).

Bagian kiri: tabel front dan rear

Tabel ini menunjukkan dua kolom yaitu FRONT dan REAR, yang berfungsi untuk menandakan posisi pertama (*front*) dan terakhir (*rear*) elemen dalam antrian di setiap tingkat prioritas.

- Baris 1 (Prioritas 1):

FRONT = 3 dan REAR = 3, yang berarti elemen pertama dan terakhir dalam queue dengan prioritas 1 berada di indeks 3. Ini menunjukkan bahwa antrian prioritas 1 hanya memiliki satu elemen, yaitu "A", yang berada dalam indeks ke-3.

- Baris 2 (Prioritas 2):

FRONT = 1 dan REAR = 3, berarti elemen pertama dalam prioritas 2 terletak di indeks 1, dan elemen terakhir berada di indeks 3. Ini menunjukkan bahwa elemen-elemen "B", "C", dan "D" berada dalam posisi ini, yaitu [B, C, D].

- Baris 3 (Prioritas 3):

FRONT = 4 dan REAR = 5, berarti elemen pertama di prioritas 3 terletak di indeks 4, dan elemen terakhir berada di indeks 5. Elemen-elemen "E" dan "F" terletak dalam posisi ini, yaitu [E, F].

- Baris 4 (Prioritas 4):

FRONT = 4 dan REAR = 1, berarti elemen pertama dalam prioritas 4 berada di indeks 4, dan elemen terakhir berada di indeks 1. Hal ini menunjukkan bahwa elemen "I", "G", dan "H" terletak dalam posisi ini, yaitu [I, G, H].

Bagian Kanan: Matriks Antrian

Matriks ini berisi elemen-elemen yang telah dimasukkan ke dalam antrian berdasarkan prioritas masing-masing. Bagian kanan, angka-angka dalam matriks ini menunjukkan elemen-elemen yang disusun berdasarkan prioritas yang berbeda.

- Baris 1 (Prioritas 1):

Elemen yang berada di baris ini adalah "A" yang terletak di posisi 3. Baris ini menunjukkan bahwa hanya ada satu elemen dengan prioritas tertinggi (prioritas 1).

- Baris 2 (Prioritas 2):

Elemen yang berada di baris ini adalah "B", "C", dan "D" yang terletak di posisi [1, 2, 3]. Ini menunjukkan bahwa elemen-elemen ini memiliki prioritas yang lebih rendah daripada elemen di baris 1, namun masih diproses lebih dulu dari elemen dengan prioritas lebih rendah.

- Baris 3 (Prioritas 3):

Elemen yang berada di baris ini adalah "E" dan "F" yang terletak di posisi [4, 5]. Elemen-elemen ini memiliki prioritas yang lebih rendah daripada baris 1 dan 2.

- Baris 4 (Prioritas 4):

Elemen yang berada di baris ini adalah "I", "G", dan "H" yang terletak di posisi [4, 5, 6]. Baris ini menunjukkan bahwa elemen-elemen ini memiliki prioritas paling rendah.

Penyisipan

Untuk menyisipkan elemen baru dengan prioritas K ke dalam antrian prioritas, elemen tersebut ditambahkan di ujung belakang dari baris K, di mana K adalah nomor baris sekaligus nomor prioritas elemen tersebut. Sebagai contoh, sisipkan elemen R dengan nomor prioritas 3, maka antrian prioritas akan diperbarui seperti yang ditunjukkan dalam Gambar 6.15.

FRONT	REAR
3	3
1	3
4	1
4	1

	1	2	3	4	5
1			A		
2	B	C	D		
3	R			E	F
4	I			G	H

Gambar 6.15: Penyisipan Elemen dalam Matriks *Priority Queue* (Thareja, 2014)

Penghapusan

Untuk menghapus elemen, cari antrian pertama yang tidak kosong dan kemudian memproses elemen di bagian depan antrian tersebut. Dalam antrian prioritas ini, antrian pertama yang tidak kosong adalah antrian dengan nomor prioritas 1, dan elemen di depannya adalah A, sehingga A akan dihapus dan diproses terlebih dahulu. Dalam istilah teknis, cari elemen dengan nilai K terkecil, sehingga FRONT[K] != NULL, yang menandakan bahwa antrian tersebut tidak kosong dan siap diproses.

Contoh latihan. Diberikan *priority queue* dengan 3 prioritas yaitu prioritas tinggi → singa, harimau. prioritas sedang → kucing, gajah. prioritas rendah: ayam, bebek. Urutkan elemen-elemen berikut menggunakan priority queue. Berikut prosedur yang harus dijalankan:

Insert: menambahkan elemen dengan prioritas tertentu ke dalam priority queue.

Dequeue: menghapus elemen dengan prioritas tertinggi (front).

Peek: melihat elemen yang berada di depan antrian tanpa menghapusnya.

Penyelesaian:

Langkah 1: Inisialisasi antrian kosong
Queue: []
Front: -1
Rear: -1

Langkah 2: Insert Singa (Prioritas Tinggi)
Queue: [Singa]
Front: 0
Rear: 0

Langkah 3: Insert Kucing (Prioritas Sedang)
Queue: [Singa, Kucing]
Front: 0
Rear: 1

Langkah 4: Insert Ayam (Prioritas Rendah)
Queue: [Singa, Kucing, Ayam]
Front: 0
Rear: 2

Langkah 5: Insert Harimau (Prioritas Tinggi)
Queue: [Singa, Harimau, Kucing, Ayam]
Front: 0
Rear: 3

Langkah 6: Dequeue (Hapus elemen dengan prioritas tertinggi, Singa)
Queue: [Harimau, Kucing, Ayam]
Front: 1
Rear: 3

Langkah 7: Insert Gajah (Prioritas Sedang)
Queue: [Harimau, Kucing, Ayam, Gajah]
Front: 1
Rear: 4

Langkah 8: Dequeue (Hapus elemen dengan prioritas tertinggi berikutnya, Harimau)
Queue: [Kucing, Ayam, Gajah]
Front: 2
Rear: 4

Langkah 9: Peek (Melihat elemen yang berada di depan antrian, yaitu Kucing)
Queue: [Kucing, Ayam, Gajah]
Front: 2
Rear: 4

Langkah 10: Insert Bebek (Prioritas Rendah)
Queue: [Kucing, Ayam, Gajah, Bebek]
Front: 2
Rear: 5

Langkah 11: Dequeue (Hapus elemen dengan prioritas tertinggi berikutnya, Kucing)
Queue: [Ayam, Gajah, Bebek]
Front: 3
Rear: 5

Langkah 12: Dequeue (Hapus elemen dengan prioritas tertinggi, Ayam)
Queue: [Gajah, Bebek]
Front: 4
Rear: 5

Langkah 13: Insert Ular (Prioritas Tinggi)
Queue: [Ular, Gajah, Bebek]
Front: 4
Rear: 6

Langkah 14: Dequeue (Hapus elemen dengan prioritas tertinggi, Gajah)
Queue: [Bebek, Ular]
Front: 5
Rear: 6

Langkah 15: Insert Sapi (Prioritas Sedang)
Queue: [Bebek, Ular, Sapi]
Front: 5
Rear: 7

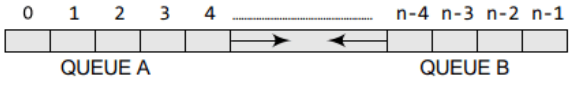
Langkah 16: Dequeue (Hapus elemen dengan prioritas tertinggi, Bebek)
Queue: [Ular, Sapi]
Front: 6
Rear: 7

Contoh *priority queue* di atas menunjukkan bagaimana elemen-elemen diproses berdasarkan prioritas masing-masing. Elemen dengan prioritas lebih tinggi selalu dihapus terlebih dahulu. Singa dan harimau memiliki prioritas tertinggi, disusul oleh kucing dan gajah yang memiliki prioritas sedang, dan terakhir ayam dan bebek yang memiliki prioritas lebih rendah. Di setiap insert dan dequeue, posisi *front* dan *rear* berubah sesuai dengan antrian yang terjadi, dengan *front* selalu menunjuk ke elemen yang sedang diproses (yang memiliki prioritas tertinggi untuk dequeue). Berikut contoh program *priority queue* dapat dilihat di link https://colab.research.google.com/drive/17GkK5OnAKKGeBHTwFxRmW5NO92xxwfi9?usp=sharing.

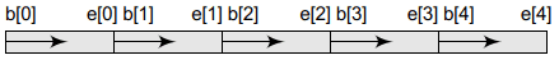
6.5 Multiple Queue

Multiple Queues adalah suatu konsep dalam implementasi antrian menggunakan array, di mana ukuran array harus diketahui terlebih dahulu. Jika ruang yang dialokasikan untuk antrian terlalu kecil, maka sering kali kondisi overflow akan terjadi. Untuk mengatasi masalah ini, kode perlu dimodifikasi agar dapat mengalokasikan lebih banyak ruang array. Namun, jika mengalokasikan ruang yang terlalu besar untuk antrian, hal ini akan menyebabkan pemborosan memori yang signifikan, terutama ketika frekuensi overflow sangat rendah.

Solusi yang lebih baik untuk masalah ini adalah dengan memiliki beberapa antrian dalam satu array dengan ukuran yang cukup. Seperti yang ditunjukkan dalam Gambar 6.16 (a), sebuah array QUEUE[n] digunakan untuk mewakili dua antrian, yaitu QUEUE A dan QUEUE B. Nilai n yang digunakan array tersebut akan memastikan bahwa ukuran gabungan kedua antrian tersebut tidak melebihi n. Saat operasi antrian-antrian ini, perlu dicatat bahwa QUEUE A akan berkembang dari kiri ke kanan, sedangkan QUEUE B akan berkembang dari kanan ke kiri secara bersamaan. Lebih lanjut, konsep ini dapat diperluas untuk lebih dari dua antrian. Sebagai contoh, sebuah array dapat digunakan untuk merepresentasikan n antrian dalam array yang sama. Dengan cara ini, setiap QUEUE i akan dialokasikan ruang yang sama, yang dibatasi oleh indeks b[i] dan e[i], seperti dalam Gambar 6.16 (b). Dengan pendekatan ini, dapat mengelola banyak antrian dalam satu array dengan lebih efisien, menghindari pemborosan ruang memori, dan mengurangi potensi overflow yang terjadi.



(a)



(b)

Gambar 6.16: Multiple Queue (Thareja, 2014)

Berikut contoh program *multiple queue* dapat dilihat di link https://colab.research.google.com/drive/1ZqBYIpJqB0E9e7s0_xgSOxSsW3NEhnZ5?usp=sharing.

Latihan:
1. Tulis sebuah program queue yang menggunakan data bunga dengan kapasitas maksimal 10 elemen. Program tersebut harus mencakup semua operasi yang ada dalam queue.

2. Buat program circular queue, deque, priority queue, dan multiple queue. Masing-masing program harus mencakup semua operasi yang ada dalam queue.

Latihan:
3. Misal maxqueue = 7
Ikuti perintah-perintah berikut dan ilustrasikan hasilnya (Linier Queue)
Initqueue
Insert L
Insert A
Insert T
Insert I
Insert H
Insert A
Insert N
Delete
Delete
Delete
Delete
Delete
Delete
Insert I
Insert H
Delete
Insert D
Insert F
Delete
Sebutkan posisi front dan rear disetiap pengerjaan

4. dari soal no 3, kerjakan menggunakan circular queue

BAB 7
Stack (Tumpukan)

Stack adalah struktur data yang krusial dan menyimpan elemen-elemen secara teratur dan bertumpuk. Konsep stack dapat dijelaskan melalui sebuah analogi yang mudah dipahami. Sebagai contoh, stack dapat disamakan dengan tumpukan piring atau buku di dapur atau rak buku. Piring atau buku ditempatkan dalam tumpukan, dan ketika diperlukan, piring atau buku diambil dari bagian atas tumpukan. Piring atau buku terakhir yang ditambahkan ke dalam tumpukan akan menjadi piring atau buku pertama yang diambil (Agarwal, 2022). Ilustrasi stack dapat dilihat di Gambar 7.1. Konsep stack dapat diibaratkan seperti tumpukan piring, buku, atau gelang yang dikenakan di tangan seseorang. Barang pertama yang dimasukkan ke dalam tumpukan akan berada di posisi paling bawah, sedangkan barang yang ditambahkan terakhir akan berada di bagian paling atas. Prinsip kerja stack mengikuti aturan bahwa barang yang berada di posisi teratas (barang terakhir yang dimasukkan) akan dikeluarkan terlebih dahulu, sedangkan barang yang berada di posisi terbawah (barang pertama yang dimasukkan) akan dikeluarkan terakhir.

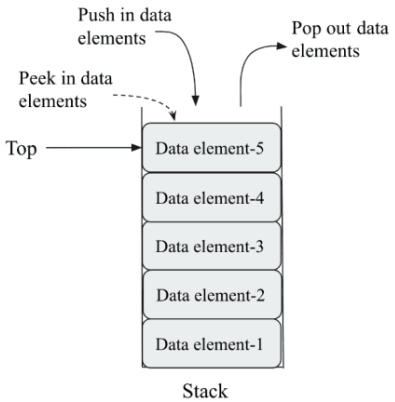
Gambar 7.1: Analogi Stack (Karimov, 2022)

Dalam konteks ini, stack adalah struktur data yang menyimpan elemen-elemen dalam urutan tertentu, mirip dengan struktur data array dan linked list, namun dengan beberapa batasan yang lebih spesifik, yaitu:

- Elemen dalam stack hanya dapat ditambahkan di ujung tumpukan melalui operasi push.
- Elemen dalam stack hanya dapat dihapus dari ujung tumpukan tersebut melalui operasi pop.
- Hanya elemen yang berada di bagian atas stack yang dapat dibaca menggunakan operasi peek.

Stack memungkinkan penyimpanan dan pembacaan data hanya dari satu sisi, di mana elemen yang terakhir dimasukkan adalah yang pertama diambil. Oleh karena itu, stack mengimplementasikan prinsip *Last-In-First-Out* (LIFO), yang dikenal sebagai *Last-In-Last-Out* (LILO) dalam beberapa kasus.

Operasi utama yang dilakukan dalam stack adalah **push** dan **pop**. **Ketika elemen ditambahkan ke atas stack, disebut operasi push**, sedangkan ketika **elemen diambil atau dihapus dari bagian atas stack**, disebut **operasi pop**. Operasi peek digunakan untuk melihat elemen di bagian atas stack tanpa menghapusnya. Semua



Gambar 7.2: Demonstrasi Operasi Push, Pop, dan Peek dalam Stack (Agarwal, 2022)

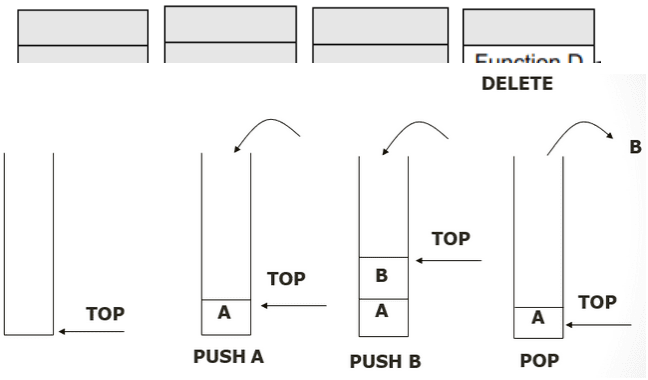
operasi dalam stack dilakukan menggunakan pointer yang disebut **top**. Semua operasi ini ditunjukkan dalam diagram yang terlampir. Seluruh operasi tersebut dijelaskan di Gambar 7.2.

Contoh pertama penambahan dan penghapusan stack disajikan di Tabel 7.1.

Tabel 7.1: Contoh Pertama Penambahan dan PenghapusanStack

Operasi Stack	Ukuran	Konten	Hasil Operasi
stack()	0	[]	Objek stack dibuat, yang kosong.
push "Semeru"	1	['Semeru']	Satu gunung Semeru ditambahkan ke dalam stack.
push "Rinjani"	2	['Semeru', 'Rinjani']	Satu lagi gunung Rinjani ditambahkan ke dalam stack.
push "Bromo"	3	['Semeru', 'Rinjani', 'Bromo']	Gunung Bromo ditambahkan ke dalam stack.
push "Krakatau"	4	['Semeru', 'Rinjani', 'Bromo', 'Krakatau']	Gunung Krakatau ditambahkan ke dalam stack.
peek()	4	['Semeru', 'Rinjani', 'Bromo', 'Krakatau']	Gunung yang ada di atas stack, yaitu Krakatau.
pop()	3	['Semeru', 'Rinjani', 'Bromo']	Gunung Krakatau dikeluarkan (gunung ini ditambahkan terakhir, jadi dihapus pertama)
pop()	2	['Semeru', 'Rinjani']	Gunung Bromo dikeluarkan (gunung ini ditambahkan terakhir, jadi dihapus pertama)
pop()	1	['Semeru']	Gunung Rinjani dikeluarkan (gunung ini ditambahkan terakhir, jadi dihapus pertama)
pop()	0	[]	Gunung Semeru dikeluarkan (ini adalah gunung pertama yang ditambahkan, jadi dikembalikan terakhir)

Gambar 7.3 menggambarkan demonstrasi dari operasi push, pop, dan peek dalam struktur data stack. Setiap tumpukan merepresentasikan urutan eksekusi fungsi yang terjadi dalam sistem stack. Proses dimulai dengan penambahan elemen pertama, yaitu Function A, yang kemudian diikuti dengan operasi push untuk menambahkan Function B dan Function C ke dalam tumpukan. Setelah itu, operasi peek digunakan untuk memeriksa elemen yang berada di bagian atas stack tanpa menghapusnya, dalam hal ini adalah Function D. Selanjutnya, operasi pop dilakukan untuk menghapus elemen yang berada di atas stack (yaitu, Function D), sehingga kontrol sistem kembali ke Function C, Function B dan Function A, sesuai dengan urutan yang ada dalam tumpukan.



Gambar 7.4: Contoh Ketiga Penambahan dan Penghapusan Stack

Gambar 7.4 menggambarkan contoh ketiga operasi penambahan dan penghapusan elemen dalam struktur data stack. Gambar pertama, dilakukan operasi push A, di mana elemen "A" dimasukkan ke dalam stack. Proses ini mengubah keadaan stack dengan "A" sebagai elemen teratas, yang ditandai dengan posisi TOP dalam diagram. Kemudian, operasi push B dilakukan, di mana elemen "B" ditambahkan ke dalam stack. Setelah operasi ini, elemen "B" berada di posisi teratas, dan "A" berada di bawahnya, dengan TOP yang menunjukkan elemen paling atas saat ini. Selanjutnya, dilakukan operasi pop untuk menghapus elemen yang berada di posisi teratas stack. Dalam hal ini, "B" dihapus dari stack, dan elemen "A" kembali menjadi elemen teratas. Proses pop ini

mengilustrasikan bagaimana stack bekerja berdasarkan prinsip LIFO, di mana elemen terakhir yang dimasukkan, yaitu "B", adalah yang pertama dikeluarkan.

Penggunaan Stack (Use)

Stack ideal digunakan ketika fungsionalitas LIFO dibutuhkan. Artinya, elemen yang terakhir dimasukkan ke dalam stack adalah elemen pertama yang akan dikeluarkan. Fungsionalitas ini sangat berguna dalam situasi-situasi seperti sistem undo-redo, pemrosesan rekursif, atau dalam algoritma backtracking, di mana elemen terakhir harus diproses terlebih dahulu. Selain itu, stack memiliki keunggulan dalam mengurangi risiko kerusakan data, karena data hanya dimanipulasi di satu sisi, yaitu puncak stack. Ini memberikan kontrol yang lebih baik terhadap integritas data.

Hindari Penggunaan Stack (Avoid)

Sebaliknya, penggunaan stack harus dihindari dalam situasi yang membutuhkan akses acak ke elemen-elemen di dalamnya. Karena stack hanya memungkinkan pengaksesan data dari puncak tumpukan, akses ke elemen lainnya tidak memungkinkan tanpa terlebih dahulu menghapus elemen-elemen yang berada di atasnya. Hal ini membatasi fleksibilitas stack dalam aplikasi yang membutuhkan akses langsung ke elemen-elemen tertentu di dalam tumpukan. Oleh karena itu, stack tidak cocok digunakan ketika struktur data yang memungkinkan akses acak lebih diperlukan.

7.1 Operasi-Operasi Stack

Beberapa operasi dalam stack yaitu:

- Create Stack

```
Algoritma create stack:
Step 1: SET TOP = -1
Step 2: SET MAX = MAX_SIZE
Step 3: SET STACK[MAX] (Inisialisasi stack kosong)
Step 4: END
```

Membuat sebuah stack baru yang kosong dan siap digunakan. Dalam proses ini, nilai TOP diatur ke -1, yang menunjukkan bahwa stack belum berisi elemen. Fungsi ini juga menetapkan ukuran maksimum stack yang ditentukan oleh MAX_SIZE, yang membatasi jumlah elemen yang dapat dimasukkan ke dalam stack. Stack ini siap untuk operasi lebih lanjut seperti push dan pop, seperti yang terlihat dalam algoritma create stack.

- Push

```
Algoritma push:
Step 1: IF TOP = MAX-1
   PRINT "Overflow"
   Goto Step 4
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

Menambahkan elemen baru ke dalam stack. Operasi push menempatkan elemen di posisi TOP dan kemudian meningkatkan nilai TOP untuk menandakan bahwa ada elemen baru yang ditambahkan. Fungsi ini bekerja berdasarkan prinsip LIFO, di mana elemen yang terakhir dimasukkan akan menjadi yang pertama dikeluarkan. Sebelum menambahkan elemen, fungsi ini memeriksa apakah stack penuh (overflow). Jika penuh, operasi push akan gagal, seperti yang terlihat dalam algoritma push.

- Pop

```
Algoritma pop:
Step 1: IF TOP = -1
   PRINT "Underflow"
   Goto Step 4
Step 2: SET REMOVED = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

Menghapus elemen yang ada di posisi TOP stack dan mengurangi nilai TOP. Operasi pop mengeluarkan elemen pertama yang masuk (yang terakhir dimasukkan), sesuai dengan prinsip LIFO. Sebelum menghapus, operasi ini memeriksa apakah stack kosong (*underflow*). Jika kosong, maka operasi pop akan gagal karena tidak ada elemen yang bisa dihapus, seperti yang terlihat dalam algoritma pop.

- Peek

```
Algoritma peek:
Step 1: IF TOP = -1
   PRINT "Stack is empty"
Step 2: ELSE
   PRINT STACK[TOP]
Step 3: END
```

Melihat elemen yang ada di posisi **TOP** stack tanpa menghapusnya. Fungsi peek untuk memeriksa elemen yang akan diproses berikutnya tanpa mengubah isi stack. Jika stack kosong, fungsi ini akan memberikan informasi bahwa stack tidak memiliki elemen untuk dilihat, seperti yang terlihat dalam algoritma peek.

- isEmpty

```
Algoritma isEmpty:
Step 1: IF TOP = -1
   RETURN True
Step 2: ELSE
   RETURN False
Step 3: END
```

Memeriksa apakah stack kosong. Fungsi ini sangat berguna sebelum melakukan operasi seperti pop atau peek untuk memastikan bahwa tidak ada operasi yang dilakukan dalam stack kosong. Fungsi ini mengembalikan nilai True jika TOP bernilai -1 (menandakan bahwa stack kosong), dan False jika sebaliknya, seperti yang terlihat dalam algoritma isEmpty.

- isFull

```
Algoritma isFull:
Step 1: IF TOP = MAX-1
   RETURN True
Step 2: ELSE
   RETURN False
Step 3: END
```

Memeriksa apakah stack sudah penuh. Fungsi ini penting untuk mencegah operasi push jika stack sudah mencapai kapasitas maksimumnya. Jika TOP mencapai MAX-1 (kapasitas penuh), maka fungsi ini mengembalikan True, menandakan bahwa tidak ada lagi ruang untuk menambah elemen ke dalam stack, seperti yang terlihat dalam algoritma isFull.

- deleteStack

```
Algoritma deleteStack:
Step 1: SET TOP = -1
Step 2: PRINT "Stack deleted"
Step 3: END
```

Menghapus semua elemen dalam stack dan mengembalikan stack ke keadaan kosong. Fungsi ini mengatur nilai TOP kembali ke -1, yang menandakan bahwa tidak ada elemen di stack. Ini berguna jika ingin mengosongkan stack sepenuhnya dan memulai dari awal, seperti yang terlihat dalam algoritma deleteStack.

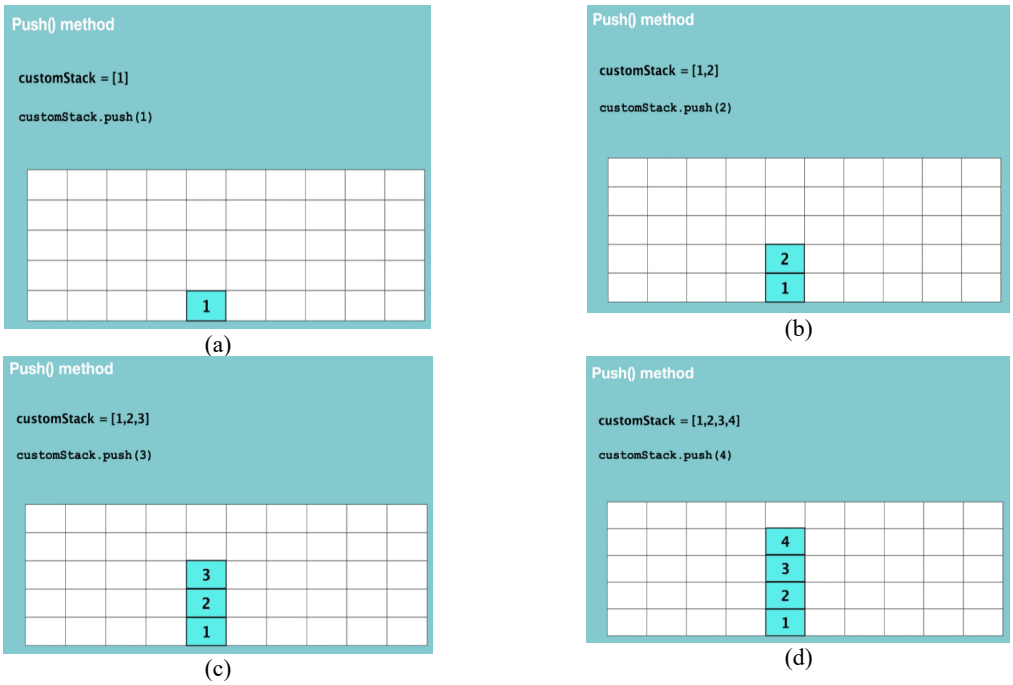
- customStack()

```
Algoritma customStack:
Step 1: DEFINE STACK
Step 2: DEFINE MAX_SIZE
Step 3: DEFINE OPERATIONS
(Push, Pop, Peek, isEmpty, isFull, deleteStack)
Step 4: EXECUTE OPERATIONS BASED ON USER INPUT
Step 5: END
```

Fungsi customStack() untuk pembuatan stack yang lebih fleksibel dan kustomisasi operasinya. Fungsi ini untuk membuat stack dengan ukuran yang ditentukan oleh pengguna (MAX_SIZE), serta memilih operasi yang ingin dijalankan seperti push, pop, peek, isEmpty, isFull, dan deleteStack.

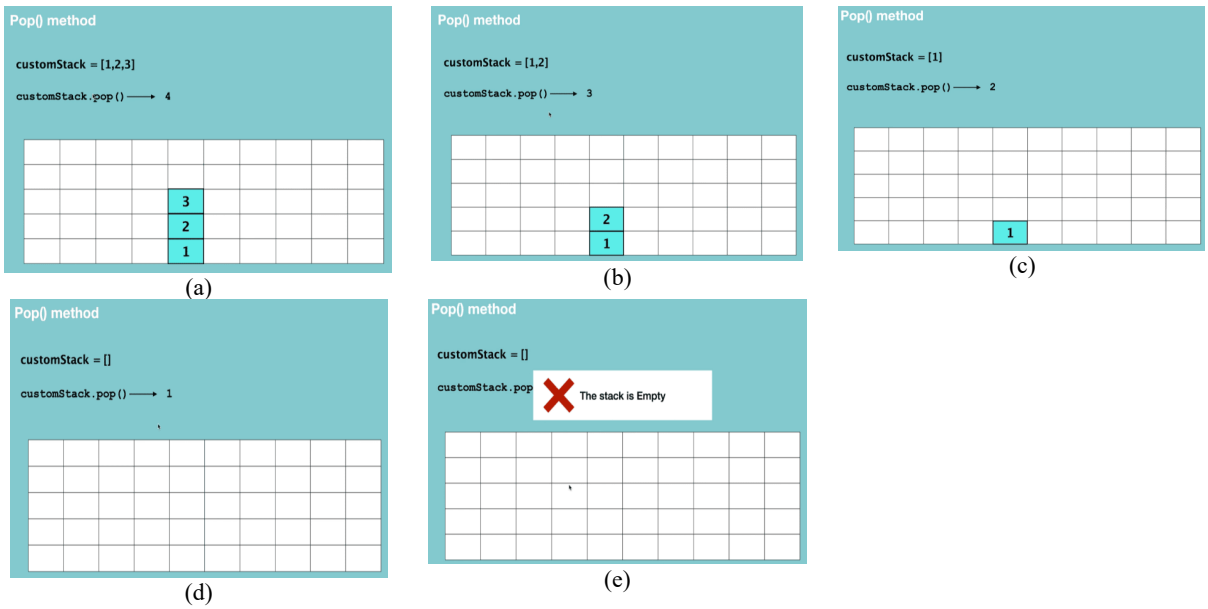
Seperti yang terlihat dalam algoritma customStack. Ini memberikan fleksibilitas lebih dalam mengelola stack sesuai dengan kebutuhan pengguna.

Gambar 7.5 menggambarkan operasi Push dalam struktur data stack. Gambar (a) merupakan sebuah stack kosong diinisialisasi dengan elemen pertama, yaitu angka 1, yang dimasukkan menggunakan metode push().



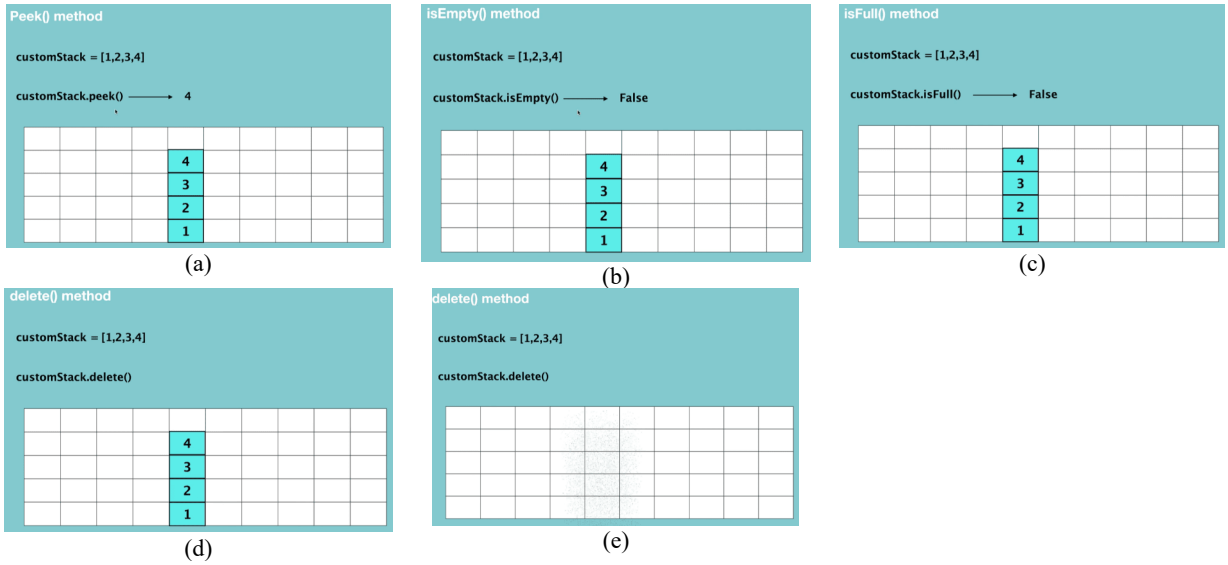
Gambar 7.5: Operasi Push dalam Stack (Karimov, 2022)

Elemen tersebut kemudian ditempatkan di bagian bawah stack. Gambar (b), setelah angka 2 dimasukkan ke dalam stack dengan menggunakan metode push(), angka 2 berada di atas angka 1, mengikuti prinsip LIFO. Gambar (c), setelah angka 3 ditambahkan, angka 3 ditempatkan di posisi teratas stack, diikuti oleh angka 2 dan 1, menunjukkan bahwa elemen yang terakhir dimasukkan berada di posisi paling atas. Gambar (d), angka 4 ditambahkan ke dalam stack, menempatkan angka 4 di posisi paling atas, dengan angka 3, 2, dan 1 mengikuti di bawahnya, menunjukkan bahwa setiap elemen baru ditambahkan di puncak stack.



Gambar 7.6: Operasi Pop dalam Stack (Karimov, 2022)

Gambar 7.6 menggambarkan ilustrasi operasi Pop dalam struktur data stack. Gambar (a), stack diinisialisasi dengan elemen-elemen 1, 2, dan 3, dengan angka 3 berada di posisi teratas. Ketika operasi pop() dilakukan, elemen yang ada di puncak stack, yaitu angka 3, dihapus, dan angka 2 menjadi elemen teratas berikutnya. Hal ini ditunjukkan dengan angka 2 yang sekarang berada di atas angka 1, mengikuti prinsip LIFO. Gambar (b), setelah angka 3 dihapus, operasi pop() kembali dijalankan oleh stack yang kini berisi angka 1 dan 2. Angka 2 dihapus, dan angka 1 menjadi elemen teratas yang tersisa di stack. Gambar (c), setelah angka 2 dikeluarkan, hanya angka 1 yang tersisa di stack. Ketika operasi pop() diterapkan lagi, angka 1 dikeluarkan dari stack, menjadikan stack kosong. Gambar (d) menunjukkan keadaan stack yang kosong setelah seluruh elemen dihapus, mengilustrasikan bahwa operasi pop() menghapus elemen dari puncak stack sesuai dengan prinsip LIFO. Gambar (e), ketika perintah customStack.pop() dijalankan dalam stack yang kosong, maka akan muncul pesan kesalahan "The stack is Empty". Hal ini mengindikasikan bahwa operasi pop() tidak dapat dilakukan karena tidak ada elemen yang dapat dihapus dari stack. Dalam konsep struktur data stack, operasi pop() digunakan untuk menghapus dan mengembalikan elemen teratas dari stack. Namun, jika stack dalam keadaan kosong, operasi ini akan menghasilkan error.



Gambar 7.7: Ilustrasi Beberapa Operasi dalam Stack (Karimov, 2022)

Gambar 7.7 menggambarkan beberapa operasi dasar yang dilakukan dalam struktur data stack. Gambar (a) menunjukkan operasi peek() yang digunakan untuk melihat elemen teratas dari stack tanpa menghapusnya. Dengan customStack.peek(), nilai teratas, yaitu 4, ditampilkan, sementara stack tetap tidak berubah. Gambar (b) ditampilkan operasi isEmpty() yang digunakan untuk memeriksa apakah stack kosong. Dengan perintah customStack.isEmpty(), hasilnya adalah False, yang menunjukkan bahwa stack tidak kosong, karena masih terdapat elemen-elemen di dalamnya. Gambar (c) menggambarkan operasi isFull(), yang memeriksa apakah stack sudah penuh. Dalam contoh ini, customStack.isFull() menghasilkan False, yang berarti stack belum penuh dan masih bisa menampung elemen baru. Gambar (d) menunjukkan operasi delete(), yang berfungsi untuk menghapus elemen teratas dari stack. Setelah operasi customStack.delete(), elemen teratas (yaitu 4) dihapus, dan stack menjadi [3, 2, 1]. Gambar (e) menunjukkan keadaan stack setelah operasi delete() dilakukan berulang kali hingga stack menjadi kosong. Dalam titik ini, stack tidak lagi memiliki elemen yang tersisa.

Berikut contoh program stack dengan berbagai operasi stack.

```
class Stack:
    def __init__(self, capacity):
        self.capacity = capacity  # Kapasitas stack
        self.stack = []  # Menyimpan data stack
        self.front = None  # Elemen paling depan
        self.rear = None  # Elemen paling belakang
        self.top = None  # Elemen teratas

    def push(self, item):
        """Menambahkan elemen ke dalam stack"""
        if len(self.stack) < self.capacity:
            self.stack.append(item)
            self.rear = len(self.stack) - 1  # Posisi rear
            self.top = self.stack[-1]  # Elemen teratas adalah elemen
terakhir
            if len(self.stack) == 1:
                self.front = 0  # Posisi front
            print(f"Elemen '{item}' ditambahkan ke stack.")
            print(f"Front = {self.front}, Rear = {self.rear}")
        else:
            print("Stack penuh! Tidak bisa menambah elemen lagi.")

    def pop(self):
        """Menghapus elemen teratas dari stack"""
        if not self.is_empty():
            popped_item = self.stack.pop()
            if len(self.stack) == 0:
                self.front = None
                self.rear = None
                self.top = None
            else:
                self.rear = len(self.stack) - 1  # Rear diperbarui
                self.top = self.stack[-1]  # Elemen teratas diperbarui
            print(f"Elemen '{popped_item}' dihapus dari stack.")
            print(f"Front = {self.front}, Rear = {self.rear}")
        else:
            print("Stack kosong! Tidak ada elemen yang bisa
dihapus.")

    def peek(self):
        """Melihat elemen teratas stack tanpa menghapusnya"""
        if not self.is_empty():
            return self.top
        else:
            return "Stack kosong!"

    def is_empty(self):
        """Memeriksa apakah stack kosong"""
        return len(self.stack) == 0

    def is_full(self):
        """Memeriksa apakah stack penuh"""
        return len(self.stack) == self.capacity

    def display(self):
        """Menampilkan seluruh elemen dalam stack"""
        if self.is_empty():
            return "Stack kosong!"
        return "Stack contents: " + " <- ".join(map(str, self.stack))
```

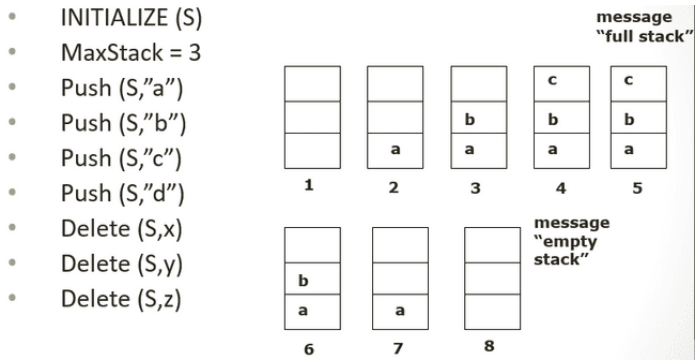
```
# Program Utama
stack_capacity = 5
galaxies_stack = Stack(stack_capacity)

# Menambahkan elemen ke dalam stack
galaxies_stack.push("Milky Way")
galaxies_stack.push("Andromeda")
galaxies_stack.push("Triangulum")
galaxies_stack.push("Whirlpool")
galaxies_stack.push("Sombrero")

# Menampilkan status stack
print(galaxies_stack.display())  # Menampilkan isi stack
print(f"Front: {galaxies_stack.front}, Rear:
{galaxies_stack.rear}, Top: {galaxies_stack.peek()}")

# Menampilkan operasi pop
galaxies_stack.pop()  # Menghapus elemen teratas
(Sombrero)
print(galaxies_stack.display())  # Menampilkan stack
setelah pop

# Menambahkan elemen baru setelah pop
galaxies_stack.push("Tarantula")  # Menambah elemen
setelah ada slot kosong
print(galaxies_stack.display())  # Menampilkan stack
setelah menambah Tarantula
print(f"Front: {galaxies_stack.front}, Rear:
{galaxies_stack.rear}, Top: {galaxies_stack.peek()}")

# Mencoba menambah elemen lebih dari kapasitas
stack
galaxies_stack.push("Tarantula")  # Menambah elemen
lagi (tidak bisa karena penuh)
```

```
Output:
Elemen 'Milky Way' ditambahkan ke stack.
Front = 0, Rear = 0
Elemen 'Andromeda' ditambahkan ke stack.
Front = 0, Rear = 1
Elemen 'Triangulum' ditambahkan ke stack.
Front = 0, Rear = 2
Elemen 'Whirlpool' ditambahkan ke stack.
Front = 0, Rear = 3
Elemen 'Sombrero' ditambahkan ke stack.
Front = 0, Rear = 4
Stack contents: Milky Way <- Andromeda <- Triangulum
<- Whirlpool <- Sombrero
Front: 0, Rear: 4, Top: Sombrero
Elemen 'Sombrero' dihapus dari stack.
Stack contents: Milky Way <- Andromeda <- Triangulum
<- Whirlpool
Front: 0, Rear: 3, Top: Whirlpool
Elemen 'Tarantula' ditambahkan ke stack.
Stack contents: Milky Way <- Andromeda <- Triangulum
<- Whirlpool <- Tarantula
Front: 0, Rear: 4, Top: Tarantula
Stack penuh! Tidak bisa menambah elemen lagi.
```

Secara umum, stack dapat diimplementasikan menggunakan array dan linked list. Implementasi berbasis array akan memiliki panjang stack yang tetap, sedangkan implementasi berbasis linked list dapat memiliki stack dengan panjang yang bervariasi.

7.2 Representasi Stack dalam Array

Dalam memori komputer, stack dapat digambarkan sebagai array linier. Saat implementasi stack berbasis array dengan ukuran tetap, perlu dilakukan pengecekan untuk memastikan apakah stack sudah penuh, karena upaya untuk menambahkan elemen ke dalam stack yang penuh akan menghasilkan kesalahan overflow. Sebaliknya, melakukan operasi pop dalam stack kosong akan memicu kesalahan underflow. Setiap stack dilengkapi dengan variabel bernama top, yang berfungsi menyimpan alamat elemen teratas di stack, sebagai posisi untuk penambahan atau penghapusan elemen. Selain itu, terdapat variabel max yang menyimpan batas maksimum jumlah elemen yang dapat disimpan dalam stack. Jika top = null, ini menandakan stack kosong, sedangkan jika top = max-1, stack dinyatakan penuh.



Gambar 7.8: Representasi Stack dalam Array

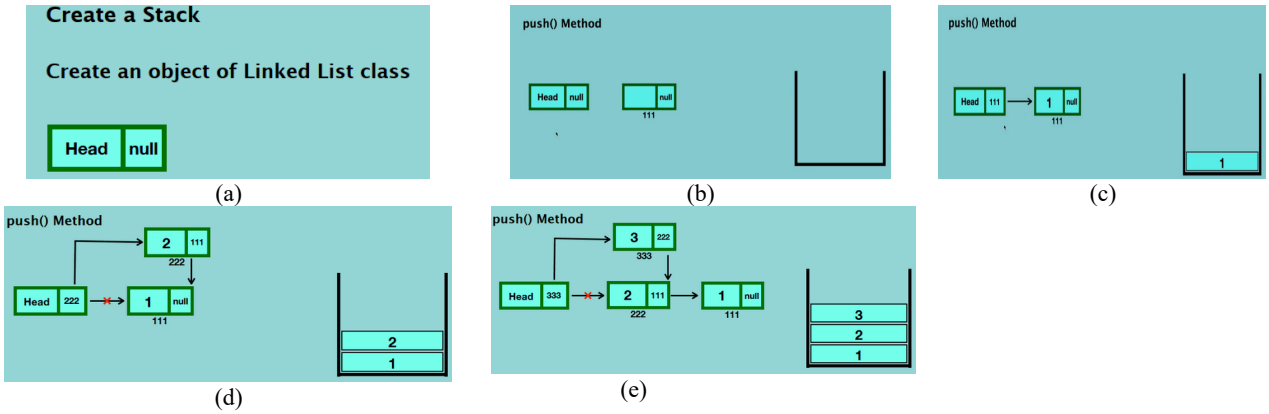
Gambar 7.8 menggambarkan penggunaan vektor untuk implementasi stack dengan batas kapasitas yang telah ditentukan. Dimulai dengan inisialisasi stack dengan nama S dan kapasitas maksimum MaxStack = 3. Langkah pertama adalah menambahkan elemen ke stack menggunakan operasi Push. Elemen pertama yang ditambahkan adalah "a", diikuti dengan elemen "b" dan "c". Langkah keempat, ketika mencoba menambahkan elemen "d", pesan "full stack" muncul, karena stack sudah mencapai kapasitas maksimum, yaitu 3. Setelah itu, operasi Delete dilakukan untuk menghapus elemen dari stack. Operasi pertama menghapus elemen teratas "c", diikuti oleh penghapusan elemen "b" dan terakhir penghapusan elemen "a". Setelah ketiga operasi penghapusan, stack menjadi kosong, dan pesan "empty stack" ditampilkan, yang menunjukkan bahwa tidak ada elemen yang tersisa dalam stack.

7.3 Representasi Stack dalam Linked List

Stack yang dibangun dengan linked list menawarkan kinerja yang lebih cepat. Hal ini karena linked list memungkinkan elemen-elemen untuk ditambahkan atau dihapus secara dinamis tanpa perlu memindahkan elemen-elemen lainnya. Namun, kelemahan dari metode ini adalah implementasinya yang lebih kompleks dibandingkan dengan menggunakan list biasa.

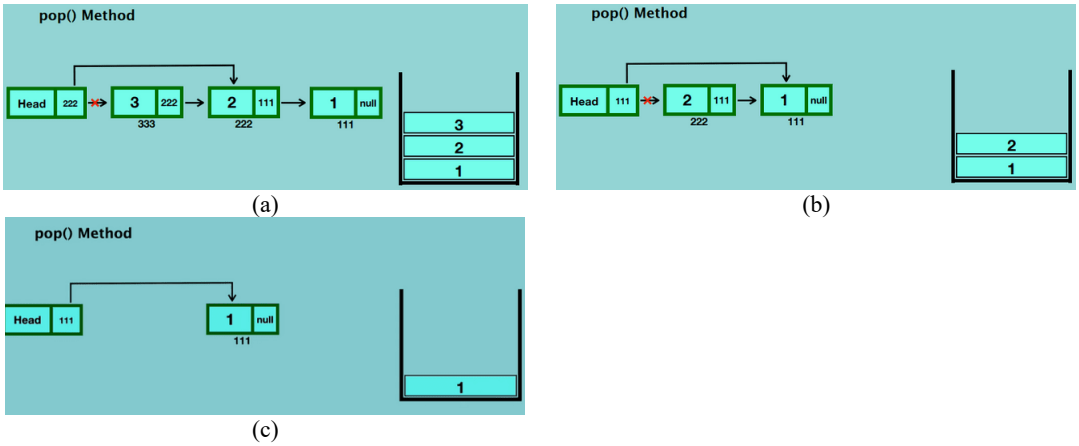
Gambar 7.9 menggambarkan proses operasi push dalam stack yang diimplementasikan menggunakan linked list. Bagian (a) merupakan sebuah stack kosong dibuat dengan objek linked list, yang ditandai dengan Head yang menunjuk ke null, menandakan bahwa stack belum memiliki elemen. Bagian (b) menunjukkan operasi push pertama kali dilakukan dengan menambahkan elemen 111 ke dalam stack. Kepala stack (Head) sekarang menunjuk ke elemen 111 yang baru ditambahkan, sementara elemen lainnya masih belum ada. Bagian (c) menampilkan operasi push kedua dilakukan untuk menambahkan elemen 1 ke stack. Dengan demikian, elemen baru (1) ditempatkan di atas elemen 111, dan Head sekarang menunjuk ke elemen 1. Panah yang dicoret menunjukkan bahwa hubungan antara Head yang sebelumnya menunjuk ke elemen 111 terputus, dan Head kini mengarah ke elemen 1 yang baru ditambahkan. Elemen 111 menjadi elemen kedua dalam stack. Bagian (d),

operasi push dilanjutkan dengan menambahkan elemen 2 ke dalam stack. Sekarang, elemen 2 berada di bagian paling atas, diikuti oleh 1, dan 111 berada di bagian bawah. Head stack kini menunjuk ke elemen 2. Di bagian ini, panah yang dicoret menunjukkan perubahan hubungan antara Head dan elemen sebelumnya. Head yang sebelumnya menunjuk ke elemen 1 kini terputus, dan Head mengarah ke elemen 2 yang menjadi elemen teratas stack. 1 dan 111 tetap berada di bawahnya, mengikuti urutan LIFO.



Gambar 7.9: Representasi Stack dalam Linked List untuk Operasi Push (Karimov, 2022)

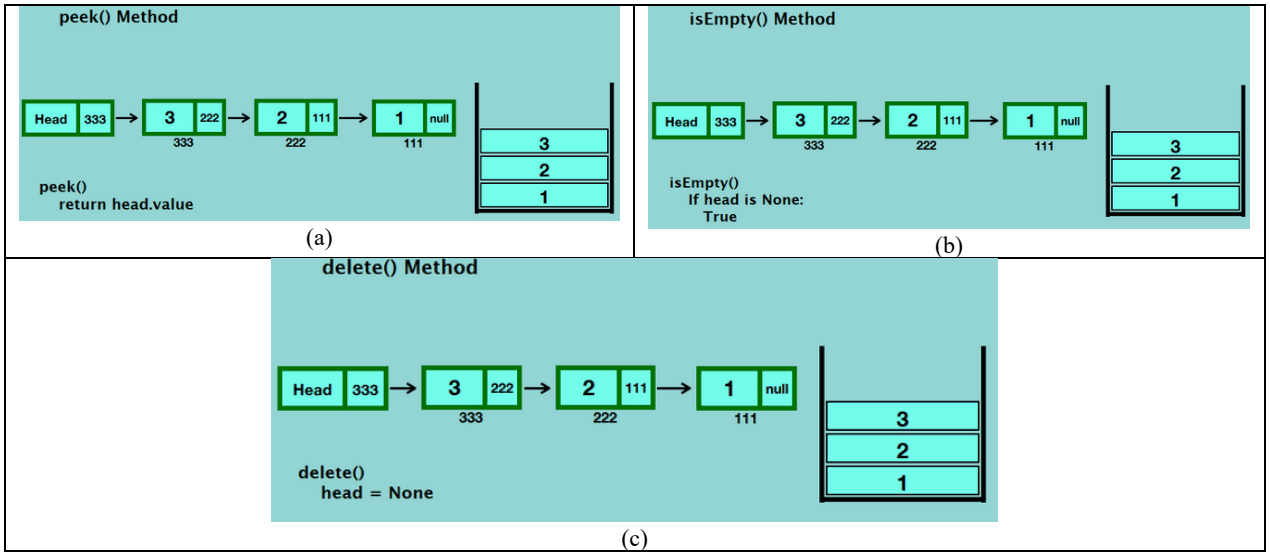
Gambar 7.10 menggambarkan proses operasi pop dalam stack yang diimplementasikan menggunakan linked list. Bagian (a) menunjukkan operasi pop pertama kali dilakukan untuk menghapus elemen teratas, yaitu 3, dari stack. Setelah operasi tersebut, elemen 2 menjadi elemen teratas yang baru. Kepala (Head) stack sekarang menunjuk ke elemen 2, yang terletak di puncak tumpukan, sedangkan elemen 1 dan 111 tetap berada di bawahnya. Bagian ini menunjukkan bahwa setelah elemen 3 dihapus, stack hanya berisi elemen 2, 1, dan 111. Bagian (b) menampilkan operasi pop diterapkan lagi untuk menghapus elemen teratas yang baru, yaitu 2. Setelah penghapusan elemen 2, elemen 1 menjadi elemen teratas yang baru, dan Head stack kini menunjuk langsung ke elemen 1. Elemen 111 tetap berada di bagian bawah stack, mengikuti prinsip LIFO. Bagian (c) menunjukkan operasi pop terakhir dilakukan untuk menghapus elemen teratas yang tersisa, yaitu 1. Setelah elemen 1 dihapus, Head stack menunjuk ke null, menandakan bahwa stack sekarang kosong.



Gambar 7.10: Representasi Stack dalam Linked List untuk Operasi Pop (Karimov, 2022)

Gambar 7.11 menggambarkan tiga operasi dalam stack yang diimplementasikan menggunakan linked list: peek, isEmpty, dan delete. Bagian (a) menunjukkan operasi peek dilakukan untuk memeriksa nilai elemen teratas stack tanpa menghapusnya. Head stack menunjuk ke elemen 3, yang merupakan elemen teratas, dan metode peek mengembalikan nilai 3 tanpa mengubah struktur stack. Ini memungkinkan untuk memeriksa elemen teratas tanpa memodifikasi stack itu sendiri. Visualisasi di sebelah kanan menunjukkan bahwa stack masih berisi elemen-elemen 3, 2, 1, dan 111, dengan elemen 3 di bagian atas. Bagian (b) menggambarkan operasi isEmpty dilakukan

untuk memeriksa apakah stack kosong. Dalam gambar ini, Head menunjuk ke elemen 3, menandakan bahwa stack tidak kosong. Metode isEmpty mengembalikan nilai False, karena Head tidak mengarah ke null atau kosong. Di sisi kanan, stack berisi tiga elemen, yaitu 3, 2, dan 1, dengan elemen 3 berada di atas. Bagian (c) menampilkan operasi delete dilakukan untuk menghapus seluruh stack. Proses ini menghapus semua elemen di stack, yang terlihat ketika Head stack terhapus dan ditetapkan ke None. Dalam gambar ini, semua elemen dalam stack (3, 2, 1, dan 111) dihapus, dan Head akhirnya menunjuk ke None, menandakan bahwa stack sudah kosong. Hal ini memperlihatkan bagaimana operasi delete dapat digunakan untuk menghapus seluruh stack secara keseluruhan.



Gambar 7.11: Representasi Stack dalam Linked List untuk Operasi Peek, isEmpty, dan Delete (Karimov, 2022)

Tabel 7.2 menunjukkan kompleksitas waktu dan ruang dari operasi stack yang diimplementasikan menggunakan linked list. Kompleksitas waktu dan ruang untuk membuat stack adalah O(1), yang berarti bahwa waktu dan ruang yang dibutuhkan untuk membuat stack tidak tergantung terhadap ukuran stack. Operasi push memiliki kompleksitas waktu dan ruang O(1). Ini berarti operasi tersebut memerlukan waktu konstan dan hanya membutuhkan ruang untuk menambahkan satu elemen baru. Operasi pop memiliki kompleksitas waktu dan ruang O(1). Ini menunjukkan bahwa penghapusan elemen teratas tidak mempengaruhi waktu atau ruang secara signifikan. Operasi peek memiliki kompleksitas waktu dan ruang O(1). Operasi ini juga tidak mempengaruhi waktu atau ruang secara signifikan. Operasi isEmpty, memiliki kompleksitas waktu dan ruang O(1). Pemeriksaan ini dilakukan dalam waktu konstan dan tidak memerlukan ruang tambahan. Operasi delete untuk menghapus seluruh stack juga memiliki kompleksitas waktu dan ruang O(1). Meskipun semua elemen dihapus, operasi ini memerlukan waktu dan ruang konstan karena proses penghapusan dilakukan dalam satu langkah tanpa mempengaruhi ruang secara besar.

Tabel 7.2: Kompleksitas Waktu dan Ruang dari Operasi Stack dalam Linked List

Operasi	Kompleksitas Waktu	Kompleksitas Ruang
Create Stack	O(1)	O(1)
Push	O(1)	O(1)
Pop	O(1)	O(1)
Peek	O(1)	O(1)
isEmpty	O(1)	O(1)
Delete Entire Stack	O(1)	O(1)

7.4 Penerapan Stack Menggunakan Polish Nation

Stack dapat diterapkan dalam konversi notasi matematika ke dalam polish notation, yang terdiri dari tiga jenis notasi utama, yaitu **infix, prefix, dan postfix**.

Notasi infix adalah bentuk ekspresi matematika yang paling umum digunakan, di mana operator ditempatkan di antara dua operand (**operand operator operand**). **Operand** adalah nilai atau variabel yang dikenakan operasi oleh operator, misalnya bilangan (2, 5, 100) dan variabel (A, B, C). **Operator** adalah simbol yang digunakan untuk melakukan operasi terhadap operand, misalnya operator aritmatika, operator pembanding, operator logika, dan operator bitwise. Contohnya adalah "**A + B**", di mana operator '+' berada di antara operand A dan B. **Notasi prefix** yang meletakkan operator sebelum operand (**operator operand operand**). Sebagai contoh, ekspresi "A + B" dalam notasi prefix akan ditulis sebagai "**+ AB**". Notasi **postfix** atau *Reverse Polish Notation* (RPN), meletakkan operator setelah operand (**operand operand operator**). Sebagai contoh, "A + B" dalam notasi postfix akan ditulis sebagai "**AB +**".

Tabel 7.3: Contoh Polish Nation

Contoh Operasi	Infix	Prefix	Postfix
A + B	A + B	+AB	AB+
A+B-C	((A + B) – C)	-+ABC	AB+C-

Ada beberapa hal yang perlu diperhatikan dalam mengerjakan polish notation, yaitu: 1) ketika mengerjakan notasi prefix atau postfix, bagian yang diberi tanda kurung () merupakan bagian yang dikerjakan terlebih dahulu. 2) Proses pengerjaan mengikuti urutan prioritas operator matematika. Berikut contoh perhitungan menggunakan Polish Notation.

Soal 1: Ubah ekspresi matematika dari infix ke prefix dan postfix.

Infix: 2 * 3 ^ 4 + 1 – 5 = 158

Prefix:
- langkah I: 2 * (^34) + 1 – 5
- Langkah II: (*2^34) + 1 – 5
- Langkah III: (+*2^341) – 5
- Langkah IV: – + * 2 ^ 3 4 1 5

Jadi, notasi prefix yaitu:
– + * 2 ^ 3 4 1 5

Postfix :
- Langkah I: 2 * (34^) + 1 – 5
- Langkah II: (234^*) + 1 – 5
- Langkah III: (234^*1+) – 5
- Langkah IV: 2 3 4 ^ * 1 + 5 –

Jadi, notasi postfix yaitu:
2 3 4 ^ * 1 + 5 –

Soal 2: Mengubah notasi prefix ke postfix dan infix.

Prefix: - + 3 * 2 4 / 5 ^ 6 2

Infix:
- Langkah I: - + 3 * 2 4 / 5 (6 ^ 2)
- Langkah II: - + 3 (2 * 4) / 5 (6 ^ 2)
- Langkah III: - + 3 (2 * 4) (5 / (6 ^ 2))
- Langkah IV: - (3 + (2 * 4)) (5 / (6 ^ 2))
- Langkah V: (3 + (2 * 4)) - (5 / (6 ^ 2))

Jadi, notasi infix yaitu:
3 + 2 * 4 - 5 / 6 ^ 2 = 10.9

Postfix :
- Langkah I: 3 + 2 * 4 - 5 / (6 2 ^)
- Langkah II: 3 + 2 * 4 – (5 (6 2 ^) /)
- Langkah III: 3 + (2 4 *) – (5 (6 2 ^) /)
- Langkah IV: 3 + (2 4 *) – (5 (6 2 ^) /)
- Langkah V: 3 (2 4 *) +) – (5 (6 2 ^) /))
- Langkah VI: (3 (2 4 *) +) (5 (6 2 ^) /)) -)

Jadi, notasi postfix yaitu:
(3 (2 4 *) +) (5 (6 2 ^) /)) -)

Soal 3: Mengubah notasi postfix ke prefix dan infix.

Postfix: 8 3 2 ^ * 9 6 / + 7 -

Infix:
Langkah I: 8 (3 ^ 2) * 9 6 / + 7 -
Langkah II: (8 * (3 ^ 2)) 9 6 / + 7 -
Langkah III: (8 * (3 ^ 2)) (9 / 6) + 7 -
Langkah IV: (8 * (3 ^ 2)) + (9 / 6) 7 -
Langkah V: (8 * (3 ^ 2)) + (9 / 6) - 7

Jadi, notasi infix yaitu:
(8 * (3 ^ 2)) + (9 / 6) – 7 = 66.5

Prefix:
Langkah I: 8 * (^ 3 2) + 9 / 6 – 7
Langkah II: (* 8 (^ 3 2)) + 9 / 6 – 7
Langkah III: (* 8 (^ 3 2)) + (/ 9 6) - 7
Langkah IV: (+(* 8 (^ 3 2)) (/ 9 6)) - 7
Langkah V: (- (+(* 8 (^ 3 2)) (/ 9 6)) 7)

Jadi, notasi prefix yaitu:
(- (+(* 8 (^ 3 2)) (/ 9 6)) 7)

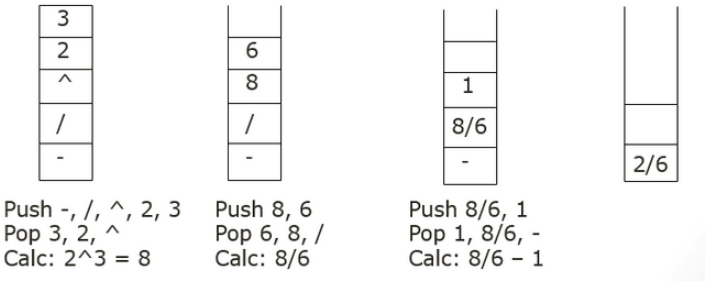
♣ **Penerapan Stack dalam Ekspresi Prefix**

Cara menerapkan stack dalam ekspresi prefix, dilakukan dengan cara yaitu:

1. Dalam proses ini, operator terlebih dahulu dimasukkan (push) ke dalam stack.
2. Ketika ditemukan dua operand yang berurutan, kedua operand tersebut diambil (pop) dari stack untuk dilakukan operasi sesuai dengan operator yang berlaku.
3. Hasil dari operasi tersebut kemudian dimasukkan kembali (push) ke dalam stack.
4. Proses ini dilakukan secara berulang hingga seluruh elemen dalam ekspresi selesai diproses dan tersisa satu nilai sebagai hasil akhir perhitungan.

Dengan demikian, penggunaan stack dalam evaluasi notasi prefix memungkinkan proses komputasi ekspresi matematika dilakukan secara sistematis dan efisien sesuai dengan urutan operasi yang ditentukan.

Contoh: Terdapat infix: 2 ^ 3 / 6 – 1 dan prefix: - / ^ 2 3 6 1. Masukkan prefix ke dalam stack. Berikut penyelesaiannya:



Gambar 7.12: Penerapan Stack dalam Ekspresi Prefix

Gambar 7.12 menggambarkan proses penerapan stack dalam ekspresi prefix. Proses ini dilakukan dengan memasukkan elemen ke stack dan pop (mengambil elemen dari stack), untuk menghitung hasil ekspresi secara bertahap. Tahap pertama, operator dan operand dimasukkan ke dalam stack sesuai dengan urutan ekspresi prefix. Ketika dua operand ditemukan secara berurutan, keduanya diambil (pop) dari stack untuk dilakukan operasi sesuai dengan operator yang berlaku. Sebagai contoh, operand 2 dan 3 diambil dari stack dan dihitung menggunakan operator pangkat (^) sehingga menghasilkan nilai $2^3 = 8$, kemudian hasil tersebut dimasukkan kembali (push) ke dalam stack. Tahap berikutnya, operand 8 dan 6 dimasukkan ke dalam stack, kemudian dilakukan operasi pembagian menggunakan operator / sehingga menghasilkan 8/6. Hasil operasi ini juga dimasukkan kembali ke dalam stack untuk diproses ke tahap selanjutnya. Selanjutnya, nilai 8/6 dan operand 1 diambil dari stack untuk dilakukan operasi pengurangan menggunakan operator -, sehingga menghasilkan ekspresi 8/6 − 1. Tahap terakhir menunjukkan hasil akhir setelah operasi tersebut disederhanakan, yaitu menjadi 2/6. Nilai 2/6 inilah yang ditampilkan sebagai hasil akhir dari proses evaluasi ekspresi prefix menggunakan stack.

Proses ini menunjukkan bagaimana stack digunakan untuk mengelola operand dan operator secara sistematis hingga diperoleh hasil perhitungan akhir.

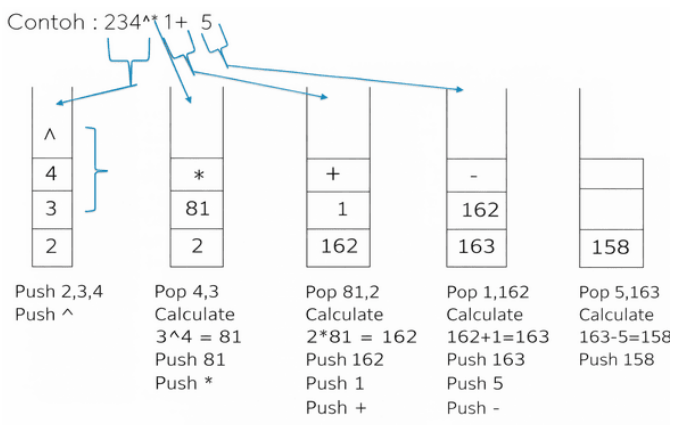
♣ **Penerapan Stack dalam Ekspresi Postfix**

Cara menerapkan stack dalam ekspresi postfix, dilakukan dengan cara yaitu:

1. Apabila elemen yang dibaca merupakan operand, maka nilai tersebut dimasukkan ke dalam stack menggunakan operasi push. 2
2. Apabila elemen yang ditemukan merupakan operator, maka dua operand yang berada di bagian paling atas stack diambil menggunakan operasi pop.
3. Kedua operand tersebut kemudian diproses menggunakan operator yang bersangkutan untuk menghasilkan suatu nilai hasil perhitungan.
4. Hasil dari operasi tersebut selanjutnya dimasukkan kembali ke dalam stack untuk digunakan untuk proses perhitungan berikutnya.
5. Proses ini berlangsung secara berulang hingga seluruh elemen dalam ekspresi postfix selesai diproses.
6. Akhir proses evaluasi, hanya akan tersisa satu nilai di dalam stack yang merupakan hasil akhir dari ekspresi matematika tersebut.

Dengan demikian, penggunaan stack dalam evaluasi notasi postfix memungkinkan proses perhitungan ekspresi dilakukan secara sistematis, efisien, dan sesuai dengan urutan operasi yang telah ditentukan dalam ekspresi postfix.

Contoh: dari soal 1 diatas, masukkan postfix ke dalam stack. Berikut penyelesaiannya:



Gambar 7.13: Penerapan Stack dalam Ekspresi Postfix

Gambar 7.13 menunjukkan penerapan stack dalam evaluasi ekspresi postfix menggunakan struktur data stack untuk melakukan perhitungan bertahap. Proses dimulai dengan memasukkan angka atau operand ke dalam stack dengan operasi push. Ekspresi 234^*1+5, langkah pertama adalah memasukkan angka 2, 3, dan 4 ke dalam stack, diikuti dengan operator ^ yang menunjukkan operasi pangkat antara 3 dan 4. Hasil dari 3^4 yang dihitung menjadi 81, kemudian dimasukkan kembali ke dalam stack. Selanjutnya, operand 81 dan 2 dimasukkan untuk melakukan operasi perkalian 2 * 81, yang menghasilkan 162. Setelah itu, hasil operasi 162 dimasukkan kembali ke dalam stack. Langkah berikutnya melibatkan operator +, yang menambahkan operand 162 dengan 1, menghasilkan 163. Kemudian, 163 dan 5 dihitung dengan operator - yang menghasilkan 158, dan akhirnya hasil akhir 158 dipush kembali ke dalam stack.

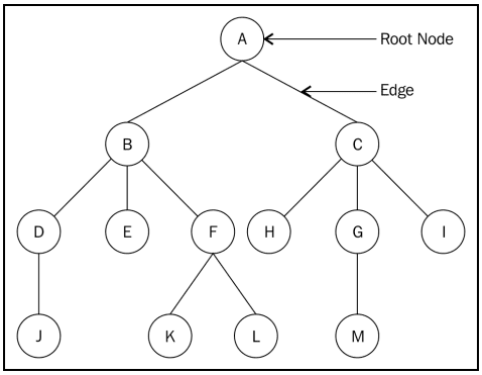
Latihan:

1. Dari contoh soal no 2, ilustrasikan postfix dan prefix ke dalam stack.
2. Dari contoh soal no 3, ilustrasikan postfix dan prefix ke dalam stack.
3. Buat notasi prefix & postfixnya hitung hasilnya bila: A = 5, B = 2, C = 1, D =4, E = 4. Ilustrasikan postfix dan prefix ke dalam stack.
   A + B * C * D – E
   A + (B – C) * D ^ E
   A ^ B + C ^ D * E
4. Diberikan polish notation sebagai berikut:
   a. 4 ^ 2 / 8 + 5 * 2 – 4
   b. 7 3 * 5 2 * - 4 + 3 /
   c. / ^ + - * 6 3 5 2 1 5
   i) Buat ke dalam bentuk 2 polish notation lainnya
   ii) konversi ke bentuk stack dengan algoritma prefix & hitung hasilnya
   iii) konversi ke bentuk stack dengan algoritma postfix & hitung hasilnya
5. Buat code yang merubah prefix ke infix dan ilustrasikan ke stack.
6. Buat code yang merubah postfix ke infix dan ilustrasikan ke stack.
7. Buat code yang merubah prefix ke postfix dan ilustrasikan ke stack.
8. Buat code yang merubah infix ke postfix dan ilustrasikan ke stack.

Berikut contoh program stack sederhana, contoh code merubah infix ke prefix secara dinamis, serta contoh code penerapan stack menggunakan polish nation secara dinamis dapat dilihat di link https://colab.research.google.com/drive/1WKxhQQHHfl_eL6rMEiqzCWyY4_4l8V4T?usp=sharing.

BAB 8
Tree

Tree merupakan struktur data nonlinear yang merepresentasikan hubungan hierarkis antarelemen tanpa membentuk siklus. Berbeda dari pohon di dunia nyata yang umumnya digambarkan tumbuh dari bawah ke atas, struktur tree dalam ilmu komputer divisualisasikan secara terbalik, yaitu berawal dari satu simpul utama di bagian atas dan bercabang ke bawah. Dengan demikian, tree menggambarkan hubungan one-to-many, yakni satu elemen dapat memiliki beberapa elemen turunan di bawahnya dalam susunan yang bertingkat (Agarwal, 2022). Berikut istilah yang berkaitan dengan tree:



Gambar 8.1: Contoh Tree (Agarwal, 2022)

- **Node** (simpul) adalah setiap elemen dalam tree yang berfungsi sebagai tempat penyimpanan data.
- **Root node** (simpul akar) merupakan simpul pertama yang menjadi asal seluruh simpul lain dalam tree. Simpul ini tidak memiliki parent. Dalam setiap tree hanya terdapat satu root node yang bersifat unik. Simpul A adalah root node.
- **Subtree** adalah bagian dari tree yang juga membentuk tree tersendiri, dengan simpul-simpulnya merupakan turunan dari tree yang lebih besar. Sebagai contoh, simpul F, K, dan L membentuk sebuah subtree dari tree utama.
- **Degree** (derajat) adalah jumlah anak (children) yang dimiliki oleh suatu simpul. Tree yang hanya terdiri atas satu simpul memiliki derajat 0. Dalam contoh tersebut, derajat simpul A adalah 2, derajat simpul B adalah 3, derajat simpul C adalah 3, dan derajat simpul G adalah 1.
- **Leaf node** (simpul daun) adalah simpul terminal yang tidak memiliki anak, sehingga derajatnya selalu 0. Dalam contoh tersebut, simpul J, E, K, L, H, M, dan I termasuk leaf node.
- **Edge** (sisi) adalah hubungan yang menghubungkan dua simpul dalam tree. Jumlah edge dalam suatu tree paling banyak adalah satu lebih sedikit daripada jumlah simpulnya.
- **Parent** (induk) adalah simpul yang memiliki subtree atau simpul turunan di bawahnya. Sebagai contoh, simpul B merupakan parent dari simpul D, E, dan F, sedangkan simpul F merupakan parent dari simpul K dan L.
- **Child** (anak) adalah simpul yang merupakan turunan langsung dari suatu parent. Dalam contoh tersebut, simpul B dan C merupakan child dari simpul A, sedangkan simpul H, G, dan I merupakan child dari simpul C.
- **Sibling** (saudara) adalah simpul-simpul yang memiliki parent yang sama. Sebagai contoh, simpul B dan C adalah sibling, demikian pula simpul D, E, dan F.
- **Level** (tingkat) menunjukkan posisi suatu simpul dalam tree berdasarkan jaraknya dari root node. Root node berada di level 0, anak-anak root berada di level 1, anak-anak dari simpul level 1 berada di level

2, dan seterusnya. Simpul A berada di level 0, simpul B dan C berada di level 1, sedangkan simpul D, E, F, H, G, dan I berada di level 2.

- **Height** (tinggi tree) adalah jumlah simpul dalam lintasan terpanjang dalam tree. Tinggi tree adalah 4, karena lintasan terpanjang seperti A-B-D-J, A-C-G-M, dan A-B-F-K masing-masing terdiri atas empat simpul.
- **Depth** (kedalaman) adalah jumlah edge dari root node ke suatu simpul tertentu. Dalam contoh tersebut, simpul H memiliki depth sebesar 2.

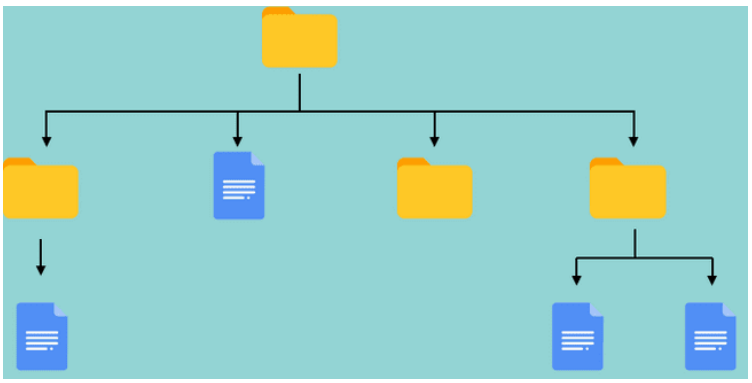
Sifat-sifat tree yaitu:

- **Tree** digunakan untuk merepresentasikan data yang bersifat **hierarkis**.
- Setiap simpul (**node**) dalam tree umumnya memiliki dua komponen utama, yaitu **data** dan **tautan** yang menghubungkannya dengan subkategori atau simpul turunannya.
- Struktur tree terdiri atas **kategori dasar** sebagai akar dan **subkategori** yang berada di bawahnya dalam susunan bertingkat.

Alasan Penggunaan tree yaitu:

- Tree memberikan **akses terhadap data yang lebih cepat dan lebih mudah** dalam kondisi tertentu.
- Struktur ini sangat sesuai untuk menyimpan data yang memiliki hubungan **hierarkis**, seperti **struktur folder**, **struktur organisasi**, serta data **XML** dan **HTML**.
- Terdapat berbagai jenis struktur tree yang dirancang untuk memberikan kinerja lebih baik sesuai dengan kebutuhan dan situasi tertentu.
- Beberapa contoh struktur tree yang sering digunakan adalah **Binary Search Tree**, **AVL Tree**, **Red-Black Tree**, dan **Trie**.

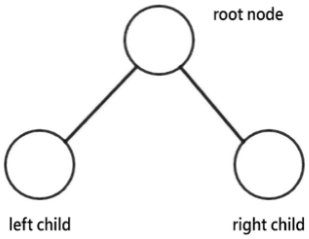
Salah satu contoh penerapan struktur tree dalam kehidupan komputasi adalah **sistem berkas (file system)** komputer, yang menyusun folder dan file dalam hubungan bertingkat, seperti Gambar 8.2.



Gambar 8.2: Contoh Penerapan Tree (Karimov, 2022)

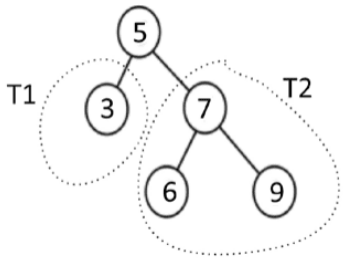
8.1 Binary Tree

Binary tree merupakan kumpulan simpul yang setiap simpulnya dapat memiliki **nol, satu, atau dua simpul anak**. Dalam bentuk paling sederhana, sebuah binary tree memiliki paling banyak dua anak di setiap simpul, yaitu **anak kiri** (*left child*) dan **anak kanan** (*right child*). Dengan demikian, hubungan antar simpul dalam binary tree tersusun secara hierarkis dan terbatas dengan dua cabang utama untuk setiap simpul. Gambar 8.3 merupakan contoh binary tree, terdapat sebuah **root node** yang memiliki dua anak, yaitu satu anak di sisi kiri dan satu anak di sisi kanan (Agarwal, 2022).



Gambar 8.4: Binary Tree (Agarwal, 2022)

Binary tree dapat memiliki simpul-simpul disusun dalam bentuk **subtree kiri** (*left subtree*) dan **subtree kanan** (*right subtree*). Setiap simpul dapat berperan sebagai akar bagi dua subtree tersebut, sehingga struktur binary tree membentuk susunan hierarkis yang bercabang ke dua arah. Sebagai contoh, suatu tree yang terdiri atas lima simpul dapat memiliki sebuah **root node** bernama **R**, dengan dua subtree, yaitu **subtree kiri T1** dan **subtree kanan T2**. Susunan ini menunjukkan bahwa binary tree dibangun dari satu simpul utama yang menjadi pusat,

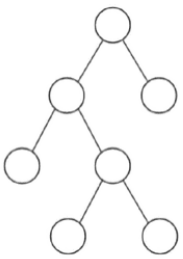


Gambar 8.3: Contoh Subtree dari Binary Tree (Agarwal, 2022)

kemudian bercabang ke bagian kiri dan kanan sebagai struktur turunannya.

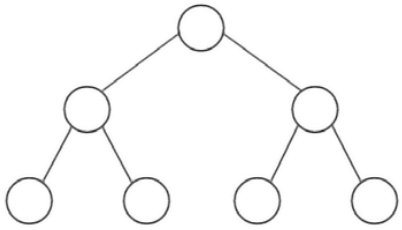
Gambar 8.4 memperlihatkan contoh subtree dalam suatu binary tree. Simpul 5 berperan sebagai root node atau simpul akar. Dari simpul root ini terbentuk dua cabang utama, yaitu cabang kiri dan cabang kanan. Cabang kiri ditandai sebagai T1, yang terdiri atas simpul 3 sebagai subtree kiri. Sementara itu, cabang kanan ditandai sebagai T2, yang berakar di simpul 7 dan memiliki dua anak, yaitu simpul 6 di sebelah kiri serta simpul 9 di sebelah kanan. Sebuah tree dapat diuraikan menjadi subtree-subtree yang lebih kecil, dan masing-masing subtree tetap memiliki sifat sebagai tree.

Secara umum, binary tree merupakan struktur data yang setiap simpulnya memiliki paling banyak dua anak, yaitu anak kiri dan anak kanan. Binary tree biasa tidak memiliki aturan tambahan mengenai bagaimana elemen harus disusun, selama setiap simpul tidak memiliki lebih dari dua anak. Adapun **full binary tree** merupakan bentuk yang lebih khusus, yaitu **tree yang seluruh simpulnya hanya memiliki nol atau dua anak, tanpa ada simpul yang memiliki tepat satu anak**. Dengan demikian, full binary tree dapat dipahami sebagai binary tree yang memenuhi syarat keseimbangan struktural tertentu di setiap percabangan. Gambar 8.1



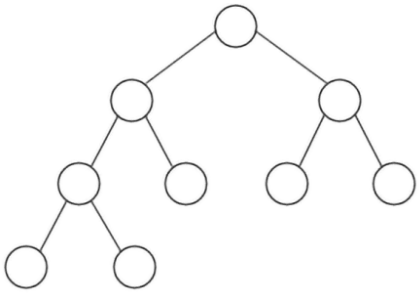
Gambar 8.5: Full Binary Tree (Agarwal, 2022)

menunjukkan contoh full binary tree, yaitu bentuk khusus dari binary tree yang setiap simpulnya hanya memiliki dua kemungkinan, yakni tidak memiliki anak sama sekali atau memiliki tepat dua anak. Simpul paling atas berperan sebagai root node dan memiliki dua anak. Simpul di sisi kiri selanjutnya juga memiliki dua anak, sedangkan simpul di sisi kanan tidak memiliki anak sehingga berperan sebagai leaf node. Di tingkat berikutnya, salah satu simpul cabang kembali memiliki dua anak, sementara simpul-simpul lain berada di posisi terminal. Dengan susunan seperti ini, tidak ada satu pun simpul yang hanya mempunyai satu anak. Ciri inilah yang menegaskan bahwa struktur gambar tersebut termasuk full binary tree.



Gambar 8.7: Perfect Binary Tree (Agarwal, 2022)

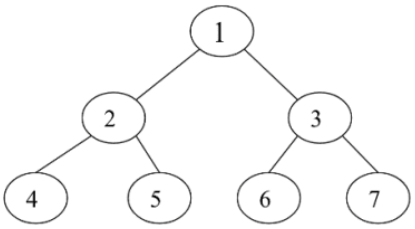
Secara konsep, **perfect binary tree** adalah **binary tree yang semua simpul internalnya memiliki dua anak dan seluruh simpul daunnya berada di kedalaman yang sama**. Dengan demikian, tidak ada ruang kosong yang tersedia untuk menambahkan simpul baru di tingkat yang sudah ada. Apabila ingin menambahkan simpul baru, penambahan tersebut hanya dapat dilakukan dengan membentuk tingkat baru, sehingga tinggi tree harus bertambah. Gambar 8.6 menegaskan bahwa perfect binary tree memiliki struktur yang sangat teratur dan simetris, karena setiap level terisi sepenuhnya tanpa ada kekosongan. Struktur seperti ini sering dijadikan model ideal dalam pembahasan tree karena menunjukkan bentuk binary tree yang paling lengkap dan seimbang.



Gambar 8.6: Complete Binary Tree (Agarwal, 2022)

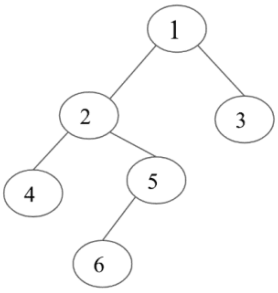
Complete binary tree merupakan jenis **binary tree yang seluruh tingkatnya terisi penuh**, **kecuali tingkat paling bawah**. Tree tingkat terakhir memiliki simpul-simpul diisi secara berurutan mulai dari sisi kiri, sehingga tidak terdapat kekosongan di bagian kiri selama masih ada simpul yang dapat ditempatkan. Complete binary tree memiliki susunan yang hampir penuh, tetapi tetap mempertahankan aturan bahwa pengisian simpul dilakukan terlebih dahulu dari kiri ke kanan. Struktur ini menjadikan complete binary tree berbeda dari **perfect binary tree**, **karena tingkat terakhirnya tidak harus terisi sepenuhnya, tetapi tetap harus tersusun rapat dari sisi kiri**. Gambar 8.7 menampilkan complete binary tree.

Binary tree dapat dibedakan menjadi **balanced** dan **unbalanced**. **Balanced binary tree tidak** memiliki selisih tinggi antara **subtree kiri** dan **subtree kanan**. Kondisi ini menunjukkan bahwa penyebaran simpul di kedua sisi tree relatif seimbang, sehingga struktur tree tidak terlalu condong ke salah satu sisi. Keseimbangan tersebut penting karena dapat mendukung efisiensi operasi dalam tree, seperti pencarian, penyisipan, dan penghapusan data. Gambar 8.8 menampilkan balanced tree.



Gambar 8.9: Balanced Tree (Agarwal, 2022)

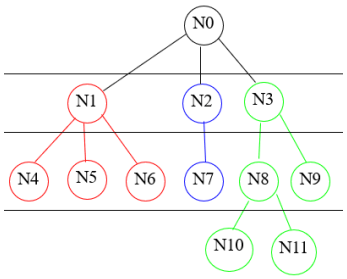
Unbalanced binary tree adalah binary tree yang memiliki selisih tinggi lebih dari satu antara subtree kiri dan subtree kanan. Kondisi ini menunjukkan bahwa distribusi simpul tree tidak seimbang, sehingga salah satu sisi tree cenderung lebih dalam atau lebih panjang dibandingkan sisi lainnya. Ketidakseimbangan tersebut dapat memengaruhi efisiensi operasi tree, karena jalur pencarian dalam salah satu cabang menjadi lebih panjang. Gambar 8.9 menampilkan unbalanced tree.



Gambar 8.8: Unbalanced Tree (Agarwal, 2022)

8.2 Representasi Tree

Representasi tree merupakan cara menyajikan struktur data hierarkis yang terdiri atas simpul (node) dan hubungan induk-anak (parent-child) sehingga keterkaitan antar elemen dapat dipahami secara sistematis. Dalam ilmu komputer, tree digunakan untuk menggambarkan data yang memiliki susunan bertingkat, mulai dari satu simpul akar (root), bercabang ke simpul-simpul turunan, hingga mencapai simpul daun (leaf) yang tidak memiliki anak. Representasi tree menjadi penting karena tidak hanya membantu proses visualisasi, tetapi juga memudahkan analisis hubungan, penelusuran jalur, identifikasi tingkat kedalaman, serta pengelolaan data yang tersusun secara bertahap. Dengan demikian, pemilihan bentuk representasi tree harus disesuaikan dengan tujuan penyajian, apakah untuk memperjelas hubungan hierarkis, menampilkan urutan keturunan, menekankan level, atau menyederhanakan bentuk struktur ke dalam notasi yang lebih ringkas.



Gambar 8.10: Contoh Binary Tree

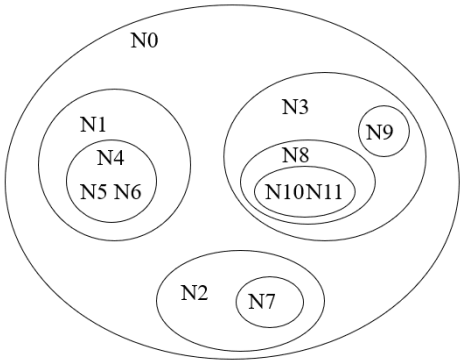
Gambar 8.10 menunjukkan sebuah contoh binary tree yang digunakan untuk menjelaskan representasi struktur tree. Struktur tersebut memiliki simpul root berada di N0. Simpul daun (leaf) terdiri atas N4, N5, N6, N7, N9, N10, dan N11, yaitu simpul-simpul yang tidak memiliki anak. Adapun simpul cabang (branch node) meliputi N1, N2, N3, dan N8, karena simpul-simpul tersebut masih memiliki turunan. Pohon ini juga dapat dibagi ke dalam

tiga subtree, yaitu T1 = N1, N4, N5, N6; T2 = N2, N7; dan T3 = N3, N8, N9, N10, N11. Selain itu, kedalaman pohon (depth) atau tingkat maksimum (max level) dalam struktur tersebut adalah 3, yang menunjukkan bahwa lintasan terpanjang dari akar menuju daun mencapai tiga level.

Jenis representasi tree yaitu:

♥ Diagram Venn

Diagram Venn lebih dikenal untuk menunjukkan himpunan dan irisan antar kelompok, dalam konteks tertentu diagram ini dapat digunakan untuk membantu menggambarkan hubungan pengelompokan bertingkat yang menyerupai struktur tree. Setiap lingkaran dapat mewakili kategori tertentu, sedangkan lingkaran di dalam menunjukkan keterkaitan dan posisi bagian terhadap keseluruhan. Namun demikian, representasi ini kurang tepat untuk menunjukkan hubungan induk-anak secara eksplisit dibandingkan bentuk tree lainnya, karena Diagram Venn lebih menonjolkan relasi keanggotaan dan irisan daripada arah cabang hierarkis.



Gambar 8.11: Representasi Tree dengan Diagram Venn

Gambar 8.11 merupakan representasi tree dengan Diagram Venn. Diagram Venn dari suatu tree digambarkan dengan cara **menempatkan setiap simpul anak di dalam lingkaran yang merepresentasikan simpul induknya**. Lingkaran terbesar bertanda **N0** menggambarkan bahwa simpul ini merupakan akar yang mencakup seluruh simpul lain di dalam tree. Di dalam lingkaran **N0** terdapat tiga kelompok utama yang mewakili subtree yaitu **N1**, **N2**, dan **N3**. Lingkaran **N1** memuat **N4, N5,** dan **N6**, yang berarti ketiga simpul tersebut merupakan turunan langsung dari **N1**. Lingkaran **N2** memuat **N7** menunjukkan bahwa **N7** adalah anak dari **N2**. Lingkaran **N3** memuat **N8** dan **N9**, lalu di dalam lingkaran **N8** masih terdapat **N10** dan **N11**, yang menandakan bahwa kedua simpul tersebut adalah anak dari **N8**.

Kelebihan representasi Diagram Venn adalah kemampuannya menonjolkan relasi inklusif antara induk dan anak, sehingga sangat berguna untuk menjelaskan konsep subtree dan cakupan simpul. Namun, dibandingkan tampilan tree biasa, representasi ini kurang menonjolkan urutan cabang kiri dan kanan serta tidak memperlihatkan garis hubungan secara

♥ Pedigree Chart

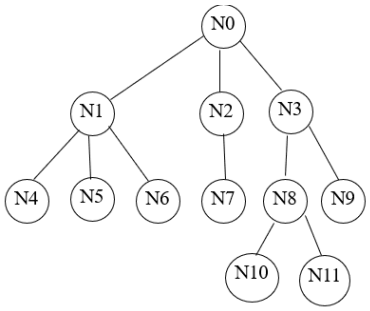
Pedigree Chart merupakan bentuk representasi tree yang umum digunakan untuk memperlihatkan garis keturunan, hubungan keluarga, atau pewarisan sifat. Dalam bagan ini, setiap simpul mewakili individu, sedangkan garis penghubung menunjukkan relasi biologis atau genealogis antaranggota. Pedigree Chart sangat efektif untuk memperlihatkan perkembangan generasi dari leluhur ke keturunan berikutnya secara runtut. Dalam perspektif struktur data, bentuk ini memperjelas konsep akar sebagai nenek moyang utama, cabang sebagai garis keturunan, serta daun sebagai anggota generasi terakhir. Oleh karena itu, Pedigree Chart banyak digunakan tidak hanya dalam studi keluarga, tetapi juga dalam genetika, biologi, dan analisis pewarisan karakteristik tertentu.

Dalam penyusunan Pedigree Chart, posisi vertikal menunjukkan hubungan generasi atau level kedalaman, sedangkan posisi horizontal digunakan untuk memisahkan simpul-simpul yang berada di tingkat yang sama agar tidak saling bertumpang tindih. Struktur tree yang semula bersifat abstrak dapat divisualisasikan sebagai bagan

yang lebih mudah dibaca. Cara menyusun Pedigree Chart dari tree dilakukan dengan mengubah struktur bercabang tree menjadi bagan bertingkat yang menyerupai silsilah, berikut langkahnya yaitu:

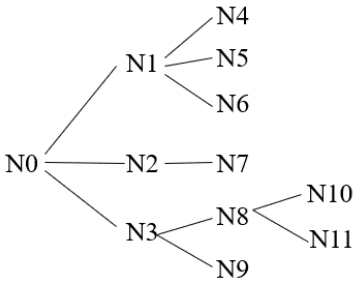
- ♣ Langkah pertama adalah menentukan root sebagai simpul paling atas, karena simpul ini menjadi titik asal seluruh hubungan dalam tree.
- ♣ Langkah kedua adalah menempatkan seluruh anak langsung dari root di baris atau tingkat di bawahnya secara sejajar.
- ♣ Langkah ketiga, untuk setiap simpul yang memiliki anak, seluruh anak tersebut ditempatkan tepat di bawah simpul induknya dan dihubungkan dengan garis.
- ♣ Proses ini diulangi terus untuk setiap tingkat hingga semua simpul daun terletak di bagian terbawah bagan.

Gambar 8.12 merupakan representasi Pedigree Chart dari tree. Struktur tree disajikan menyerupai bagan



Gambar 8.12: Representasi Tree dengan Pedigree Chart

silsilah, sehingga hubungan antar simpul tampak lebih menyerupai hubungan generasional dari satu induk ke beberapa keturunan. Simpul N0 tetap ditempatkan di posisi paling atas sebagai pusat asal seluruh cabang. Simpul N1, N2, dan N3 diletakkan sejajar untuk menunjukkan bahwa ketiganya merupakan anak langsung dari N0. Selanjutnya, keturunan dari masing-masing simpul ditata di tingkat bawah simpul induknya, yakni N4, N5, dan N6 di bawah N1; N7 di bawah N2; serta N8 dan N9 di bawah N3. Karena N8 masih memiliki anak, maka N10 dan N11 ditempatkan lagi di tingkat berikutnya di bawah N8.



Gambar 8.13: Representasi Tree dengan Linier Chart

♥ Linear chart

Linear Chart merupakan representasi tree yang menyusun simpul-simpul dalam bentuk urutan garis atau daftar terstruktur. Representasi ini biasanya digunakan ketika hubungan hierarkis perlu ditampilkan secara sederhana tanpa banyak percabangan visual yang kompleks. Setiap elemen dalam Linear Chart diletakkan berurutan dengan penanda tertentu, misalnya indentasi, simbol, atau nomor, untuk menunjukkan posisi relatif terhadap induknya. Kelebihannya adalah kesederhanaannya, sehingga mudah dibaca dalam teks atau dokumen yang tidak memungkinkan penggunaan gambar bercabang. Meskipun demikian, ketika jumlah simpul dan tingkat

kedalaman semakin besar, Linear Chart dapat menjadi kurang intuitif karena pembaca harus lebih cermat menafsirkan tingkat hierarkinya dari format penulisan.

Cara menyusun Linier Chart dari suatu tree dilakukan dengan menempatkan simpul root sebagai titik awal di sisi kiri, kemudian menggambar seluruh anak langsungnya ke arah kanan sebagai cabang pertama. Setelah itu, untuk setiap simpul yang memiliki anak, simpul-simpul turunannya kembali ditempatkan di sebelah kanan simpul tersebut dan dihubungkan dengan garis. Proses ini dilakukan secara berulang hingga seluruh simpul daun tergambar. Dalam penyusunannya, anak-anak dari satu simpul biasanya ditempatkan berdekatan agar menunjukkan bahwa asalnya dari induk yang sama, sedangkan subtree yang berbeda diberi jarak secukupnya supaya struktur tetap mudah dibaca. Dengan cara tersebut, hubungan hierarkis dalam tree diubah menjadi pola yang lebih memanjang, namun relasi induk-anak tetap terjaga. Jadi, prinsip penyusunan Linier Chart dari suatu tree yaitu setiap simpul anak digambarkan di sebelah kanan simpul induknya sesuai urutan percabangan, sehingga keseluruhan struktur pohon dapat divisualisasikan dalam bentuk yang sederhana, sistematis, dan efisien.

Gambar 8.13 menunjukkan representasi tree dalam bentuk Linier Chart, yaitu bentuk penyajian struktur pohon yang disusun secara memanjang dari kiri ke kanan dengan tetap mempertahankan hubungan hierarkis antar simpul. Simpul N0 ditempatkan sebagai root yang berada di sisi paling kiri. Dari simpul ini terbentuk tiga cabang utama yang masing-masing menuju ke simpul N1, N2, dan N3. Simpul N1 kemudian bercabang lagi ke tiga simpul, yaitu N4, N5, dan N6. Simpul N2 hanya memiliki satu cabang menuju N7. Sementara itu, simpul N3 bercabang ke dua simpul, yaitu N8 dan N9, lalu simpul N8 masih memiliki dua cabang lanjutan menuju N10 dan N11. Gambar tersebut memperlihatkan susunan hubungan induk-anak secara berurutan, di mana arah percabangan bergerak dari kiri sebagai asal struktur menuju kanan sebagai turunan-turunannya.

♥ Nested Parentheses

Struktur hierarkis dalam Nested Parentheses dituliskan menggunakan tanda kurung bertingkat, di mana suatu simpul diikuti oleh simpul-simpul anak yang diletakkan di dalam pasangan kurung. Representasi ini bersifat ringkas dan formal, sehingga sangat sesuai untuk keperluan komputasional, penulisan ekspresi struktur, serta pemrosesan sintaksis. Nested Parentheses memungkinkan hubungan induk-anak dan subtree dinyatakan secara eksplisit dalam satu baris atau beberapa baris teks. Keunggulannya adalah efisiensi penyimpanan dan kemudahan pemrosesan oleh program komputer, tetapi kelemahannya yaitu keterbacaan visual yang menurun apabila struktur tree sangat besar atau memiliki banyak tingkat kedalaman.

Cara menyusun Nested Parentheses dari suatu tree dilakukan dengan mengikuti hubungan hierarkis dari akar hingga ke simpul daun. Langkah pertama adalah menuliskan simpul root terlebih dahulu. Setelah itu, setiap anak dari simpul tersebut dituliskan di dalam tanda kurung setelah nama simpul induknya. Jika suatu simpul anak masih memiliki turunan, maka nama simpul itu diikuti lagi oleh tanda kurung baru yang berisi anak-anaknya. Proses ini dilakukan secara rekursif sampai seluruh simpul daun selesai dituliskan. Simpul daun cukup dituliskan sebagai nama simpul di dalam kurung tanpa tambahan turunan lagi, karena simpul tersebut tidak memiliki anak. Dengan demikian, struktur tree diterjemahkan menjadi susunan ekspresi bertingkat yang setiap lapisannya menunjukkan hubungan antara simpul induk dan simpul anak.

Gambar 8.14 menunjukkan representasi tree dalam bentuk Nested Parentheses, yaitu cara penyajian struktur pohon dengan menggunakan tanda kurung bertingkat untuk menunjukkan hubungan hierarkis antar

(N0(N1(N4)(N5)(N6))(N2(N7))
(N3(N8(N10)(N11))(N9)))

Gambar 8.14: Representasi Tree dengan Nested Parentheses

simpul. Dalam representasi ini, simpul root dituliskan terlebih dahulu, kemudian seluruh simpul anaknya ditempatkan di dalam tanda kurung setelah simpul induk tersebut. Penyusunan Nested Parentheses dimulai dengan menuliskan N0 sebagai akar. Selanjutnya, karena N0 memiliki tiga anak, maka setelah N0 dituliskan tiga

kelompok berturut-turut, yaitu (N1(N4)(N5)(N6)), (N2(N7)), dan (N3(N8(N10)(N11))(N9)). Kelompok pertama, N1 diikuti tiga pasangan kurung yang memuat N4, N5, dan N6 sebagai anak-anaknya. Kelompok kedua, N2 diikuti satu pasangan kurung yang memuat N7. Kelompok ketiga, N3 diikuti dua kelompok anak, yaitu N8 dan N9; kemudian karena N8 masih memiliki dua anak, maka setelah N8 ditambahkan lagi dua pasangan kurung yang memuat N10 dan N11. Hasil akhir dari proses tersebut adalah notasi bertingkat yang padat, tetapi tetap merepresentasikan seluruh struktur tree secara utuh dan akurat.

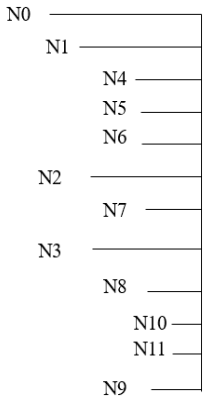
♥ Bar Chart

Bar Chart dapat dimanfaatkan untuk menunjukkan level dalam struktur tree melalui batang-batang yang tersusun menurut kategori atau tingkatan tertentu. Berbeda dengan representasi cabang yang memperlihatkan hubungan langsung antar simpul, Bar Chart lebih menekankan perbandingan kuantitatif antarbagian dalam suatu struktur bertingkatBentuk ini lebih tepat digunakan ketika tujuan penyajian bukan hanya menunjukkan hierarki, tetapi juga mengomunikasikan proporsi, frekuensi, atau ukuran dari setiap bagian dalam tree. Bar Chart berfungsi sebagai representasi pendukung yang memperjelas distribusi data dalam struktur hierarkis.

Cara menyusun Bar Chart dari suatu tree dilakukan dengan mengubah hubungan bercabang tree menjadi susunan garis vertikal dan horizontal yang merepresentasikan level dan keterhubungan simpul. Prinsip dasarnya adalah setiap simpul anak harus diletakkan sesuai induknya, sedangkan kedalaman tree dinyatakan melalui pergeseran posisi ke bawah dan ke kanan. Berikut langkah-langkahnya:

- Langkah pertama adalah menempatkan simpul root di bagian paling atas sebagai titik awal representasi.
- Setelah itu, dibuat garis vertikal utama di sisi kanan sebagai sumbu yang akan menghubungkan simpul-simpul dalam struktur tree.
- Menempatkan seluruh anak langsung dari akar di tingkat bawahnya, lalu masing-masing dihubungkan ke garis vertikal utama dengan garis horizontal.
- Setiap simpul yang masih memiliki anak, dibuat lagi garis-garis horizontal tambahan di posisi yang sesuai, dengan penempatan simpul anak sedikit lebih ke kanan atau lebih rendah agar tingkat kedalamannya terbaca dengan jelas.
- Proses ini diulangi sampai seluruh simpul daun selesai ditampilkan.

Gambar 8.15 menunjukkan representasi tree dalam bentuk Bar Chart, yaitu penyajian struktur pohon secara ringkas dengan menggunakan satu garis vertikal utama dan sejumlah garis horizontal sebagai penanda



Gambar 8.15: Representasi Tree dengan Bar Chart

percabangan antar simpul. Penyusunan Bar Chart dimulai dengan menempatkan N0 di bagian paling atas. Setelah itu, N1, N2, dan N3 ditempatkan sebagai turunan langsung di tingkat berikutnya. Kemudian, di bawah N1 ditambahkan N4, N5, dan N6; di bawah N2 ditambahkan N7; dan di bawah N3 ditambahkan N8 serta N9. Karena N8 masih memiliki anak, maka N10 dan N11 ditempatkan lagi di tingkat berikutnya sebagai turunan dari N8.

Hasil akhirnya adalah diagram batang yang menunjukkan bahwa semakin dalam posisi suatu simpul di dalam tree, semakin lanjut pula letaknya di susunan garis. Dengan demikian, Bar Chart dari tree disusun dengan mempertahankan urutan hierarkis, tetapi menyajikannya dalam bentuk yang lebih ringkas, sistematis, dan mudah dibaca.

♥ Level-Number Chart

Level-Number Chart menampilkan struktur tree berdasarkan tingkat (level) dan penomoran simpul. Setiap simpul diberi nomor tertentu sesuai urutan kemunculannya, lalu dikelompokkan menurut level kedalamannya dari akar. Bentuk ini sangat bermanfaat dalam analisis struktur data karena memudahkan identifikasi posisi node, hubungan antarlevel, dan perhitungan karakteristik seperti tinggi pohon, jumlah simpul untuk setiap level, serta pola penyebaran cabang. Level-Number Chart juga sering digunakan dalam pembelajaran struktur data karena memberikan gambaran yang sistematis dan terukur mengenai organisasi tree. Dibandingkan representasi visual bercabang, bentuk ini lebih abstrak, tetapi sangat membantu ketika fokus analisis diarahkan dalam susunan level dan keteraturan numerik.

Cara menyusun Level-Number Chart dari suatu tree dilakukan dengan menentukan terlebih dahulu level setiap simpul berdasarkan jaraknya dari akar. Simpul root diberi level 1. Setiap anak langsung dari akar diberi

```
1 N0
2 N1
3 N4
3 N5
3 N6
2 N2
3 N7
2 N3
3 N8
4 N10
4 N11
3 N9
```

Gambar 8.16: Representasi Tree dengan Level-Number Chart

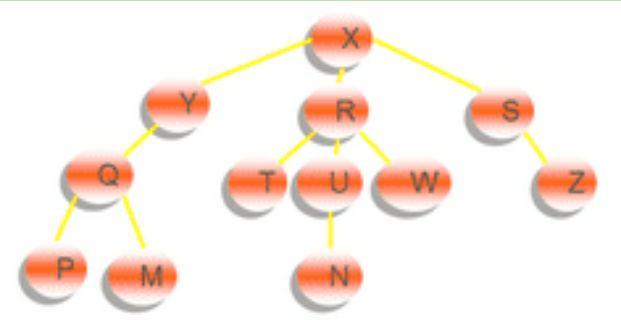
level 2, anak dari level 2 diberi level 3, dan seterusnya hingga seluruh simpul memperoleh nomor tingkatnya. Setelah itu, simpul-simpul dituliskan secara berurutan sesuai alur penelusuran tree, umumnya dimulai dari akar, lalu diikuti subtree pertama, subtree berikutnya, dan seterusnya. Setiap baris, nomor level ditulis lebih dahulu, kemudian diikuti nama simpul. Dengan cara ini, setiap entri dalam daftar memuat dua informasi penting sekaligus, yaitu identitas simpul dan kedudukannya dalam hierarki.

Gambar 8.16 menunjukkan representasi tree dalam bentuk Level-Number Chart, yaitu penyajian struktur pohon dengan menuliskan setiap simpul bersama nomor levelnya. Angka yang terletak di sebelah kiri nama simpul menyatakan tingkat kedalaman simpul dalam tree. Simpul root N0 diberi nomor 1, yang menandakan bahwa simpul tersebut berada di level pertama. Selanjutnya, simpul N1, N2, dan N3 diberi nomor 2 karena ketiganya merupakan turunan langsung dari akar dan berada satu tingkat di bawahnya. Simpul-simpul N4, N5, N6, N7, N8, dan N9 diberi nomor 3 karena semuanya berada di tingkat berikutnya. Adapun simpul N10 dan N11 diberi nomor 4, yang menunjukkan bahwa keduanya terletak dalam level terdalam dalam struktur tersebut.

Keunggulan utama yaitu kesederhanaan dan efisiensinya. Gambar Level-Number Chart tidak membutuhkan ruang visual yang besar, tetapi tetap mampu menunjukkan organisasi hierarkis tree secara jelas. Nomor level memudahkan pembaca untuk segera mengetahui posisi relatif suatu simpul dalam struktur, sedangkan urutan penulisan membantu mengidentifikasi kelompok subtree. Dengan melihat data gambar, dapat dipahami bahwa N0 berada di level tertinggi, N1, N2, dan N3 berada di level kedua, simpul-simpul berikutnya tersebar di level ketiga, dan N10 serta N11 mencapai level keempat.

Program tree dengan berbagai representasi ditampilkan di https://colab.research.google.com/drive/1u2NWPNISglAak3s6fS7IQKYavtkg41Ys?usp=sharing.

Latihan:
SOAL 1:



Buat representasi tree di atas :
diagram venn, linear chart, nested parentheses, bar chart, level-number chart.

SOAL 2:
Buat code tentang tree dan representasi tree dari soal no 1.

8.3 Penelusuran Tree (Traversal)

Traversal tree merupakan metode untuk mengunjungi seluruh simpul yang terdapat dalam suatu tree. Dalam struktur data linear, proses penelusuran elemen data relatif sederhana karena seluruh elemen disusun secara berurutan, sehingga setiap elemen cukup dikunjungi satu kali sesuai urutan penyimpanannya. Sebaliknya, dalam struktur data non-linear seperti tree dan graph, proses penelusuran memerlukan algoritma khusus karena hubungan antar elemen tidak tersusun secara sekuensial, melainkan bercabang. Untuk memahami konsep traversal, penelusuran dapat dimulai dari subtree kiri dalam sebuah binary tree. Proses tersebut dilakukan dengan memulai dari simpul akar, kemudian simpul dikunjungi atau ditampilkan, lalu penelusuran bergerak ke simpul paling kiri berikutnya. Langkah ini dilakukan secara berulang hingga seluruh bagian subtree kiri selesai ditelusuri.

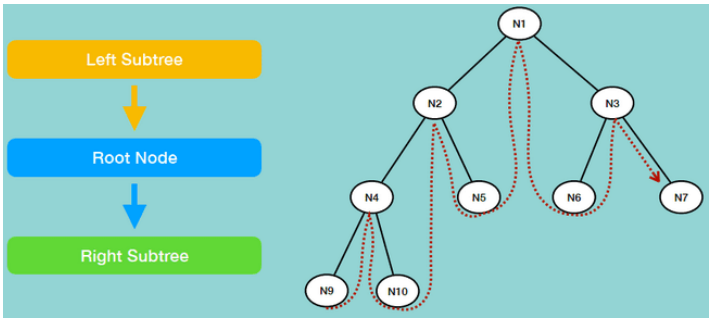
Secara umum, terdapat beberapa cara untuk memproses dan menelusuri tree, yang dibedakan berdasarkan urutan kunjungan terhadap simpul akar, subtree kiri, dan subtree kanan. Terdapat dua pendekatan utama dalam traversal tree. Pendekatan pertama dilakukan dengan memulai dari suatu simpul, kemudian menelusuri seluruh anak yang tersedia sebelum berlanjut ke simpul saudara berikutnya. Pendekatan ini memiliki tiga variasi utama, yaitu **in-order**, **pre-order**, dan **post-order**. Pendekatan kedua dilakukan dengan memulai dari simpul akar, kemudian mengunjungi seluruh simpul di setiap level secara bertahap, sehingga penelusuran berlangsung dari level atas ke level bawah. Pendekatan ini menekankan pemrosesan simpul berdasarkan tingkat kedalamannya dalam tree.

⌘ In-Order Traversal

Traversal **in-order** dalam tree dilakukan dengan urutan **penelusuran subtree kiri, kemudian simpul akar, dan terakhir subtree kanan**. Mekanisme ini berlangsung secara rekursif, artinya proses yang sama diterapkan berulang untuk setiap subtree hingga seluruh simpul selesai dikunjungi. Secara umum, traversal in-order terdiri atas tiga langkah utama (langkah ini diilustrasikan dalam Gambar 8.17), yaitu:

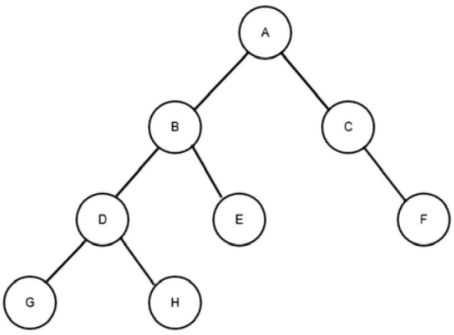
1. Menelusuri subtree kiri terlebih dahulu dengan memanggil fungsi penelusuran secara rekursif.
2. Kemudian mengunjungi simpul akar.

3. Selanjutnya menelusuri subtree kanan secara rekursif.



Gambar 8.17: Ilustrasi In-Order Traversal (Karimov, 2022)

Pola dasar traversal ini dapat diringkas sebagai urutan **kiri–akar–kanan**. Metode ini banyak digunakan dalam binary tree karena mampu menampilkan simpul dalam urutan yang sistematis sesuai struktur pohonnya.



Gambar 8.18: Contoh Binary Tree dengan In-Order Traversal (Agarwal, 2022)

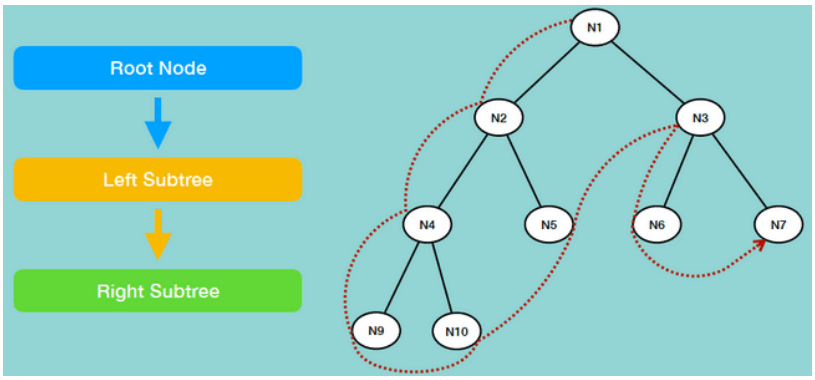
Sebagai ilustrasi Gambar 8.18, penelusuran in-order dimulai dengan mengunjungi subtree kiri dari simpul root **A** secara rekursif. Subtree kiri tersebut, simpul **B** menjadi akar, sehingga penelusuran kembali diarahkan ke subtree kiri dari **B**, yaitu simpul **D**. Dari simpul **D**, penelusuran dilanjutkan lagi ke subtree kirinya hingga mencapai simpul **G**. Setelah simpul **G** dikunjungi, proses kembali ke simpul **D** untuk mengunjungi akar tersebut, lalu dilanjutkan ke anak kanannya, yaitu **H**. Sesudah seluruh bagian subtree **D** selesai ditelusuri, penelusuran kembali ke simpul **B**, kemudian dilanjutkan ke anak kanannya, yaitu **E**. Dengan demikian, seluruh subtree kiri dari simpul **A** telah selesai dikunjungi. Tahap berikutnya adalah mengunjungi simpul root **A**, lalu penelusuran berlanjut ke subtree kanan dari **A**, yaitu subtree yang berakar di **C**. Bagian ini, subtree kiri dari **C** tidak memiliki simpul, sehingga simpul **C** langsung dikunjungi, kemudian dilanjutkan ke anak kanannya, yaitu **F**. Berdasarkan urutan tersebut, hasil traversal in-order dalam tree tersebut adalah **G-D-H-B-E-A-C-F**.

Program tree dengan in-order traversal dapat dilihat di https://colab.research.google.com/drive/14P7E2Sts7wF-5oHLDdNAn8kGxJm6NGkj?usp=sharing.

Pre-Order Traversal

Traversal **pre-order** dalam tree dilakukan dengan urutan kunjungan **simpul root, subtree kiri, kemudian subtree kanan**. Proses Traversal pre-order dimulai dari (langkah ini diilustrasikan dalam Gambar 8.19):

1. Root sebagai simpul yang pertama dikunjungi.
2. Setelah itu dilanjutkan dengan menelusuri seluruh bagian subtree kiri secara rekursif.
3. Akhirnya menelusuri subtree kanan secara rekursif pula.



Gambar 8.19: Ilustrasi Pre-Order Traversal (Karimov, 2022)

Oleh karena itu, pola dasar traversal pre-order dapat diringkas sebagai **root–kiri–kanan**. Metode ini menempatkan simpul induk untuk diproses lebih dahulu sebelum semua turunannya, sehingga sangat sesuai digunakan ketika struktur tree perlu dibaca mulai dari elemen utama menuju percabangan-percabangan berikutnya.

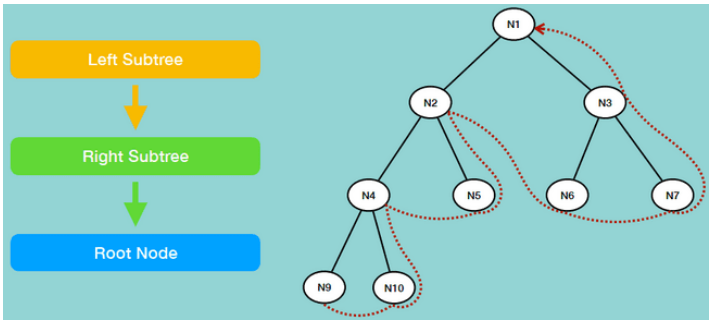
Berdasarkan contoh binary tree, traversal pre-order dimulai dengan mengunjungi simpul root **A**. Setelah itu, penelusuran bergerak ke subtree kiri dari **A**, yaitu subtree yang berakar di **B**. Simpul **B** dikunjungi terlebih dahulu, kemudian penelusuran dilanjutkan ke subtree kirinya, yaitu simpul **D**. Selanjutnya, simpul **D** juga dikunjungi lebih dahulu, lalu penelusuran diteruskan ke subtree kirinya hingga mencapai simpul **G**. Karena simpul **G** tidak memiliki anak, proses kembali ke simpul **D** dan berlanjut ke subtree kanannya, yaitu simpul **H**. Setelah seluruh subtree dari **D** selesai, penelusuran kembali ke simpul **B** dan dilanjutkan ke anak kanannya, yaitu **E**. Dengan demikian, bagian kiri dari akar **A** telah selesai ditelusuri. Tahap berikutnya adalah menelusuri subtree kanan dari **A**, yaitu subtree yang berakar di **C**. Simpul **C** dikunjungi terlebih dahulu, kemudian karena subtree kirinya kosong, penelusuran langsung dilanjutkan ke subtree kanannya, yaitu **F**. Berdasarkan urutan tersebut, hasil traversal pre-order dalam tree tersebut adalah **A-B-D-G-H-E-C-F**.

Program tree dengan pre-order traversal dapat dilihat di https://colab.research.google.com/drive/1FUgYB1uauJ3oYQ4hrB8XxSwLUY7Nj3Lp?usp=sharing.

Post-Order Traversal

Traversal **post-order** dalam tree dilakukan dengan urutan kunjungan **subtree kiri, subtree kanan, kemudian simpul root**. Mekanisme langkah post-order berikut berlangsung secara rekursif, (langkah ini diilustrasikan dalam Gambar 8.20), yaitu:

1. Penelusuran selalu dimulai dari subtree kiri suatu simpul.
2. Dilanjutkan ke subtree kanan.
3. Simpul root baru dikunjungi setelah seluruh turunannya selesai diproses.



Gambar 8.20: Ilustrasi Post-Order Traversal (Karimov, 2022)

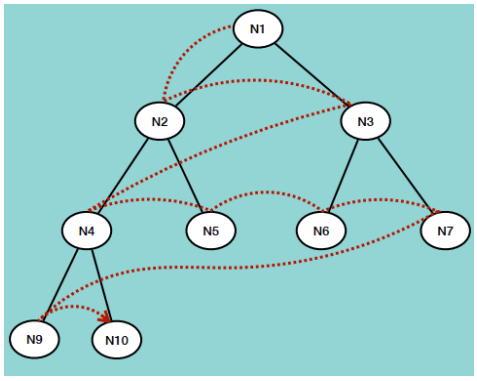
Pola dasar traversal post-order dapat diringkas sebagai kiri–kanan–akar. Metode ini menempatkan simpul induk di tahap akhir penelusuran, sehingga sangat sesuai digunakan ketika pemrosesan suatu simpul bergantung dari selesainya pemrosesan seluruh anak atau subtree yang berada di bawahnya.

Berdasarkan contoh binary tree, traversal post-order dimulai dengan menelusuri subtree kiri dari simpul root **A** secara rekursif. Penelusuran bergerak hingga mencapai bagian terdalam dari subtree kiri, yaitu simpul **D**, kemudian mengunjungi anak kirinya, yaitu **G**. Setelah itu, penelusuran berlanjut ke anak kanan dari **D**, yaitu **H**, dan setelah kedua anak tersebut selesai dikunjungi, simpul **D** baru diproses. Selanjutnya, aturan yang sama diterapkan di simpul **B**, sehingga setelah subtree kiri selesai, penelusuran berpindah ke anak kanannya, yaitu **E**, dan kemudian simpul **B** dikunjungi. Setelah seluruh subtree kiri dari **A** selesai, penelusuran dilanjutkan ke subtree kanan dari **A**. Penelusuran mencapai simpul paling kanan terdalam, yaitu **F**, kemudian kembali untuk mengunjungi simpul **C**. Akhirnya, setelah seluruh subtree kiri dan kanan dari akar selesai diproses, simpul root **A** dikunjungi di tahap terakhir. Berdasarkan urutan tersebut, hasil traversal post-order dalam tree tersebut adalah **G-H-D-E-B-F-C-A**.

Program tree dengan post-order traversal dapat dilihat di https://colab.research.google.com/drive/1Otbd5zqgkGONzN7RSmGOTxGW2LOl0RS6?usp=sharing.

Level-Order Traversal

Traversal level-order merupakan metode penelusuran tree yang dilakukan dengan mengunjungi simpul root terlebih dahulu, kemudian seluruh simpul level berikutnya, lalu berlanjut secara bertahap ke level-level selanjutnya hingga seluruh simpul dalam tree selesai diproses. **Penelusuran ini berlangsung berdasarkan urutan tingkat kedalaman, dimulai dari tingkat paling atas menuju tingkat yang lebih bawah**. Pola kerja traversal ini sejalan dengan konsep breadth-first traversal dalam graph, yaitu memperluas penelusuran seluruh simpul dalam satu level sebelum bergerak lebih dalam ke level berikutnya. Oleh karena itu, level-order traversal menekankan keluasan cakupan setiap tingkat tree, bukan penelusuran mendalam untuk satu cabang tertentu. Langkah ini diilustrasikan dalam Gambar 8.21.

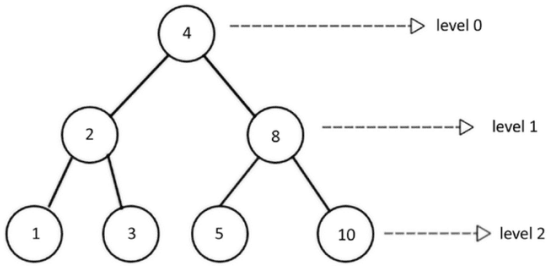


Gambar 8.21: Ilustrasi Level-Order Traversal (Karimov, 2022)

Sebagai contoh, tree yang ditunjukkan dalam Gambar 8.22, proses traversal dimulai dengan mengunjungi simpul root level 0, yaitu simpul bernilai 4. Setelah nilai simpul tersebut diproses, penelusuran berlanjut ke level 1 dengan mengunjungi seluruh simpul tingkat tersebut, yakni simpul bernilai 2 dan 8. Selanjutnya, penelusuran diteruskan ke level berikutnya, yaitu level yang memuat simpul 1, 3, 5, dan 10, sehingga semua simpul tingkat tersebut dikunjungi secara berurutan. Berdasarkan urutan tersebut, hasil traversal level-order tree tersebut adalah 4, 2, 8, 1, 3, 5, dan 10. Contoh ini memperlihatkan bahwa traversal level-order selalu memproses simpul berdasarkan kelompok levelnya, dimulai dari akar hingga level terdalam.

Implementasi traversal level-order umumnya menggunakan struktur data queue. Prosesnya dimulai dengan mengunjungi simpul akar dan memasukkannya ke dalam queue. Simpul yang berada di bagian depan queue kemudian diambil (dequeue) untuk diproses, misalnya ditampilkan atau disimpan untuk keperluan lain.

Setelah simpul akar dimasukkan, anak kiri dari simpul tersebut dimasukkan terlebih dahulu ke dalam queue, kemudian diikuti oleh anak kanannya. Dengan mekanisme ini, seluruh simpul suatu level akan lebih dahulu masuk ke queue dari kiri ke kanan sebelum dikunjungi satu per satu. Proses tersebut diulangi untuk setiap level hingga queue menjadi kosong, yang menandakan bahwa seluruh simpul dalam tree telah selesai ditelusuri.



Gambar 8.22: Contoh Binary Tree dengan Level-Order Traversal (Agarwal, 2022)

Jika algoritma tersebut diterapkan terhadap tree sebelumnya, maka simpul akar 4 mula-mula dimasukkan ke dalam queue, kemudian dikeluarkan untuk dikunjungi. Setelah itu, simpul 2 dan 8 dimasukkan ke queue sebagai anak kiri dan anak kanan di level berikutnya. Simpul 2 kemudian dikeluarkan untuk diproses, lalu anak kiri dan kanannya, yaitu 1 dan 3, dimasukkan ke dalam queue. Tahap berikutnya, simpul yang berada di depan queue adalah 8, sehingga simpul ini dikeluarkan dan dikunjungi, lalu anak-anaknya juga dimasukkan ke dalam queue. Langkah tersebut terus dilakukan sampai queue tidak lagi berisi simpul. Dalam implementasi Python, traversal breadth-first semacam ini dilakukan dengan memasukkan simpul root ke dalam queue dan mencatat simpul-simpul yang telah dikunjungi dalam sebuah daftar, sedangkan queue dikelola menggunakan kelas deque.

Program tree dengan traversal level-order dapat dilihat di https://colab.research.google.com/drive/1wdiG1cI16YGW-EmXbhRQWVzOtmqmE2dl?usp=sharing.

Secara umum, berbagai algoritma traversal tree dapat digunakan sesuai dengan kebutuhan aplikasi. Traversal in-order sangat bermanfaat ketika isi tree perlu diperoleh dalam urutan terurut. Prinsip ini juga dapat diterapkan untuk memperoleh urutan menurun dengan membalik pola penelusuran menjadi subtree kanan, akar, lalu subtree kiri, yang dikenal sebagai reverse in-order traversal. Sementara itu, traversal pre-order digunakan ketika simpul akar perlu diproses sebelum seluruh simpul daun, sedangkan traversal post-order lebih sesuai apabila simpul daun perlu diperiksa terlebih dahulu sebelum simpul akar. Dengan demikian, pemilihan metode traversal sangat bergantung tujuan pemrosesan data yang diinginkan.

Binary tree sendiri memiliki berbagai aplikasi penting dalam ilmu komputer yaitu:

- ✓ Struktur ini digunakan sebagai expression tree dalam compiler.
- ✓ Dimanfaatkan dalam Huffman coding untuk kompresi data.
- ✓ Diterapkan dalam binary search tree guna mendukung proses pencarian, penyisipan, dan penghapusan data secara efisien.
- ✓ Implementasi Priority Queue (PQ), yang memungkinkan pencarian dan penghapusan elemen minimum atau maksimum dalam waktu logaritmik dalam kasus terburuk.

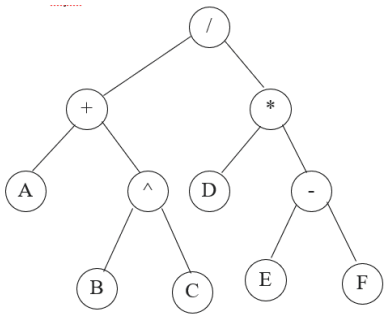
Hal tersebut menunjukkan bahwa binary tree bukan hanya penting dari sisi konsep struktur data, tetapi juga memiliki peran yang luas dalam berbagai aplikasi komputasi praktis.

Menganalisis Ekspresi Polish Nation dalam Tree

Untuk menganalisis ekspresi polish notation dalam struktur tree, digunakan algoritma traversal sebagai mekanisme penelusuran simpul. Setiap bentuk notasi memiliki keterkaitan dengan jenis traversal tertentu. Notasi prefix dianalisis menggunakan algoritma **pre-order, yaitu dengan urutan mencetak isi simpul yang sedang dikunjungi terlebih dahulu, kemudian mengunjungi anak kiri (*left child*), dan selanjutnya anak kanan (*right child*)**. Notasi infix dianalisis menggunakan algoritma **in-order, yaitu dengan urutan mengunjungi anak kiri terlebih dahulu, kemudian mencetak isi simpul yang sedang dikunjungi, lalu mengunjungi anak**

kanan. Sementara itu, notasi postfix dianalisis menggunakan algoritma **post-order, yaitu dengan urutan mengunjungi anak kiri, kemudian anak kanan, dan akhirnya mencetak isi simpul yang sedang dikunjungi**. Dengan demikian, perbedaan urutan traversal akan menghasilkan bentuk representasi ekspresi yang berbeda sesuai dengan jenis notasinya.

Gambar 8.23 menampilkan ekspresi polish nation berbentuk tree. Dari bentuk tree tersebut akan diubah



Gambar 8.23: Contoh Polish Nation Berbentuk Tree (Agarwal, 2022)

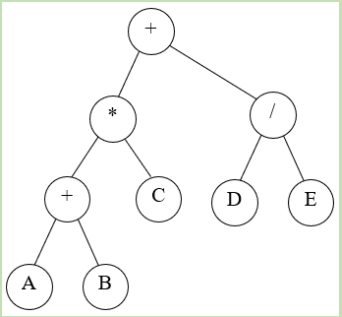
menjadi bentuk notasi infix, prefix, dan postfix. Apabila tree tersebut diubah ke dalam bentuk **infix**, maka digunakan traversal **in-order**, yaitu dengan urutan menelusuri subtree kiri, kemudian root, lalu subtree kanan. Dalam subtree kiri, simpul + membentuk ekspresi **A + (B ^ C)** karena anak kanannya berupa operator ^ dengan operand **B** dan **C**. Dalam subtree kanan, simpul * membentuk ekspresi **D * (E - F)** karena anak kanannya berupa operator - dengan operand **E** dan **F**. Setelah kedua subtree diperoleh, simpul akar / ditempatkan di antara keduanya. Dengan demikian, bentuk **infix** dari tree tersebut adalah **((A + (B ^ C)) / (D * (E - F)))**. Tanda kurung diperlukan agar urutan operasi yang tersimpan dalam tree tetap terjaga secara jelas.

Jika tree diubah ke dalam bentuk **prefix**, maka digunakan traversal **pre-order**, yaitu dengan urutan root, subtree kiri, kemudian subtree kanan. Penelusuran dimulai dari akar /, lalu bergerak ke subtree kiri sehingga diperoleh urutan **+ A ^ B C**, kemudian diteruskan ke subtree kanan sehingga diperoleh urutan *** D - E F**. Setelah seluruh hasil digabungkan sesuai urutan traversal, bentuk **prefix** yang dihasilkan adalah **/ + A ^ B C * D - E F**. Bentuk ini juga disebut notasi Polandia karena operator selalu dituliskan lebih dahulu daripada operand yang dioperasikannya.

Sementara itu, jika tree diubah ke dalam bentuk **postfix**, maka digunakan traversal **post-order**, yaitu dengan urutan subtree kiri, subtree kanan, kemudian root. Dalam subtree kiri, urutan traversal menghasilkan **A B C ^ +**, karena operand **B** dan **C** diproses terlebih dahulu sebelum operator ^, lalu hasilnya digabungkan dengan **A** menggunakan operator +. Dalam subtree kanan, traversal menghasilkan **D E F - ***, karena operand **E** dan **F** diproses terlebih dahulu sebelum operator **-**, lalu hasilnya digabungkan dengan **D** menggunakan operator *****. Selanjutnya, operator akar / ditempatkan di bagian akhir. Dengan demikian, bentuk **postfix** dari tree tersebut adalah **A B C ^ + D E F - * /**. Bentuk ini dikenal juga sebagai **Reverse Polish Notation**, karena operator dituliskan setelah operand.

Program merubah bentuk tree tersebut akan diubah menjadi bentuk notasi infix, prefix, dan postfix disajikan di https://colab.research.google.com/drive/1qxv_fv1hkM_fs753d6H_qtgUG_lJRL-Q?usp=sharing.

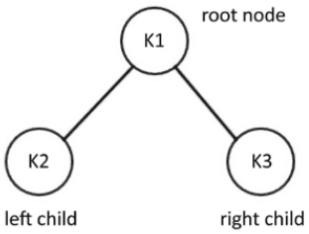
Latihan:
SOAL 1:
Buatlah notasi Infix, Prefix dan Postfixnya dari bentuk tree berikut.



SOAL 2:
Buat code perubahan notasi Infix, Prefix dan Postfixnya dari bentuk tree dalam soal no 1.

8.4 Binary Search Tree (BST)

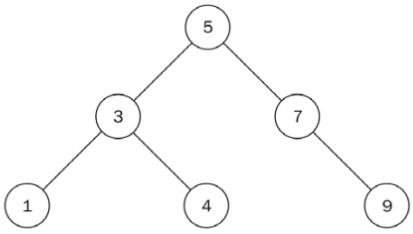
Binary search tree (BST) merupakan salah satu bentuk khusus dari **binary tree** yang banyak digunakan dalam berbagai aplikasi ilmu komputer. Struktur data ini dirancang untuk menyimpan data secara efisien di setiap simpul sehingga mampu mendukung proses pencarian, penyisipan, dan penghapusan dengan cepat. **Sebuah** binary tree **dikatakan sebagai** binary search tree **jika nilai di setiap simpul lebih besar daripada seluruh nilai yang berada di** subtree **kiri dan lebih kecil atau sama dengan seluruh nilai yang berada di** subtree **kanan**. Oleh karena itu, susunan nilai di setiap simpul harus selalu mengikuti aturan keterurutan tertentu. Sebagai contoh yang disajikan dalam Gambar 8.24, jika terdapat tiga nilai kunci, yaitu **K1, K2,** dan **K3**, maka agar memenuhi sifat BST, nilai **K2** harus lebih kecil atau sama dengan **K1**, sedangkan **K3** harus lebih besar daripada **K1**.



Gambar 8.24: Ilustrasi BST (Agarwal, 2022)

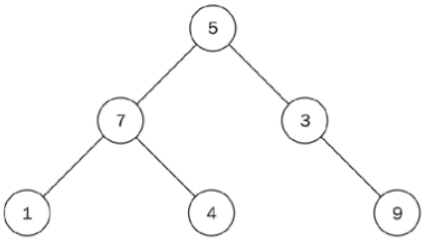
Agar konsep tersebut lebih mudah dipahami, dapat diperhatikan contoh sebuah **binary search tree** yang memperlihatkan bahwa seluruh simpul di **subtree** kiri selalu memiliki nilai yang lebih kecil atau sama dengan nilai simpul induknya, sedangkan seluruh simpul di **subtree** kanan selalu memiliki nilai yang lebih besar daripada nilai induknya. Sebagai ilustrasi yang ditampilkan di Gambar 8.25, jika simpul akar bernilai **5**, maka semua simpul di **subtree** kiri harus bernilai kurang dari **5**, sedangkan semua simpul di **subtree** kanan harus bernilai lebih besar dari **5**. Aturan ini tidak hanya berlaku untuk root, tetapi juga harus dipenuhi oleh semua simpul lain di dalam **tree** tanpa pengecualian. Misalnya, jika dipilih simpul bernilai **3**, maka seluruh simpul di **subtree** kirinya harus bernilai lebih kecil dari **3**, sedangkan seluruh simpul di **subtree** kanannya harus bernilai lebih besar dari **3**. Ketaatan terhadap aturan tersebut di seluruh simpul menjadi ciri utama yang membedakan BST dari **binary tree** biasa.

Meskipun demikian, tidak semua **binary tree** dapat digolongkan sebagai **binary search tree**, walaupun bentuk visualnya terlihat serupa. Sebuah **binary tree** tidak lagi memenuhi kriteria BST jika



Gambar 8.25: Contoh BST (Agarwal, 2022)

terdapat simpul yang melanggar aturan keterurutan nilai. Sebagai contoh yang disajikan di Gambar 8.26, terdapat simpul bernilai **7** yang berada di **subtree** kiri dari akar bernilai **5**, maka struktur tersebut tidak memenuhi sifat BST karena nilai di **subtree** kiri seharusnya lebih kecil atau sama dengan nilai akar. Hal yang sama juga terjadi jika simpul bernilai **4** ditempatkan di **subtree** kanan dari induk bernilai **7**, karena **subtree** kanan semestinya berisi nilai yang lebih besar daripada induknya. Dengan demikian, sebuah **binary tree** hanya dapat dinyatakan sebagai **binary search tree** jika seluruh simpul di dalamnya secara konsisten memenuhi aturan hubungan nilai antara induk, **subtree** kiri, dan **subtree** kanan.



Gambar 8.26: Contoh Binary Tree yang BUKAN BST (Agarwal, 2022)

Operasi-operasi dalam BST yaitu menyisipkan node, pencarian tree, menghapus node, menemukan node minimum dan maksimum.

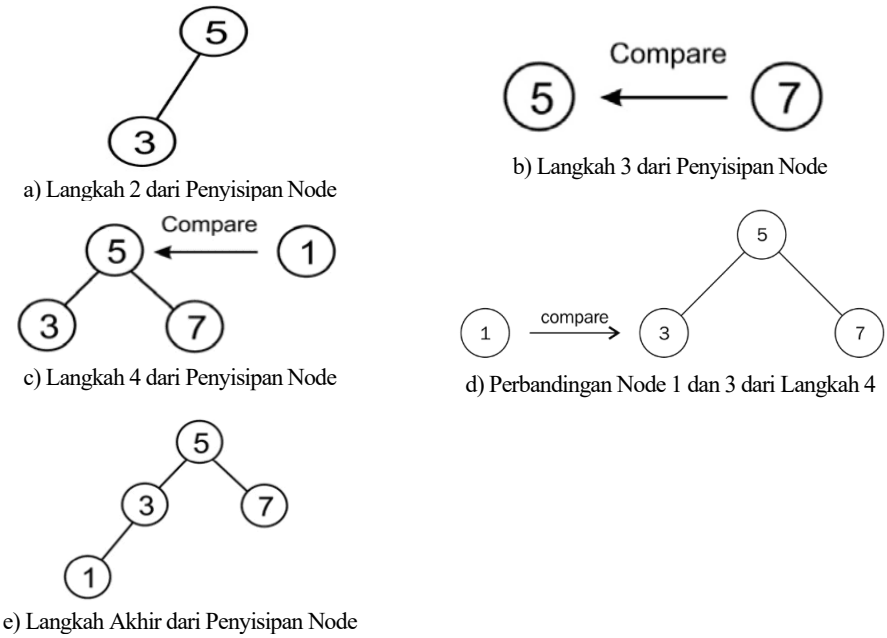
Menyisipkan Node (*Inserting*) dalam BST

Salah satu operasi utama dalam BST ialah penyisipan elemen ke dalam tree. Proses ini harus dilakukan secara cermat agar sifat dasar BST tetap terjaga setelah elemen baru ditambahkan**. Penyisipan elemen baru diawali dengan membandingkan nilai elemen tersebut dengan nilai simpul root. Jika nilainya lebih kecil daripada akar, elemen itu diarahkan ke subtree kiri. Sebaliknya, jika nilainya lebih besar atau sama, elemen itu diarahkan ke subtree kanan**. Perbandingan tersebut dilakukan berulang kali dari satu simpul ke simpul berikutnya hingga ditemukan posisi kosong yang sesuai. Setelah letak yang tepat ditemukan, elemen baru ditempatkan di bagian itu tanpa melanggar aturan BST.

```
Algoritma InsertBST(root, dataBaru):
1. Jika root = NULL maka
2.     root ← Node(dataBaru)
3.     return root
4. EndIf

5. Jika dataBaru < root.data maka
6.     root.left ← InsertBST(root.left, dataBaru)
7. Jika tidak maka
8.     root.right ← InsertBST(root.right, dataBaru)
9. EndIf
```

Sebagai ilustrasi yang disajikan di Gambar 8.27, pembentukan BST dapat dilakukan melalui penyisipan data 5, 3, 7, dan 1 secara berurutan, yaitu:

1. Penyisipan 5 dilakukan lebih dahulu karena elemen ini merupakan data pertama. Oleh sebab itu, simpul bernilai 5 langsung dibentuk sebagai root tree.



a) Langkah 2 dari Penyisipan Node
b) Langkah 3 dari Penyisipan Node
c) Langkah 4 dari Penyisipan Node
d) Perbandingan Node 1 dan 3 dari Langkah 4
e) Langkah Akhir dari Penyisipan Node

Gambar 8.27: Langkah Penyisipan Node (Agarwal, 2022)

2. Penyisipan 3 dilakukan dengan membandingkan nilai 3 terhadap nilai akar, yaitu 5. Karena 3 lebih kecil dari 5, simpul 3 ditempatkan di subtree kiri milik simpul 5. Susunan ini telah memenuhi aturan BST karena nilai di sisi kiri lebih kecil daripada nilai induknya.
3. Penyisipan 7 dilakukan dengan membandingkan nilai 7 dengan root 5. Karena 7 lebih besar dari 5, simpul 7 ditempatkan di subtree kanan milik simpul 5.
4. Penyisipan 1 dimulai dengan membandingkan nilai 1 dengan akar 5. Hasil perbandingan menunjukkan bahwa 1 lebih kecil dari 5, sehingga penelusuran dilanjutkan ke subtree kiri yang berisi simpul 3. Langkah berikutnya ialah membandingkan 1 dengan 3. Karena 1 lebih kecil dari 3, penelusuran bergerak lagi ke sisi kiri milik simpul 3. Karena posisi itu masih kosong, dibentuk simpul baru bernilai 1, lalu simpul tersebut ditempatkan sebagai anak kiri milik simpul 3.
5. Hasil akhir proses itu membentuk sebuah binary search tree dengan empat simpul, yaitu 5 sebagai akar, 3 di sisi kiri akar, 7 di sisi kanan akar, dan 1 di sisi kiri milik simpul 3.

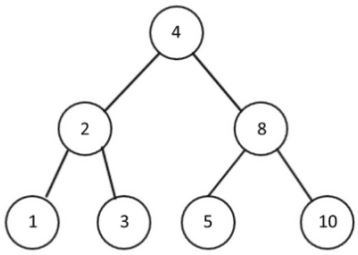
Contoh tersebut menggunakan data bertipe bilangan bulat. Namun, BST juga dapat digunakan untuk menyimpan data string. Dalam kasus string, proses perbandingan dilakukan berdasarkan urutan alfabetis, sehingga aturan penyisipannya tetap mengikuti prinsip keterurutan BST. Program inserting BST disajikan di https://colab.research.google.com/drive/13cq2yvzzVSKUv-K79dnPOqsPNTDl9-Oi?usp=sharing.

Pencarian Tree (*Searching*) dalam BST

Binary search tree merupakan struktur data berbentuk tree yang mengikuti aturan bahwa seluruh simpul di subtree kiri dari suatu simpul selalu memiliki nilai kunci lebih kecil, sedangkan seluruh simpul di subtree kanan memiliki nilai kunci lebih besar. Susunan seperti ini membuat proses pencarian elemen dengan nilai kunci tertentu menjadi lebih mudah dan efisien. Sebagai contoh di Gambar 8.28, dalam sebuah binary search tree yang berisi simpul 1, 2, 3, 4, 8, 5, dan 10, pencarian simpul bernilai 5 dilakukan dengan memulai penelusuran dari simpul akar. Nilai yang dicari terlebih dahulu dibandingkan dengan nilai akar, yaitu 4. Karena 5 lebih besar daripada 4, penelusuran diarahkan ke subtree kanan.

Algoritma SearchBST(root, key):
1. Jika root = NULL
   return "data tidak ditemukan"
2. Jika key = root.data
   return "data ditemukan"
3. Jika key < root.data
   return SearchBST(root.left, key)
4. Jika tidak
   return SearchBST(root.right, key)

Di bagian itu terdapat simpul 8 sebagai akar subtree kanan, sehingga nilai 5 dibandingkan dengan 8. Karena 5 lebih kecil daripada 8, penelusuran dilanjutkan ke subtree kiri. Selanjutnya ditemukan simpul bernilai 5, yang nilainya sama dengan nilai yang dicari. Kecocokan tersebut menunjukkan bahwa elemen berhasil ditemukan. Program pemcarian node dapat dilihat di https://colab.research.google.com/drive/1xXk3yb-HpsZIKdzB98XcX9VWni-_8QHj?usp=sharing.



Gambar 8.28: Pencarian BST (Agarwal, 2022)

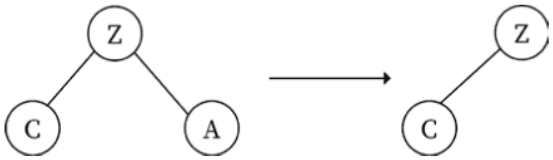
Menghapus Node (*Deleting*) dalam BST

Penghapusan simpul merupakan salah satu operasi utama dalam binary search tree. Proses ini mencakup tiga kemungkinan kondisi yang perlu diperhatikan, yaitu:

- ✓ Simpul yang tidak mempunyai anak. Jika simpul tersebut tidak mempunyai anak, simpul dapat langsung dihapus.
- ✓ Simpul dengan satu anak. Jika simpul mempunyai satu anak, nilai simpul itu dapat dipertukarkan dengan nilai anaknya, kemudian simpul tersebut dihapus.
- ✓ Simpul dengan dua anak. Adapun jika simpul mempunyai dua anak, penghapusan dilakukan dengan terlebih dahulu mencari in-order successor atau in-order predecessor, menukar nilainya dengan simpul yang akan dihapus, lalu menghapus simpul pengganti itu.

Kondisi pertama merupakan kasus yang paling sederhana. Jika simpul yang akan dihapus tidak mempunyai anak, simpul itu cukup dilepaskan dari induknya. Sebagai contoh Gambar 8.29, apabila sebuah simpul daun seperti A hendak dihapus, simpul tersebut dapat langsung dihilangkan dari hubungan dengan simpul induknya, misalnya Z. Karena simpul itu tidak mempunyai anak, proses penghapusan tidak memengaruhi struktur subtree lain dalam binary search tree.

Kondisi kedua terjadi saat simpul yang akan dihapus mempunyai satu anak. Dalam situasi ini, simpul induk



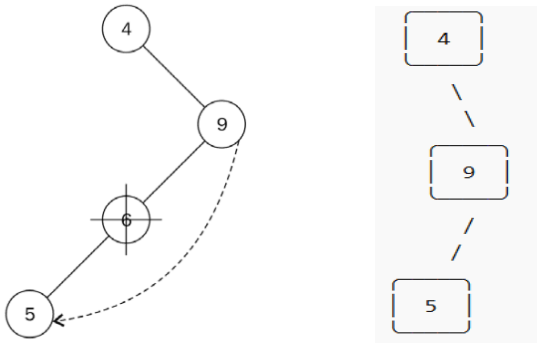
Gambar 8.29: Penghapusan Node yang Tidak mempunyai Anak (Agarwal, 2022)

dari simpul yang dihapus harus diarahkan langsung ke anak dari simpul tersebut. Sebagai ilustrasi Gambar 8.30, jika simpul 6 akan dihapus dan simpul itu hanya mempunyai satu anak, yaitu 5, maka penunjuk kiri milik simpul 9 diarahkan ke simpul 5. Langkah ini harus dilakukan sambil tetap menjaga agar hubungan antara induk dan anak sesuai dengan sifat keterurutan dalam binary search tree.

Kondisi ketiga muncul saat simpul yang akan dihapus mempunyai dua anak. Dalam keadaan seperti ini, penghapusan tidak dapat dilakukan secara langsung. Langkah pertama ialah mencari simpul pengganti, yaitu successor. Simpul successor merupakan simpul dengan nilai terkecil di subtree kanan dari simpul yang akan dihapus. Dengan kata lain, successor adalah elemen pertama yang diperoleh jika traversal in-order diterapkan ke

subtree kanan tersebut. Setelah successor ditemukan, isi simpul successor dipindahkan ke simpul yang akan dihapus, lalu simpul successor dihapus dari tree.

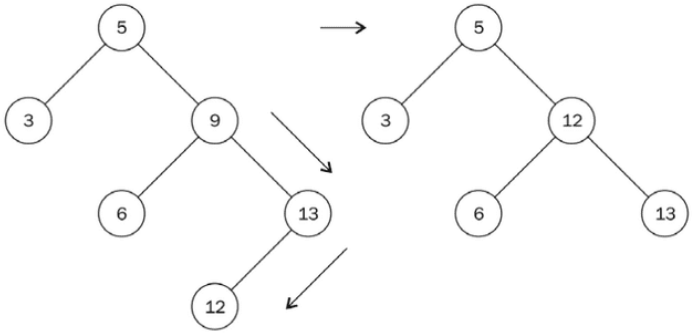
Sebagai contoh Gambar 8.31, jika simpul 9 yang mempunyai dua anak akan dihapus, maka terlebih dahulu



Gambar 8.30: Penghapusan Node dengan Satu Anak (Agarwal, 2022)

dicari elemen terkecil di subtree kanannya. Elemen terkecil tersebut ialah simpul 12. Setelah itu, nilai 9 diganti dengan nilai 12, kemudian simpul 12 dihapus. Karena simpul 12 tidak mempunyai anak, proses penghapusannya mengikuti aturan penghapusan simpul tanpa anak.

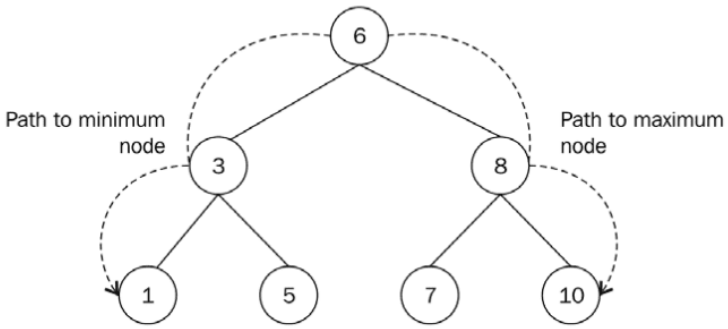
Program penghapusan node dalam BST disajikan di https://colab.research.google.com/drive/14gxQLRkCVfjwJfNB6tRJI2XKVqDtq14w?usp=sharing.



Gambar 8.31: Penghapusan Node dengan Dua Anak (Agarwal, 2022)

Menemukan Node Minimum Dan Maksimum (*Finding*) dalam BST

Pencarian simpul dengan nilai minimum dan maksimum dalam binary search tree dapat dilakukan dengan sangat mudah karena struktur tree ini telah tersusun secara terurut. **Untuk menemukan simpul yang memiliki nilai terkecil, penelusuran dimulai dari root lalu terus bergerak ke simpul paling kiri hingga mencapai**



Gambar 8.32: Penemuan Node Minimum dan Maksimum dalam BST (Agarwal, 2022)

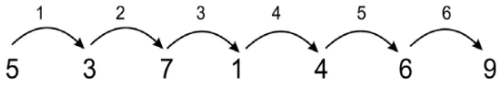
ujung tree. Sebaliknya, **untuk menemukan simpul yang memiliki nilai terbesar, penelusuran dimulai dari**

akar lalu terus bergerak ke simpul paling kanan hingga mencapai ujung tree. Sebagai ilustrasi disajikan di Gambar 8.32, jika digunakan sebuah tree dengan akar bernilai 6, maka pencarian nilai minimum dilakukan dengan bergerak dari simpul 6 ke 3, lalu dari 3 ke 1, sehingga diperoleh simpul bernilai 1 sebagai nilai terkecil. Dengan cara yang sama, pencarian nilai maksimum dilakukan dengan bergerak dari simpul 6 ke 8, kemudian dari 8 ke 10, sehingga diperoleh simpul bernilai 10 sebagai nilai terbesar. Dengan demikian, struktur binary search tree memungkinkan proses pencarian elemen minimum dan maksimum dilakukan secara efisien hanya dengan mengikuti cabang kiri untuk nilai minimum dan cabang kanan untuk nilai maksimum. Program menemukan node minimum dan maksimum dalam BST disajikan di link https://colab.research.google.com/drive/1KL2RDd7UAnesPfmM6jYwXvB9FHUzJ4Xa?usp=sharing.

Manfaat Binary Search Tree

Binary search tree secara umum menjadi pilihan yang lebih baik dibandingkan array dan linked list ketika sebuah aplikasi lebih sering melakukan akses terhadap elemen data. Struktur ini mendukung sebagian besar operasi, seperti pencarian, penyisipan, dan penghapusan, dengan kinerja yang relatif cepat. Array memang unggul dalam proses pencarian, tetapi cenderung kurang efisien untuk penyisipan dan penghapusan elemen. Sebaliknya, linked list lebih efisien untuk penyisipan dan penghapusan, tetapi lebih lambat saat digunakan untuk pencarian. Kompleksitas waktu terbaik dalam binary search tree untuk pencarian adalah O(log n), sedangkan kompleksitas terburuknya adalah O(n). Sementara itu, pencarian dalam list, baik dalam kondisi terbaik maupun terburuk, sama-sama memiliki kompleksitas O(n). Hal ini menunjukkan bahwa binary search tree dapat memberikan keuntungan efisiensi, terutama jika struktur tree tersusun dengan baik.

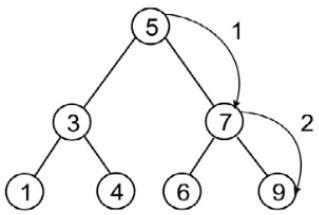
Keunggulan binary search tree dapat dipahami melalui contoh penyimpanan data 5, 3, 7, 1, 4, 6, dan 9. Jika data tersebut disimpan dalam bentuk list, pencarian elemen tertentu dalam kondisi terburuk mengharuskan penelusuran seluruh elemen yang ada. Sebagai ilustrasi Gambar 3.33, pencarian nilai 9 dapat memerlukan



Gambar 8.33: Contoh Pencarian dalam List (Agarwal, 2022)

pemeriksaan hampir semua elemen dalam list hingga elemen yang dicari ditemukan. Sebaliknya, jika data yang sama disimpan dalam binary search tree, jumlah perbandingan yang dibutuhkan untuk menemukan elemen tersebut menjadi jauh lebih sedikit. Dengan demikian, binary search tree mampu mempercepat proses pencarian karena setiap langkah pencarian langsung mengarahkan penelusuran ke subtree kiri atau subtree kanan sesuai hasil perbandingan nilai.

Meskipun demikian, efisiensi pencarian dalam binary search tree sangat dipengaruhi oleh cara tree tersebut dibangun. Jika elemen-elemen dimasukkan dengan urutan yang kurang tepat, tree dapat menjadi tidak seimbang sehingga kinerjanya menurun dan bahkan tidak lebih baik daripada list. Sebagai contoh Gambar 8.34, jika elemen

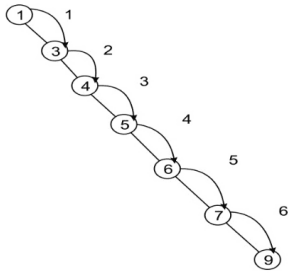


Gambar 8.34: Contoh Pencarian dalam BST (Agarwal, 2022)

dimasukkan secara berurutan dari nilai terkecil ke nilai terbesar, struktur tree cenderung membentuk rantai memanjang ke satu sisi. Kondisi ini menyebabkan proses pencarian kehilangan keunggulan utamanya karena

jumlah langkah yang diperlukan menjadi semakin besar. Oleh sebab itu, binary search tree akan memberikan hasil yang optimal apabila struktur tree tetap seimbang.

Namun, perlu diperhatikan bahwa efisiensi proses pencarian juga bergantung terhadap cara binary search tree dibangun. Jika struktur tree tidak disusun dengan baik, kinerjanya dapat menjadi lambat. Sebagai contoh



Gambar 8.35: Pohon Pencarian Biner Terurut (Agarwal, 2022)

Gambar 8.35, apabila elemen-elemen dimasukkan secara berurutan, misalnya 1, 3, 4, 5, 6, 7, dan 9, maka bentuk tree akan cenderung memanjang ke satu sisi, sehingga efisiensinya tidak akan lebih baik daripada list.

Berdasarkan uraian tersebut, dapat dipahami bahwa binary search tree merupakan pilihan yang baik untuk penyimpanan data jika tree berada dalam kondisi seimbang. Untuk menjaga kondisi tersebut, diperlukan metode tertentu yang mampu membentuk self-balancing tree, sehingga struktur tree tetap proporsional dan operasi pencarian dapat berlangsung lebih efisien.

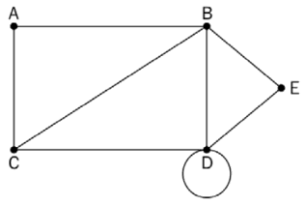
Tabel berikut menyajikan perbandingan struktur data array, linked list, dan binary search tree:

Tabel 8.1: Perbandingan Struktur Data Array, Linked List, dan Binary Search Tree

Properti	Array	Linked List	BST
Struktur data	Linear	Linear	Non-linear
Kemudahan penggunaan	Mudah dibuat dan digunakan. Kompleksitas rata-rata untuk pencarian, penyisipan, dan penghapusan adalah **O(n)**.	Penyisipan dan penghapusan berlangsung cepat, terutama **doubly linked list**.	Akses elemen, penyisipan, dan penghapusan berlangsung cepat, dengan kompleksitas rata-rata **O(log n)**.
Kompleksitas akses	Akses elemen mudah. Kompleksitasnya **O(1)**.	Hanya mendukung akses berurutan sehingga lebih lambat. Kompleksitas rata-rata dan terburuk adalah **O(n)**.	Akses berlangsung cepat, tetapi menjadi lambat jika tree tidak seimbang, dengan kompleksitas terburuk **O(n)**.
Kompleksitas pencarian	Kompleksitas rata-rata dan terburuk adalah **O(n)**.	Pencarian lambat karena harus dilakukan secara berurutan. Kompleksitas rata-rata dan terburuk adalah **O(n)**.	Kompleksitas terburuk untuk pencarian adalah **O(n)**.
Kompleksitas penyisipan	Penyisipan lambat. Kompleksitas rata-rata dan terburuk adalah **O(n)**.	Kompleksitas rata-rata dan terburuk adalah **O(1)**.	Kompleksitas terburuk untuk penyisipan adalah **O(n)**.
Kompleksitas penghapusan	Penghapusan lambat. Kompleksitas rata-rata dan terburuk adalah **O(n)**.	Kompleksitas rata-rata dan terburuk adalah **O(1)**.	Kompleksitas terburuk untuk penghapusan adalah **O(n)**.

BAB 9
Graph

Graf merupakan himpunan yang terdiri atas sejumlah **simpul (vertex atau node) dan sisi (edge)**, di mana sisi berfungsi sebagai penghubung antar simpul, serta setiap sisi dalam graf menghubungkan dua simpul yang berbeda (Agarwal, 2022). Selain itu, graf juga dapat dipahami sebagai representasi matematis formal dari suatu jaringan. Secara notasi matematis, sebuah **graf G** dinyatakan sebagai pasangan terurut dari **himpunan simpul V** dan **himpunan sisi E**, yang dituliskan sebagai **G = (V, E)**. Graf menyediakan kerangka konseptual yang sistematis untuk menggambarkan hubungan antar elemen dalam suatu jaringan atau struktur yang saling terhubung.



Gambar 9.1: Contoh Graf (Agarwal, 2022)

Grafik G = (V, E) di Gambar 9.1 dapat dijelaskan sebagai berikut:

- V = {A, B, C, D, E}
- E = {{A, B}, {A, C}, {B, C}, {B, D}, {C, D}, {D, D}, {B, E}, {D, E}}
- G = (V, E)

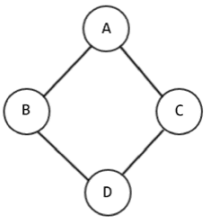
Beberapa terminologi graf yaitu:

- **Node** atau **vertex** adalah titik dalam suatu graf yang disebut sebagai simpul. Dalam representasi graf, simpul umumnya digambarkan sebagai titik. Sebagai contoh, simpul dapat diberi label **A, B, C, D,** dan **E**.
- **Edge** adalah garis atau relasi yang menghubungkan dua simpul dalam graf. Dengan kata lain, **edge** menunjukkan adanya keterkaitan langsung antardua **vertex**. Sebagai contoh, garis yang menghubungkan simpul **A** dan **B** merupakan sebuah **edge**.
- **Loop** adalah sisi yang berawal dari suatu simpul dan kembali lagi ke simpul yang sama. **Loop** menunjukkan bahwa sebuah simpul memiliki hubungan dengan dirinya sendiri.
- **Derajat simpul** (*degree of a vertex/node*) adalah jumlah seluruh **edge** yang terhubung dengan suatu simpul tertentu. Nilai derajat ini menunjukkan banyaknya hubungan langsung yang dimiliki simpul tersebut dengan simpul lain atau dengan dirinya sendiri dalam kasus **loop**.
- **Adjacency** atau ketetanggaan mengacu terhadap hubungan langsung antardua simpul. Dua simpul dikatakan bertetangga jika terdapat **edge** yang menghubungkan keduanya. Sebagai contoh, simpul **C** dikatakan bertetangga dengan simpul **A** jika terdapat sisi yang menghubungkan **C** dan **A**.
- **Path** adalah urutan simpul dan **edge** yang membentuk lintasan dari satu simpul ke simpul lainnya. **Path** merepresentasikan jalur yang dapat ditempuh dalam graf untuk mencapai simpul tujuan dari simpul awal.
- **Leaf vertex** atau **pendant vertex** adalah simpul yang memiliki derajat tepat satu. Artinya, simpul tersebut hanya terhubung dengan satu **edge**, sehingga umumnya berada di bagian ujung struktur graf.

GRAF BERARAH DAN TAK BERARAH

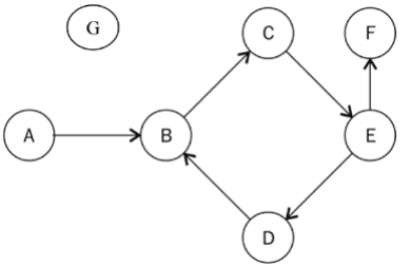
Graf direpresentasikan melalui sisi-sisi yang menghubungkan antar simpul. Sisi penghubung tersebut dapat bersifat tidak berarah maupun berarah. **Apabila sisi-sisi dalam suatu graf tidak memiliki orientasi tertentu, maka graf tersebut disebut graf tak berarah**, sedangkan **jika setiap sisi memiliki arah yang jelas**

dari satu simpul ke simpul lain, maka graf itu disebut graf berarah. Dalam graf tak berarah, sisi umumnya digambarkan sebagai garis yang menghubungkan dua simpul tanpa menunjukkan arah hubungan tertentu. Representasi ini hanya menegaskan bahwa dua simpul memiliki keterhubungan, tanpa memberikan informasi tambahan mengenai orientasi relasi di antara keduanya. Sebagai ilustrasi Gambar 9.2, suatu graf tak berarah dapat terdiri atas empat simpul, yaitu A, B, C, dan D, yang saling terhubung melalui sejumlah sisi.



Gambar 9.2: Graf Tak Berarah (Agarwal, 2022)

Dalam graf berarah, setiap sisi memuat informasi mengenai arah hubungan antara dua simpul yang terhubung. Oleh karena itu, hubungan dari simpul A ke simpul B berbeda dengan hubungan dari simpul B ke simpul A, sehingga pasangan (A,B) (tidak dapat disamakan dengan (B,A). Secara visual, **sisi berarah digambarkan dalam bentuk garis yang dilengkapi anak panah untuk menunjukkan arah keterhubungan antar simpul.** Arah panah di sisi menentukan alur perpindahan yang dapat dilakukan, misalnya perpindahan hanya dapat berlangsung dari A ke B dan bukan sebaliknya. Sebagai contoh, Gambar 9.3 menunjukkan grafik berarah dimana banyak simpul terhubung menggunakan sisi berarah:



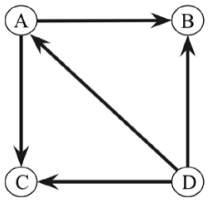
Gambar 9.3: Graf Berarah (Agarwal, 2022)

Dalam graf berarah, simpul memiliki karakteristik tertentu yaitu:

- **Indegree** adalah jumlah seluruh sisi yang masuk ke suatu simpul dalam graf. Sebagai contoh, simpul E memiliki indegree sebesar 1 karena terdapat satu sisi, yaitu CE, yang menuju ke simpul tersebut.
- **Outdegree** adalah jumlah seluruh sisi yang keluar dari suatu simpul dalam graf. Sebagai ilustrasi, simpul E mempunyai outdegree sebesar 2 karena terdapat dua sisi, yaitu EF dan ED, yang berawal dari simpul tersebut.
- **Simpul terisolasi** (**isolated vertex**) adalah simpul yang memiliki derajat nol, sehingga tidak mempunyai sisi masuk maupun sisi keluar. Contohnya adalah simpul G yang digambarkan sebagai simpul terpisah.
- **Simpul sumber** (**source vertex**) adalah simpul yang memiliki indegree nol, sehingga tidak menerima sisi dari simpul lain. Sebagai contoh, simpul A merupakan source vertex karena tidak ada sisi yang masuk ke simpul tersebut.
- **Simpul tujuan akhir** (**sink vertex**) adalah simpul yang memiliki outdegree nol, sehingga tidak memiliki sisi yang keluar menuju simpul lain. Contohnya, simpul F merupakan sink vertex karena semua hubungan berakhir di simpul itu.

GRAF ACYCLIC TERARAH

Graf berarah tak bersiklus atau *directed acyclic graph* (DAG) merupakan graf berarah yang tidak mengandung siklus. Dalam struktur ini, setiap sisi memiliki arah dari satu simpul ke simpul lain, tetapi rangkaian sisi yang terbentuk tidak pernah kembali ke simpul asal sehingga tidak membentuk lintasan tertutup. Suatu siklus dalam graf terjadi apabila simpul awal dari sisi pertama sama dengan simpul akhir dari sisi terakhir dalam suatu urutan lintasan. Setiap penelusuran jalur yang dimulai dari suatu simpul tertentu tidak akan pernah berakhir kembali di simpul tersebut. Karakteristik ini menjadikan DAG banyak dimanfaatkan dalam berbagai bidang, seperti penjadwalan pekerjaan, graf sitasi, dan kompresi data.

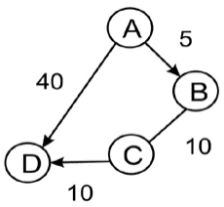


Gambar 9.4: Graf Acyclic Terarah (Agarwal, 2022)

Gambar 9.4 memperlihatkan sebuah DAG yang terdiri atas empat simpul, yaitu A, B, C, dan D. Hubungan antarsimpul ditunjukkan oleh sisi berarah, yakni dari A menuju B, dari A menuju C, dari D menuju A, dari D menuju B, serta dari D menuju C. Arah panah di setiap sisi menunjukkan urutan keterhubungan yang hanya dapat diikuti sesuai orientasinya. **Struktur ini disebut graf berarah tak bersiklus karena meskipun terdapat beberapa jalur yang menghubungkan simpul-simpul tersebut, tidak ada satu pun lintasan yang kembali ke simpul asal sehingga tidak membentuk putaran tertutup**. Karakter utama DAG yaitu adanya arah yang jelas di setiap relasi tanpa keberadaan siklus dalam graf.

BOBOT GRAF

Graf berbobot adalah graf yang setiap sisinya memiliki nilai numerik tertentu sebagai bobot. Bobot tersebut dapat merepresentasikan berbagai makna, seperti jarak, biaya, waktu, atau ukuran lain, sesuai dengan tujuan penggunaan graf. Graf berbobot dapat berbentuk graf berarah maupun graf tak berarah. Keberadaan bobot sisi memungkinkan analisis yang lebih rinci terhadap hubungan antarsimpul, karena penentuan jalur terbaik tidak hanya didasarkan ada atau tidaknya koneksi, tetapi juga nilai bobot yang melekat di setiap sisi. Sebagai contoh Gambar 9.5, untuk mencapai simpul D dari simpul A, dapat tersedia dua jalur, yaitu jalur langsung A-D atau jalur tidak langsung A-B-C-D melalui simpul B dan C. Apabila bobot dalam graf menyatakan jarak antarsimpul, maka pemilihan jalur terbaik didasarkan total bobot terkecil. Jalur langsung A-D memiliki total bobot 40, sedangkan jalur A-B-C-D memiliki total bobot 25. Oleh sebab itu, jalur A-B-C-D dinilai lebih baik karena memberikan jarak tempuh yang lebih pendek.

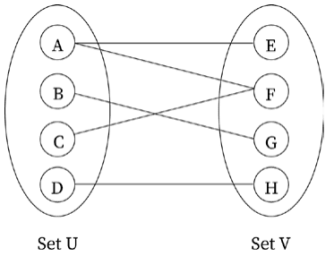


Gambar 9.5: Bobot Graf (Agarwal, 2022)

GRAF BIPARTIT

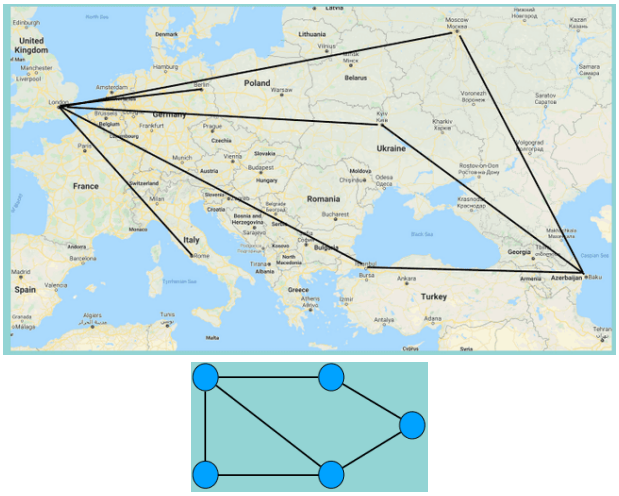
Graf bipartit, dikenal sebagai bigraph, merupakan jenis graf khusus yang seluruh simpulnya dapat dipartisi ke dalam dua himpunan terpisah sehingga setiap sisi hanya menghubungkan simpul dari himpunan pertama ke simpul dari himpunan kedua. Dengan kata lain, apabila himpunan simpul dinyatakan

sebagai himpunan U dan himpunan V (Gambar 9.6), maka setiap sisi dalam graf harus memiliki satu ujung dalam simpul di himpunan U dan ujung lainnya dalam simpul di himpunan V. Dalam struktur ini, tidak terdapat sisi yang menghubungkan dua simpul yang berada dalam himpunan yang sama. Karakteristik tersebut menjadikan graf bipartit sangat berguna untuk memodelkan hubungan antara dua kelas objek yang berbeda. Sebagai contoh, graf bipartit digunakan untuk merepresentasikan hubungan antara pelamar kerja dan pekerjaan, sehingga keterkaitan antara individu pelamar dengan posisi yang dilamar dapat dianalisis secara sistematis. Contoh lain adalah hubungan antara pemain sepak bola dan klub, yang dapat digunakan untuk menunjukkan apakah seorang pemain pernah bermain untuk klub tertentu atau tidak. Graf bipartit merupakan representasi efektif untuk menggambarkan relasi antardua kelompok entitas yang berbeda secara jelas dan terstruktur.



Gambar 9.6: Graf Bipartit (Agarwal, 2022)

ALASAN PENGGUNAAN GRAF



Gambar 9.7: Contoh Graf (Karimov, 2022)

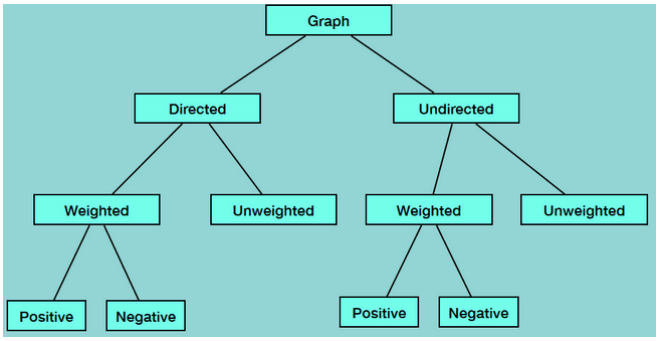
Gambar 9.7 menyajikan ilustrasi konseptual tentang alasan penggunaan graf sebagai model representasi hubungan antarobjek dalam konteks dunia nyata. Bagian atas memperlihatkan peta kawasan Eropa dan wilayah di sekitarnya yang dilengkapi sejumlah garis penghubung berwarna hitam antartitik lokasi. Garis-garis tersebut merepresentasikan relasi antara satu titik geografis dan titik geografis lainnya, seperti jalur perjalanan, rute distribusi, atau jaringan transportasi. Visualisasi ini menunjukkan bahwa persoalan dunia nyata yang mula-mula tampak kompleks dalam bentuk peta dapat disederhanakan menjadi himpunan titik dan garis penghubung. Melalui cara pandang semacam ini, wilayah geografis tidak lagi dipahami semata-mata sebagai ruang fisik, melainkan sebagai jaringan hubungan yang dapat dikaji secara sistematis untuk mengidentifikasi keterhubungan, alternatif lintasan, dan kemungkinan rute yang paling efisien.

Bagian bawah memperlihatkan hasil abstraksi dari representasi berbasis peta ke dalam bentuk graf yang lebih sederhana. Setiap lingkaran biru menggambarkan simpul (vertex), sedangkan garis hitam yang menghubungkan antarsimpul menunjukkan sisi (edge). Penyederhanaan tersebut menegaskan bahwa hakikat

utama suatu jaringan tidak terletak bentuk visual petanya, melainkan pola hubungan antartitik yang membentuk struktur tertentu. Melalui abstraksi itu, proses analisis menjadi lebih mudah dilakukan, misalnya untuk menentukan jalur terpendek, menghitung jumlah koneksi suatu simpul, mengidentifikasi simpul yang berperan penting, atau mengevaluasi struktur jaringan secara menyeluruh. Fungsi graf sebagai alat pemodelan yang efektif untuk mentransformasikan persoalan nyata yang kompleks, seperti jaringan wilayah dan rute perjalanan, ke dalam struktur matematis yang lebih terorganisasi, terukur, dan mudah dianalisis (Karimov, 2022).

TIPE GRAF

Terdapat tipe graf yaitu:



Gambar 9.8: Tipe Graf (Karimov, 2022)

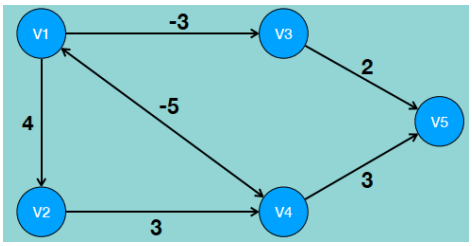
- Graf berarah (*directed graph*) merupakan graf yang setiap sisinya memiliki orientasi atau arah tertentu, sehingga relasi antara dua simpul dinyatakan secara satu arah.
- Graf tak berarah (*undirected graph*) ialah graf yang sisi-sisinya tidak memiliki orientasi, sehingga hubungan antara dua simpul bersifat timbal balik.

Masing-masing kategori itu kemudian dipilah lagi menjadi graf berbobot (*weighted graph*) dan graf tak berbobot (*unweighted graph*).

- Graf berbobot adalah graf yang setiap sisinya memiliki nilai numerik tertentu.
- Graf tak berbobot hanya menunjukkan adanya keterhubungan tanpa nilai tambahan.

Dalam graf berbobot, bobot sisi masih dapat dikelompokkan menjadi bobot positif dan bobot negatif. Bobot positif umumnya digunakan untuk merepresentasikan jarak, biaya, atau waktu tempuh yang bernilai nonnegatif, sedangkan bobot negatif digunakan untuk menunjukkan pengurangan nilai, penalti, atau kondisi tertentu dalam pemodelan matematis dan komputasional.

Dari tipe graf dapat dibentuk enam kombinasi utama jenis graf, yaitu unweighted-undirected, unweighted-



Gambar 9.9: Ilustrasi Tipe Graf (Agarwal, 2022)

directed, positive-weighted-undirected, positive-weighted-directed, negative-weighted-undirected, dan negative-weighted-directed. Daftar itu menunjukkan bahwa jenis graf dibentuk oleh tiga unsur utama, yakni keberadaan arah di sisi, keberadaan bobot di sisi, dan tanda bobot tersebut. Suatu graf dapat dipahami melalui ada atau tidaknya arah dalam relasi antarsimpul, ada atau tidaknya nilai kuantitatif yang menyertai hubungan, serta sifat nilai tersebut, apakah positif atau negatif. Klasifikasi ini penting dalam kajian struktur data dan teori graf karena

setiap jenis graf memiliki karakteristik, metode analisis, dan ranah penerapan yang berbeda. Sebagai contoh, graf tak berbobot lebih sesuai untuk memodelkan hubungan sederhana, sedangkan graf berbobot lebih tepat digunakan untuk persoalan optimasi jalur, distribusi, transportasi, dan analisis jaringan.

Gambar 9.9 memperlihatkan contoh graf berarah berbobot dengan simpul V1, V2, V3, V4, dan V5. Arah panah di setiap sisi menunjukkan bahwa hubungan antarsimpul tidak bersifat dua arah, melainkan hanya dapat diikuti sesuai orientasi panah. Sementara itu, angka-angka yang tercantum di setiap sisi menunjukkan bobot yang menyertai hubungan tersebut. Sebagai ilustrasi, sisi dari V1 ke V3 memiliki bobot −3, sisi dari V1 ke V2 berbobot 4, sisi dari V2 ke V4 berbobot 3, sisi dari V3 ke V5 berbobot 2, sisi dari V4 ke V5 berbobot 3, serta terdapat sisi berbobot negatif −5 yang menghubungkan simpul lain dalam jaringan tersebut. Kehadiran bobot positif dan negatif dalam satu graf memperlihatkan bahwa hubungan antarsimpul tidak selalu merepresentasikan penambahan biaya atau jarak, tetapi juga dapat melambangkan pengurangan nilai atau keuntungan tertentu. Teori graf tidak hanya membahas pola keterhubungan, tetapi juga memfasilitasi representasi hubungan yang lebih kaya dan realistis dalam berbagai persoalan komputasi.

9.1 Representasi Graf

Representasi graf merupakan cara penyimpanan struktur graf di dalam memori komputer, yakni pengaturan simpul, sisi, dan bobot—apabila graf bersifat berbobot—agar dapat diproses secara efisien. Secara umum, terdapat dua metode utama, yakni:

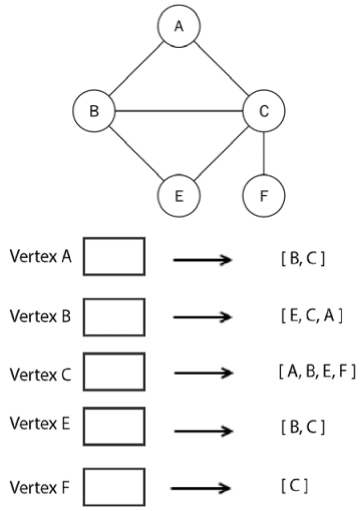
- **Daftar ketetanggaan (adjacency list)**
  Adjacency list dibangun dengan memanfaatkan struktur linked list melalui penyimpanan daftar simpul tetangga bagi setiap simpul dalam graf
- **Matriks ketetanggaan (adjacency matrix)**.
  Adjacency matrix memanfaatkan sebuah matriks untuk menunjukkan hubungan ketetanggaan antarsimpul, sehingga setiap sel di dalam matriks menyatakan ada atau tidaknya sisi yang menghubungkan dua simpul tertentu.

Kedua bentuk representasi tersebut dapat digunakan sesuai kebutuhan, tetapi pemilihannya sangat bergantung karakteristik graf serta tujuan penggunaannya. Adjacency list lebih sesuai **untuk graf jarang (sparse graph), yaitu graf dengan jumlah sisi yang relatif sedikit dibandingkan jumlah simpul**. Sebagai ilustrasi, graf yang memiliki 200 simpul dan hanya 100 sisi akan lebih efisien jika disimpan dalam bentuk daftar ketetanggaan, sebab penggunaan matriks ketetanggaan akan menghasilkan matriks berukuran 200 × 200 dengan sebagian besar elemen bernilai nol. Sebaliknya, **adjacency matrix lebih tepat untuk graf rapat (dense graph), yaitu graf yang memiliki banyak sisi**. Selain itu, matriks ketetanggaan memberikan kemudahan dalam memeriksa keberadaan maupun ketiadaan suatu sisi, karena informasi tersebut dapat diakses secara langsung melalui indeks matriks. Pemilihan teknik representasi graf perlu mempertimbangkan efisiensi ruang penyimpanan sekaligus kebutuhan operasi yang akan diterapkan terhadap graf tersebut.

Adjacency List

Adjacency lists merupakan salah satu bentuk representasi graf yang menuliskan seluruh nodes yang terhubung langsung dengan suatu node x ke dalam adjacent list milik node tersebut. Dalam bentuk ini, graf direpresentasikan dengan menampilkan adjacent list untuk seluruh nodes yang terdapat di dalam graf. Dua nodes, misalnya A dan B, dikatakan saling adjacent apabila terdapat hubungan langsung di antara keduanya. Implementasi adjacency list dapat menggunakan linked list. Untuk merepresentasikan graf, diperlukan sejumlah linked list yang sama dengan jumlah seluruh nodes di dalam graf. Setiap indeks menyimpan adjacent nodes dari vertex yang bersangkutan. Sebagai contoh Gambar 9.10, node pertama merepresentasikan vertex A dengan

adjacent nodes berupa B dan C. Node kedua merepresentasikan vertex B dengan adjacent nodes berupa E, C, dan A. Dengan cara serupa, vertex lain, yaitu C, E, dan F, direpresentasikan beserta adjacent nodes masing-masing.



Gambar 9.10: Graf dan Adjacency List (Agarwal, 2022)

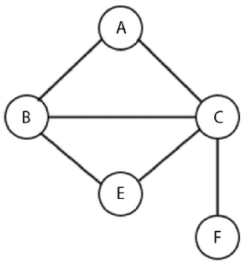
Penggunaan struktur list dalam representasi ini tergolong cukup terbatas karena label vertex tidak dapat dimanfaatkan secara langsung. Oleh karena itu, untuk mengimplementasikan graf secara efisien menggunakan Python, struktur data dictionary lebih sesuai digunakan karena lebih tepat untuk merepresentasikan graf. Program disajikan di https://colab.research.google.com/drive/1TabKmwYiiDQZn30zbtvKBifc1yhcgJAb?usp=sharing.

Adjacency Matrix

Adjacency matrix merupakan pendekatan lain untuk merepresentasikan graf. Dalam metode ini, graf digambarkan melalui **nodes** beserta keterhubungannya melalui **edges**. Representasi tersebut menggunakan sebuah matriks berukuran V×V, dengan setiap sel di dalam matriks menyatakan keberadaan suatu **edge** dalam graf. Karena matriks merupakan **two-dimensional array**, gagasan utamanya adalah mengisi sel-sel matriks dengan nilai **1** atau **0**, bergantung ada atau tidaknya hubungan **edge** antara dua **nodes**. Apabila dua **nodes** terhubung, maka sel yang bersesuaian diisi dengan nilai **1**; sebaliknya, apabila tidak terdapat hubungan, maka sel tersebut diisi dengan nilai **0**. Dengan cara ini, **Adjacency matrix** mampu memberikan gambaran yang sistematis mengenai pola keterhubungan seluruh **nodes** dalam graf.

Adjacency matrix juga dapat diimplementasikan dengan memanfaatkan **adjacency list** yang telah dibentuk sebelumnya. Untuk membangun **adjacency matrix**, dapat digunakan implementasi graf berbasis **dictionary** yang telah tersedia. Langkah awal dalam proses ini adalah memperoleh **key elements** yang akan digunakan dalam penyusunan matriks. Hal yang penting untuk diperhatikan ialah bahwa elemen-elemen matriks tersebut merupakan **vertices** dari graf. Dengan demikian, setiap baris dan setiap kolom dalam **Adjacency matrix** merepresentasikan **vertices**, sedangkan nilai perpotongan baris dan kolom menunjukkan ada atau tidaknya **edge** yang menghubungkan dua **vertices** tersebut.

Gambar 9.11 menampilkan hubungan antara representasi visual graf dan representasi Adjacency Matrix dari graf yang sama. Di sisi kiri, terlihat sebuah graf tak berarah yang terdiri atas lima simpul utama, yaitu A, B, C, E, dan F. Simpul A terhubung dengan B dan C, simpul B terhubung dengan A, C, dan E, simpul C terhubung dengan A, B, E, dan F, simpul E terhubung dengan B dan C, sedangkan simpul F hanya terhubung dengan C. Garis-garis yang menghubungkan simpul-simpul tersebut menunjukkan adanya sisi atau hubungan langsung antarsimpul. Karena graf ini tidak menampilkan arah panah, maka relasi antarsimpul bersifat dua arah atau timbal balik, sehingga graf tersebut termasuk graf tak berarah.



Adjacency Matrix

	A	B	C	E	F
A	0	1	1	0	0
B	1	0	1	1	0
C	1	1	0	1	1
E	0	1	1	0	0
F	0	0	1	0	0

Gambar 9.11: Graf dan Adjacency Matrix (Agarwal, 2022)

Di sisi kanan, graf yang sama direpresentasikan ke dalam bentuk Adjacency Matrix, yaitu matriks persegi yang baris dan kolomnya sama-sama diberi label A, B, C, E, dan F. Setiap sel dalam matriks menunjukkan ada atau tidaknya hubungan antara dua simpul. Nilai 1 menyatakan bahwa dua simpul saling terhubung secara langsung, sedangkan nilai 0 menunjukkan bahwa keduanya tidak memiliki hubungan langsung. Sebagai contoh, sel perpotongan baris A dan kolom B bernilai 1 karena terdapat sisi antara A dan B. Demikian pula, sel baris C dan kolom F bernilai 1 karena C terhubung langsung dengan F. Sebaliknya, sel baris A dan kolom F bernilai 0 karena tidak terdapat sisi yang menghubungkan kedua simpul tersebut. Nilai diagonal utama, seperti A-A, B-B, atau C-C, bernilai 0 karena dalam graf tersebut tidak terdapat loop, yaitu sisi yang menghubungkan simpul dengan dirinya sendiri.

Matriks tersebut juga tampak simetris terhadap diagonal utama. Kondisi ini menunjukkan ciri khas graf tak berarah, karena hubungan antara simpul X ke simpul Y selalu sama dengan hubungan antara simpul Y ke simpul X. Sebagai contoh, nilai posisi B-C sama dengan nilai posisi C-B, yaitu sama-sama 1. Melalui gambar ini, terlihat dengan jelas bahwa Adjacency Matrix merupakan cara sistematis untuk menyajikan informasi keterhubungan graf dalam bentuk tabel numerik. Representasi ini sangat berguna dalam analisis komputasional karena memudahkan pemeriksaan hubungan antar simpul secara langsung, meskipun untuk graf dengan jumlah sisi yang sedikit, metode ini dapat menghasilkan banyak elemen nol. Satu graf dapat direpresentasikan baik secara visual maupun dalam bentuk matriks, dan keduanya saling melengkapi dalam membantu pemahaman terhadap struktur hubungan antarsimpul.

Programm adjacency matriks ditampilkan di https://colab.research.google.com/drive/1EATSyH-PUPyJr3eB3jb1rML53wFn8lH6?usp=sharing.

Adjacency Multi List

Adjacency Multi List adalah salah satu struktur representasi graf, khususnya untuk **graf tak berarah (undirected graph)**, yang menyimpan setiap **edge** hanya **satu kali**, tetapi tetap dapat diakses dari dua **vertex** yang dihubungkannya. Struktur ini dikembangkan untuk mengatasi keterbatasan **adjacency list** biasa, karena dalam **adjacency list** setiap **edge** dalam graf tak berarah umumnya dicatat dua kali, yakni dari vertex pertama ke vertex kedua dan dari vertex kedua ke vertex pertama. Akibatnya, terjadi duplikasi penyimpanan. Melalui **Adjacency Multi List**, satu **edge** cukup direpresentasikan dalam satu simpul data, lalu simpul tersebut dihubungkan ke dua vertex yang berinsiden dengannya.

Secara konseptual, **Adjacency Multi List** terdiri atas dua komponen utama, yaitu **vertex node** dan **edge node**. **Vertex node** menyimpan informasi tentang vertex, misalnya label vertex, serta pointer yang mengarah ke edge pertama yang berhubungan dengan vertex tersebut. Sementara itu, **edge node** menyimpan pasangan vertex yang dihubungkannya, misalnya **ivex** dan **jvex**, serta dua pointer, yaitu **ilink** dan **jlink**. Pointer **ilink** mengarah ke edge lain yang juga berhubungan dengan **ivex**, sedangkan **jlink** mengarah ke edge lain yang berhubungan dengan **jvex**. Dengan susunan ini, satu edge dapat menjadi bagian dari daftar keterhubungan dua vertex sekaligus.

Keunggulan utama **Adjacency Multi List** yaitu efisiensi penyimpanan untuk graf tak berarah, karena setiap edge tidak perlu dicatat dua kali. Struktur ini juga memudahkan penghapusan edge, sebab informasi edge berada dalam satu simpul data yang sama. Meskipun demikian, strukturnya lebih kompleks dibandingkan

adjacency list biasa, karena setiap edge harus memiliki dua pointer keterhubungan. Oleh sebab itu, **Adjacency Multi List** lebih sesuai digunakan ketika efisiensi penyimpanan edge menjadi pertimbangan penting dan operasi manipulasi graf dilakukan secara intensif.

Struktur umum Adjacency Multi List yaitu:

1. Vertex node

Setiap vertex node biasanya memuat:

data: label vertex.

firstedge: pointer ke edge pertama yang berhubungan dengan vertex tersebut.

2. Edge node

Setiap edge node biasanya memuat:

ivex: vertex ujung pertama.

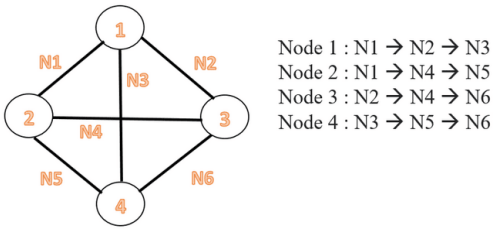
jvex: vertex ujung kedua.

ilink: pointer ke edge lain yang terkait dengan ivex.

jlink: pointer ke edge lain yang terkait dengan jvex.

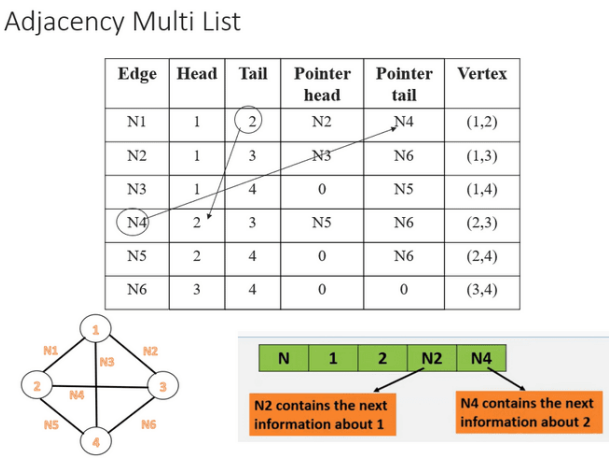
info: data tambahan, jika diperlukan, misalnya bobot.

Gambar 9.12 menampilkan contoh graf tak berarah dengan empat **vertex** yaitu 1, 2, 3, dan 4, yang



Gambar 9.12: Contoh Graf Tak Berarah

dihubungkan oleh enam **edge** bernama **N1**, **N2**, **N3**, **N4**, **N5**, dan **N6**. Struktur tersebut merepresentasikan graf lengkap dengan empat simpul, karena setiap **node** terhubung dengan seluruh **node** lainnya. Di sisi kanan, daftar "Node 1: N1 → N2 → N3", "Node 2: N1 → N4 → N5", "Node 3: N2 → N4 → N6", dan "Node 4: N3 → N5 → N6" menunjukkan urutan **edge** yang berinsiden dengan masing-masing **vertex** .



Gambar 9.13: Tabular Adjacency Multi List

Gambar 9.13 memperlihatkan bentuk tabular dari **Adjacency Multi List** untuk graf yang sama. Kolom **Edge** berisi identitas hubungan, kolom **Head** dan **Tail** berisi pasangan **vertex** yang dihubungkan, kolom **Pointer head** menunjukkan **edge** berikutnya yang masih terkait dengan **Head**, kolom **Pointer tail** menunjukkan **edge**

berikutnya yang masih terkait dengan **Tail**, sedangkan kolom **Vertex** menuliskan pasangan simpul yang dihubungkan oleh **edge** tersebut.

- **Head** adalah **vertex** pertama yang terhubung oleh suatu **edge**. Jadi, jika sebuah **edge** menghubungkan vertex 1 dan vertex 2, maka salah satu vertex ditetapkan sebagai **Head**. Penetapan ini bukan berarti vertex tersebut lebih penting, melainkan hanya sebagai penanda ujung pertama dari edge agar struktur data dapat disusun secara sistematis.
- **Tail** adalah **vertex** kedua yang terhubung oleh **edge** yang sama. Dengan demikian, jika sebuah **edge** ditulis menghubungkan 1 dan 2, lalu 1 dipilih sebagai **Head**, maka 2 menjadi **Tail**. Jadi, **Head** dan **Tail** bersama-sama menyatakan dua ujung dari satu **edge**.
- **Pointer head** adalah penunjuk ke **edge** berikutnya yang masih berhubungan dengan **Head**. Artinya, jika suatu **edge** memiliki **Head = 1**, maka **Pointer head** akan mengarah ke **edge** lain yang juga terhubung dengan vertex 1. Tujuannya agar seluruh edge yang berkaitan dengan vertex 1 dapat ditelusuri secara berantai. Misalnya, **N1** yang menghubungkan vertex 1 dan 2, nilai **Pointer head = N2** berarti setelah **N1**, masih ada **edge** berikutnya yang juga berkaitan dengan vertex 1, yaitu **N2**.
- **Pointer tail** adalah penunjuk ke **edge** berikutnya yang masih berhubungan dengan **Tail**. Jadi, jika **Tail** dari **N1** adalah vertex 2, maka **Pointer tail** akan mengarah ke **edge** lain yang juga terhubung dengan node 2. Contoh, **N1** memiliki **Pointer tail = N4**, yang berarti sesudah **N1**, masih ada **edge** lain yang berkaitan dengan vertex 2, yaitu **N4**.

Berikut penjelasan tabular adjacency list tiap baris yaitu:

- Baris pertama, N1 adalah edge yang menghubungkan vertex 1 dan vertex 2. Karena Head = 1, maka dari sisi vertex 1, edge berikutnya yang masih berhubungan dengan vertex 1 adalah N2, sehingga Pointer head = N2. Karena Tail = 2, maka dari sisi vertex 2, edge berikutnya yang masih berhubungan dengan vertex 2 adalah N4, sehingga Pointer tail = N4. Jadi, baris pertama menunjukkan bahwa N1 dapat ditelusuri ke N2 jika mengikuti jalur keterhubungan vertex 1, dan dapat ditelusuri ke N4 jika mengikuti jalur keterhubungan vertex 2.
- Beris kedua, N2 adalah edge yang menghubungkan vertex 1 dan vertex 3. Dari sisi Head, yaitu vertex 1, edge berikutnya yang masih berhubungan dengan vertex 1 adalah N3, sehingga Pointer head = N3. Dari sisi Tail, yaitu vertex 3, Pointer tail = N6, yang berarti dari vertex 3, sesudah N2 penelusuran langsung dilanjutkan ke N6. Jadi, N2 menghubungkan 1 dan 3, lalu dari sisi 1 berlanjut ke N3, sedangkan dari sisi 3 berlanjut ke N6.
- Baris ketiga, N3 adalah edge yang menghubungkan vertex 1 dan vertex 4. Dari sisi Head, yaitu vertex 1, tidak ada lagi edge berikutnya yang masih terkait, sehingga Pointer head = 0. Nilai 0 berarti rantai penelusuran berhenti. Dari sisi Tail, yaitu vertex 4, edge berikutnya yang masih berhubungan adalah N5, sehingga Pointer tail = N5. Dengan demikian, N3 merupakan edge terakhir untuk jalur vertex 1, tetapi belum merupakan edge terakhir untuk jalur vertex 4.
- Baris keempat, N4 adalah edge yang menghubungkan vertex 2 dan vertex 3. Dari sisi Head, yaitu vertex 2, edge berikutnya yang masih berhubungan adalah N5, sehingga Pointer head = N5. Dari sisi Tail, yaitu vertex 3, edge berikutnya yang masih berhubungan adalah N6, sehingga Pointer tail = N6. Jadi, setelah N4, penelusuran dapat dilanjutkan ke N5 jika mengikuti vertex 2, atau ke N6 jika mengikuti vertex 3.
- Baris kelima, N5 adalah edge yang menghubungkan vertex 2 dan vertex 4. Dari sisi Head, yaitu vertex 2, tidak ada lagi edge berikutnya yang masih terkait, sehingga Pointer head = 0. Dari sisi Tail, yaitu vertex 4, masih ada edge berikutnya yang berhubungan, yaitu N6, sehingga Pointer tail = N6. Jadi, N5 merupakan edge terakhir untuk jalur vertex 2, tetapi masih dilanjutkan ke N6 untuk jalur vertex 4.

❋ Baris keenam, N6 adalah edge yang menghubungkan vertex 3 dan vertex 4. Dari sisi Head maupun Tail, tidak ada lagi edge berikutnya yang terhubung, sehingga kedua pointer bernilai 0. Dengan demikian, N6 merupakan edge terakhir dalam rantai penelusuran untuk vertex 3 maupun vertex 4.

Gambar Adjacency Multi List bagian bawah kanan mempertegas makna ini dengan menunjukkan bahwa N2 menyimpan informasi berikutnya mengenai vertex 1, sedangkan N4 menyimpan informasi berikutnya mengenai vertex 2. Melalui tabel ini, terlihat bahwa setiap edge bertindak sebagai simpul penghubung ganda yang dapat ditelusuri dari sisi Head maupun dari sisi Tail.

Program Adjacency Multi List dapat dilihat di https://colab.research.google.com/drive/15DLWjMDQD7GLu6qZaNbJVpth9Mt53J3i?usp=sharing.

9.2 Graf Traversal

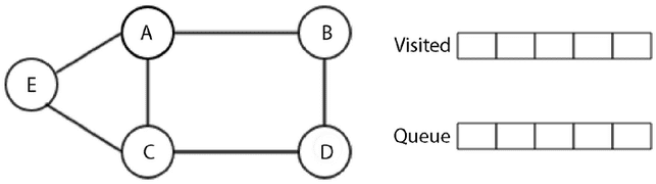
Graph traversal merupakan proses mengunjungi seluruh vertices di dalam suatu graf sambil mencatat nodes atau vertices mana yang telah dikunjungi dan mana yang belum. Suatu graph traversal algorithm dikatakan efisien apabila mampu menelusuri seluruh nodes dalam waktu sekecil mungkin. Graph traversal, yang juga dikenal sebagai graph search algorithm, memiliki kemiripan dengan algoritma traversal dalam tree, seperti preorder, inorder, postorder, dan level order. Kesamaannya yaitu cara kerja yang dimulai dari sebuah node, kemudian penelusuran dilanjutkan melalui edges menuju nodes lain di dalam graf. Salah satu strategi umum dalam graph traversal ialah mengikuti suatu jalur hingga mencapai titik buntu, lalu menelusuri kembali jalur sebelumnya sampai ditemukan jalur alternatif. Selain itu, penelusuran juga dapat dilakukan secara iteratif dari satu node ke node lain untuk menjangkau seluruh graf atau hanya sebagian darinya. Keberadaan graph traversal algorithms sangat penting untuk menjawab berbagai persoalan mendasar, misalnya menentukan cara berpindah dari satu vertex ke vertex lain serta memilih jalur dari A node ke B node yang lebih baik dibandingkan jalur lainnya. Dalam jaringan kota, misalnya, graph traversal algorithms dapat dimanfaatkan untuk mencari rute terpendek dari satu kota ke kota lain. Dua algoritma penting yang banyak digunakan dalam graph traversal adalah **breadth-first search (BFS) dan depth-first search (DFS)**.

BREADTH-FIRST SEARCH (BFS)

Breadth-first search (BFS) bekerja dengan cara yang sangat mirip dengan algoritma level order traversal dalam struktur data tree. Algoritma BFS menelusuri graf secara bertingkat atau level by level. Prosesnya dimulai dengan:

1. Mengunjungi root node di level 0.
2. Kemudian seluruh nodes di level pertama yang terhubung langsung dengan root node dikunjungi di level 1.
3. Setiap level 1 node memiliki jarak 1 dari root node. Setelah seluruh nodes di level 1 selesai dikunjungi, penelusuran dilanjutkan ke level 2 nodes.
4. Proses yang sama terus dilakukan hingga seluruh nodes dalam graf selesai ditelusuri.

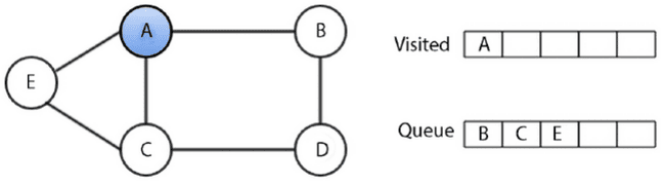
Dengan demikian, breadth-first traversal algorithms bekerja secara melebar atau breadthwise di dalam graf. Dalam pelaksanaannya, queue data structure digunakan untuk menyimpan informasi mengenai vertices yang akan dikunjungi dalam graf. Penelusuran dimulai dari starting node. Mula-mula, node tersebut dikunjungi, kemudian seluruh neighboring atau adjacent vertices-nya dicari. Selanjutnya, adjacent vertices tersebut dikunjungi satu per satu, sambil menambahkan neighbors dari masing-masing vertex ke dalam daftar vertices yang akan dikunjungi berikutnya. Proses ini terus dilanjutkan sampai seluruh vertices dalam graf telah selesai dikunjungi, dengan memastikan bahwa tidak ada vertex yang dikunjungi lebih dari satu kali.



Gambar 9.14: Graf Sederhana (Agarwal, 2022)

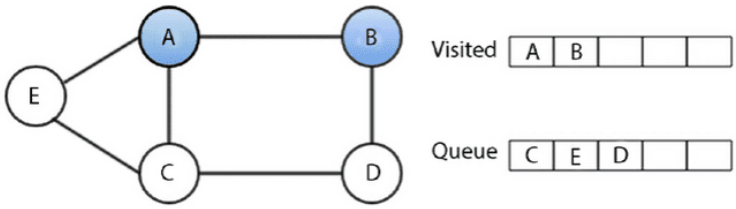
Untuk memahami cara kerja breadth-first traversal graf secara lebih jelas, dapat digunakan sebuah contoh graf seperti yang ditunjukkan dalam Figure 9.14. Dalam ilustrasi tersebut, sisi kiri memperlihatkan sebuah graph yang terdiri atas lima nodes, sedangkan sisi kanan menampilkan queue data structure yang digunakan untuk menyimpan vertices yang akan dikunjungi.

Proses penelusuran dimulai dari node A, kemudian seluruh adjacent vertices miliknya, yaitu B, C, dan E, dimasukkan ke dalam queue. Perlu diperhatikan bahwa terdapat lebih dari satu kemungkinan urutan untuk memasukkan adjacent nodes tersebut ke dalam queue, misalnya BCE, CEB, CBE, BEC, atau ECB, dan masing-masing urutan tersebut dapat menghasilkan hasil tree traversal yang berbeda. Meskipun demikian, seluruh kemungkinan itu tetap dianggap benar dalam graph traversal. Dalam contoh ini, nodes dimasukkan ke dalam queue berdasarkan urutan alfabet, yaitu BCE, agar proses penelusuran menjadi lebih sederhana. Dengan demikian, A node menjadi vertex pertama yang dikunjungi. Node dalam graf, yaitu A, telah diberi warna biru yang menandakan bahwa semuanya sudah dikunjungi. seperti terlihat di Gambar 9.15.



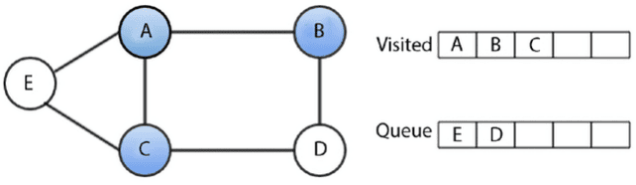
Gambar 9.15: Kunjungan Node A dalam BFS (Agarwal, 2022)

Setelah A vertex dikunjungi, langkah berikutnya adalah mengunjungi adjacent vertex pertama miliknya, yaitu B, lalu menambahkan adjacent vertices dari vertex B yang belum dikunjungi dan belum tercatat di dalam queue. Dalam kasus ini, D vertex perlu dimasukkan ke dalam queue, karena vertex B terhubung dengan A dan D, sedangkan A telah lebih dahulu dikunjungi, seperti terlihat di Gambar 9.16.



Gambar 9.16: Kunjungan Node B dalam BFS (Agarwal, 2022)

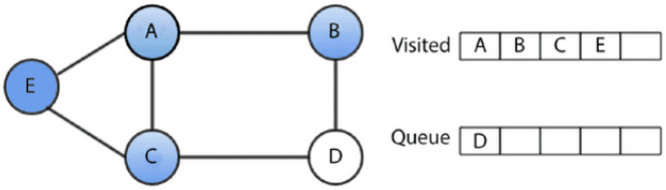
Sesudah B vertex selesai diproses, vertex berikutnya yang diambil dari queue adalah C vertex. Dari C vertex, penelusuran kembali memeriksa adjacent vertices yang belum tercatat di dalam queue, tetapi dalam contoh



Gambar 9.17: Kunjungan Node C dalam BFS (Agarwal, 2022)

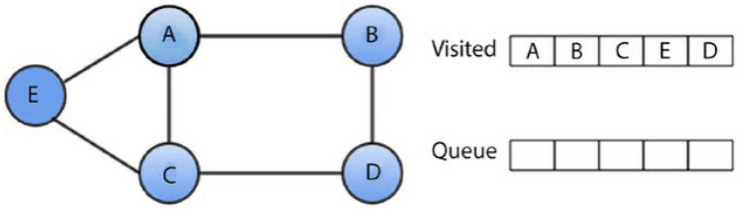
ini tidak ada lagi vertices baru yang perlu ditambahkan. bagian Queue menampilkan E dan D, seperti terlihat di Gambar 9.17.

Tahap berikutnya adalah mengunjungi E vertex dan bagian Queue menampilkan D sebagai satu-satunya node yang masih menunggu untuk diproses, seperti Gambar 9.18.



Gambar 9.18: Kunjungan Node E dalam BFS (Agarwal, 2022)

Kemudian dilanjutkan dengan D vertex sebagai langkah terakhir. Seluruh node dalam graf, yaitu A, B, C, D, dan E, telah diberi warna biru yang menandakan bahwa semuanya sudah dikunjungi. Bagian Visited memperlihatkan urutan kunjungan A, B, C, E, D, sehingga dapat dipahami bahwa node D merupakan node terakhir yang diproses dalam penelusuran ini. Sementara itu, bagian Queue terlihat kosong, yang berarti tidak ada lagi node yang menunggu untuk dikunjungi. Kondisi ini menegaskan bahwa proses BFS telah selesai karena seluruh node yang terhubung dalam graf telah berhasil ditelusuri, seperti terlihat di Gambar 9.19.



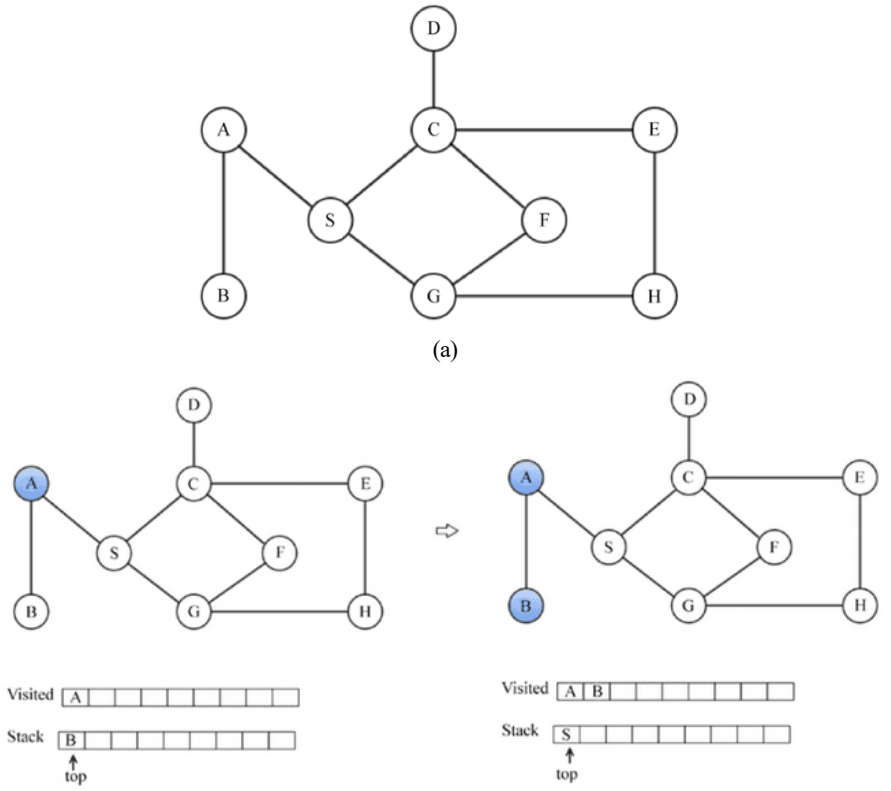
Gambar 9.19: Kunjungan Node D dalam BFS (Agarwal, 2022)

Berdasarkan urutan tersebut, algoritma BFS untuk menelusuri graph ini menghasilkan urutan kunjungan A-B-C-E-D. Urutan ini merupakan salah satu kemungkinan hasil BFS traversal, sedangkan kemungkinan lain tetap dapat muncul bergantung cara adjacent nodes dimasukkan ke dalam queue. Program BFS dapat dilihat di https://colab.research.google.com/drive/1A-V8H70Iaj6IMB8gjczx6DrsVACuodyB?usp=sharing.

DEPTH-FIRST SEARCH (DFS)

Sesuai dengan namanya, Depth-First Search (DFS) atau algoritma traversal menelusuri graf dengan cara yang serupa dengan cara kerja algoritma preorder traversal dalam tree. Dalam algoritma DFS, penelusuran dilakukan dengan mendalami suatu jalur tertentu di dalam graf terlebih dahulu. Dengan demikian, child nodes dikunjungi lebih dahulu dibandingkan sibling nodes. Berikut langkah DFS yaitu:

- Mulai dari root node.
- Kunjungi root node.
- Periksa seluruh adjacent vertices dari current node.
- Pilih salah satu adjacent node yang belum dikunjungi.
- Kunjungi unvisited node tersebut.
- Ulangi proses penelusuran secara mendalam dari node yang baru dikunjungi.
- Jika menemukan visited node, lakukan backtrack ke current node.
- Jika mencapai dead end, lakukan backtrack ke previous node.
- Lanjutkan penelusuran ke jalur lain yang masih memiliki unvisited node.
- Hentikan proses ketika seluruh node telah dikunjungi dan penelusuran kembali selesai di root node.



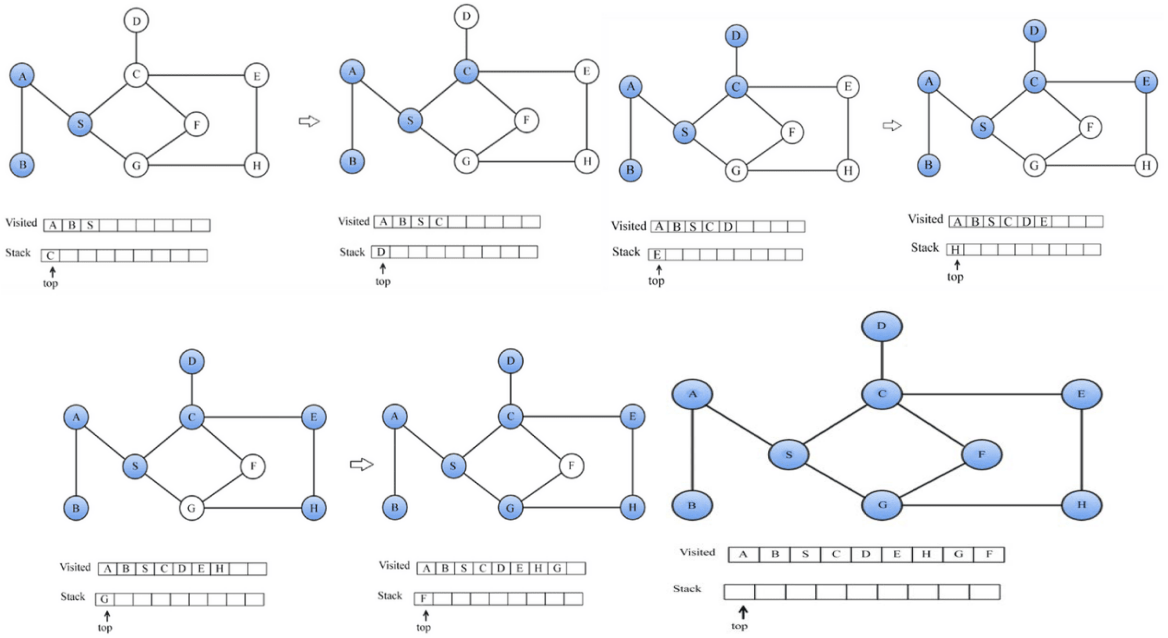
Gambar 9.20: (a) Contoh Graf (b) Penelusuran Node A,B dengan DFS (Agarwal, 2022)

Gambar 9.20 menyajikan ilustrasi proses DFS melalui dua bagian utama, yaitu (a) contoh graf dan (b) tahapan awal penelusuran DFS. Bagian (a) menampilkan sebuah graf yang terdiri atas sembilan simpul, yaitu A, B, S, C, D, E, F, G, dan H. Hubungan antarsimpul digambarkan melalui sejumlah sisi, misalnya A terhubung dengan B dan S, S terhubung dengan C dan G, C terhubung dengan D, E, dan F, sedangkan G terhubung dengan F dan H, serta E juga terhubung dengan H. Struktur ini memperlihatkan bahwa graf memiliki beberapa cabang dan jalur alternatif, sehingga sesuai digunakan untuk menjelaskan karakteristik DFS, yaitu penelusuran yang mendahulukan kedalaman lintasan sebelum berpindah ke cabang lain. Dengan susunan tersebut, simpul awal A menjadi titik masuk penelusuran, lalu algoritma bergerak ke simpul-simpul yang bertetangga secara bertahap sambil mempertimbangkan urutan kunjungan tertentu.

Bagian (b) memperlihatkan dua kondisi awal proses DFS beserta perubahan isi Visited dan Stack. Ilustrasi sebelah kiri menunjukkan tahap ketika simpul A telah dikunjungi lebih dahulu. Hal ini ditandai dengan warna biru di simpul A, sedangkan simpul lain masih berwarna putih. Baris Visited berisi A, yang berarti hanya simpul tersebut yang telah tercatat sebagai simpul yang sudah dikunjungi. Sementara itu, baris Stack menampilkan B di posisi teratas (top), yang menunjukkan bahwa B menjadi simpul berikutnya yang akan diproses. Kondisi ini menggambarkan mekanisme DFS yang menyimpan kandidat penelusuran dalam stack, sehingga simpul yang terakhir dimasukkan akan diproses lebih dahulu. Karena A terhubung langsung dengan B dan S, maka salah satu simpul tetangga dipilih untuk ditelusuri lebih dulu, dan dalam ilustrasi ini B ditempatkan sebagai elemen teratas stack.

Ilustrasi sebelah kanan menunjukkan langkah berikutnya, yaitu ketika simpul B telah berhasil dikunjungi. Hal ini ditandai dengan perubahan warna simpul B menjadi biru, sehingga kini terdapat dua simpul yang telah dikunjungi, yaitu A dan B. Baris Visited juga berubah menjadi A B, sedangkan Stack kini menampilkan S di posisi teratas. Keadaan ini menunjukkan bahwa setelah DFS bergerak dari A ke B, algoritma memeriksa tetangga simpul B. Karena B tidak memiliki simpul tetangga lain yang belum dikunjungi selain A, maka proses tidak dapat

dilanjutkan lebih dalam dari B. Oleh sebab itu, algoritma melakukan backtracking ke simpul sebelumnya, kemudian melanjutkan penelusuran ke cabang lain yang masih tersedia, yaitu S.



Gambar 9.21: Penelusuran Lanjutan dengan DFS (Agarwal, 2022)

Gambar 9.21 memperlihatkan penelusuran lanjutan DFS setelah simpul A, B, dan S dikunjungi. Ilustrasi disajikan secara bertahap dari kiri ke kanan serta dari atas ke bawah untuk menunjukkan perubahan Visited dan Stack di setiap langkah. Dalam penjelasan ini, Visited menyatakan urutan simpul yang benar-benar telah dikunjungi oleh algoritma, sedangkan Stack menunjukkan simpul yang akan diproses berikutnya menurut prinsip LIFO. Perlu dipahami bahwa gambar tersebut menyederhanakan tampilan Stack dengan menonjolkan simpul yang berada di posisi top, sehingga yang tampak terutama ialah simpul tujuan penelusuran berikutnya. Dengan demikian, fokus utama ilustrasi bukan menampilkan seluruh isi Stack secara rinci, melainkan memperlihatkan perubahan simpul teratas yang mengarahkan jalannya DFS.

Dalam subgambar pertama di baris atas sebelah kiri, simpul yang telah berwarna biru ialah A, B, dan S, sehingga baris Visited berisi A-B-S. Keadaan ini menunjukkan bahwa algoritma berangkat dari A, bergerak ke B, lalu melakukan backtracking karena B tidak memiliki tetangga baru yang dapat diteruskan, kemudian beralih ke S. Sesudah mencapai S, algoritma memeriksa tetangganya, yaitu C dan G. Karena ilustrasi mengikuti urutan tertentu, simpul C dipilih lebih dahulu. Itulah alasan Stack menampilkan C di posisi top. Maknanya, C merupakan simpul berikutnya yang akan diproses.

Dalam subgambar kedua, C telah dikunjungi sehingga simpul itu berubah menjadi biru dan Visited bertambah menjadi A-B-S-C. Setelah C masuk ke daftar Visited, algoritma memeriksa tetangga C, yaitu S, D, E, dan F. Karena S telah dikunjungi, jalur baru yang dipilih ialah D. Akibatnya, Stack berubah dan kini D berada di posisi top.

Dalam subgambar ketiga, D telah dikunjungi, ditandai oleh perubahan warna menjadi biru dan penambahan isi Visited menjadi A-B-S-C-D. Sesudah D diproses, algoritma memeriksa tetangga D. Namun, D hanya terhubung kembali ke C, sedangkan C sudah termasuk simpul yang telah dikunjungi. Keadaan ini menunjukkan bahwa cabang melalui D telah mencapai dead end. Karena tidak ada simpul baru yang dapat didatangi dari D, algoritma melakukan backtracking ke C dan mencari cabang lain yang belum ditelusuri. Dari C, simpul berikutnya yang masih tersedia ialah E, sehingga Stack menampilkan E sebagai elemen top.

Subgambar keempat memperlihatkan hasil langkah tersebut, E telah dikunjungi, simpulnya berubah menjadi biru, dan Visited menjadi A-B-S-C-D-E. Dari E, algoritma memeriksa tetangga yang terhubung. Karena

hubungan kembali ke C tidak dapat dipakai lagi, jalur yang dapat diteruskan mengarah ke H. Itulah sebabnya Stack dalam tahap tersebut menampilkan H di posisi teratas.

Dalam subgambar kelima di baris bawah sebelah kiri, H telah dikunjungi sehingga Visited bertambah menjadi A-B-S-C-D-E-H. Setelah itu, DFS memeriksa tetangga H, yaitu E dan G. Karena E sudah dikunjungi, satu-satunya jalur baru yang dapat diteruskan ialah ke G. Konsekuensinya, Stack berubah dan menempatkan G di posisi top.

Subgambar keenam menunjukkan bahwa G telah dikunjungi, sehingga urutan Visited berubah menjadi A-B-S-C-D-E-H-G. Dari G, algoritma memeriksa tetangga S, F, dan H. Dua simpul, yaitu S dan H, telah lebih dahulu dikunjungi, sehingga cabang yang masih dapat dilanjutkan tinggal F. Oleh karena itu, Stack kini menampilkan F sebagai elemen teratas. Tahap tersebut memperlihatkan dengan jelas prinsip DFS, yakni algoritma terus bergerak sedalam mungkin melalui satu jalur sebelum kembali mencari cabang alternatif lain.

Subgambar ketujuh di sebelah kanan bawah memperlihatkan kondisi akhir traversal. Dalam tahap tersebut, F telah berhasil dikunjungi sehingga Visited lengkap menjadi A-B-S-C-D-E-H-G-F. Setelah F diproses, algoritma memeriksa tetangganya, tetapi seluruh tetangga F telah masuk ke dalam daftar Visited. Akibatnya, tidak ada lagi simpul baru yang dapat dimasukkan ke dalam Stack. Karena semua cabang telah selesai ditelusuri, Stack menjadi kosong. Kekosongan Stack menandakan bahwa proses DFS telah berakhir.

Dengan demikian, gambar ini tidak hanya menunjukkan urutan kunjungan simpul, tetapi juga memperlihatkan peran dua komponen penting dalam DFS. Visited berfungsi mencatat simpul-simpul yang telah diproses agar algoritma tidak mengunjungi simpul yang sama berulang kali, sedangkan Stack berfungsi menentukan simpul mana yang harus diproses berikutnya sekaligus menjadi sarana utama untuk melakukan backtracking ketika suatu jalur tidak lagi memiliki kelanjutan. Secara keseluruhan, urutan Visited yang terbentuk ialah A-B-S-C-D-E-H-G-F, sedangkan perubahan Stack yang tampak dalam gambar bergerak dari C, lalu D, kemudian E, H, G, F, dan akhirnya kosong, yang seluruhnya mencerminkan mekanisme inti DFS secara sistematis.

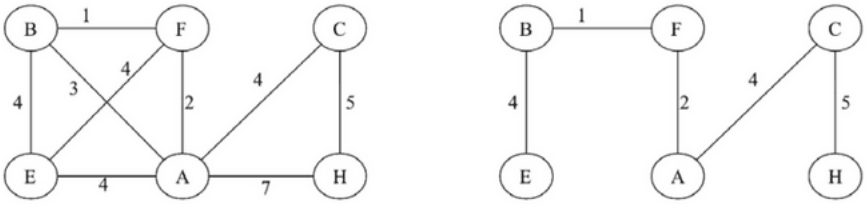
Program DFS dapat dilihat di link https://colab.research.google.com/drive/102jI5OvMGZJJLd5HVXFtPN-k5JztDUwa?usp=sharing.

9.3 Penggunaan Metode Graf

Graf sangat sering digunakan untuk menemukan path antara dua nodes. Dalam beberapa kasus, perlu menentukan seluruh paths yang mungkin menghubungkan dua nodes, sedangkan dalam situasi lain yang dibutuhkan ialah shortest path di antara keduanya. Sebagai contoh, dalam routing applications, berbagai algoritma umumnya digunakan untuk menentukan shortest path dari source node ke destination node. Untuk unweighted graph, penentuan jalur dapat dilakukan dengan memilih path yang memiliki jumlah edges paling sedikit. Sebaliknya, jika yang digunakan adalah weighted graph, maka perhitungan harus mempertimbangkan total weight dari sekumpulan edges yang dilalui. Oleh karena itu, dalam kondisi yang berbeda, pencarian longest path maupun shortest path memerlukan algoritma berbeda, salah satunya Minimum Spanning Tree.

Minimum Spanning Tree

Minimum Spanning Tree (MST) adalah subset dari **edges** dalam **connected graph** berbobot yang menghubungkan seluruh **nodes** di dalam graf dengan total **edge weights** serendah mungkin serta tanpa membentuk **cycle**. Secara lebih formal, jika diberikan **connected graph** GGG, dengan G=(V,E)G = (V, E)G=(V,E) dan **edge weights** bernilai riil, maka **MST** merupakan subgraf yang tersusun atas subset **edges** T⊆ET \subseteq ET⊆E sedemikian sehingga jumlah seluruh **edge weights** bernilai minimum dan tidak mengandung **cycle**. Terdapat banyak kemungkinan **spanning trees** yang dapat menghubungkan seluruh **nodes** tanpa **cycle**, tetapi **minimum weight spanning tree** adalah **spanning tree** yang memiliki total **edge weight**, atau **cost**, paling rendah dibandingkan seluruh kemungkinan **spanning trees** lainnya.



Gambar 9.22: Contoh Minimum Spanning Tree (Agarwal, 2022)

Gambar 9.22 menampilkan graf beserta **MST**-nya, dapat diamati bahwa semua **nodes** tetap terhubung melalui subset **edges** yang diambil dari graf asal, sementara total bobot seluruh **edges** dalam **MST** bernilai paling kecil, yaitu (1+4+2+4+5=16, jika dibandingkan dengan **spanning trees** lain yang mungkin terbentuk. **MST** memiliki beragam aplikasi nyata, terutama dalam **network design**, seperti pengaturan kemacetan jalan, jaringan kabel hidraulik, jaringan kabel listrik, hingga **cluster analysis**. Salah satu algoritma penting untuk membangun **MST** adalah algoritma **Kruskal's minimum spanning tree**.

Algoritma Kruskal's Minimum Spanning Tree

Kruskal's Minimum Spanning Tree algorithm merupakan algoritma yang banyak digunakan untuk menemukan spanning tree dari suatu weighted, connected, and undirected graph. Algoritma ini mengacu ke greedy approach, karena prosesnya dimulai dengan memilih edge yang memiliki weight paling rendah, lalu menambahkannya ke dalam tree. Selanjutnya, dalam setiap iterasi, algoritma kembali memilih edge dengan weight paling rendah untuk dimasukkan ke dalam spanning tree, dengan syarat bahwa penambahan edge tersebut tidak membentuk cycle. Tahap awal seluruh vertices dalam graf diperlakukan sebagai tree yang terpisah. Kemudian, di setiap iterasi dipilih edge dengan weight terendah sedemikian rupa sehingga tidak menghasilkan cycle. Tree-tree yang semula terpisah tersebut secara bertahap digabungkan dan berkembang hingga membentuk sebuah spanning tree. Proses ini diulangi sampai seluruh nodes selesai diproses.

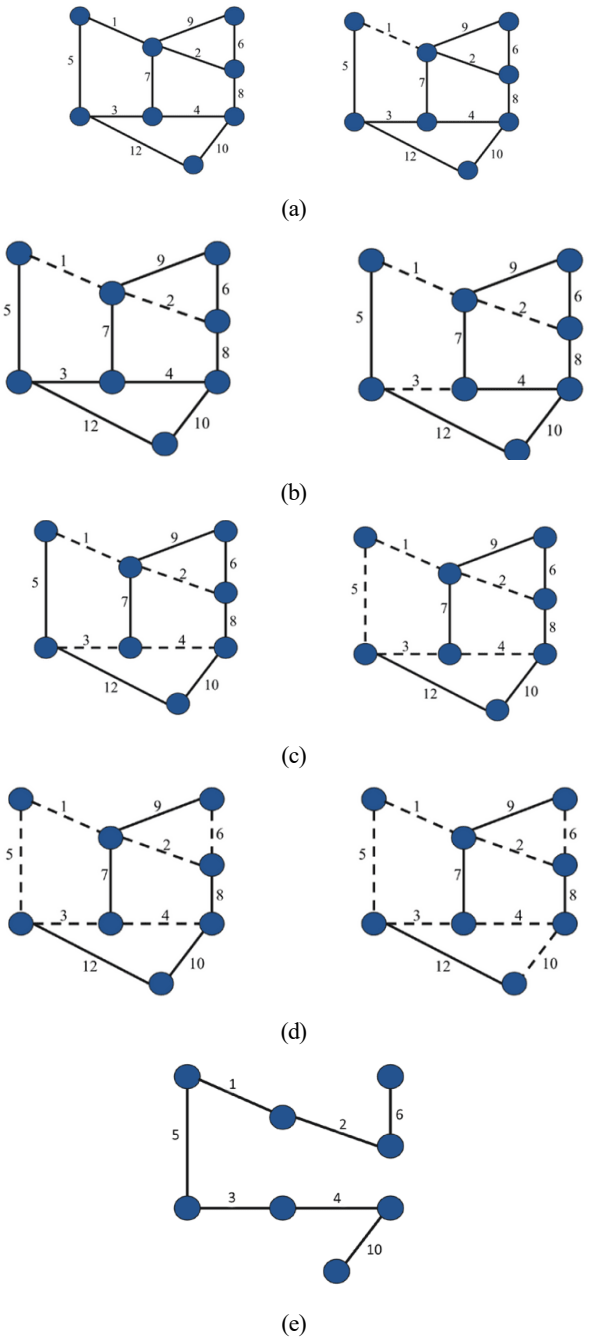
Secara umum, langkah kerja algoritma ini dapat dijelaskan sebagai berikut:

1. Mula-mula, dibentuk empty MST (M) yang belum memiliki edges sama sekali.
2. Setelah itu, seluruh edges **diurutkan** berdasarkan weights masing-masing.
3. Selanjutnya, setiap edge dari daftar yang telah diurutkan tersebut ditambahkan satu per satu ke dalam MST (M), dengan ketentuan bahwa penambahan itu tidak boleh membentuk cycle.

Untuk memahami proses tersebut secara lebih jelas, dapat digunakan sebuah contoh. Penelusuran dimulai dengan memilih edge yang memiliki weight paling kecil, yaitu weight 1. Contoh direpresentasikan oleh garis putus-putus dalam ilustrasi Gambar 8.23 yang menyajikan contoh kerja Kruskal's Minimum Spanning Tree secara bertahap melalui subgambar (a) hingga (e). Secara umum, ilustrasi ini memperlihatkan sebuah weighted, connected, and undirected graph yang memiliki delapan simpul dan sejumlah edge dengan bobot berbeda, lalu menunjukkan bagaimana algoritma Kruskal memilih edge satu demi satu berdasarkan urutan bobot terkecil sambil menjaga agar cycle tidak terbentuk. Tanda garis putus-putus menandai edge yang dipilih ke dalam kandidat MST, sedangkan subgambar terakhir menampilkan hasil akhir berupa spanning tree dengan total bobot minimum. Dengan cara ini, gambar tidak hanya memperlihatkan struktur graf asal, tetapi juga menunjukkan logika seleksi edge menurut prinsip greedy, yaitu selalu memilih edge termurah yang masih aman untuk ditambahkan.

Subgambar (a) menampilkan graf awal beserta tahap seleksi pertama. Graf asal memperlihatkan keterhubungan antarsimpul dengan bobot 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, dan 12. Sesuai prinsip Kruskal, algoritma mula-mula mengurutkan seluruh edge dari bobot terkecil ke terbesar. Edge pertama yang dipilih adalah edge berbobot 1, karena bobot ini merupakan nilai paling kecil di seluruh graf dan penambahannya jelas tidak

membentuk cycle, mengingat proses baru dimulai. Dalam ilustrasi, edge berbobot 1 itu ditandai dengan garis putus-putus sebagai edge pertama yang masuk ke dalam kandidat MST.



Gambar 9.23: Contoh Algoritma Kruskal's Minimum Spanning Tree (Agarwal, 2022)

Subgambar (b) memperlihatkan kelanjutan proses seleksi. Setelah edge berbobot 1 dipilih, algoritma memeriksa edge berikutnya menurut urutan bobot, yaitu 2. Edge berbobot 2 ditambahkan karena menghubungkan simpul yang belum berada dalam satu lintasan tertutup, sehingga cycle belum terbentuk. Sesudah itu, edge berbobot 3 juga dipilih dengan alasan yang sama, yaitu masih menghubungkan komponen yang berbeda dan tetap mempertahankan struktur acyclic. Tahap ini menunjukkan bahwa dalam Kruskal, penambahan edge tidak sekadar didasarkan bobot kecil, melainkan juga harus memenuhi syarat bahwa struktur yang terbentuk tetap berupa forest yang belum mengandung cycle.

Subgambar (c) menggambarkan tahap lanjutan ketika algoritma menambahkan edge berbobot 4 dan 5. Kedua edge tersebut dipilih karena masih menghubungkan bagian-bagian graf yang sebelumnya terpisah tanpa menghasilkan lintasan tertutup. Penambahan edge berbobot 4 memperluas keterhubungan di bagian bawah dan kanan graf, sedangkan edge berbobot 5 menghubungkan komponen kiri atas dengan komponen kiri bawah sehingga dua bagian besar graf mulai menyatu. Sampai tahap ini, struktur hasil seleksi telah menghubungkan semakin banyak simpul, tetapi masih belum mencakup seluruh graf. Dengan kata lain, forest yang dibangun secara bertahap sedang tumbuh menuju spanning tree, namun proses belum selesai karena masih ada simpul yang belum tersambung ke jaringan utama.

Subgambar (d) menunjukkan tahap kritis ketika algoritma menyeleksi edge berikutnya. Edge berbobot 6 dipilih karena masih diperlukan untuk menghubungkan simpul di bagian kanan atas dengan komponen utama yang telah terbentuk. Sebaliknya, beberapa edge lain seperti bobot 7, 8, dan 9 tidak dijadikan bagian dari MST, sebab penambahannya akan menghubungkan simpul-simpul yang sebenarnya sudah tersambung melalui jalur lain, sehingga cycle akan muncul. Di sinilah esensi Kruskal tampak jelas: algoritma tidak selalu mengambil seluruh edge berbobot kecil secara otomatis, melainkan hanya memilih edge yang benar-benar menambah keterhubungan tanpa melanggar syarat acyclic. Sesudah simpul-simpul bagian atas dan tengah tersambung, masih tersisa satu simpul di bagian bawah yang belum masuk ke struktur utama. Untuk menghubungkannya, algoritma memilih edge berbobot 10 karena edge ini lebih murah daripada alternatif lain yang berbobot 12. Dengan demikian, edge berbobot 12 tidak dipilih karena tidak lagi memberikan solusi minimum.

Subgambar (e) menampilkan hasil akhir Minimum Spanning Tree. Seluruh simpul dalam graf telah terhubung menggunakan subset edge berbobot 1, 2, 3, 4, 5, 6, dan 10, tanpa membentuk cycle. Jumlah edge yang dipilih adalah tujuh, sesuai dengan sifat spanning tree untuk graf yang memiliki delapan simpul, yaitu selalu terdiri atas n−1 edge. Jika seluruh bobot tersebut dijumlahkan, totalnya menjadi 31, dan nilai inilah yang merepresentasikan cost minimum untuk menghubungkan semua simpul dalam graf.

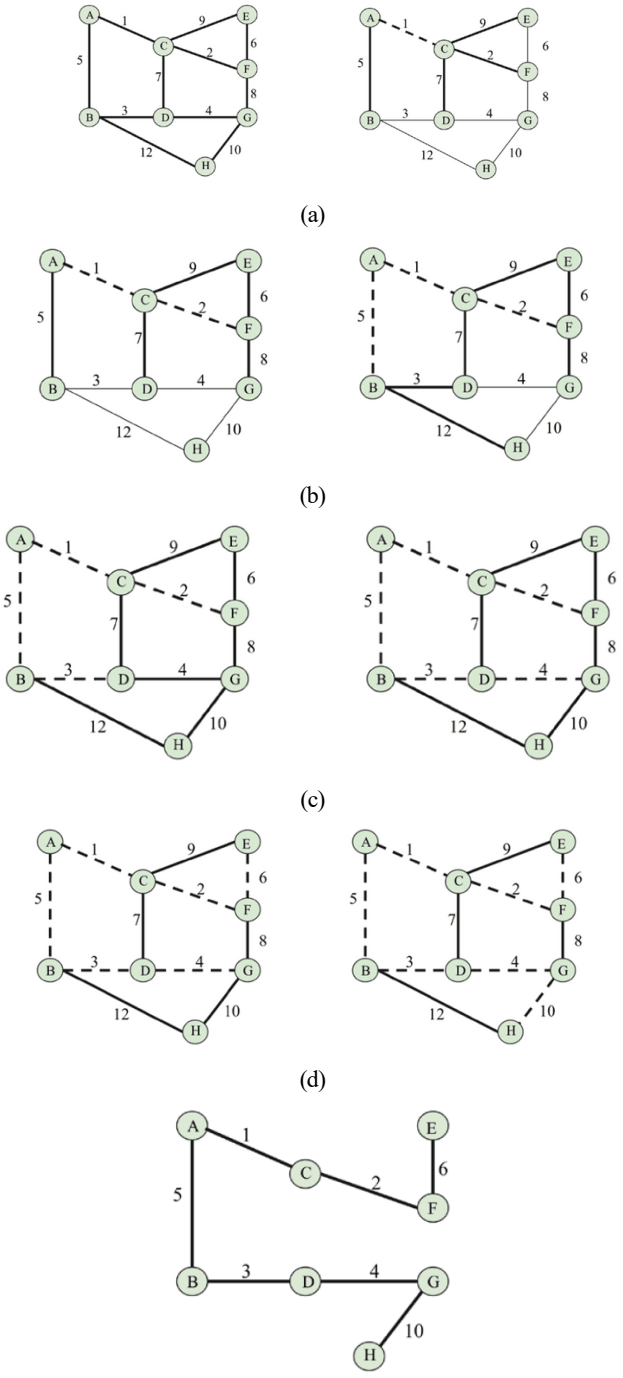
Program Algoritma Kruskal's Minimum Spanning Tree dapat dilihat di https://colab.research.google.com/drive/1sv28Ui_szho7ypCbeY3HBcLraoTTCwzt?usp=sharing.

Algoritma Prim's Minimum Spanning Tree

Prim's Minimum Spanning Tree algorithm juga mengacu ke greedy approach untuk menemukan minimum cost spanning tree. Algoritma Prim memiliki kemiripan yang sangat kuat dengan algoritma Dijkstra yang digunakan untuk menentukan shortest path dalam graf. Dalam algoritma ini, proses dimulai dari sebuah arbitrary node sebagai titik awal, kemudian seluruh outgoing edges dari selected nodes diperiksa, dan penelusuran dilanjutkan melalui edge yang memiliki cost atau weight paling rendah. Dalam konteks ini, istilah cost dan weight digunakan secara bergantian dengan makna yang sama. Setelah memulai dari selected node, tree dikembangkan sedikit demi sedikit dengan memilih edges satu per satu yang memiliki weight paling kecil dan tidak membentuk cycle. Secara umum, langkah kerja algoritma ini meliputi:

1. Pembuatan dictionary yang memuat seluruh edges beserta weights-nya.
2. Kemudian mengambil edges satu demi satu yang memiliki cost paling rendah dari dictionary tersebut untuk menumbuhkan tree dengan syarat cycle tidak terbentuk.
3. Ulangi proses itu sampai seluruh vertices telah dikunjungi.

Untuk memahami mekanisme kerja algoritma Prim secara lebih jelas, dapat digunakan sebuah contoh. Jika dipilih A node secara arbitrer sebagai titik awal, maka seluruh outgoing edges dari A diperiksa. Dalam kasus ini tersedia dua pilihan, yaitu AB dan AC, lalu dipilih edge AC karena memiliki cost/weight yang lebih kecil, yaitu weight 1.



Gambar 9.24: Contoh Graf (Agarwal, 2022)

Gambar 9.24 memperlihatkan tahapan kerja Prim's Minimum Spanning Tree algorithm dalam membentuk MST dari sebuah weighted, connected, and undirected graph. Ilustrasi disusun bertahap melalui subgambar (a) hingga (e) untuk menunjukkan bagaimana tree dibangun sedikit demi sedikit dengan prinsip greedy, yaitu selalu memilih edge dengan weight paling kecil yang menghubungkan simpul yang sudah masuk ke tree dengan simpul lain yang belum masuk, tanpa membentuk cycle. Garis putus-putus dalam setiap subgambar menandai edge yang dipilih ke dalam tree, sedangkan subgambar terakhir memperlihatkan hasil akhir MST. Berbeda dari Kruskal's algorithm yang mengurutkan seluruh edges lebih dahulu secara global, Prim's

algorithm memulai proses dari satu starting node tertentu, lalu menumbuhkan tree secara bertahap dari simpul awal tersebut ke simpul-simpul lain yang terdekat menurut bobot minimum.

Subgambar (a) menunjukkan tahap awal pemilihan starting node, yaitu A. Dari A, terdapat dua outgoing edges, yakni A–B dengan weight 5 dan A–C dengan weight 1. Sesuai prinsip Prim, algoritma memilih edge dengan weight paling kecil, sehingga A–C dipilih lebih dahulu. Setelah A dan C masuk ke tree, algoritma memeriksa seluruh outgoing edges yang keluar dari himpunan simpul terpilih tersebut menuju simpul yang belum tergabung. Pilihan yang tersedia mencakup A–B (5), C–D (7), C–E (9), dan C–F (2). Di antara pilihan tersebut, C–F memiliki weight terkecil, sehingga edge itulah yang ditambahkan berikutnya. Sampai tahap ini, tree sementara tersusun dari A–C dan C–F.

Subgambar (b) memperlihatkan kelanjutan proses ketika tree yang telah berisi A, C, dan F diperluas ke simpul lain. Dari himpunan simpul tersebut, outgoing edges yang masih mungkin dipilih adalah A–B (5), C–D (7), C–E (9), F–E (6), dan F–G (8). Karena A–B memiliki weight terendah, edge ini dipilih dan B masuk ke tree. Setelah itu, frontier baru yang menghubungkan tree dengan simpul luar menjadi lebih kaya, terutama karena B menyediakan edge B–D berbobot 3 dan B–H berbobot 12. Dalam kondisi tersebut, B–D (3) menjadi pilihan termurah sehingga ditambahkan ke dalam tree. Sampai akhir tahap ini, tree telah mencakup A, B, C, D, dan F, dengan edges terpilih A–C, C–F, A–B, dan B–D.

Subgambar (c) menunjukkan proses perluasan tree sesudah D tergabung. Dari himpunan simpul terpilih, tersedia beberapa kandidat edge, yakni D–G (4), F–E (6), C–D (7) tidak lagi relevan karena kedua ujungnya sudah berada dalam tree, F–G (8), C–E (9), dan B–H (12). Karena D–G berbobot 4, edge tersebut dipilih lebih dahulu sehingga G masuk ke tree. Sesudah itu, algoritma kembali mengevaluasi semua outgoing edges yang menghubungkan tree dengan simpul di luar tree. Pilihan terendah berikutnya adalah F–E (6), sehingga E ditambahkan. Tahap ini menegaskan ciri penting Prim's algorithm, yaitu pemilihan edge selalu dilakukan dari batas tree yang sudah terbentuk, bukan dari seluruh graf secara bebas. Akibatnya, tree tumbuh sebagai satu struktur yang tetap terhubung sejak awal hingga akhir.

Subgambar (d) memperlihatkan fase akhir ketika hampir seluruh simpul sudah tergabung. Setelah E masuk, simpul yang belum terhubung tinggal H. Beberapa edges menuju H tersedia, yaitu G–H (10) dan B–H (12). Karena G–H memiliki weight lebih rendah, maka edge inilah yang dipilih. Dengan masuknya H, seluruh simpul dalam graf telah berhasil dihubungkan. Tidak ada lagi simpul luar yang perlu dimasukkan, sehingga proses Prim's Minimum Spanning Tree algorithm selesai. Seluruh pemilihan tersebut selalu menjaga agar tidak muncul cycle, sebab setiap edge baru selalu menghubungkan tree yang telah ada dengan satu simpul baru di luar tree.

Subgambar (e) menampilkan hasil akhir Minimum Spanning Tree. Edges yang terpilih adalah A–C (1), C–F (2), A–B (5), B–D (3), D–G (4), F–E (6), dan G–H (10). Jumlah edge yang digunakan adalah tujuh, sesuai dengan sifat spanning tree untuk graf dengan delapan simpul, yaitu selalu terdiri atas $n-1$ edges. Jika seluruh weight dijumlahkan, totalnya adalah $1 + 2 + 5 + 3 + 4 + 6 + 10 = 31$. Dengan demikian, gambar ini menjelaskan bahwa Prim's algorithm membangun MST dengan memulai dari satu starting node, lalu secara bertahap memilih edge termurah yang menghubungkan tree dengan simpul baru sampai semua simpul terhubung dalam satu tree dengan cost minimum.

Program Algoritma Prim's Minimum Spanning Tree dapat dilihat di https://colab.research.google.com/drive/1yYAImEwr-pcIG_fYT4FB8xpIPYSw-YsF?usp=sharing.

BAB 10
Sorting dan Selection

Sorting (pengurutan) adalah suatu proses pengurutan data yang sebelumnya disusun secara acak atau tidak teratur menjadi urut atau teratur menurut suatu aturan tertentu (Agarwal, 2022). Selection adalah Proses **memilih satu atau beberapa elemen tertentu** dari kumpulan data tanpa harus mengurutkan semuanya (Baka, 2017). Terdapat dua jenis pengurutan data yaitu *ascending* (pengurutan dari kecil ke besar) dan *descending* (pengurutan dari besar ke kecil). Algoritma pengurutan merupakan salah satu algoritma yang paling penting dalam ilmu komputer dan banyak diterapkan dalam algoritma yang terkait dengan basis data. Dalam berbagai aplikasi, data yang telah diurutkan memungkinkan pengambilan informasi secara lebih efisien, misalnya kumpulan nama, nomor telepon, atau daftar tugas sederhana. Contoh:

Data Acak	: 45 66 78 201 93 25 100
Ascending	: 25 45 66 78 93 100 201
Descending	: 201 100 93 78 66 45 25

Tujuan sorting adalah untuk menyusun ulang elemen-elemen dalam suatu koleksi agar berada dalam urutan tertentu, misalnya dari yang terkecil ke yang terbesar, sehingga memudahkan operasi komputasi lainnya. Dalam Python, urutan alami objek biasanya ditentukan melalui operator < yang memiliki sifat irrefleksif dan transitif. Sifat transitif untuk menyimpulkan hasil perbandingan tertentu tanpa harus melakukan semua perbandingan secara eksplisit, sehingga mendorong efisiensi algoritma. Sorting berperan penting dalam penyimpanan data agar pencarian, misalnya menggunakan algoritma binary search, menjadi lebih efisien. Selain itu, banyak algoritma lanjutan menggunakan sorting sebagai subrutin. Python menyediakan metode bawaan untuk sorting, seperti sort dalam kelas list dan fungsi sorted, yang sudah dioptimalkan secara canggih. Meskipun programmer biasanya mengandalkan fungsi bawaan ini, pemahaman mendalam tentang algoritma pengurutan tetap penting untuk menilai efisiensi dan perilaku algoritma, serta untuk menerapkan prinsip-prinsip pengembangan algoritma dalam konteks komputasi lain (Goodrich et al., 2013). Alasan Melakukan Sorting Data yaitu:

- Data yang terurut mudah untuk dicari, mudah untuk diperiksa, dan mudah untuk dibetulkan jika terdapat kesalahan.
- Data yang terurut dengan baik juga mudah untuk dihapus jika sewaktu-waktu data tersebut tidak diperlukan lagi.
- Semakin mudah untuk menyisipkan data atapun melakukan penggabungan data.

Terdapat beberapa algoritma sorting yaitu selection sort, gnome sort, cycle sort, bubble sort, shell sort, introspective sort (introsort), pancake sort, insertion sort, cocktail sort, pigeonhole sort, bitonic sort, merge sort, bucket sort, timsort, flashsort, radix sort, comb sort, timsort, dan counting sort (Rahmawati & Setiawan, 2026). Sedangkan algortima yang termasuk selection yaitu heap sort, quick sort, smoothsort, randomize selection, quick selection, dan deterministic selection. Algoritma tersebut memiliki beragam pendekatan dan keunggulan masing-masing yang relevan dalam pengembangan algoritma secara umum. Kompleksitas suatu algoritma dilihat dari *worst case, best case,* dan *average case* (telah dijelaskan di Bab 1), yang dapat memengaruhi kinerjanya (Rumapea, 2017). Algoritma sorting dibagi menjadi dua teknik, (Fenyi et al., 2020), yaitu:

A. Teknik *comparison* (perbadingan) yaitu algoritma yang bekerja dengan membandingkan elemen dalam array satu sama lain. Algoritma yang menggunakan teknik ini yaitu bubble sort, insertion sort, merge sort, heap sort, selection sort, gnome sort, cycle sort, shell sort, introsort, pancake sort, cocktail sort, bitonic sort, dan smoothsort.
B. Teknik *non-comparison* (non-perbadingan) yaitu algoritma yang mengurutkan elemen tanpa membandingkan elemen secara langsung, tetapi dengan pendekatan lain. Algoritma yang menggunakan teknik ini yaitu radix sort, pigeonhole sort, bucket sort, dan flashsort.

10.1 Insertion Sort

Insertion sort merupakan algoritma pengurutan yang sederhana, bekerja secara in-place, dan cukup efisien untuk digunakan untuk data berukuran kecil atau data yang hampir terurut. Algoritma ini bekerja dengan cara menyisipkan setiap elemen ke posisi yang tepat di dalam sub-daftar yang sudah terurut. Selain itu, insertion sort hanya memerlukan tambahan memori yang konstan. Namun, kinerjanya akan menurun cukup signifikan ketika ukuran data masukan semakin besar dibandingkan dengan algoritma pengurutan lainnya (Gill, 2018).

Cara kerja insertion sort (Aung, 2019) sebagai berikut:

- Algoritma membandingkan elemen saat ini dengan nilai terbesar yang terdapat dalam bagian array yang sudah terurut.
- Jika elemen saat ini lebih besar, maka elemen tersebut dibiarkan dalam posisinya dan proses dilanjutkan ke elemen berikutnya. Namun, jika tidak, algoritma akan mencari posisi yang sesuai bagi elemen tersebut di dalam bagian array yang telah terurut dan memindahkannya ke posisi tersebut.

Algoritma Insertion Sort:

```
Algoritma InsertionSort(A):
  n = panjang array A
  for i = 2 to n do:
    key = A[i]        // Ambil elemen yang akan disisipkan
    j = i - 1

    // Geser elemen yang lebih besar dari key ke kanan
    while j >= 1 and A[j] > key do:
      A[j + 1] = A[j]  // Geser elemen ke kanan
      j = j - 1

    // Tempatkan key di posisi yang tepat
    A[j + 1] = key
```

- Proses ini dilakukan dengan menggeser semua elemen dalam bagian array yang telah terurut yang nilainya lebih besar daripada elemen saat ini satu posisi ke depan.

Algoritma ini memerlukan sejumlah perbandingan dan penyalinan elemen selama proses pengurutan. Tahap pertama, algoritma melakukan perbandingan maksimal terhadap satu elemen. Tahap kedua, perbandingan dapat mencapai maksimal dua elemen, dan proses ini terus berlanjut hingga tahap terakhir dilakukan maksimal N−1 perbandingan. Dengan demikian, jumlah total perbandingan yang mungkin terjadi adalah $1 + 2 + 3 + \ldots + (N-1) = N(N-1)/2$.

Namun, setiap tahap umumnya hanya sekitar setengah dari jumlah maksimum elemen yang benar-benar dibandingkan sebelum posisi penyisipan ditemukan. Oleh karena itu, jumlah perbandingan rata-rata dapat dianggap lebih kecil dibandingkan jumlah maksimum tersebut. Jumlah proses penyalinan elemen umumnya hampir sama dengan jumlah perbandingan yang dilakukan. Meskipun demikian, proses penyalinan tidak memerlukan waktu sebesar operasi pertukaran (swap). Oleh karena itu, data acak algoritma insertion sort biasanya bekerja lebih cepat dibandingkan dengan bubble sort dan selection sort. Dalam hal kompleksitas waktu, insertion sort memiliki kompleksitas $O(n^2)$ kondisi terburuk (*worst case*) maupun rata-rata (*average case*) ketika data bersifat acak. Sementara itu, kondisi terbaik (*best case*), yaitu ketika data sudah hampir atau sepenuhnya terurut, kompleksitas waktunya menjadi O(n).

Kelebihan algoritma insertion sort yaitu:

- Mudah diimplementasikan.
- Memiliki efisiensi sekitar dua kali lebih baik dibandingkan dengan bubble sort.
- Memiliki kompleksitas waktu linear dalam kondisi terbaik (*best case*), yaitu ketika data sudah dalam keadaan terurut.
- Kesederhanaan → pendekatan selection sort yang jelas dan sederhana membuat algoritma ini mudah dipahami serta efektif digunakan dalam pembelajaran konsep dasar pengurutan.
- Pengurutan in-place → algoritma ini mengurutkan array secara langsung tanpa memerlukan tambahan memori yang besar.

- Jumlah pertukaran minimal → *selection sort* meminimalkan jumlah operasi pertukaran (swap), biasanya hanya memerlukan paling banyak (n − 1) pertukaran, sehingga bermanfaat ketika jumlah operasi penulisan data perlu diminimalkan.

Keterbatasan algoritma insertion sort yaitu:

- Kurang efisien untuk array berukuran besar.
- Memiliki kompleksitas waktu kuadratik dalam kondisi terburuk (*worst case*), yaitu ketika data tersusun dalam urutan terbalik.
- Kurang efisien untuk dataset besar. Kompleksitas waktu $o(n^2)$ algoritma ini kurang efisien dibandingkan algoritma yang lebih optimal seperti quick sort atau merge sort ketika digunakan dalam dataset yang besar.
- Tidak stabil. Selection sort termasuk algoritma yang tidak stabil karena tidak menjamin urutan relatif elemen yang memiliki nilai sama tetap terjaga setelah proses pengurutan.
- Perbandingan yang berulang. Algoritma ini melakukan banyak perbandingan, bahkan ketika sebagian data sudah dalam keadaan terurut, sehingga dapat menimbulkan inefisiensi.

Contoh perhitungan manual insertion sort secara *ascending*:

```
Data awal: 9 4 6 1 8 3 7 2
Mulai dari i = 2
9 4 6 1 8 3 7 2
^
4 disisipkan di lokasi: di depan 9
jadi: 4 9 6 1 8 3 7 2

i = 3
4 9 6 1 8 3 7 2
^
6 disisipkan di lokasi: di antara 4 dan 9
jadi: 4 6 9 1 8 3 7 2

i = 4
4 6 9 1 8 3 7 2
^
1 disisipkan di lokasi: di depan 4
jadi: 1 4 6 9 8 3 7 2

i = 5
1 4 6 9 8 3 7 2
^
8 disisipkan di lokasi: di antara 6 dan 9
jadi: 1 4 6 8 9 3 7 2

i = 6
1 4 6 8 9 3 7 2
^
3 disisipkan di lokasi: di antara 1 dan 4
jadi: 1 3 4 6 8 9 7 2

i = 7
1 3 4 6 8 9 7 2
^
7 disisipkan di lokasi: di antara 6 dan 8
jadi: 1 3 4 6 7 8 9 2

i = 8
1 3 4 6 7 8 9 2
^
2 disisipkan di lokasi: di antara 1 dan 3
jadi: 1 2 3 4 6 7 8 9

Hasil akhir (ascending):
1 2 3 4 6 7 8 9
```

Contoh program insertion sort dapat dilihat di link https://colab.research.google.com/drive/1G-UF5sQVvswmsSYAqPV31H9Zc_KfmtGk?usp=sharing.

Latihan:
Soal 1
Diketahui sebuah array berikut:
12 5 9 3 15 8
Urutkan data tersebut menggunakan algoritma Insertion Sort secara *ascending*.
Tunjukkan proses setiap iterasi (i = 2 sampai n) seperti contoh berikut:
Data awal
Iterasi i = 2
Iterasi i = 3
Iterasi i = 4
Iterasi i = 5
Iterasi i = 6
Tuliskan juga posisi penyisipan elemen dalam setiap langkah.

Soal 2
Diketahui data berikut:
20 7 14 2 11 6 9
Gunakan algoritma insertion sort (*ascending*) untuk mengurutkan data tersebut.
Tunjukkan secara lengkap:
(a) Data awal sebelum pengurutan.
(b) Proses penyisipan elemen untuk setiap iterasi.
(c) Perubahan array setelah setiap langkah penyisipan.
(d) Hasil akhir setelah seluruh data terurut.
Tambahan:
Tandai elemen yang sedang diproses untuk setiap iterasi.
Jelaskan posisi elemen setelah proses penyisipan.

Soal 3
Buatkan code insertion sort dengan data dari soal 1 dan 2.

10.2 Bubble Sort

Bubble Sort adalah algoritma pengurutan yang sederhana dan paling lambat, yang bekerja berdasarkan prinsip bubbling out elemen terkecil atau terbesar dari array, tergantung apakah array perlu diurutkan dalam urutan menurun atau menaik. Algoritma ini membandingkan setiap elemen dalam array dengan elemen tetangganya, dan menukarnya jika urutannya tidak diinginkan. Jika jumlah elemen dalam array adalah 'n', maka dalam putaran pertama, dilakukan 'n-1' perbandingan nilai berdekatan, yang mengeluarkan elemen dengan peringkat 'n' dan menempatkannya di posisi terakhir. Operasi ini dijalankan sebanyak 'n-1' kali atau hingga algoritma melakukan satu putaran tanpa menukar pasangan elemen, yang menandakan bahwa array telah terurut. Ketika panjang array input sangat besar, algoritma ini bekerja secara tidak efisien dan karenanya tidak cocok digunakan untuk dataset besar (Gill, 2018).

Langkah-langkah dalam algoritma bubble sort, (Aung, 2019), sebagai berikut:

- Telusuri array dari kiri ke kanan.
- Bandingkan A[1] dan A[2], jika A[1] > A[2], maka kedua elemen tersebut ditukar.

Algoritma Buble Sort:

```
Algoritma BubbleSort(A):
  n = panjang array A
  repeat
    swapped = false
    for i = 1 to n-1 do:
      if A[i] > A[i+1] then:
        swap A[i] and A[i+1]
        swapped = true
    n = n - 1 // Mengurangi panjang array yang perlu diperiksa
  until swapped = false
// Mengulang sampai tidak ada pertukaran
```

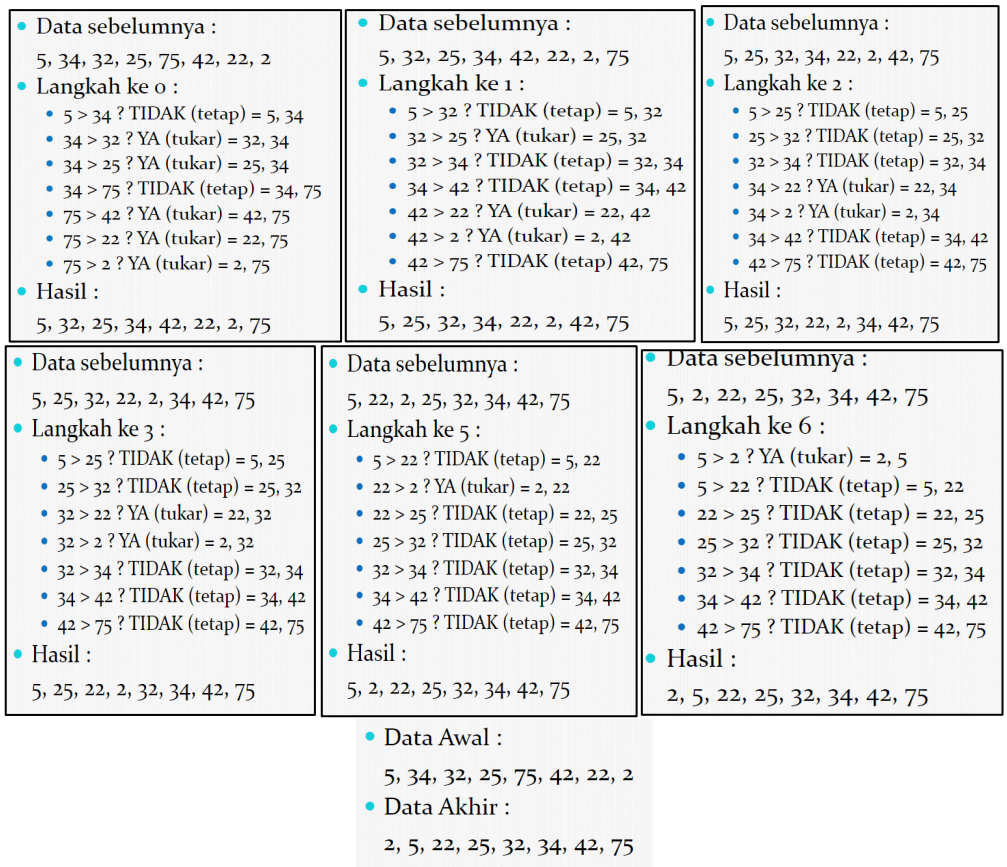
- Proses ini dilanjutkan untuk setiap pasangan elemen yang berdekatan hingga mencapai ujung kanan.
- Setelah melakukan N-1 perbandingan dan pertukaran jika diperlukan, maka A[N] akan berisi elemen terbesar.
- Kemudian, proses dimulai lagi dari dua elemen pertama dan diulang hingga tidak ada pertukaran yang terjadi di putaran terakhir.

Kelebihan algoritma bubble yaitu:

- Sederhana dan mudah untuk diimplementasikan. Bubble sort mudah dipahami dan diimplementasikan, menjadikannya pilihan yang sangat baik untuk memperkenalkan prinsip dasar pengurutan.
- Memiliki kemampuan untuk mengidentifikasi jika daftar sudah terurut, yang menghasilkan kompleksitas waktu terbaik O(n).
- Menggunakan ruang tambahan O(1).
- Efisiensi untuk dataset kecil. Bubble sort bisa menunjukkan efisiensi yang tak terduga untuk dataset yang sangat kecil, karena overhead dari algoritma yang lebih kompleks mungkin tidak dibenarkan.
- Kemampuan beradaptasi dengan daftar yang hampir terurut. Dalam kasus di mana daftar hampir terurut, bubble sort dapat mengalahkan skenario terburuknya dengan mengakhiri proses lebih awal begitu tidak ada lagi pertukaran yang diperlukan.

Kelemahan algoritma bubble yaitu:

- Tidak efisien untuk dataset besar. Bubble sort menunjukkan kompleksitas waktu terburuk dan rata-rata O(n^2), yang menjadikannya sangat tidak efisien untuk mengurutkan dataset besar dibandingkan dengan algoritma yang lebih canggih seperti quick sort atau merge sort.



Gambar 10.1: Penyelesaian Perhitungan Manual Bubble Sort secara Ascending

- Operasi yang redundan. Algoritma ini melakukan perbandingan dan pertukaran yang tidak perlu, terutama tahap awal, yang mengakibatkan waktu eksekusi yang lebih lama.
- Penggunaan praktis yang terbatas. Dikarenakan ketidakefisienannya, bubble sort jarang digunakan dalam situasi praktis untuk kinerja sangat penting, kecuali sebagai alat pedagogis atau dalam skenario tertentu yang karakteristiknya menguntungkan.

Gambar 10.1 menunjukkan contoh perhitungan manual bubble sort secara *ascending* untuk data acak yang diberikan, yaitu 5, 34, 32, 25, 75, 42, 22, 2. Proses dimulai dengan Langkah 1, di mana elemen-elemen dibandingkan berpasangan dan ditukar jika diperlukan. Setelah N-1 perbandingan di putaran pertama, elemen terbesar dipindahkan ke posisi terakhir. Langkah 2 dan seterusnya, perbandingan dilakukan terhadap elemen yang tersisa, dengan setiap iterasi mengurangi jumlah elemen yang perlu diproses. Proses ini berlanjut hingga Langkah 6 ketika tidak ada lagi pertukaran yang dilakukan, menandakan bahwa array telah terurut. Hasil akhir dari pengurutan data ini adalah 2, 5, 22, 25, 32, 34, 42, 75, yang merupakan urutan data secara menaik. Proses bubble sort ini mencakup perbandingan dan pertukaran yang berulang, dengan hasil akhir yang menunjukkan keefektifannya dalam mengurutkan data meskipun kurang efisien untuk dataset yang besar.

Berikut contoh program bubble sort dapat dilihat di link https://colab.research.google.com/drive/1Knp7lGNLMgkJN7wc2ENy1yCOu15LRhmM?usp=sharing.

Latihan:
Soal 1:
Diketahui array berikut: 60, 25, 35, 10, 50, 100, 15, 95, 75
Urutkan data tersebut menggunakan bubble sort secara *ascending*. Tunjukkan langkah-langkah perbandingan dan pertukaran untuk setiap putaran hingga array terurut.

Soal 2:
Diberikan array acak berikut: 12, 9, 5, 20, 15, 2, 17, 150, 112, 98, 99
Urutkan data tersebut menggunakan bubble sort secara *descending*. Lakukan perhitungan manual dan jelaskan langkah-langkah perbandingan dan pertukaran yang dilakukan untuk
setiap putaran.

Soal 3
Buatkan code bubble sort dengan data dari soal 1 dan 2

10.3 Selection Sort

Selection sort adalah teknik pengurutan dasar yang berbasis perbandingan, yang berfungsi dengan cara secara berulang memilih elemen terkecil dari bagian yang belum terurut dalam daftar dan menukarnya dengan elemen paling kiri yang belum terurut (Alaa et al., 2024). Proses dalam algoritma selection sort dilakukan dengan cara secara berulang memilih elemen terkecil dari bagian yang belum terurut dalam daftar dan menukarnya dengan elemen paling kiri yang belum terurut. Batas antara bagian yang terurut dan yang belum terurut digeser satu langkah ke kanan setelah setiap iterasi, dan proses ini berlanjut hingga seluruh daftar

Algoritma Selection Sort:

```
Algoritma SelectionSort(A):
  n = panjang array A
  for i = 1 to n-1 do:
    min_index = i  // Asumsi elemen terkecil berada di posisi i

    // Mencari elemen terkecil dalam sub-array yang belum terurut
    for j = i+1 to n do:
      if A[j] < A[min_index] then:
        min_index = j  // Temukan elemen terkecil

    // Tukar elemen terkecil dengan elemen di posisi i
    if min_index != i then:
      swap A[i] and A[min_index]

  return A
```

terurut. Kesederhanaan dan sifatnya yang mudah dipahami menjadikan selection sort sangat berguna untuk tujuan pendidikan. Namun, kompleksitas waktu $O(n^2)$ dalam skenario rata-rata dan terburuk membatasi efisiensinya dalam mengurutkan dataset besar. Selain itu, selection sort bersifat tidak stabil, yang berarti algoritma ini tidak mempertahankan urutan relatif dari elemen-elemen identik. Meskipun memiliki keterbatasan tersebut, kesederhanaan dan jumlah pertukaran yang minimal dapat bermanfaat dalam situasi tertentu, terutama ketika meminimalkan operasi penulisan menjadi hal yang krusial. Algoritma ini sangat cocok digunakan untuk file kecil karena setiap elemen hanya dipindahkan paling banyak satu kali. Meskipun kompleksitas waktu $O(n^2)$ dalam kasus terbaik dan terburuk menjadikannya tidak efisien untuk daftar yang besar, selection sort memiliki keunggulan dibandingkan teknik pengurutan lainnya. Meskipun melakukan banyak perbandingan, algoritma ini dapat mengurangi jumlah pertukaran yang dilakukan. Hal ini berarti, jika data input berukuran kecil dengan area data besar, maka selection sort dapat menjadi metode pengurutan yang tercepat. Selection sort adalah algoritma pengurutan in-place dan tidak dapat diimplementasikan sebagai algoritma pengurutan yang stabil.

Cara kerja algoritma selection sort, (Aung, 2019), sebagai berikut:

- Temukan elemen terkecil dalam daftar data.
- Tempatkan elemen terkecil di posisi pertama dalam daftar.
- Temukan elemen terkecil berikutnya dalam daftar.
- Tempatkan elemen tersebut dalam posisi kedua dalam daftar, dan lanjutkan proses ini hingga seluruh elemen dalam daftar terurut.

Kelebihan algoritma selection sort:

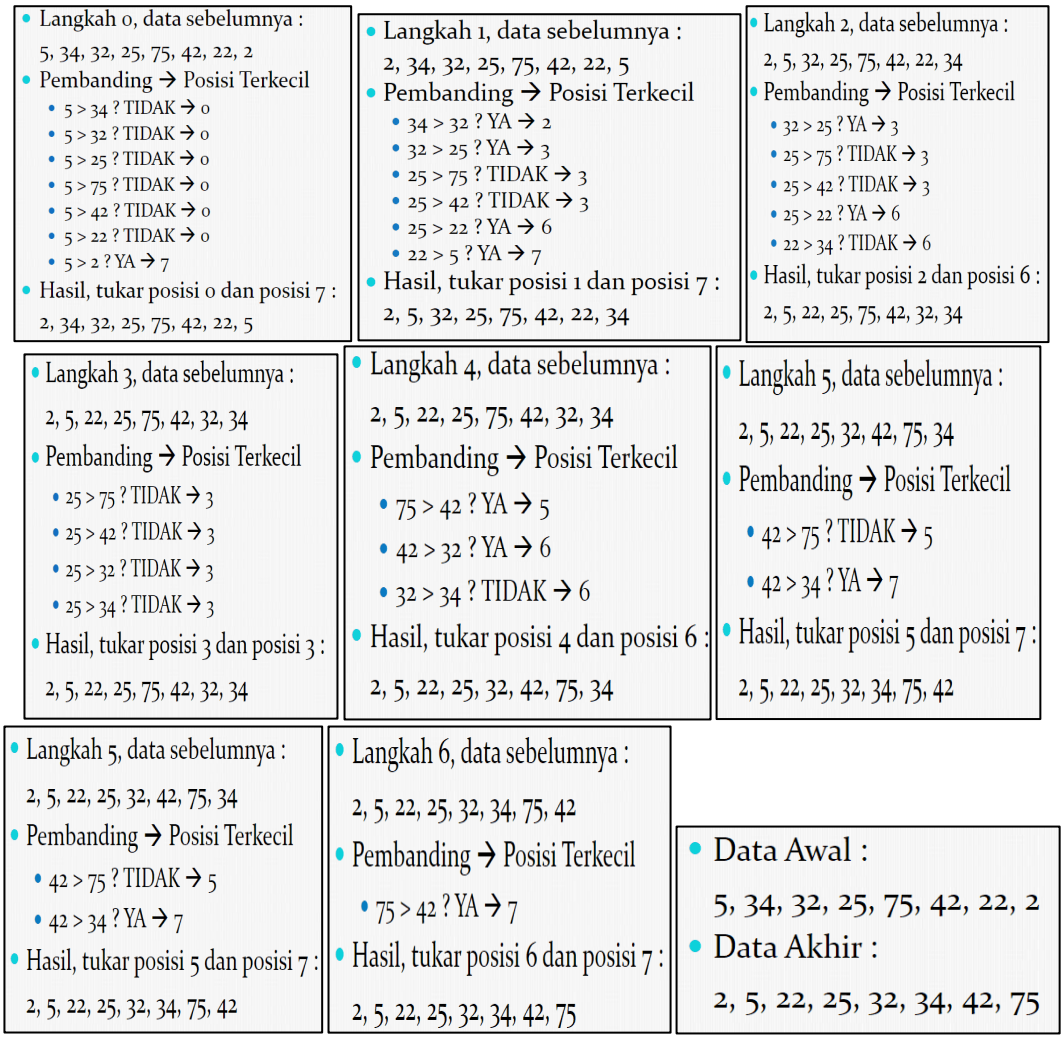
- Mudah diimplementasikan.
- Lebih efisien dibandingkan bubble sort.
- Sederhana. Pendekatan selection sort yang jelas dan langsung menjadikannya mudah dipahami dan efektif untuk mengajarkan prinsip dasar pengurutan.
- Pengurutan in-place. Algoritma ini mengurutkan array input secara langsung, memerlukan memori tambahan yang minimal.
- Pertukaran minimal. Selection sort meminimalkan jumlah pertukaran yang diperlukan, biasanya hanya memerlukan maksimal $(n - 1)$ pertukaran, yang dapat menguntungkan ketika meminimalkan operasi penulisan sangat penting.

Kekurangan algoritma selection sort:

- Tidak efisien untuk data yang besar. Dengan kompleksitas waktu $O(n^2)$, algoritma ini kurang efisien untuk mengurutkan dataset besar dibandingkan dengan algoritma yang lebih dioptimalkan seperti quick sort atau merge sort.
- Tidak memberikan kompleksitas waktu linear untuk skenario apapun.
- Pengurutan tidak stabil. Selection sort bersifat tidak stabil artinya algoritma ini tidak menjamin urutan relatif elemen-elemen yang identik tetap terjaga.
- Perbandingan yang redundan. Algoritma ini melakukan banyak perbandingan, bahkan dalam kasus dimana daftar sudah sebagian terurut, yang dapat menghasilkan inefisiensi.

Gambar 10.2 menunjukkan penyelesaian manual algoritma Selection Sort secara ascending. Dalam proses ini, data yang diberikan adalah 5, 34, 32, 25, 75, 42, 22, 2, dan algoritma bekerja dengan cara memilih elemen terkecil untuk setiap iterasi, lalu menukarnya dengan elemen yang ada di posisi yang tepat. Langkah 1, algoritma membandingkan setiap elemen untuk mencari elemen terkecil. Setelah perbandingan, elemen terkecil (2) ditempatkan di posisi pertama, mengubah urutan menjadi 2, 34, 32, 25, 75, 42, 22, 5. Langkah 2, proses serupa dilakukan, dan elemen terkecil berikutnya (5) dipindahkan ke posisi kedua, menghasilkan urutan 2, 5, 32, 25, 75, 42, 22, 34. Proses ini berlanjut hingga Langkah 6, elemen-elemen dipindahkan ke posisi yang sesuai satu per satu, hingga array terurut sepenuhnya. Langkah 6, elemen terbesar (75) dipindahkan ke posisi terakhir, dan akhirnya

array terurut menjadi 2, 5, 22, 25, 32, 34, 42, 75. Proses ini menggambarkan cara Selection Sort bekerja, yaitu memilih elemen terkecil secara berulang dan menukarnya hingga seluruh array terurut secara ascending.



Gambar 10.2: Penyelesaian Perhitungan Manual Selection Sort secara Ascending

Berikut contoh program selection sort di link https://colab.research.google.com/drive/1zjY0U4McL9Gow0QSjekfDZFWXGX87933?usp=sharing.

Latihan:
Soal 1:
Diketahui sebuah array berikut: [72, 15, 33, 4, 23, 56, 18]
Urutkan data tersebut menggunakan Selection Sort secara ascending. Tunjukkan langkah-langkah perbandingan dan pertukaran untuk setiap putaran hingga array terurut.

Soal 2:
Diberikan array berikut: [91, 28, 47, 56, 12, 65, 73, 9]
Urutkan array tersebut menggunakan Selection Sort secara ascending. Lakukan perhitungan manual dan jelaskan langkah-langkah perbandingan dan pertukaran yang dilakukan untuk setiap putaran.

Soal 3
Buatkan code selection sort dengan data dari soal 1 dan 2.

10.4 Merge Sort

Merge sort adalah teknik pengurutan yang menggunakan pendekatan *divide-and-conquer*. Teknik ini membagi array menjadi dua sub-array, kemudian mengurutkan setiap sub-array secara rekursif, dan akhirnya menggabungkan sub-array yang sudah terurut untuk menghasilkan array yang terurut. Proses ini bersifat rekursif; array awal dibagi menjadi dua bagian yang hampir sama hingga tidak bisa dibagi lagi, yang berarti hanya tersisa satu elemen di setiap sub-array (Sundaramoorthy & Karunanidhi, 2025),.

Merge sort merupakan algoritma yang sangat efisien dan stabil yang cocok untuk mengurutkan dataset besar. Pendekatan *divide-and-conquer* memastikan data diurutkan dengan kompleksitas waktu O(nlogn), meskipun membutuhkan memori tambahan untuk proses penggabungan. Algoritma ini ideal untuk aplikasi di mana stabilitas sangat penting dan ketersediaan memori bukanlah kendala (Alaa et al., 2024).

Cara kerja algoritma merge sort sebagai berikut:

- *Splitting*. Bagi array yang belum terurut menjadi dua bagian yang sama (sub-array) hingga setiap sub-array hanya berisi satu elemen.
- *Sorting*. Secara rekursif urutkan sub-array yang telah dibuat dalam langkah sebelumnya. Proses rekursif ini berlanjut hingga semua sub-array hanya berisi satu elemen, karena satu elemen dianggap sudah terurut.
- *Merging*. Gabungkan sub-array yang sudah terurut untuk menghasilkan sub-array yang lebih besar dan terurut. Proses penggabungan ini menggabungkan dua sub-array terurut menjadi satu array terurut. Proses ini terus berlanjut hingga akhirnya terbentuk satu array terurut, yang merupakan array terurut akhir.

Berikut algoritma merge sort berdasarkan cara kerjanya.

Algoritma Merge Sort:

```
Algoritma MergeSort(A):
  if panjang A > 1 then:
    # Langkah 1: Splitting
    mid = panjang A // 2  // Tentukan titik tengah array
    left_half = A[0:mid]  // Bagi array menjadi dua bagian
    right_half = A[mid:]  // Bagi array menjadi dua bagian

    # Langkah 2: Sorting
    MergeSort(left_half)   // Urutkan sub-array kiri secara rekursif
    MergeSort(right_half)  // Urutkan sub-array kanan secara rekursif

    # Langkah 3: Merging
    i = 0  # Indeks untuk sub-array kiri
    j = 0  # Indeks untuk sub-array kanan
    k = 0  # Indeks untuk array gabungan

    # Gabungkan dua sub-array yang sudah terurut
    while i < panjang left_half and j < panjang right_half:
      if left_half[i] < right_half[j]:
        A[k] = left_half[i]  # Ambil elemen terkecil dari kiri
        i += 1
      else:
        A[k] = right_half[j]  # Ambil elemen terkecil dari kanan
        j += 1
      k += 1

    # Salin sisa elemen kiri (jika ada)
    while i < panjang left_half:
      A[k] = left_half[i]
      i += 1
      k += 1

    # Salin sisa elemen kanan (jika ada)
    while j < panjang right_half:
      A[k] = right_half[j]
      j += 1
      k += 1

  return A
```

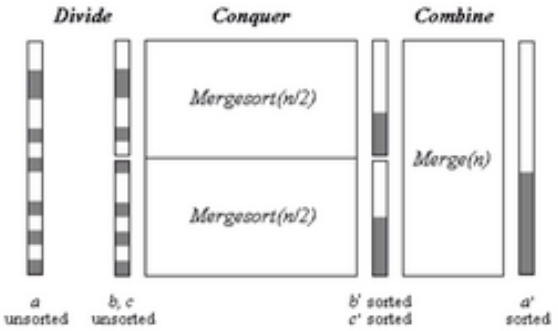
Kelebihan algoritma merge sort adalah:

- Efisiensi, merge sort secara konsisten mencapai kompleksitas waktu o(nlogn) di semua skenario (terbaik, rata-rata, dan terburuk), menjadikannya sangat efektif untuk menangani dataset yang besar.
- Stabilitas, algoritma ini mempertahankan urutan elemen dengan kunci yang sama, memastikan stabilitas dalam pengurutan.
- Potensi untuk paralelisasi. Strategi *divide-and-conquer* dalam merge sort memungkinkan paralelisasi yang mudah, meningkatkan skalabilitasnya dalam sistem multi-core.

Kelemahan algoritma merge sort yaitu:

- Kompleksitas Ruang → merge sort membutuhkan ruang memori tambahan yang proporsional dengan ukuran array selama fase penggabungan. Ini dapat menjadi tantangan dalam lingkungan dengan keterbatasan memori.

- Kompleksitas versus kesederhanaan. Tidak seperti metode pengurutan yang lebih sederhana seperti bubble sort atau insertion sort, merge sort lebih kompleks, sehingga kurang cocok untuk pengaturan pendidikan yang lebih mengutamakan kesederhanaan.



Gambar 10.3: Proses Kerja Algoritma Merge Sort

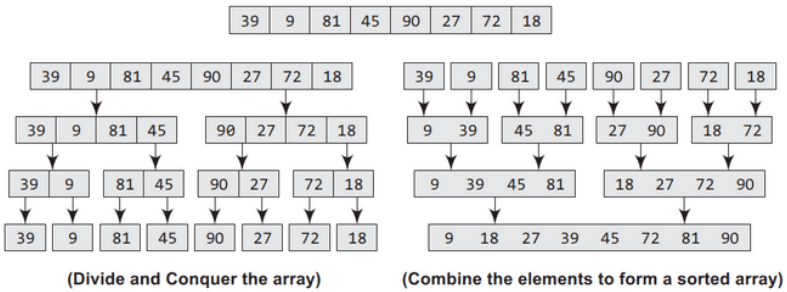
Gambar 10.3 menggambarkan proses kerja algoritma Merge Sort dalam tiga tahapan utama: Divide, Conquer, dan Combine.

1. *Divide* (Pembagian)
   Array yang tidak terurut (ditandai dengan a) dibagi menjadi dua bagian yang lebih kecil. Bagian pertama adalah b, dan bagian kedua adalah c. Pembagian ini dilakukan secara rekursif hingga setiap sub-array hanya berisi satu elemen, yang dianggap sudah terurut.
2. *Conquer* (Pengurutan)
   Setiap sub-array (b dan c) kemudian diproses secara rekursif dengan algoritma MergeSort(n/2). Proses ini membagi lagi hingga seluruh elemen terurut. Hasil akhirnya adalah dua sub-array yang terurut (b' dan c').
3. *Combine* (Penggabungan)
   Setelah dua sub-array terurut (b' dan c'), keduanya digabungkan menggunakan proses Merge(n) untuk menghasilkan array yang terurut secara keseluruhan (a').

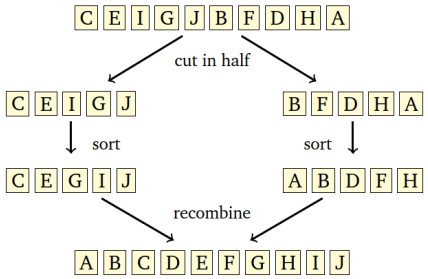
Dengan demikian, algoritma Merge Sort bekerja dengan prinsip *divide-and-conquer*, yaitu membagi masalah besar menjadi sub-masalah yang lebih kecil, mengurutkan masing-masing sub-masalah, dan kemudian menggabungkan hasilnya untuk mendapatkan solusi akhir yang terurut.

Gambar 10.4 menggambarkan tahapan algoritma merge sort dengan dua data berbeda, Gambar (a) dengan data angka, Gambar (b) menggunakan data karakter. Gambar (a) terbagi menjadi dua bagian utama, yaitu *Divide and Conquer* dan *Combine*. Gambar (a) tahap pertama, *Divide and Conquer*, array yang belum terurut dibagi menjadi dua sub-array yang lebih kecil secara berulang hingga setiap sub-array hanya berisi satu elemen. Proses ini memastikan bahwa setiap elemen sudah terpisah dan siap untuk diurutkan. Gambar (a) tahap kedua, yaitu *Combine*, sub-array yang telah terurut digabungkan kembali dengan membandingkan elemen-elemen dari setiap sub-array. Elemen yang lebih kecil ditempatkan terlebih dahulu, dan proses ini terus berlanjut hingga menghasilkan satu array yang terurut. Contoh yang diberikan menunjukkan bagaimana array awal [39, 9, 81, 45, 90, 27, 72, 18] dibagi menjadi bagian-bagian lebih kecil, kemudian digabungkan kembali menjadi array terurut [9, 18, 27, 39, 45, 72, 81, 90]. Algoritma ini sangat efisien dengan kompleksitas waktu O(n log n), yang membuatnya sangat cocok untuk mengurutkan dataset besar, mengingat proses pembagian dan penggabungan yang terstruktur dengan baik. Gambar (b) menggambarkan ilustrasi algoritma Merge Sort dengan fokus yaitu tahapan Divide (pembagian), Sort (pengurutan), dan Combine (penggabungan). Tahap pertama, array yang tidak terurut dibagi menjadi dua bagian, yaitu sub-array CEIGJ dan BFDHAA. Proses ini dilakukan berulang hingga setiap sub-array hanya berisi satu elemen. Setelah itu, setiap sub-array yang telah terpisah disortir secara rekursif, dengan CEIGJ disortir menjadi CEGIJ dan BFDHAA disortir menjadi ABDFH. Setelah kedua sub-array terurut, langkah berikutnya adalah *Combine*, di mana dua sub-array yang sudah terurut digabungkan untuk membentuk

array yang lebih besar dan terurut. Hasil penggabungan kedua sub-array tersebut adalah ABCDEFGHIJ, yang merupakan array akhir yang telah terurut sepenuhnya. Ilustrasi ini menunjukkan bagaimana Merge Sort bekerja dengan membagi masalah menjadi sub-masalah yang lebih kecil, mengurutkannya secara terpisah, dan kemudian menggabungkannya untuk membentuk solusi yang utuh.



(a)



(b)

Gambar 10.4: Ilustrasi Algoritma Merge Sort (a) (Thareja, 2014) (b) (Heinold, 2025)

Latihan:
Soal 1:
Diketahui array berikut: [45, 12, 89, 34, 23, 56, 71, 4]
Urutkan data tersebut menggunakan Merge Sort secara ascending. Tunjukkan langkah-langkah pembagian dan penggabungan dalam setiap tahap hingga array terurut.

Soal 2:
Diberikan array berikut: [17, 39, 58, 23, 89, 45, 12, 62]
Urutkan data tersebut menggunakan Merge Sort secara descending. Lakukan perhitungan manual dan jelaskan langkah-langkah perbandingan, pembagian, dan penggabungan yang dilakukan dalam setiap putaran hingga array terurut dalam urutan menurun.

Soal 3
Buatkan code merge sort dengan data dari soal 1 dan 2.

Berikut contoh program algoritma merge sort dapat dilihat di link https://colab.research.google.com/drive/1l24cQoDw1CdCFN_Aiy6pevLHdQTyIssD?usp=sharing.

10.5 Radix Sort

Radix Sort merupakan algoritma pengurutan yang memiliki sifat linier, stabil, dan tidak bergantung terhadap perbandingan antar elemen. Algoritma ini bekerja dengan cara mengurutkan array berdasarkan posisi digitnya secara terpisah, bukan dengan membandingkan elemen satu per satu. Setiap elemen dalam array ditempatkan ke kelompok yang sesuai dengan nilai digit tertentu (radix). Proses pengurutan berlanjut dengan mengurutkan elemen-elemen tersebut secara berulang, dimulai dari digit yang paling tidak signifikan hingga yang paling signifikan (Sundaramoorthy & Karunanidhi, 2025).

Cara kerja radix sort secara singkat adalah membagi data dalam beberapa kolom (pocket), selanjutnya mengurutkan data berdasarkan karakter dalam posisi kolom terakhir sampai kolom pertama. Apabila maksimum jumlah digit dari data tersebut adalah "m", maka ada m tahap pengurutan.

Algoritma Radix Sort:

```
Algoritma RadixSort(A):
  m = jumlah digit maksimum dalam data A

  for digit_position = 1 to m do:
    buat wadah (bucket) untuk setiap digit (0-9)

    for setiap elemen dalam A do:
      tentukan digit posisi digit_position dari elemen tersebut
      masukkan elemen ke dalam wadah sesuai digitnya

    A = gabungkan semua elemen dari wadah ke dalam array A sesuai urutan
  return A
```

Algoritma dimulai dengan mengidentifikasi nilai elemen tertinggi dalam array, kemudian digunakan untuk menentukan jumlah iterasi yang dibutuhkan. Jumlah iterasi ini untuk mengurutkan array berdasarkan digit terbesar. Jumlah digit dalam elemen tertinggi menentukan banyaknya iterasi, dengan setiap iterasi mengurutkan satu posisi digit. Wadah (*buckets*) dibuat sebanyak basis angka yang digunakan untuk pengurutan, misalnya untuk bilangan bulat dengan basis 10, maka k = 10. Algoritma mengurutkan array dengan memproses digit per digit dimulai dari digit yang paling tidak signifikan, mendistribusikan elemen-elemen ke dalam wadah sesuai nilai digitnya. Setelah itu, elemen dikumpulkan kembali dari wadah sesuai urutan asalnya, mempertahankan urutan di dalam setiap wadah, hingga array terurut.

Radix sort muncul sebagai solusi yang efisien untuk mengurutkan dataset yang memiliki panjang tetap, seperti bilangan bulat atau string. Keefektifannya sangat terlihat ketika rentang dataset tidak jauh melebihi jumlah item yang akan diurutkan. Dalam dataset relatif terbatas, radix sort dengan efisien mengorganisasi elemen-elemen dengan memeriksa digit atau karakter secara terpisah, yang memperlancar proses pengurutan. Pendekatan ini memastikan bahwa radix sort dapat menangani dataset besar dengan efisiensi yang luar biasa, memberikan solusi yang andal untuk berbagai macam tugas pengurutan.

Kelebihan algoritma radix sort yaitu:

- Dapat menangani kunci yang besar dengan lebih efisien.
- Dapat diimplementasikan untuk mengurutkan data alfabetik.
- Radix sort dapat sangat cepat untuk mengurutkan angka dengan panjang tetap, mengungguli algoritma pengurutan berbasis perbandingan seperti quick sort dan merge sort.
- Algoritma ini tidak membandingkan elemen satu sama lain (non-Komparatif), yang dalam beberapa situasi bisa lebih efisien.

Keterbatasan algoritma radix sort yaitu:

- Kurang fleksibel.
- Membutuhkan ruang memori yang lebih banyak.
- Lingkup Terbatas. Radix sort hanya dapat diterapkan dalam elemen yang dapat diurutkan berdasarkan kunci yang menyerupai digit (misalnya, bilangan bulat, string).
- Algoritma ini memerlukan ruang tambahan untuk menyimpan array penghitung dan array output, yang bisa menjadi signifikan terhadap dataset besar.

Gambar 10.5 menunjukkan proses pengurutan data menggunakan algoritma Radix Sort secara ascending terhadap deretan bilangan 73, 65, 52, 77, 24, 83, 17, 35, 96, 62, 41, 87, 9, dan 11. Proses pengurutan dilakukan berdasarkan nilai digit secara bertahap, dimulai dari digit dengan bobot terendah, yaitu satuan, kemudian dilanjutkan ke digit yang lebih tinggi, yaitu puluhan. Iterasi pertama menunjukkan setiap bilangan dikelompokkan ke dalam pocket 0 sampai 9 sesuai angka satuannya. Setelah seluruh elemen ditempatkan, isi pocket digabungkan

kembali secara berurutan sehingga diperoleh susunan sementara, yaitu 41, 11, 52, 62, 73, 83, 24, 65, 35, 96, 77, 17, 87, dan 09. Iterasi kedua menunjukkan hasil iterasi pertama kembali dikelompokkan berdasarkan angka puluhannya. Setelah semua pocket digabungkan secara berurutan, diperoleh hasil akhir pengurutan, yaitu 09, 11, 17, 24, 35, 41, 52, 62, 65, 73, 77, 83, 87, dan 96.

Contoh:
73, 65, 52, 77, 24, 83, 17, 35, 96, 62, 41, 87, 9, 11
Jumlah iterasi (m) = jumlah digit terbanyak
Iterasi 1 :

Pocket:	0	1	2	3	4	5	6	7	8	9
		41	52	73	24	65	96	77		09
Langkah-1		11	62	83		35		17		
(satuan)								87		

Gabungkan isi semua elemen:
41, 11, 52, 62, 73, 83, 24, 65, 35, 96, 77, 17, 87, 09

(a)

- Iterasi Ke 2 :
  Hasil iterasi 1 (satuan) :
  41, 11, 52, 62, 73, 83, 24, 65, 35, 96, 77, 17, 87, 09

Pocket:	0	1	2	3	4	5	6	7	8	9
	09	11	24	35	41	52	62	73	83	96
Langkah-2		17					65	77	87	
(puluhan)										

- Gabungkan isi semua elemen:
  09, 11, 17, 24, 35, 41, 52, 62, 65, 73, 77, 83, 87, 96 (hasil sorting)

(b)

Gambar 10.5: Proses Radix Sort secara Ascending

Contoh program radix sort terdapat di link https://colab.research.google.com/drive/1aQqB-V8heD0ZEEsTqrFAMnJAACKLtK0m?usp=sharing.

Latihan:
Soal 1:
Diketahui sebuah array berikut: [170, 45, 75, 90, 802, 24, 2, 66]
Urutkan data tersebut menggunakan Radix Sort secara ascending. Tunjukkan langkah-langkah pembagian berdasarkan digit satuan, puluhan, dan ratusan, serta hasil setelah setiap tahap pengurutan.

Soal 2:
Diberikan array berikut: [53, 87, 34, 12, 91, 43, 28]
Urutkan data tersebut menggunakan Radix Sort secara descending. Lakukan perhitungan manual dan jelaskan langkah-langkah perbandingan, pembagian berdasarkan digit satuan, puluhan, dan seterusnya, serta hasil setelah setiap tahap pengurutan.

Soal 3
Buatkan code radix sort dengan data dari soal 1 dan 2.

10.6 Heap Sort

Heapsort merupakan algoritma sorting yang memanfaatkan struktur data heap untuk mengurutkan elemen dalam sebuah array. Heap adalah struktur data yang memungkinkan akses yang cepat terhadap elemen dengan nilai tertentu, khususnya elemen dengan nilai terkecil atau terbesar. Dalam proses sorting, elemen-elemen dalam array terlebih dahulu dimasukkan ke dalam heap satu per satu. Setelah seluruh elemen berada dalam heap, elemen yang berada di bagian puncak heap kemudian diambil secara bertahap. Karena elemen bagian atas heap merupakan elemen dengan nilai paling kecil atau paling besar (tergantung jenis heap yang digunakan), maka elemen-elemen yang dikeluarkan akan membentuk urutan yang teratur (Heinold, 2025).

Heapsort termasuk dalam algoritma pengurutan berbasis perbandingan (*comparison-based*) dan dapat dipandang sebagai bentuk pengembangan yang lebih efisien dari selection sort. Algoritma ini membagi data menjadi dua bagian, yaitu bagian yang telah terurut dan bagian yang belum terurut. Setiap iterasi, elemen terbesar dari bagian yang belum terurut akan dipilih dan dipindahkan ke bagian yang telah terurut. Peningkatan efisiensi dibandingkan dengan selection sort dicapai melalui penggunaan struktur data heap, sehingga pencarian elemen maksimum tidak perlu dilakukan secara linear.

Secara umum, heapsort memanfaatkan binary heap, khususnya max-heap, di mana setiap simpul induk memiliki nilai yang lebih besar atau sama dengan simpul anaknya. Proses pengurutan dimulai dengan membangun struktur max-heap dari data yang akan diurutkan. Selanjutnya, elemen terbesar dalam akar heap dipindahkan ke posisi akhir array, kemudian struktur heap diperbaiki kembali agar sifat max-heap tetap terpenuhi. Proses ini dilakukan secara berulang hingga seluruh elemen tersusun secara terurut, dengan elemen-elemen terbesar secara bertahap berpindah ke bagian akhir array.

Dari segi kinerja, heapsort memiliki kompleksitas waktu O(n log n) baik kondisi terbaik, rata-rata, maupun terburuk. Kompleksitas ini muncul karena setiap elemen dimasukkan ke dalam heap melalui operasi yang membutuhkan waktu O(log n), dan proses penghapusan elemen dari heap juga memerlukan waktu O(log n). Dengan demikian, keseluruhan proses pengurutan memerlukan waktu O(n log n). Keunggulan lain dari heapsort adalah sifatnya sebagai in-place sorting algorithm, sehingga hanya membutuhkan tambahan memori yang relatif kecil. Namun demikian, algoritma ini tidak bersifat stabil karena tidak menjamin urutan relatif elemen yang memiliki nilai yang sama tetap dipertahankan.

Meskipun implementasinya relatif lebih kompleks dibandingkan beberapa algoritma sorting sederhana, heapsort tetap menjadi metode pengurutan yang kuat dan andal. Konsistensi performa dengan kompleksitas waktu O(n log n) menjadikannya sangat cocok digunakan untuk mengurutkan dataset berukuran besar. Dengan kombinasi efisiensi waktu dan penggunaan memori yang optimal, heapsort banyak dimanfaatkan dalam berbagai aplikasi pengolahan dan pengurutan data.

Algoritma HeapSort(A, n):
1. Bentuk binary tree lengkap dari array A
   - Node dimasukkan dari kiri ke kanan
   - Anak kiri diisi terlebih dahulu sebelum anak kanan
2. Bangun struktur Max-Heap
   Untuk i ← n/2 − 1 sampai 0 lakukan
      Heapify(A, n, i)
3. Lakukan proses pengurutan
   Untuk i ← n − 1 sampai 1 lakukan
      Tukar A[0] dengan A[i] // root dipindah ke posisi akhir
      Heapify(A, i, 0) // perbaiki heap
4. Selesai

Algoritma Heapify(A, n, i):
1. largest ← i
2. left ← 2*i + 1
3. right ← 2*i + 2
4. Jika left < n dan A[left] > A[largest]
      largest ← left
5. Jika right < n dan A[right] > A[largest]
      largest ← right
6. Jika largest ≠ i
      Tukar A[i] dengan A[largest]
      Heapify(A, n, largest)

Cara kerja algoritma heap sort yang disertai dengan algoritmanya yaitu:

1. Masukkan data ke dalam struktur binary tree lengkap (*complete binary tree*).
   - Penyusunan node binary tree dilakukan secara berurutan dari kiri ke kanan.
   - Node anak kiri diisi terlebih dahulu sebelum node anak kanan.
2. Bentuk struktur max-heap, yaitu kondisi di mana nilai parent (*root*) lebih besar atau sama dengan nilai node anaknya.
   Setelah struktur heap terbentuk, kondisi tersebut menjadi inisialisasi heap yang akan digunakan dalam proses pengurutan.
3. Lakukan proses heap sort, yaitu dengan langkah-langkah berikut:
   - Elemen root (elemen terbesar) dihapus dari heap.
   - Posisi root kemudian digantikan oleh node paling kanan di level terbawah.
   - Struktur heap diperbaiki kembali agar sifat max-heap tetap terpenuhi.

4. Proses ini diulang hingga seluruh elemen heap telah dipindahkan ke bagian array yang terurut.

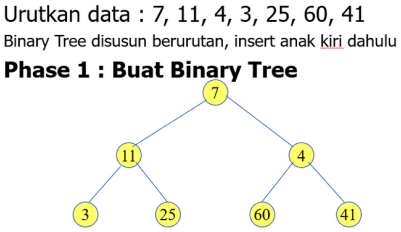
Kelebihan algoritma heap sort yaitu:

- Heap sort memiliki kinerja yang efisien karena algoritma ini memiliki kompleksitas waktu O(n log n) sehingga algoritma ini mampu menangani dataset berukuran besar secara efektif.
- Heap sort menggunakan memori secara efisien karena algoritma ini termasuk in-place sorting, sehingga algoritma ini hanya membutuhkan tambahan ruang memori yang konstan.
- Heap sort tidak memiliki risiko kasus terburuk $O(n^2)$ karena algoritma ini tetap mempertahankan kompleksitas waktu O(n log n) dalam berbagai kondisi data.

Kekurangan algoritma heap sort yaitu:

- Heap sort tidak bersifat stabil karena algoritma ini tidak mempertahankan urutan relatif elemen-elemen yang memiliki nilai sama.
- Heap sort memiliki implementasi yang relatif lebih kompleks dibandingkan dengan algoritma pengurutan yang lebih sederhana seperti bubble sort.
- Heap sort dapat mengalami penurunan kinerja cache karena algoritma ini mengakses memori secara tidak berurutan sehingga kinerja cache dalam arsitektur komputer modern dapat menjadi kurang optimal.

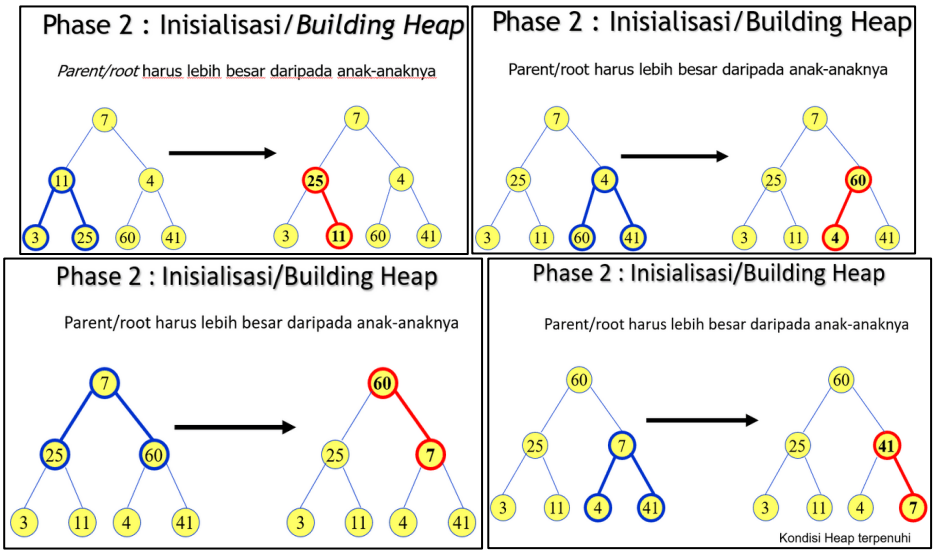
Gambar 10.6 menunjukkan tahap pertama dalam algoritma heap sort, yaitu proses membentuk binary tree



Gambar 10.6: Tahap Pertama Algoritma Heap Sort

lengkap (*complete binary tree*) dari data yang akan diurutkan. Data awal yang digunakan adalah 7, 11, 4, 3, 25, 60, dan 41. Kemudian elemen-elemen tersebut disusun ke dalam struktur pohon biner secara berurutan dari kiri ke kanan. Pengisian node dilakukan dengan menempatkan anak kiri terlebih dahulu sebelum anak kanan, sehingga struktur pohon yang terbentuk memenuhi sifat complete binary tree, yaitu setiap level pohon terisi secara berurutan dari kiri ke kanan. Tahap ini bertujuan untuk merepresentasikan data dalam bentuk struktur pohon sebelum dilakukan proses pembentukan heap, yang selanjutnya akan digunakan dalam proses pengurutan menggunakan algoritma heap sort.

Gambar tersebut menunjukkan tahap kedua dalam algoritma heap sort, yaitu proses inisialisasi atau pembentukan heap (*building heap*) dari struktur binary tree yang telah dibuat sebelumnya. **Setiap node diperiksa dan disesuaikan agar memenuhi sifat max-heap, yaitu kondisi di mana nilai parent atau root harus lebih besar daripada nilai node anak-anaknya**. **Proses ini dilakukan dengan membandingkan nilai parent dengan nilai anak kiri dan anak kanan, kemudian menukar posisi node apabila ditemukan nilai anak yang lebih besar daripada parent**. Proses penyesuaian tersebut dilakukan secara bertahap dari bagian bawah pohon menuju ke bagian atas hingga seluruh struktur pohon memenuhi aturan max-heap. Setelah semua node memenuhi kondisi tersebut, struktur pohon akan memiliki elemen terbesar di posisi root, yang menandakan bahwa proses building heap telah selesai dan heap siap digunakan dalam tahap pengurutan berikutnya dalam algoritma heap sort.



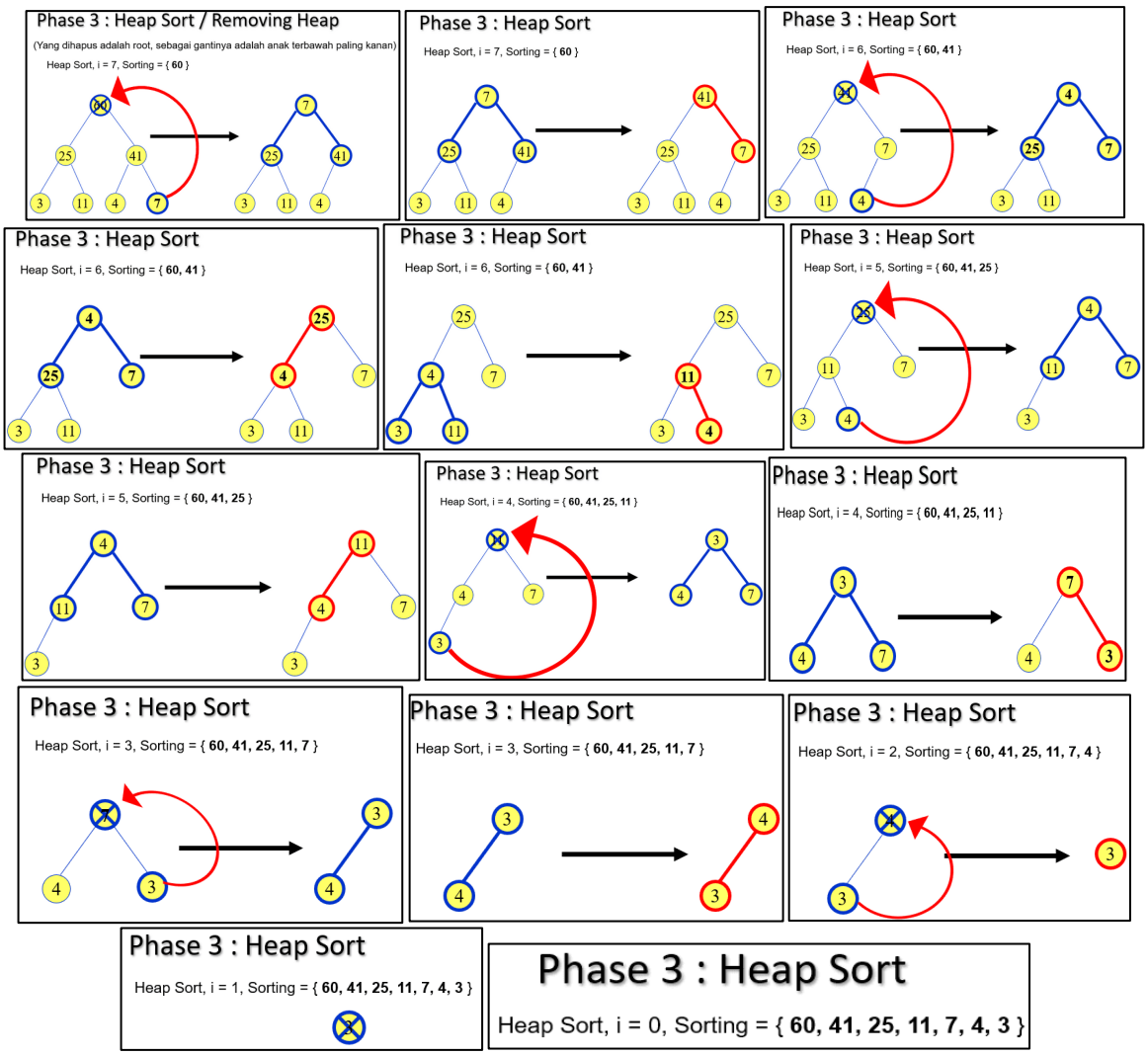
Gambar 10.7: Tahap Kedua Algoritma Heap Sort

Gambar 10.8 menunjukkan **tahap ketiga dalam algoritma heap sort**, yaitu proses **pengurutan dengan cara menghapus elemen di root heap secara berulang**. Setelah struktur **max-heap** terbentuk, elemen terbesar yang berada di **root** dipindahkan ke bagian akhir array sebagai bagian dari elemen yang telah terurut. Selanjutnya, posisi root digantikan oleh **node yang berada paling kanan di level terbawah** dari heap. Setelah penggantian tersebut, struktur heap diperbaiki kembali agar sifat **max-heap** tetap terpenuhi, yaitu nilai parent harus lebih besar daripada nilai anak-anaknya. Proses ini dilakukan secara berulang, di mana setiap kali elemen terbesar dipindahkan ke bagian akhir array dan ukuran heap dikurangi. Melalui langkah-langkah tersebut, elemen-elemen terbesar secara bertahap terkumpul di bagian akhir array hingga seluruh data tersusun secara terurut. Hasil akhir dari proses heap sort adalah urutan **{60, 41, 25, 11, 7, 4, 3}**, yang menunjukkan bahwa seluruh elemen telah berhasil diurutkan.

Latihan:
Soal 1:
Diketahui sebuah array berikut: 12, 7, 25, 15, 28, 33, 5
Urutkan data tersebut menggunakan algoritma heap sort secara ascending.

Soal 2:
Diberikan array berikut: 18, 42, 9, 27, 31, 6, 14, 50
Urutkan data tersebut menggunakan algoritma heap sort secara descending.

Soal 3
Buatkan code heap sort dengan data dari soal 1 dan 2.



Gambar 10.8: Tahap Ketiga Algoritma Heap Sort

Contoh program algoritma heap sort dapat dilihat di link https://colab.research.google.com/drive/1PV2xa4bEUQGjvgCs14fYejTfXmC2tmrP?usp=sharing.

10.7 Quick Sort

Quicksort merupakan salah satu algoritma pengurutan yang sangat efisien dan banyak digunakan dalam berbagai bahasa pemrograman serta pustaka pengurutan. Algoritma ini pertama kali diperkenalkan oleh C. A. R. Hoare dan termasuk dalam kategori algoritma *divide and conquer*, yaitu pendekatan yang memecah suatu permasalahan besar menjadi beberapa subpermasalahan yang lebih kecil dan lebih mudah diselesaikan. Dalam penerapannya, quicksort bekerja dengan membagi sebuah array yang belum terurut menjadi dua subarray yang lebih kecil, kemudian masing-masing subarray tersebut diurutkan secara rekursif sebelum akhirnya digabungkan kembali sehingga menghasilkan urutan data yang teratur (Heinold, 2025).

Prinsip utama dari algoritma **quicksort** adalah proses **partitioning**, yaitu pemilihan sebuah elemen yang disebut pivot dari dalam array. Elemen pivot ini digunakan sebagai titik pembanding untuk mengelompokkan elemen-elemen lain. Seluruh elemen yang memiliki nilai lebih kecil dari pivot akan ditempatkan di sebelah kiri pivot, sedangkan elemen yang memiliki nilai lebih besar akan ditempatkan di sebelah kanan pivot. Setelah proses partisi selesai, dua bagian array tersebut akan diurutkan kembali menggunakan prosedur yang sama secara rekursif hingga seluruh elemen berada di posisi yang tepat.

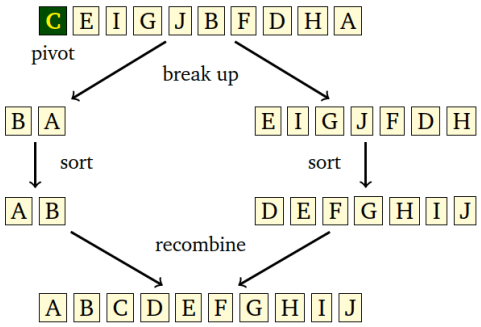
Secara teoritis, quicksort memiliki kompleksitas waktu rata-rata sebesar O(n log n) untuk mengurutkan array yang terdiri dari n elemen. Namun, saat kondisi terburuk, misalnya ketika pemilihan pivot tidak optimal atau data sudah berada dalam pola tertentu, kompleksitas waktunya dapat meningkat menjadi $O(n^2)$. Meskipun demikian, kondisi terburuk tersebut relatif jarang terjadi dalam praktik. Oleh karena itu, quicksort sering kali menunjukkan kinerja lebih cepat dibandingkan beberapa algoritma pengurutan lain yang memiliki kompleksitas rata-rata sama, seperti merge sort. Selain itu, quicksort biasanya diimplementasikan secara in-place, sehingga tidak memerlukan ruang memori tambahan yang besar, meskipun algoritma ini bersifat tidak stabil. Quicksort menjadi salah satu algoritma pengurutan yang paling populer dalam praktik komputasi, terutama ketika digunakan untuk mengolah kumpulan data berukuran besar, karena efisiensi dan performanya yang tinggi. Pendekatan pembagian masalah secara rekursif serta mekanisme partisi yang efektif menjadikan quicksort mampu menghasilkan proses pengurutan yang cepat dan efisien dalam berbagai aplikasi pengolahan data.

Cara kerja algoritma quick sort dapat dijelaskan sebagai berikut:

1. Tentukan sebuah elemen dari array sebagai pivot.
2. Susun kembali elemen-elemen dalam array sehingga:
   - Elemen yang bernilai lebih kecil dari pivot ditempatkan di sebelah kiri pivot.
   - Elemen yang bernilai lebih besar dari pivot ditempatkan di sebelah kanan pivot.
   - Elemen dengan nilai yang sama dapat ditempatkan di salah satu sisi.
3. Setelah proses tersebut selesai, pivot berada di posisi akhirnya dalam array. Proses ini disebut operasi partisi (*partition*).
4. Lakukan proses pengurutan secara rekursif terhadap dua subarray yang terbentuk, yaitu:
   - Subarray yang berisi elemen dengan nilai lebih kecil dari pivot.
   - Subarray yang berisi elemen dengan nilai lebih besar dari pivot.
5. Proses rekursi berhenti ketika subarray memiliki nol atau satu elemen, karena dalam kondisi tersebut data sudah dalam keadaan terurut.
6. Setiap iterasi, satu elemen pivot akan berada di posisi akhirnya sehingga jumlah elemen yang masih perlu diurutkan akan semakin berkurang.

```
Algoritma QUICK_SORT(A, awal, akhir)
1. Jika awal < akhir maka
2.    pivot_index ← PARTITION(A, awal, akhir)
3.    QUICK_SORT(A, awal, pivot_index - 1)
4.    QUICK_SORT(A, pivot_index + 1, akhir)
5. Selesai
```

```
Algoritma PARTITION(A, awal, akhir)
1. pivot ← A[awal]
2. left ← awal + 1
3. right ← akhir

4. selama left ≤ right lakukan
5.    selama left ≤ right dan A[left] ≤ pivot lakukan
6.       left ← left + 1
7.    selama left ≤ right dan A[right] ≥ pivot lakukan
8.       right ← right - 1
9.
10.   jika left < right maka
11.      tukar A[left] dengan A[right]

12. tukar A[awal] dengan A[right]

13. kembalikan right sebagai posisi pivot
```



Gambar 10.9: Contoh Quick Sort (Heinold, 2025)

Gambar 10.9 menunjukkan contoh proses kerja algoritma quick sort dalam melakukan pengurutan data. Tahap awal, dipilih sebuah elemen sebagai pivot, yaitu elemen yang dijadikan acuan untuk membagi data. Dalam ilustrasi tersebut, huruf C dipilih sebagai pivot dari sekumpulan elemen yang belum terurut. Selanjutnya dilakukan proses break up atau partisi, yaitu membagi elemen-elemen lainnya menjadi dua kelompok berdasarkan nilai pivot. Elemen yang memiliki nilai lebih kecil dari pivot ditempatkan di bagian kiri, sedangkan elemen yang memiliki nilai lebih besar ditempatkan di bagian kanan. Elemen B dan A berada di sisi kiri karena nilainya lebih kecil dari C, sementara elemen E, I, G, J, B, F, D, H berada di sisi kanan karena nilainya lebih besar.

Setelah proses partisi selesai, kedua kelompok data tersebut kemudian diurutkan kembali secara rekursif menggunakan prosedur yang sama. Subarray di sisi kiri, yaitu B dan A, diurutkan sehingga menjadi A, B. Sementara itu, subarray di sisi kanan juga mengalami proses pengurutan hingga menghasilkan urutan D, E, F, G, H, I, J. Setelah kedua bagian tersebut selesai diurutkan, tahap berikutnya adalah recombine, yaitu menggabungkan kembali hasil pengurutan dengan pivot yang telah berada di posisi yang benar. Hasil akhirnya adalah urutan data yang telah tersusun secara teratur, yaitu A, B, C, D, E, F, G, H, I, J. Dengan demikian, gambar tersebut menggambarkan prinsip dasar algoritma Quick Sort, yaitu memilih pivot, melakukan partisi, mengurutkan subarray secara rekursif, dan kemudian menggabungkan hasilnya hingga seluruh elemen berada dalam keadaan terurut.

Kelebihan algoritma quick sort yaitu:

- Efisiensi. Quick sort umumnya memiliki kinerja yang baik dengan kompleksitas waktu rata-rata sebesar O(n log n), sehingga lebih unggul dibandingkan algoritma sederhana seperti bubble sort atau insertion sort.
- Pengurutan In-Place. Proses pengurutan dilakukan langsung dalam masukan tanpa memerlukan ruang penyimpanan tambahan yang signifikan.
- Fleksibilitas. Quick sort dapat diterapkan dalam berbagai tipe data dan dapat dimodifikasi sesuai dengan kebutuhan atau kondisi tertentu.

Kelemahan algoritma quick sort yaitu:

- Kompleksitas kasus terburuk. Kompleksitas waktu quick sort dapat menurun menjadi $o(n^2)$ apabila pemilihan pivot tidak dilakukan dengan baik, sehingga dapat mengurangi efisiensi algoritma secara signifikan.
- Tidak stabil. quick sort termasuk algoritma pengurutan yang tidak stabil, sehingga urutan relatif elemen-elemen yang memiliki nilai kunci yang sama dapat berubah setelah proses pengurutan.

Program quick sort dapat dilihat di link https://colab.research.google.com/drive/1DcZ9QmJ7TORpexg9TZY0IvGqp0MDevs9?usp=sharing.

Latihan:
Soal 1:
Diketahui sebuah array berikut: 35, 12, 43, 8, 51, 26, 17
Urutkan data tersebut menggunakan algoritma quick sort secara ascending.

Soal 2:
Diberikan array berikut: 64, 25, 12, 22, 11, 90, 34
Urutkan data tersebut menggunakan algoritma quick sort secara descending.

Soal 3
Buatkan code quick sort dengan data dari soal 1 dan 2.

10.8 Shell Sort

Shell sort merupakan algoritma pengurutan yang diperkenalkan oleh Donald Shell tahun 1959 sebagai pengembangan dari algoritma insertion sort. Algoritma insertion sort diketahui bahwa metode tersebut bekerja cukup efektif apabila data yang diurutkan sudah berada dalam kondisi hampir terurut (*almost sorted*). Namun, insertion sort memiliki kelemahan karena setiap proses pemindahan elemen hanya dilakukan satu posisi dalam satu langkah, sehingga menjadi kurang efisien ketika elemen yang seharusnya berada di awal justru terletak di bagian akhir array. Untuk mengatasi keterbatasan tersebut, shell sort memperkenalkan konsep perbandingan elemen yang dipisahkan oleh jarak tertentu (gap). Dengan cara ini, suatu elemen dapat bergerak lebih jauh menuju posisi yang seharusnya dalam satu langkah, sehingga proses pengurutan dapat berlangsung lebih cepat dibandingkan dengan insertion sort biasa (Heinold, 2025).

Algoritma shell sort adalah proses pengurutan dilakukan melalui beberapa tahap (*pass*). Setiap tahap, elemen-elemen dibandingkan berdasarkan jarak tertentu yang disebut gap atau increment. Nilai gap tersebut secara bertahap diperkecil hingga akhirnya bernilai satu. Ketika gap telah mencapai nilai satu, proses pengurutan yang terjadi sama dengan insertion sort. Sebagian besar elemen biasanya sudah berada dalam kondisi hampir terurut akibat proses pengurutan sebelumnya, sehingga insertion sort tahap akhir dapat bekerja dengan lebih efisien. Sebagai ilustrasi, apabila elemen terkecil berada di ujung array, algoritma seperti bubble sort atau insertion sort membutuhkan banyak perbandingan dan pertukaran untuk memindahkannya ke posisi yang benar. Sebaliknya, shell sort dapat memindahkan elemen tersebut dengan langkah yang lebih besar di tahap awal sehingga jumlah operasi yang diperlukan menjadi lebih sedikit.

Secara konseptual, shell sort merupakan algoritma perbandingan (*comparison-based*) yang bekerja secara in-place, yaitu proses pengurutan dilakukan langsung dalam array tanpa memerlukan memori tambahan yang signifikan. Algoritma ini membentuk urutan yang disebut h-sorted, yaitu kondisi ketika elemen-elemen yang berjarak h posisi dari suatu titik telah berada dalam keadaan terurut. Dengan cara ini, array secara bertahap menjadi semakin terstruktur hingga akhirnya sepenuhnya terurut. Kinerja shell sort sangat dipengaruhi oleh pemilihan urutan gap (gap sequence) yang digunakan. Beberapa urutan gap yang umum digunakan antara lain urutan asli yang diusulkan oleh Shell, yaitu N/2,N/4,…,1 serta variasi lain seperti Hibbard increments dan Sedgewick increments yang dirancang untuk meningkatkan efisiensi algoritma.

Dari sisi kompleksitas waktu, performa shell sort dapat bervariasi tergantung dalam urutan gap yang dipilih. Dalam kondisi terburuk, kompleksitas waktunya dapat mencapai O(n^2). Namun dengan pemilihan gap sequence yang lebih optimal, kompleksitas tersebut dapat ditingkatkan menjadi sekitar O(n^{2/3}) atau bahkan mendekati O(n \ log^2 n). Dengan karakteristik tersebut, shell sort sering dianggap sebagai metode pengurutan yang lebih efisien dibandingkan insertion sort dalam banyak kasus, terutama ketika data awal tidak sepenuhnya acak atau telah mengalami sebagian proses pengurutan sebelumnya.

Cara kerja Shell Sort yaitu:

Langkah 1:

- Susun elemen-elemen array dalam bentuk tabel.
- Setiap kolom tabel diurutkan menggunakan insertion sort.
- Kolom tersebut sebenarnya mewakili elemen yang memiliki jarak (gap) tertentu dalam array.

Langkah 2:

- Ulangi langkah pertama dengan:
  - ♥ Jumlah kolom lebih sedikit.
  - ♥ Panjang kolom lebih besar.
- Hal ini berarti gap semakin kecil.

Langkah 3:

- Proses ini terus diulang hingga hanya tersisa satu kolom.
- Tahap ini seluruh elemen berada dalam satu urutan dan dilakukan pengurutan terakhir.

Algoritma ShellSort:

```
Input  : Array A berisi n elemen
Output : Array A terurut

1. Tentukan nilai gap awal
   gap ← n / 2

2. Selama gap > 0 lakukan

   // Langkah 1 : membentuk kolom (visualisasi tabel)
   Untuk i dari gap sampai n-1 lakukan

      temp ← A[i]
      j ← i

      // Mengurutkan elemen dalam satu kolom
      Selama j ≥ gap dan A[j-gap] > temp lakukan
         A[j] ← A[j-gap]
         j ← j - gap
      Selesai

      A[j] ← temp

   Selesai

   // Langkah 2 : memperkecil jumlah kolom
   gap ← gap / 2

3. Selesai
```

Kelebihan algoritma shell sort yaitu:

- Efisiensi. Shell sort memiliki kinerja yang lebih cepat dibandingkan bubble sort, insertion sort, dan selection sort ketika digunakan dalam array berukuran menengah.
- In-place. Algoritma ini hanya membutuhkan tambahan memori yang sangat kecil dan bersifat konstan karena proses pengurutan dilakukan langsung dalam array yang sama.
- Adaptif. Kinerja algoritma meningkat secara signifikan ketika diterapkan dalam data yang sudah hampir terurut.

Kekurangan algoritma shell sort yaitu:

- Kompleksitas penentuan increment. Pemilihan urutan nilai increment atau gap sangat memengaruhi performa algoritma, sementara menentukan urutan gap yang optimal bukanlah hal yang sederhana.
- Tidak stabil. Shell sort tidak menjamin mempertahankan urutan relatif dari elemen-elemen yang memiliki nilai yang sama.
- Kurang optimal untuk data sangat besar. Untuk dataset berukuran sangat besar, algoritma lain seperti quick sort atau merge sort umumnya memberikan performa yang lebih baik.

Gambar 10.10 menunjukkan proses pengurutan data dengan algoritma Shell Sort dalam deretan huruf A, Q, D, O, M, C, R, B, J, I, H, P, L, K, F, G, E, N. Secara umum, Shell Sort bekerja dengan membandingkan elemen-elemen yang dipisahkan oleh jarak tertentu (gap), kemudian mengurutkannya menggunakan prinsip insertion sort, lalu secara bertahap memperkecil nilai gap hingga akhirnya menjadi 1. Jumlah data adalah 18 elemen, sehingga tahap pertama menggunakan gap 4 dan tahap berikutnya menggunakan gap 2. Hal ini tampak dari pola elemen yang diberi warna hijau. Bagian atas gambar, elemen yang dipilih setiap empat posisi sekali membentuk kelompok-kelompok yang kemudian diurutkan. Misalnya, kelompok pertama adalah A, M, J, L, E yang berasal dari posisi 1, 5, 9, 13, dan 17. Kelompok ini kemudian diurutkan menjadi A, E, J, L, M, sehingga

huruf E berpindah dari posisi akhir kelompok ke posisi kedua, sedangkan M bergeser ke posisi terakhir kelompok. Perubahan ini tercermin di baris kedua, ketika susunan data berubah dari A Q D O M C R B J I H P L K F G E N menjadi A Q D O E C R B J I H P L K F G M N.

Gambar 10.10: Contoh Algoritma Shell Sort (Heinold, 2025)

Proses yang sama dilakukan dalam kelompok-kelompok lain dengan gap 4. Kelompok kedua adalah Q, C, I, K, N yang berasal dari posisi 2, 6, 10, 14, dan 18, lalu diurutkan menjadi C, I, K, N, Q. Akibatnya, C berpindah ke posisi paling awal kelompok, sedangkan Q berpindah ke posisi paling akhir. Kelompok ketiga adalah D, R, H, F, yang setelah diurutkan menjadi D, F, H, R, sehingga F berpindah mendekati awal kelompok dan R bergeser ke belakang. Kelompok keempat adalah O, B, P, G, yang setelah diurutkan menjadi B, G, O, P, sehingga B berpindah ke depan dan P tetap berada di bagian akhir kelompok. Dengan demikian, tahap gap 4 ini, elemen-elemen yang semula jauh dari posisi seharusnya dapat dipindahkan lebih cepat melalui lompatan yang lebih besar dibandingkan insertion sort biasa. Hasil tahap ini adalah data yang sudah lebih mendekati urutan akhir, meskipun belum sepenuhnya terurut.

Setelah tahap gap 4 selesai, algoritma melanjutkan ke tahap berikutnya dengan gap 2, yaitu membandingkan elemen setiap dua posisi. Bagian bawah gambar, kelompok pertama yang diwarnai hijau adalah elemen di posisi ganjil, yaitu A, D, E, F, J, H, L, R, M, yang kemudian diurutkan menjadi A, D, E, F, H, J, L, M, R. Tahap ini terlihat bahwa H berpindah ke posisi sebelum J, sedangkan M berpindah ke posisi sebelum R. Kelompok kedua adalah elemen di posisi genap, yaitu C, B, I, G, K, O, N, P, Q, yang setelah diurutkan menjadi B, C, G, I, K, O, N, P, Q. Akibatnya, B berpindah ke awal kelompok, C bergeser ke belakangnya, dan G berpindah ke depan sebelum I. Tahap gap 2 ini membuat data semakin mendekati kondisi terurut penuh, karena elemen-elemen yang sebelumnya masih berada di sekitar posisi yang salah dipindahkan kembali dengan jarak yang lebih kecil dan lebih halus.

Tahap terakhir, yaitu gap 1, tetapi tahap itu secara implisit diperlukan agar seluruh data benar-benar terurut sempurna. Saat gap menjadi 1, proses yang berlangsung sama dengan insertion sort biasa, namun karena dua tahap sebelumnya telah membuat data hampir terurut, jumlah pergeseran yang dibutuhkan menjadi jauh lebih sedikit. Oleh sebab itu, analisis gap sangat penting yaitu gap 4 digunakan untuk memindahkan elemen-elemen yang jauh dari posisi yang benar dengan lompatan besar, sedangkan gap 2 digunakan untuk memperhalus susunan data agar semakin mendekati urutan final. Dengan mekanisme inilah Shell Sort mampu bekerja lebih efisien daripada insertion sort biasa, khususnya ketika data awal masih cukup acak.

Program shell sort dapat dilihat di link https://colab.research.google.com/drive/1dL35Rs9WI4q6-7j4EH56_d-zFPnmpXcc?usp=sharing.

Latihan:
Soal 1:
Diberikan sebuah array bilangan berikut:
38, 27, 43, 3, 9, 82, 10, 14
Lakukan proses pengurutan menggunakan algoritma Shell Sort dengan aturan gap = n/2, kemudian dibagi 2 setiap iterasi hingga gap = 1.
Jumlah data n = 8, sehingga urutan gap yang digunakan adalah:
gap = 4 → gap = 2 → gap = 1

Tugas:
- Tunjukkan proses pengelompokan elemen berdasarkan gap = 4 dan lakukan pengurutan untuk setiap kelompok menggunakan insertion sort.
- Tuliskan hasil array setelah proses gap = 4 selesai.
- Lanjutkan dengan gap = 2, lalu tuliskan kembali proses pengurutan untuk setiap kelompok.
- Tuliskan hasil array setelah proses gap = 2 selesai.
- Lakukan tahap terakhir dengan gap = 1, kemudian tuliskan hasil pengurutan akhir.

Soal 2
Buatkan code quick sort dengan data dari soal 1.

10.9 Counting Sort

Counting sort merupakan salah satu algoritma pengurutan yang bersifat non-komparatif, yaitu tidak melakukan perbandingan langsung antar elemen sebagaimana algoritma pengurutan berbasis perbandingan seperti quick sort atau merge sort. **Algoritma ini bekerja dengan cara menghitung jumlah kemunculan setiap nilai unik dalam data yang akan diurutkan**. Proses dimulai dengan membuat sebuah array penghitung (*count array*) yang berisi jumlah kemunculan masing-masing elemen dalam data. Selanjutnya, array tersebut digunakan untuk membentuk array kumulatif yang menunjukkan posisi setiap elemen dalam urutan hasil pengurutan. Berdasarkan informasi tersebut, setiap elemen dari data awal kemudian ditempatkan di posisi yang sesuai dalam array keluaran sehingga menghasilkan data yang telah terurut (Heinold, 2025).

Berbeda dengan metode pengurutan tradisional yang bergantung proses perbandingan antar elemen, counting sort mengorganisasi objek berdasarkan nilai kunci (key) yang berbeda, yang prinsip kerjanya mirip dengan mekanisme hashing. Setelah frekuensi setiap nilai diketahui, algoritma menggunakan operasi aritmetika untuk menentukan indeks atau posisi setiap elemen dalam urutan hasil pengurutan. Oleh karena itu, counting sort termasuk algoritma pengurutan yang bersifat khusus dan tidak dirancang untuk digunakan secara umum untuk semua jenis data.

Counting sort bekerja sangat efisien ketika rentang nilai data yang akan diurutkan tidak jauh lebih besar dibandingkan jumlah elemen yang diproses. Dalam kondisi tersebut, algoritma ini dapat memberikan performa yang sangat baik karena kompleksitas waktunya adalah O(n + k), di mana n menyatakan jumlah elemen data dan k menunjukkan rentang nilai yang mungkin muncul dalam data. Kompleksitas waktu ini tidak bergantung dalam kondisi awal data, sehingga baik data sudah terurut maupun tidak, waktu eksekusi algoritma tetap berada di orde yang sama. Selain itu, counting sort juga dapat digunakan untuk mengurutkan data yang memiliki nilai negatif, selama dilakukan penyesuaian dalam proses pemetaan indeksnya.

Dibandingkan dengan algoritma pengurutan berbasis perbandingan, counting sort memiliki keunggulan karena tidak melakukan proses perbandingan antar elemen sehingga dapat bekerja lebih cepat dalam kondisi tertentu. Namun demikian, efisiensinya akan menurun apabila rentang nilai data sangat besar. Hal ini disebabkan oleh kebutuhan untuk membuat array penghitung dengan ukuran yang sebanding dengan rentang nilai tersebut, yang dapat mengakibatkan penggunaan memori menjadi sangat besar dan kurang praktis. Oleh karena itu,

counting sort paling efektif digunakan ketika data yang diurutkan berupa bilangan bulat dengan rentang nilai yang relatif kecil.

Proses eksekusi counting sort, (Sundaramoorthy & Karunanidhi, 2025), yaitu:

- Proses dimulai dengan **menghitung jumlah kemunculan setiap elemen unik** dalam data yang diberikan.
- Jumlah kemunculan tersebut kemudian **disimpan dalam sebuah array khusus yang disebut count array**.
- Setelah itu, dibuat **array kumulatif** dengan cara menambahkan nilai setiap elemen dalam count array dengan nilai elemen sebelumnya.
- Selanjutnya, dibuat sebuah **array keluaran (output array)** yang awalnya kosong dan akan digunakan untuk menyimpan elemen-elemen yang telah terurut.
- Array data awal kemudian **ditelusuri kembali**, dengan memanfaatkan informasi dalam count array dan array kumulatif untuk menentukan **posisi yang tepat bagi setiap elemen** output array.
- Setiap elemen ditempatkan di **indeks yang sesuai dalam output array**, kemudian nilai count array untuk elemen tersebut **dikurangi satu** untuk menangani kemungkinan adanya elemen yang sama (duplikasi).

Algoritma CountingSort:
Input : Array A berisi n elemen
Output : Array B berisi elemen yang telah terurut

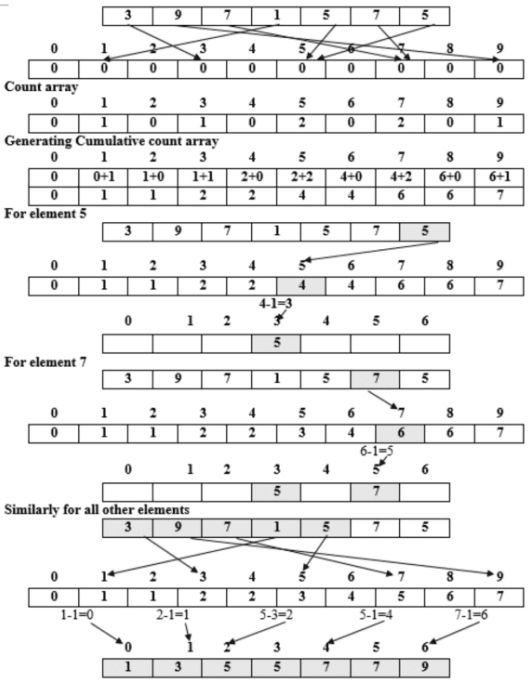
```
1. Tentukan nilai maksimum dari array A
   max ← nilai terbesar dalam A

2. Buat array penghitung C dengan ukuran max + 1
   Inisialisasi semua elemen C dengan nilai 0

3. Hitung jumlah kemunculan setiap elemen
   untuk i = 0 sampai n-1 lakukan
      C[A[i]] ← C[A[i]] + 1

4. Buat array kumulatif
   untuk i = 1 sampai max lakukan
      C[i] ← C[i] + C[i-1]

5. Buat array output B dengan ukuran n

6. Tentukan posisi setiap elemen dalam array output
   untuk i = n-1 sampai 0 lakukan
      B[C[A[i]] - 1] ← A[i]
      C[A[i]] ← C[A[i]] - 1

7. Kembalikan array B sebagai hasil pengurutan
```

- Proses ini dilakukan hingga **seluruh elemen dalam data telah diproses**.
- Setelah semua langkah selesai, **output array akan berisi elemen-elemen yang telah terurut**.
- Alur kerja algoritma counting sort ini juga dapat dijelaskan melalui **diagram alir (flowchart)** yang menggambarkan tahapan-tahapan operasinya.

Kelebihan algoritma counting sort yaitu:

- Kompleksitas waktu yang efisien. Counting sort memiliki kompleksitas waktu O(n + k), di mana n adalah jumlah elemen yang akan diurutkan dan k adalah rentang nilai data. Algoritma ini sangat efektif ketika nilai k tidak jauh lebih besar dibandingkan jumlah elemen n.
- Stabilitas algoritma. Counting sort termasuk algoritma stabil, sehingga urutan relatif elemen yang memiliki nilai yang sama tetap dipertahankan setelah proses pengurutan. Hal ini penting terutama ketika data memiliki nilai yang sama tetapi memiliki informasi tambahan yang perlu dipertahankan urutannya.
- Mudah dipahami dan diimplementasikan. Mekanisme counting sort relatif sederhana karena hanya melibatkan proses penghitungan frekuensi dan manipulasi array, sehingga mudah dipelajari dan cocok digunakan dalam pembelajaran algoritma maupun implementasi yang cepat.

Kelemahan algoritma counting sort yaitu:



Gambar 10.11: Contoh Operasi Algoritma Counting Sort (Sundaramoorthy & Karunanidhi, 2025)

- Rentang nilai yang besar. Counting sort membutuhkan ruang memori tambahan sebesar O(k) karena memerlukan array penghitung dengan ukuran sesuai rentang nilai data. Jika rentang nilai k jauh lebih besar daripada jumlah data n, maka penggunaan memori dan efisiensi waktu menjadi kurang optimal.
- Keterbatasan fleksibilitas. Algoritma ini memerlukan informasi mengenai rentang nilai data terlebih dahulu, yang dalam beberapa kasus tidak selalu diketahui atau sulit ditentukan, terutama dataset dengan rentang nilai yang sangat besar atau tidak terbatas.
- Tidak cocok untuk data non-integer. Counting sort secara langsung hanya dapat digunakan untuk data yang memiliki kunci berupa bilangan bulat non-negatif. Pengurutan tipe data lain seperti bilangan pecahan atau string, diperlukan penyesuaian tambahan terhadap algoritma.

Gambar 10.11 memperlihatkan tahapan operasi algoritma Counting Sort dalam mengurutkan sekumpulan data secara sistematis tanpa melakukan perbandingan antar elemen. Data awal yang digunakan dalam ilustrasi adalah 3, 9, 7, 1, 5, 7, 5. Algoritma ini bekerja dengan menghitung frekuensi kemunculan setiap nilai, kemudian menggunakan informasi tersebut untuk menentukan posisi akhir elemen dalam array hasil. Bagian awal gambar ditunjukkan array input yang berisi tujuh elemen tersebut. Karena nilai maksimum dalam data adalah 9, maka dibuat sebuah count array dengan indeks 0 sampai 9 yang akan digunakan untuk menyimpan jumlah kemunculan setiap nilai.

Tahap pertama adalah proses pembentukan count array dengan cara menghitung frekuensi setiap elemen array input. Setiap kali suatu nilai ditemukan, nilai indeks yang bersesuaian dalam count array akan ditambah satu. Setelah seluruh elemen dihitung, diperoleh distribusi frekuensi yaitu nilai 1 muncul satu kali, nilai 3 muncul satu kali, nilai 5 muncul dua kali, nilai 7 muncul dua kali, dan nilai 9 muncul satu kali, sedangkan indeks lainnya bernilai nol. Dengan demikian, count array yang dihasilkan menunjukkan jumlah kemunculan masing-masing nilai dalam data.

Proses pembentukan array kumulatif (*cumulative count array*). Setiap elemen count array dijumlahkan dengan nilai indeks sebelumnya sehingga menghasilkan jumlah kumulatif. Proses ini bertujuan untuk menentukan posisi akhir setiap nilai dalam array hasil pengurutan. Sebagai contoh, nilai kumulatif untuk indeks tertentu menunjukkan berapa banyak elemen yang memiliki nilai kurang dari atau sama dengan indeks tersebut.

Dengan demikian, array kumulatif memberikan informasi langsung mengenai posisi terakhir suatu elemen dalam array hasil.

Setelah array kumulatif terbentuk, proses selanjutnya adalah menempatkan setiap elemen dari array input ke dalam array output berdasarkan informasi yang terdapat dalam array kumulatif. Proses ini dimulai dari elemen tertentu, misalnya elemen 5. Nilai kumulatif untuk elemen tersebut menunjukkan posisi yang harus ditempati dalam array output. Setelah elemen ditempatkan di posisi yang sesuai, nilai count array untuk elemen tersebut dikurangi satu untuk mengakomodasi kemungkinan adanya elemen yang sama (duplikasi). Langkah yang sama kemudian diterapkan dalam elemen berikutnya, misalnya 7, dengan memanfaatkan nilai kumulatif yang tersisa untuk menentukan posisi yang tepat dalam array hasil.

Proses penempatan elemen ini dilakukan secara berulang untuk seluruh elemen dalam array input. Setiap elemen dipetakan ke posisi yang tepat dalam array output berdasarkan nilai kumulatif yang tersedia, sementara nilai count array terus dikurangi untuk menjaga konsistensi penempatan elemen yang memiliki nilai sama. Bagian akhir gambar ditunjukkan bahwa setelah semua elemen diproses, array output berisi data yang telah terurut, yaitu 1, 3, 5, 5, 7, 7, 9. Hasil ini menunjukkan bahwa counting sort berhasil menyusun data secara terurut dengan memanfaatkan frekuensi kemunculan dan posisi kumulatif elemen tanpa melakukan perbandingan antar elemen. Pendekatan ini menjadikan counting sort sangat efisien untuk data dengan rentang nilai yang relatif kecil dibandingkan jumlah elemennya.

Program counting sort dapat dilihat di https://colab.research.google.com/drive/1T3YoIL-MT7RzzRO6zy-JwVIO7do1OkiJ?usp=sharing.

Latihan:
Soal 1:
- Diberikan sebuah array bilangan berikut: 4, 2, 2, 8, 3, 3, 1
- Lakukan proses pengurutan menggunakan algoritma Counting Sort secara manual.

Soal 2
Buatkan code counting sort dengan data dari soal 1.

10.10 Bucket Sort

Bucket sort merupakan salah satu algoritma pengurutan non-komparatif yang bekerja dengan cara membagi elemen-elemen dalam suatu array ke dalam beberapa kelompok yang disebut bucket. Setiap bucket berisi nilai-nilai yang berada dalam rentang tertentu, sehingga semua nilai dalam bucket ke-i akan lebih kecil dibandingkan nilai dalam bucket ke-i+1. Setelah proses pembagian selesai, setiap bucket kemudian diurutkan secara terpisah menggunakan algoritma pengurutan berbasis perbandingan, seperti insertion sort atau merge sort. Setelah seluruh bucket selesai diurutkan, isi setiap bucket digabungkan kembali sesuai urutan nomor bucket sehingga menghasilkan array akhir yang telah terurut (Sundaramoorthy & Karunanidhi, 2025).

Metode bucket sort sangat efektif ketika nilai-nilai data terdistribusi secara merata dalam suatu rentang tertentu, karena kondisi tersebut memungkinkan pembagian data ke dalam bucket menjadi relatif seimbang. Pemilihan algoritma pengurutan yang digunakan untuk mengurutkan elemen di dalam setiap bucket biasanya disesuaikan dengan jumlah data dalam bucket tersebut. Apabila jumlah data dalam bucket relatif sedikit, maka insertion sort sering dipilih karena kesederhanaannya. Sebaliknya, jika jumlah data dalam bucket cukup banyak, maka algoritma yang lebih efisien seperti merge sort dapat digunakan untuk meningkatkan kinerja proses pengurutan.

Dalam beberapa implementasi, algoritma bucket sort mengasumsikan bahwa nilai-nilai data berada dalam rentang [0,1]. Jika data tidak berada dalam rentang tersebut, maka nilai-nilai array terlebih dahulu dinormalisasi agar berada dalam rentang tersebut sebelum dilakukan proses pembagian ke dalam bucket. Dalam proses pengurutan setiap bucket biasanya digunakan algoritma pengurutan yang bersifat stabil, sehingga urutan relatif

elemen yang memiliki nilai sama tetap terjaga sebelum seluruh bucket digabungkan kembali menjadi satu array yang telah terurut.

Karena bucket sort terlebih dahulu mendistribusikan seluruh elemen ke dalam bucket sebelum melakukan pengurutan untuk masing-masing bucket, algoritma ini memerlukan ruang penyimpanan tambahan yang sebanding dengan jumlah elemen data. Oleh karena itu, kompleksitas ruang dari bucket sort adalah O(n). Dalam kondisi ideal, ketika data terdistribusi secara merata, bucket sort dapat mencapai kompleksitas waktu O(n) sehingga sangat efisien. Namun, dalam kondisi terburuk, misalnya ketika banyak elemen masuk ke dalam satu bucket yang sama, kinerja algoritma dapat menurun hingga $O(n^2)$. Meskipun demikian, bucket sort tetap menjadi salah satu teknik pengurutan yang efektif untuk data numerik dengan distribusi nilai yang relatif seragam dalam suatu rentang tertentu.

Proses eksekusi algoritma bucket sort`yaitu:

- Menentukan nilai terkecil (minimum) dan nilai terbesar (maksimum) dari array yang belum terurut.
- Menggunakan kedua nilai tersebut untuk mengetahui rentang data yang akan menjadi dasar pembagian elemen ke dalam bucket.
- Membagi rentang nilai tersebut menjadi beberapa bucket yang sesuai.
- Membuat sejumlah bucket kosong, biasanya disesuaikan dengan jumlah elemen dalam array.
- Mendistribusikan setiap elemen dari array utama ke dalam bucket yang sesuai berdasarkan nilai elemennya.
- Mengurutkan elemen-elemen yang berada di dalam setiap bucket secara terpisah menggunakan algoritma pengurutan lain, seperti insertion sort atau metode pengurutan lainnya.
- Setelah semua bucket selesai diurutkan, menggabungkan isi setiap bucket secara berurutan sesuai dengan urutan bucketnya.
- Hasil penggabungan tersebut menghasilkan array akhir yang telah terurut secara keseluruhan.

Algoritma BucketSort(A):
Input : Array A berisi n elemen yang belum terurut
Output : Array A yang telah terurut

```
1. n ← panjang array A

2. Tentukan nilai minimum dan maksimum
   min ← nilai terkecil dalam A
   max ← nilai terbesar dalam A

3. Hitung rentang data
   range ← max - min + 1

4. Tentukan jumlah bucket
   bucket_count ← n

5. Buat bucket kosong
   Buat array B yang berisi bucket_count bucket kosong

6. Distribusikan elemen ke bucket
   Untuk setiap elemen x dalam A lakukan
      index ← floor((x - min) / range * bucket_count)
      Jika index = bucket_count maka
         index ← bucket_count - 1
      Masukkan x ke dalam bucket B[index]

7. Urutkan setiap bucket
   Untuk i ← 0 sampai bucket_count-1 lakukan
      Urutkan elemen dalam bucket B[i]
      (misalnya menggunakan insertion sort)

8. Gabungkan semua bucket
   k ← 0
   Untuk i ← 0 sampai bucket_count-1 lakukan
      Untuk setiap elemen x dalam B[i] lakukan
         A[k] ← x
         k ← k + 1

9. Kembalikan array A yang telah terurut
```

Keunggulan algoritma bucket yaitu:

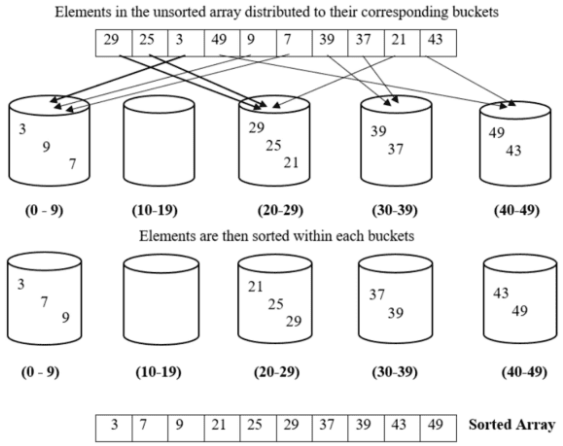
- Kompleksitas waktu linear. Bucket sort dapat mencapai kompleksitas waktu linear, sehingga proses pengurutan dapat berlangsung sangat cepat dalam kondisi tertentu.
- Mendukung pemrosesan paralel. Proses pengurutan dapat dilakukan secara paralel, karena setiap bucket dapat diproses dan diurutkan secara independen tanpa bergantung bucket lainnya.
- Efisien untuk distribusi data yang merata. Algoritma ini sangat efektif apabila data yang akan diurutkan terdistribusi secara seragam dalam suatu rentang nilai.

- Kinerja cepat dalam kondisi ideal. Jika distribusi data merata dan strategi pembagian bucket tepat, bucket sort dapat bekerja lebih cepat dibandingkan banyak algoritma pengurutan berbasis perbandingan yang memiliki kompleksitas rata-rata O(n log n).

Keterbatasan algoritma bucket yaitu:

- Implementasi relatif lebih kompleks. Implementasi bucket sort dapat menjadi lebih sulit karena setiap bucket biasanya menggunakan struktur data seperti linked list atau struktur penyimpanan tambahan lainnya.
- Bergantung terhadap distribusi data. Kinerja algoritma sangat dipengaruhi oleh cara data terdistribusi serta jumlah bucket yang digunakan. Jika distribusi data tidak merata, performa algoritma dapat menurun.
- Membutuhkan ruang memori tambahan. Bucket sort memerlukan ruang penyimpanan tambahan untuk menampung bucket-bucket yang digunakan selama proses pengurutan.

Gambar 10.12 menunjukkan ilustrasi proses kerja algoritma Bucket Sort dalam mengurutkan data numerik. Bagian paling atas terlihat sebuah array yang belum terurut yang berisi elemen-elemen data, yaitu 29,



Gambar 10.12: Ilustrasi Bucket Sort (Sundaramoorthy & Karunanidhi, 2025)

25, 3, 49, 9, 7, 39, 37, 21, dan 43. Algoritma bucket sort bekerja dengan cara membagi rentang nilai data menjadi beberapa kelompok yang disebut bucket. Rentang nilai dibagi menjadi lima bucket, yaitu (0–9), (10–19), (20–29), (30–39), dan (40–49). Setiap bucket merepresentasikan interval nilai tertentu sehingga setiap elemen dari array awal dapat dimasukkan ke dalam bucket yang sesuai dengan nilai yang dimilikinya.

Tahap pertama, seluruh elemen dari array yang belum terurut didistribusikan ke dalam bucket yang sesuai dengan rentang nilainya. Misalnya, nilai 3, 9, dan 7 dimasukkan ke dalam bucket (0–9) karena ketiganya berada dalam rentang tersebut. Nilai 29, 25, dan 21 dimasukkan ke dalam bucket (20–29), sedangkan nilai 39 dan 37 ditempatkan di bucket (30–39). Sementara itu, nilai 49 dan 43 dimasukkan ke dalam bucket (40–49). Bucket (10–19) tidak berisi elemen apa pun karena tidak terdapat nilai dari array yang berada dalam rentang tersebut. Proses distribusi ini merupakan langkah penting dalam bucket sort karena bertujuan mengelompokkan data ke dalam bagian-bagian yang lebih kecil sehingga mempermudah proses pengurutan selanjutnya.

Setelah seluruh elemen berhasil dimasukkan ke dalam bucket yang sesuai, tahap berikutnya adalah mengurutkan elemen-elemen di dalam setiap bucket secara terpisah. Setiap bucket yang berisi elemen kemudian diurutkan dari nilai terkecil ke nilai terbesar. Misalnya, bucket (0–9) elemen 3, 9, dan 7 diurutkan menjadi 3, 7, dan 9. Bucket (20–29) elemen 29, 25, dan 21 diurutkan menjadi 21, 25, dan 29. Bucket (30–39) yang berisi 37 dan 39, serta bucket (40–49) yang berisi 43 dan 49, masing-masing diurutkan sesuai urutan naik. Proses pengurutan di dalam bucket biasanya menggunakan algoritma pengurutan sederhana seperti insertion sort, karena jumlah elemen dalam setiap bucket umumnya relatif kecil.

Tahap terakhir adalah menggabungkan kembali seluruh elemen dari setiap bucket secara berurutan berdasarkan urutan bucketnya. Proses penggabungan dilakukan dengan mengambil elemen dari bucket pertama hingga bucket terakhir. Dimulai dari bucket (0–9) yang menghasilkan urutan 3, 7, 9, kemudian diikuti bucket (20–29) yang menghasilkan 21, 25, 29, dilanjutkan bucket (30–39) dengan 37, 39, dan terakhir bucket (40–49) dengan 43, 49. Hasil akhir dari proses penggabungan ini membentuk sebuah array yang telah terurut secara keseluruhan, yaitu 3, 7, 9, 21, 25, 29, 37, 39, 43, dan 49.

Program bucket sort dapat dilihat di link https://colab.research.google.com/drive/15AiJ69dmBdI0VB55CkwOPYGFZM7oz1b0?usp=sharing.

10.11 Cocktail Sort

Latihan:
Soal 1:
Diketahui sebuah array yang belum terurut sebagai berikut: A = [29, 25, 3, 49, 9, 7, 39, 37, 21, 43]
Gunakan algoritma Bucket Sort untuk mengurutkan data tersebut dengan ketentuan sebagai berikut:
Rentang bucket dibagi menjadi: (0–9), (10–19), (20–29), (30–39), (40–49)

Soal 2
Diketahui sebuah data yang belum terurut sebagai berikut: B = [42, 12, 33, 25, 7, 18, 46, 29, 4, 38]
Gunakan algoritma Bucket Sort untuk mengurutkan data tersebut dengan ketentuan berikut:
Rentang bucket dibagi menjadi:
(0–9), (10–19), (20–29), (30–39), (40–49)

Soal 3
Buatkan code bucket sort dengan data dari soal 1 dan 2.

Cocktail sort merupakan salah satu variasi dari algoritma bubble sort yang melakukan proses pengurutan secara dua arah (*bidirectional*). Algoritma ini sering juga disebut sebagai *bidirectional bubble sort* atau *shaker sort*. Berbeda dengan bubble sort konvensional yang hanya melakukan penelusuran dari satu arah, cocktail sort memungkinkan elemen-elemen dalam array bergerak baik ke arah depan maupun ke arah belakang selama proses pengurutan. Pendekatan dua arah ini bertujuan untuk memperbaiki keterbatasan bubble sort dengan mempercepat perpindahan elemen yang berada jauh dari posisi yang seharusnya (Sundaramoorthy & Karunanidhi, 2025).

Tahap pertama, algoritma melakukan iterasi maju dari kiri ke kanan, mirip dengan proses bubble sort, sehingga elemen dengan nilai terbesar akan berpindah ke posisi paling akhir dalam array. Setelah proses tersebut selesai, algoritma melanjutkan dengan iterasi mundur dari kanan ke kiri, yang bertujuan memindahkan elemen dengan nilai terkecil ke posisi paling awal dalam array. Dengan demikian, dalam satu siklus iterasi lengkap, elemen terbesar dan elemen terkecil secara bersamaan dapat ditempatkan di posisi yang tepat. Iterasi berikutnya, elemen terbesar kedua dan elemen terkecil kedua juga akan dipindahkan ke posisi yang sesuai. Proses ini terus diulang hingga seluruh elemen dalam array tersusun secara terurut.

Karena melakukan proses penelusuran secara dua arah, cocktail sort terkadang mampu mengurangi jumlah iterasi dibandingkan dengan bubble sort tradisional, khususnya ketika data tidak sepenuhnya acak atau hanya mengalami sedikit ketidakteraturan. Algoritma ini cukup efektif untuk dataset yang relatif kecil atau hampir terurut, karena elemen yang lebih kecil maupun lebih besar dapat dipindahkan ke posisi yang benar dengan lebih cepat. Meskipun demikian, dari segi kompleksitas waktu, cocktail sort masih memiliki kompleksitas $O(n^2)$ baik

dalam kasus rata-rata maupun kasus terburuk. Oleh karena itu, untuk dataset yang besar, algoritma ini umumnya kurang efisien dibandingkan algoritma pengurutan yang lebih maju seperti quick sort atau merge sort, meskipun mekanisme bidirectional yang dimilikinya tetap memberikan keunggulan dalam situasi tertentu.

Proses eksekusi cocktail sort yaitu:

- Proses eksekusi algoritma cocktail sort dalam setiap iterasi dibagi menjadi dua fase utama. Fase pertama dimulai dengan melakukan penelusuran dari sisi kiri menuju sisi kanan array, yang mekanismenya serupa dengan algoritma bubble sort.
- Selama proses penelusuran tersebut, elemen-elemen yang berdekatan dibandingkan secara berurutan.
- Apabila nilai elemen di sebelah kiri lebih besar daripada nilai elemen di sebelah kanan, maka kedua elemen tersebut ditukar posisinya (swap).
- Melalui proses perbandingan dan pertukaran tersebut, elemen dengan nilai terbesar secara bertahap akan berpindah ke bagian akhir array.
- Setelah satu kali penelusuran dari kiri ke kanan selesai dilakukan, elemen terbesar akan berada di posisi yang tepat di ujung kanan array.

Algoritma CocktailSort(A):

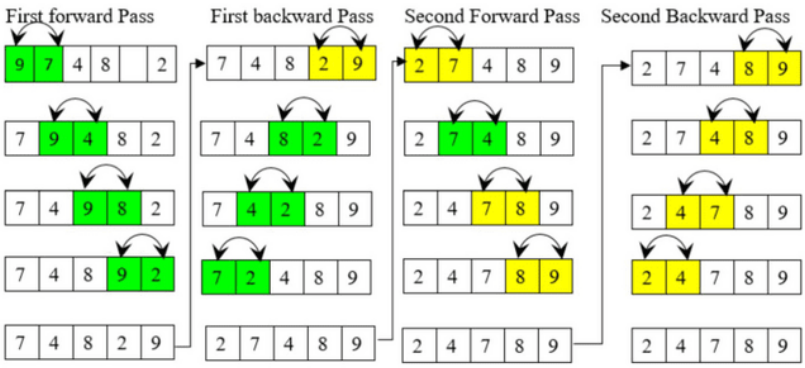
```
Input : Array A berisi n elemen yang belum terurut
Output : Array A yang telah terurut

1. n ← panjang array A
2. start ← 0
3. end ← n - 1
4. swapped ← true

5. Selama swapped = true lakukan
    swapped ← false

    // Fase 1 : Traversal kiri ke kanan
    untuk i ← start sampai end - 1 lakukan
        jika A[i] > A[i+1] maka
            tukar A[i] dengan A[i+1]
            swapped ← true
    end ← end - 1

    jika swapped = false maka
        berhenti

    swapped ← false

    // Fase 2 : Traversal kanan ke kiri
    untuk i ← end - 1 turun sampai start lakukan
        jika A[i] > A[i+1] maka
            tukar A[i] dengan A[i+1]
            swapped ← true
    start ← start + 1

6. Kembalikan array A yang telah terurut
```

Keunggulan cocktail sort yaitu:

- Sederhana. Algoritma ini mudah dipahami dan relatif mudah diimplementasikan karena mekanisme kerjanya tidak terlalu kompleks.
- Lebih baik daripada bubble sort dalam kondisi tertentu. Dalam beberapa kasus, terutama ketika data yang akan diurutkan hampir terurut, cocktail sort dapat bekerja lebih efisien dibandingkan dengan bubble sort tradisional.
- Bersifat stabil. Cocktail sort mampu mempertahankan urutan relatif elemen-elemen yang memiliki nilai yang sama, sehingga kestabilan data tetap terjaga setelah proses pengurutan.

Kelemahan cocktail sort yaitu:

- Kurang efisien. Algoritma ini memiliki kompleksitas waktu $O(n^2)$ baik dalam kasus rata-rata maupun kasus terburuk, sehingga kinerjanya kurang optimal untuk jumlah data yang besar.
- Banyak perbandingan yang tidak perlu. Seperti halnya bubble sort, algoritma ini masih melakukan banyak perbandingan dan pertukaran elemen yang sebenarnya tidak diperlukan.
- Tidak cocok untuk dataset besar: Karena kompleksitas waktunya bersifat kuadratik, cocktail sort menjadi kurang praktis untuk digunakan dengan dataset dengan jumlah elemen yang besar.



Gambar 10.13: Contoh Cocktail Sort (Sundaramoorthy & Karunanidhi, 2025)

Gambar 10.3 memperlihatkan proses pengurutan menggunakan algoritma cocktail sort melalui beberapa tahap iterasi yang dilakukan secara dua arah, yaitu dari kiri ke kanan (*forward pass*) dan dari kanan ke kiri (*backward pass*). Array awal yang belum terurut adalah [9, 7, 4, 8, 2]. Proses pengurutan dimulai dengan iterasi pertama dari kiri ke kanan (*first forward pass*), di mana elemen-elemen yang berdekatan dibandingkan satu per satu. Jika elemen di sebelah kiri lebih besar daripada elemen di sebelah kanan, maka kedua elemen tersebut ditukar posisinya. Pasangan elemen 9 dan 7 ditukar sehingga menjadi [7, 9, 4, 8, 2]. Selanjutnya, elemen 9 dan 4 dibandingkan dan ditukar menjadi [7, 4, 9, 8, 2], kemudian 9 dan 8 ditukar menjadi [7, 4, 8, 9, 2], dan terakhir 9 dan 2 ditukar sehingga menghasilkan [7, 4, 8, 2, 9]. Setelah iterasi pertama dari kiri ke kanan selesai, elemen dengan nilai terbesar, yaitu 9, telah berada dalam posisi yang tepat di bagian akhir array.

Tahap berikutnya adalah iterasi pertama dari kanan ke kiri (*first backward pass*). Proses perbandingan dilakukan secara berlawanan arah, dimulai dari elemen sebelum elemen terakhir menuju ke awal array. Pasangan elemen 8 dan 2 dibandingkan dan ditukar sehingga menjadi [7, 4, 2, 8, 9]. Kemudian elemen 4 dan 2 dibandingkan dan ditukar menjadi [7, 2, 4, 8, 9], dan terakhir elemen 7 dan 2 ditukar sehingga menghasilkan [2, 7, 4, 8, 9]. Setelah proses iterasi mundur ini selesai, elemen dengan nilai terkecil, yaitu 2, telah berpindah ke posisi awal array.

Proses pengurutan kemudian dilanjutkan dengan iterasi kedua dari kiri ke kanan (*second forward pass*). Elemen-elemen yang tersisa di bagian tengah array kembali dibandingkan. Elemen 7 dan 4 dibandingkan dan ditukar sehingga menghasilkan [2, 4, 7, 8, 9], sedangkan pasangan elemen berikutnya tidak memerlukan pertukaran karena sudah berada dalam urutan yang benar. Setelah itu dilakukan iterasi kedua dari kanan ke kiri (*second backward pass*) untuk memastikan tidak ada lagi elemen yang berada dalam posisi yang salah. Karena semua elemen sudah tersusun dengan benar, tidak terjadi pertukaran tambahan.

Hasil akhir dari proses tersebut adalah array yang telah terurut secara menaik, yaitu [2, 4, 7, 8, 9]. Ilustrasi ini menunjukkan bahwa algoritma cocktail sort bekerja dengan cara melakukan perbandingan dan pertukaran elemen secara dua arah dalam setiap iterasi, sehingga elemen terbesar dan terkecil dapat berpindah menuju posisi yang tepat secara bersamaan. Pendekatan *bidirectional* ini membuat cocktail sort dalam beberapa kasus lebih efisien dibandingkan bubble sort tradisional, terutama ketika data yang diurutkan tidak sepenuhnya acak atau hanya mengalami sedikit ketidakteraturan.

Program algoritma cocktail dapat dilihat di link https://colab.research.google.com/drive/1A4kGQ5rz3sPZfTrwlOrWOpqyttjI-AV-?usp=sharing.

Latihan:
Soal 1:

Diketahui sebuah array yang belum terurut sebagai berikut: A = [9, 7, 4, 8, 2, 23, 15, 63, 52]

Gunakan algoritma cocktail sort secara ascending.

Soal 2:
Diketahui sebuah data yang belum terurut sebagai berikut: B = [5, 1, 4, 2, 8, 3, 10, 15, 12, 25, 22, 30, 56, 45]
Gunakan algoritma cocktail sort secara descending.

Soal 3:
Buatkan code cocktail sort dengan data dari soal 1 dan 2.

10.12 Comb Sort

Comb sort merupakan salah satu algoritma pengurutan yang berbasis perbandingan (*comparison-based sorting*) dan dikembangkan sebagai perbaikan dari algoritma bubble sort. Algoritma ini bekerja dengan cara membandingkan elemen-elemen dalam array yang tidak selalu berdekatan, melainkan memiliki jarak tertentu (gap) di antara keduanya. Dengan membandingkan elemen yang berjauhan, comb sort dapat memindahkan elemen-elemen yang lebih kecil menuju bagian awal daftar dengan lebih cepat dibandingkan bubble sort yang hanya membandingkan elemen yang bersebelahan (Sundaramoorthy & Karunanidhi, 2025).

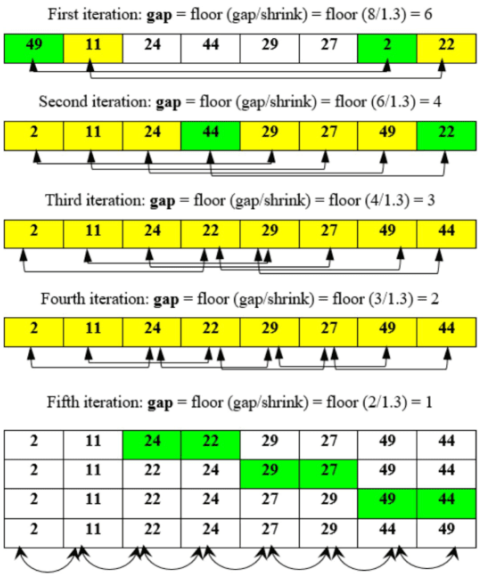
Prinsip utama dari comb sort adalah penggunaan faktor penyusutan (shrink factor) untuk secara bertahap mengurangi nilai jarak (gap) antara elemen-elemen yang dibandingkan. Proses pengurutan dimulai dengan menggunakan nilai gap yang besar, kemudian setiap iterasi nilai gap tersebut dikurangi dengan membaginya menggunakan faktor tertentu, biasanya sekitar 1,3, hingga akhirnya nilai gap menjadi 1. Ketika gap telah bernilai satu, mekanisme kerja algoritma akan menyerupai proses bubble sort.

Keunggulan utama dari comb sort yaitu penggunaan gap awal yang besar, yang memungkinkan algoritma ini lebih cepat mengatasi elemen-elemen kecil yang berada jauh di bagian akhir array, yang sering disebut sebagai turtles. Dengan pendekatan ini, comb sort mampu meningkatkan efisiensi proses pengurutan dibandingkan bubble sort tradisional.

Proses eksekusi comb sort yaitu:

- Algoritma dimulai dengan menentukan nilai gap awal, yang umumnya sama dengan panjang daftar atau array yang akan diurutkan.
- Setiap iterasi proses pengurutan, nilai gap dikurangi menggunakan faktor penyusutan (*shrink factor*), biasanya sekitar 1,3.
- Pengurangan gap ini dilakukan secara bertahap sehingga jarak antara elemen yang dibandingkan semakin kecil untuk setiap iterasi.
- Setiap iterasi, algoritma akan membandingkan pasangan elemen yang terpisah sejauh nilai gap saat itu.
- Apabila pasangan elemen yang dibandingkan tidak berada dalam urutan yang benar, maka kedua elemen tersebut ditukar posisinya (swap).
- Proses perbandingan dan pertukaran ini terus dilakukan sambil mengurangi nilai gap secara bertahap.
- Iterasi berlanjut hingga nilai gap menjadi 1, di mana proses pengurutan kemudian bekerja seperti bubble sort.
- Setelah semua perbandingan selesai dan tidak ada lagi pertukaran yang diperlukan, array akan berada dalam kondisi terurut.

```
Algoritma CombSort(A):
1. gap = panjang(A)
2. swapped = true
3. shrink = 1.3

4. while gap > 1 atau swapped = true
5.     gap = floor(gap / shrink)
6.     if gap < 1 then gap = 1
7.     swapped = false

8.     for i = 0 to panjang(A) - gap - 1
9.         if A[i] > A[i + gap]
10.            swap A[i], A[i + gap]
11.            swapped = true
12.        end if
13.    end for
14. end while
```



Gambar 10.14: Contoh Algoritma Comb Sort

Gambar 10.14 memperlihatkan proses pengurutan menggunakan algoritma Comb Sort melalui beberapa tahap iterasi dengan nilai gap yang secara bertahap semakin kecil. Data awal yang akan diurutkan adalah [49, 11, 24, 44, 29, 27, 2, 22]. Algoritma comb sort bekerja dengan cara membandingkan elemen-elemen yang memiliki jarak tertentu (gap) dalam array. Awal proses, nilai gap ditentukan berdasarkan panjang array, kemudian secara bertahap dikurangi menggunakan faktor penyusutan (*shrink factor*) sebesar sekitar 1,3 hingga akhirnya mencapai nilai 1.

Iterasi pertama, nilai gap dihitung menggunakan **rumus gap = floor(gap / shrink)**, sehingga diperoleh nilai gap = 6 dari perhitungan floor(8 / 1,3). Dengan gap tersebut, elemen yang berjarak enam posisi dibandingkan, yaitu elemen pertama dengan elemen ketujuh (49 dan 2) serta elemen kedua dengan elemen kedelapan (11 dan 22). Ketika pasangan elemen tidak berada dalam urutan yang benar, maka dilakukan pertukaran. Tahap ini, 49 dan 2 ditukar sehingga elemen yang lebih kecil berpindah ke bagian awal array.

Iterasi kedua, nilai gap kembali dikurangi menggunakan faktor penyusutan sehingga diperoleh gap = 4 dari perhitungan floor(6 / 1,3). Selanjutnya dilakukan perbandingan antara elemen yang dipisahkan oleh jarak empat posisi. Proses ini menghasilkan beberapa pertukaran sehingga elemen-elemen yang lebih kecil semakin mendekati posisi yang benar di bagian awal array, sedangkan elemen yang lebih besar bergerak menuju bagian akhir.

Iterasi ketiga, nilai gap kembali dikurangi menjadi 3, kemudian perbandingan dilakukan terhadap elemen-elemen yang berjarak tiga posisi. Proses ini membantu memperbaiki urutan data secara lebih rinci. Setelah itu, iterasi keempat, nilai gap kembali dikurangi menjadi 2, sehingga perbandingan dilakukan antara elemen yang berjarak dua posisi. Tahap ini semakin memperbaiki urutan data dalam array.

Iterasi kelima, nilai gap akhirnya menjadi 1, sehingga proses pengurutan bekerja seperti bubble sort, yaitu membandingkan elemen-elemen yang berdekatan. Tahap ini, beberapa pertukaran dilakukan hingga tidak ada lagi pasangan elemen yang berada dalam urutan yang salah. Setelah seluruh proses selesai, array akhirnya tersusun secara terurut menjadi [2, 11, 22, 24, 27, 29, 44, 49]. Ilustrasi tersebut menunjukkan bagaimana algoritma comb sort memanfaatkan pengurangan nilai gap secara bertahap untuk mempercepat perpindahan elemen yang jauh dari posisi yang seharusnya, sehingga proses pengurutan menjadi lebih efisien dibandingkan dengan metode bubble sort tradisional.

Program comb sort dapat dilihat di link https://colab.research.google.com/drive/1lj2mUf9jRlEBIsc32TsgVGbnnfc1vavm?usp=sharing.

Latihan:
Soal 1:
Diketahui sebuah array data sebagai berikut: [72, 15, 94, 38, 61, 47, 29, 83, 56]
Urutkan data tersebut menggunakan algoritma comb sort (ascending) dengan shrink factor = 1,3.

Soal 2:
Diberikan data berikut: [84, 27, 63, 19, 91, 42, 58, 36, 74, 50]
Gunakan algoritma comb sort untuk mengurutkan data tersebut secara descending dengan shrink factor = 1,3.

Soal 3:
Buatkan code comb sort dengan data dari soal 1 dan 2.

10.13 Cycle Sort

Cycle sort merupakan algoritma pengurutan berbasis perbandingan (*comparison-based sorting*) yang dirancang untuk meminimalkan jumlah operasi penulisan (*write*) ke memori. Algoritma ini bekerja dengan menempatkan setiap elemen langsung dalam posisi yang seharusnya di dalam array, sehingga setiap elemen umumnya hanya ditulis satu kali. Karakteristik tersebut menjadikan cycle sort sangat berguna untuk situasi di mana operasi penulisan ke memori memiliki biaya yang tinggi, misalnya sistem tertanam (*embedded systems*) atau media penyimpanan seperti flash memory yang memiliki keterbatasan jumlah siklus penulisan (Alaa et al., 2024).

Proses kerja cycle sort dilakukan dengan cara mengidentifikasi siklus (cycles) dalam permutasi elemen array. Setiap siklus menunjukkan sekelompok elemen yang saling bertukar posisi hingga seluruh elemen berada dalam posisi yang benar. Jika suatu elemen sudah berada dalam posisi yang tepat, maka elemen tersebut tidak perlu dipindahkan dan akan dilewati oleh algoritma. Proses ini kemudian diulangi untuk setiap elemen dalam array hingga seluruh data tersusun secara terurut.

Keunggulan utama dari cycle sort yaitu efisiensi jumlah operasi penulisan, yang secara teoritis maksimum hanya sebesar $O(n-1)$ untuk array yang berisi n elemen. Namun demikian, dari sisi kompleksitas waktu, algoritma ini masih memiliki kompleksitas $O(n^2)$ karena tetap memerlukan banyak perbandingan untuk mengidentifikasi siklus-siklus dalam array. Meskipun demikian, kemampuan cycle sort dalam meminimalkan operasi penulisan menjadikannya menarik untuk digunakan dalam konteks tertentu yang membutuhkan efisiensi akses memori.

Langkah-langkah algoritma cycle sort yaitu:

1. **Tentukan jumlah elemen array**
   Hitung panjang array A dan simpan dalam variabel n.
2. **Mulai dari elemen pertama**
   Lakukan proses pengurutan dengan iterasi indeks **cycleStart** dari **0 sampai n−2**.
3. **Ambil elemen yang akan ditempatkan**
   Simpan nilai elemen di posisi **cycleStart** ke dalam variabel **item**.
4. **Tentukan posisi yang benar untuk elemen tersebut**
   - Inisialisasi **pos = cycleStart**.
   - Bandingkan **item** dengan elemen lain dalam indeks **cycleStart + 1 sampai n − 1**.

```
Algoritma CycleSort(A):
1. n ← length(A)
2. for cycleStart ← 0 to n − 2 do
3.    item ← A[cycleStart]
4.    pos ← cycleStart
5.    for i ← cycleStart + 1 to n − 1 do
6.       if A[i] < item then
7.          pos ← pos + 1
8.       end if
9.    end for
10.   if pos = cycleStart then
11.      continue
12.   end if
13.   while item = A[pos] do
14.      pos ← pos + 1
15.   end while
16.   swap(item, A[pos])
17.   while pos ≠ cycleStart do
18.      pos ← cycleStart
19.      for i ← cycleStart + 1 to n − 1 do
20.         if A[i] < item then
21.            pos ← pos + 1
22.         end if
23.      end for

24.      while item = A[pos] do
25.         pos ← pos + 1
26.      end while
27.      swap(item, A[pos])
28.   end while
29. end for

30. return A
```

- Setiap kali ditemukan elemen yang **lebih kecil dari item**, maka **pos ditambah 1**.

5. **Periksa apakah elemen sudah berada di posisi yang benar**
   - Jika **pos = cycleStart**, berarti elemen sudah berada di posisi yang tepat.
   - Lanjutkan ke elemen berikutnya.
6. **Tangani elemen yang bernilai sama (duplicate)**
   Jika nilai **item sama dengan A[pos]**, maka **pos ditambah 1** sampai menemukan posisi yang berbeda.
7. **Tukar elemen dengan posisi yang benar**
   Tukarkan nilai **item** dengan elemen di posisi **A[pos]**.
8. **Lanjutkan rotasi siklus (cycle rotation)**
   Selama **pos ≠ cycleStart**, lakukan langkah berikut:
   - Set kembali **pos = cycleStart**.
   - Cari kembali posisi yang benar untuk **item** dengan membandingkan elemen dari **cycleStart + 1 sampai n − 1**.
   - Tambahkan **pos** jika ditemukan elemen yang lebih kecil dari **item**.
   - Jika ditemukan elemen yang bernilai sama, geser **pos** ke indeks berikutnya.
   - Tukarkan **item** dengan **A[pos]**.
9. **Selesaikan satu siklus**
   Proses ini akan memindahkan semua elemen yang berada dalam satu **cycle** hingga setiap elemen berada di posisi yang benar.
10. **Ulangi proses untuk elemen berikutnya**
    Lakukan langkah yang sama hingga seluruh elemen array diperiksa.
11. **Hasil akhir**
    Setelah semua siklus selesai diproses, array akan **terurut secara ascending**.

Kelebihan algoritma cycle sort yaitu:

- Meminimalkan operasi penulisan. Cycle sort mampu mengurangi jumlah operasi penulisan yang diperlukan selama proses pengurutan. Hal ini menjadikannya sangat bermanfaat untuk media penyimpanan yang sensitif terhadap jumlah penulisan, seperti flash memory atau perangkat penyimpanan dengan keterbatasan siklus tulis.
- In-place. Algoritma ini bekerja langsung dalam array yang diurutkan tanpa memerlukan alokasi memori tambahan di luar array tersebut.
- Optimal secara teoretis dalam jumlah penulisan. Setiap elemen dipindahkan ke posisi yang benar dengan jumlah operasi penulisan yang sangat sedikit, bahkan secara teoretis setiap elemen cukup dipindahkan satu kali untuk mencapai posisi yang tepat.

Kekurangan algoritma cycle sort yaitu:

- Kompleksitas waktu. Meskipun efisien dalam mengurangi operasi penulisan, algoritma ini memiliki kompleksitas waktu sebesar O(n^2) baik untuk kasus rata-rata maupun kasus terburuk.
- Tidak stabil. Algoritma ini tidak mempertahankan urutan relatif elemen-elemen yang memiliki nilai sama, sehingga termasuk dalam kategori algoritma pengurutan yang tidak stabil.
- Implementasi lebih kompleks. Proses implementasi cycle sort relatif lebih rumit dibandingkan dengan algoritma pengurutan yang lebih sederhana, seperti bubble sort atau insertion sort.

Secara keseluruhan, cycle sort merupakan algoritma pengurutan yang memiliki karakteristik unik karena mampu meminimalkan jumlah operasi penulisan selama proses pengurutan. Keunggulan tersebut menjadikannya sesuai digunakan dalam kondisi tertentu yang menuntut efisiensi akses memori. Meskipun demikian, algoritma

ini tetap memiliki kompleksitas waktu kuadratik, sehingga penggunaannya perlu mempertimbangkan kebutuhan dan karakteristik sistem yang digunakan.

Program algoritma cycle sort dapat dilihat di link https://colab.research.google.com/drive/1oRTfT3qWa5b5mOCoYAaLn56JZ1JsX-Gb?usp=sharing.

10.14 Gnome Sort

Gnome sort, yang terkadang dijuluki sebagai "*stupid sort*", merupakan algoritma pengurutan yang sederhana dan memiliki kemiripan dengan insertion sort, meskipun menggunakan pendekatan yang berbeda dalam menata elemen-elemen data. Algoritma ini bekerja dengan cara menelusuri array secara berurutan, kemudian menukar elemen-elemen yang tidak berada dalam urutan yang benar, lalu bergerak mundur untuk memeriksa kembali elemen sebelumnya hingga susunan yang tepat tercapai. Mekanisme tersebut membuat gnome sort cukup mudah dipahami dan diimplementasikan. Namun demikian, kinerja algoritma ini menjadi kurang efisien ketika diterapkan terhadap kumpulan data yang berukuran besar, karena memiliki kompleksitas waktu rata-rata dan kasus terburuk sebesar O(n^2). Oleh karena itu, gnome sort lebih sering digunakan sebagai sarana pembelajaran untuk memahami konsep dasar algoritma pengurutan atau diterapkan dalam array yang berukuran kecil (Alaa et al., 2024).

Kelebihan algoritma gnome sort yaitu:

- Sederhana (*Simplicity*). Gnome sort merupakan algoritma yang mudah dipahami karena langkah-langkahnya sederhana. Selain itu, proses implementasinya dalam berbagai bahasa pemrograman juga relatif mudah dilakukan.
- In-Place. Algoritma ini bekerja langsung dalam array yang akan diurutkan tanpa memerlukan tambahan memori di luar array tersebut, sehingga penggunaan memori menjadi lebih efisien.
- Adaptif terhadap data kecil atau hampir terurut. Gnome sort menunjukkan kinerja yang cukup baik ketika menggunakan dataset berukuran kecil atau array yang sebagian besar elemennya sudah berada dalam kondisi hampir terurut.

Kekurangan algoritma gnome sort yaitu:

- Kurang efisien untuk data besar. Gnome sort memiliki kompleksitas waktu sebesar O(n^2) baik dalam kasus rata-rata maupun kasus terburuk, sehingga kurang efisien apabila digunakan untuk mengurutkan dataset yang berukuran besar.
- Tidak stabil (*Lack of Stability*). Algoritma ini tidak selalu mempertahankan urutan relatif elemen-elemen yang memiliki nilai sama, sehingga termasuk dalam kategori algoritma pengurutan yang tidak stabil.
- Banyak perbandingan yang tidak perlu. Proses pengurutan gnome sort sering melibatkan perbandingan dan pertukaran elemen yang sebenarnya tidak diperlukan, sehingga dapat menurunkan efisiensi algoritma secara keseluruhan.

Secara keseluruhan, gnome sort merupakan algoritma pengurutan dasar yang mudah dipahami dan diimplementasikan. Algoritma ini lebih cocok digunakan untuk tujuan pembelajaran atau untuk mengurutkan dataset berukuran kecil maupun data yang hampir terurut. Namun, karena efisiensinya rendah terhadap dataset besar, penggunaannya dalam aplikasi praktis yang menangani data dalam jumlah besar menjadi terbatas.

Langkah-langkah Algoritma gnome sort yaitu:

1. Menentukan jumlah elemen array. Hitung panjang array A dan simpan nilainya dalam variabel n.
2. Inisialisasi indeks awal. Tetapkan variabel index = 0 sebagai posisi awal yang akan diperiksa dalam array.

3. Mulai proses pengurutan. Lakukan proses pengurutan selama nilai index < n.
4. Periksa posisi indeks pertama atau kondisi terurut. Jika index = 0 atau A[index] ≥ A[index − 1], berarti elemen saat ini sudah berada dalam posisi yang benar dibandingkan dengan elemen sebelumnya.
5. Geser indeks ke kanan. Apabila langkah sebelumnya terpenuhi, maka index ditambah 1 untuk memeriksa elemen berikutnya.

```
Algoritma GnomeSort(A):
1. n ← length(A)
2. index ← 0
3. while index < n do
4.    if index = 0 OR A[index] ≥ A[index − 1] then
5.       index ← index + 1
6.    else
7.       swap A[index] dengan A[index − 1]
8.       index ← index − 1
9.    end if
10. end while
11. return A
```

6. Periksa apakah elemen tidak terurut. Jika A[index] < A[index − 1], berarti urutan elemen salah.
7. Tukar elemen yang tidak terurut. Tukarkan elemen A[index] dengan A[index − 1].
8. Kembali memeriksa elemen sebelumnya. Setelah pertukaran dilakukan, kurangi nilai index sebesar 1 untuk memeriksa kembali posisi elemen yang baru saja dipindahkan.
9. Ulangi proses hingga seluruh elemen diperiksa. Langkah-langkah di atas diulangi sampai index mencapai nilai n.
10. Hasil akhir. Setelah seluruh proses selesai, array akan terurut secara *ascending*.

Inti mekanisme gnome sort yaitu algoritma ini bekerja dengan cara bergerak maju dan mundur dalam array, mirip seperti seseorang yang memeriksa pot bunga satu per satu. Jika ditemukan dua elemen yang tidak terurut, keduanya ditukar dan algoritma kembali satu langkah ke belakang untuk memastikan urutan tetap benar. Proses ini terus dilakukan hingga seluruh elemen berada dalam urutan yang benar.

Program gnome sort dapat dilihat di link https://colab.research.google.com/drive/143wCQ8pJz0ciy5av9dRLnL66KzCct-Ti?usp=sharing.

10.15 Timsort

Timsort merupakan algoritma pengurutan yang digunakan sebagai metode standar dalam Python sejak versi 2.3 dan seterusnya. Algoritma ini dirancang untuk memberikan kinerja yang optimal dalam pengolahan daftar data berukuran besar yang sering ditemukan dalam aplikasi nyata. Timsort menggabungkan keunggulan dari dua algoritma pengurutan, yaitu merge sort dan insertion sort. Insertion sort sangat efektif digunakan ketika ukuran data relatif kecil atau ketika sebagian data sudah berada dalam kondisi hampir terurut. Sementara itu, metode penggabungan (merge) dalam merge sort bekerja dengan cepat ketika diperlukan penggabungan beberapa bagian data yang telah terurut (Agarwal, 2022).

Konsep utama dari Timsort adalah dengan terlebih dahulu menggunakan insertion sort untuk mengurutkan bagian-bagian kecil dari data yang disebut sebagai blok atau chunks. Setelah setiap blok data tersebut tersusun secara terurut, algoritma kemudian memanfaatkan mekanisme merge sort untuk menggabungkan seluruh blok yang telah terurut menjadi satu daftar yang sepenuhnya terurut. Salah satu karakteristik penting dari Timsort adalah kemampuannya dalam memanfaatkan pola data yang sudah sebagian terurut, yang dikenal sebagai natural runs. Pola ini sering muncul dalam data dunia nyata, sehingga Timsort mampu bekerja lebih efisien dengan memanfaatkan kondisi tersebut selama proses pengurutan.

Langkah-langkah algoritma Timsort yaitu:

1. Membagi array menjadi beberapa blok (**run**).

Array yang berisi elemen data terlebih dahulu dibagi menjadi beberapa bagian kecil yang disebut run atau blok data.

2. Menentukan ukuran run (**minrun**).

Umumnya ukuran run yang digunakan adalah 32 atau 64 elemen, karena ukuran tersebut dianggap paling sesuai untuk algoritma Timsort. Namun ukuran ini juga dapat dihitung berdasarkan panjang array N. Nilai minrun merupakan panjang minimum setiap run dan ditentukan dengan beberapa prinsip berikut:

a. Ukuran minrun tidak boleh terlalu besar, karena setiap blok kecil akan diurutkan menggunakan insertion sort yang lebih efektif untuk daftar dengan jumlah elemen sedikit.
b. Panjang run juga tidak boleh terlalu kecil, karena hal ini akan menghasilkan jumlah run yang terlalu banyak sehingga proses penggabungan (merge) menjadi lebih lambat.
c. Karena merge sort bekerja paling efisien ketika jumlah run merupakan pangkat dua, maka sebaiknya jumlah run yang diperoleh dari perhitungan **N/minrun** mendekati atau merupakan pangkat dua.

3. Menentukan jumlah run berdasarkan ukuran run.

Sebagai contoh, jika ukuran run adalah 32, maka jumlah run dapat dihitung dengan rumus **jumlah run = ukuran array / 32**. Jika hasilnya merupakan pangkat dua, maka proses penggabungan akan berjalan lebih efisien. Dengan kata lain, penentuan minrun didasarkan oleh panjang array (N), rentang minrun (biasanya 32–64), efisiensi insertion sort untuk data kecil, serta efisiensi merge ketika jumlah run mendekati pangkat dua.

```
Algoritma TimSort(A):
Input  : Array A berisi n elemen yang belum terurut
Output : Array A yang telah terurut

1. n ← panjang(A)
2. minrun ← tentukan ukuran minimum run
   (misalnya 32 atau 64, atau dihitung dari n)
3. Bagi array A menjadi beberapa run
   setiap run memiliki panjang minrun
   atau sisa elemen terakhir
4. Untuk setiap run lakukan
     urutkan run tersebut menggunakan InsertionSort
5. size ← minrun
6. Selama size < n lakukan
     left ← 0

     Selama left < n lakukan
        mid ← left + size - 1
        right ← minimum(left + 2*size - 1, n - 1)

        Jika mid < right maka
            merge A[left..mid] dan A[mid+1..right]
        EndIf

        left ← left + 2*size
     EndWhile

     size ← 2 * size
   EndWhile
7. Kembalikan array A yang telah terurut
```

4. Mengurutkan setiap run menggunakan insertion sort.

Setelah run terbentuk, masing-masing run diurutkan satu per satu menggunakan algoritma insertion sort.

5. Menggabungkan run yang telah terurut.

Semua run yang sudah terurut kemudian digabungkan secara bertahap menggunakan metode merge dari algoritma merge sort.

6. Melipatgandakan ukuran subarray untuk setiap iterasi.

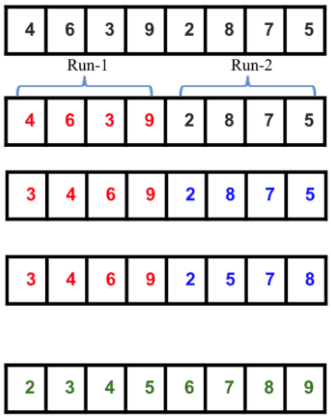
Setiap tahap penggabungan, ukuran subarray yang digabungkan akan menjadi dua kali lebih besar hingga seluruh elemen dalam array berhasil diurutkan.

```
Algoritma InsertionSort(A, kiri, kanan):
1. Untuk i ← kiri + 1 sampai kanan lakukan
2.    key ← A[i]
3.    j ← i - 1

4.    Selama j ≥ kiri dan A[j] > key lakukan
5.       A[j + 1] ← A[j]
6.       j ← j - 1
7.    EndWhile

8.    A[j + 1] ← key
9. EndFor
```

Algoritma Merge(A, kiri, tengah, kanan):
1. Salin subarray kiri A[kiri..tengah] ke array sementara L
2. Salin subarray kanan A[tengah+1..kanan] ke array sementara R
3. i ← 0
4. j ← 0
5. k ← kiri
6. Selama i < panjang(L) dan j < panjang(R) lakukan
7. Jika L[i] ≤ R[j] maka
8. A[k] ← L[i]
9. i ← i + 1
10. Jika tidak maka
11. A[k] ← R[j]
12. j ← j + 1
13. EndIf
14. k ← k + 1
15. EndWhile

16. Salin sisa elemen L ke A jika masih ada
17. Salin sisa elemen R ke A jika masih ada

Gambar 10.15 menjelaskan contoh proses kerja algoritma Timsort dalam mengurutkan sebuah array data. Terdapat array awal [4, 6, 3, 9, 2, 8, 7, 5]. Ukuran run ditentukan sebesar 4, sehingga array dibagi menjadi dua bagian kecil (run), yaitu Run-1 = [4, 6, 3, 9] dan Run-2 = [2, 8, 7, 5]. Pembagian ini merupakan tahap awal dalam algoritma Timsort, yaitu memecah data menjadi blok-blok kecil agar lebih mudah diurutkan.



Gambar 10.15: Ilustrasi Algoritma Timsort (Agarwal, 2022)

Tahap berikutnya adalah mengurutkan setiap run menggunakan algoritma Insertion Sort. Data Run-1 [4, 6, 3, 9] diurutkan menggunakan insertion sort hingga menjadi [3, 4, 6, 9]. Proses ini dilakukan dengan membandingkan setiap elemen dengan elemen sebelumnya, lalu menempatkannya di posisi yang tepat sehingga urutan menjadi menaik. Setelah Run-1 selesai diurutkan, proses yang sama dilakukan terhadap Run-2, yaitu data [2, 8, 7, 5] diurutkan menggunakan insertion sort hingga menghasilkan [2, 5, 7, 8]. Tahap ini kedua run sudah dalam kondisi terurut secara terpisah.

Setelah masing-masing run selesai diurutkan, langkah berikutnya adalah menggabungkan (merge) run-run yang telah terurut menggunakan metode merge dari algoritma Merge Sort. Proses penggabungan ini, elemen dari kedua run dibandingkan secara bertahap mulai dari elemen terkecil. Elemen yang memiliki nilai lebih kecil akan dimasukkan terlebih dahulu ke dalam array hasil penggabungan. Proses perbandingan dan pemindahan elemen dilakukan secara berulang hingga seluruh elemen dari kedua run telah digabungkan.

Hasil akhir dari proses penggabungan tersebut menghasilkan array yang telah terurut secara keseluruhan, yaitu [2, 3, 4, 5, 6, 7, 8, 9]. Timsort bekerja dengan cara menggabungkan dua pendekatan algoritma, yaitu Insertion Sort untuk mengurutkan blok-blok kecil data (run) dan Merge Sort untuk menggabungkan blok-blok yang telah

terurut tersebut. Pendekatan ini membuat Timsort sangat efisien terutama ketika data memiliki bagian yang sudah hampir terurut sebelumnya.

Algoritma Timsort dikenal sangat efisien untuk berbagai aplikasi di dunia nyata karena memiliki kompleksitas waktu terburuk sebesar O(n log n). Algoritma ini menjadi salah satu pilihan terbaik dalam proses pengurutan data, baik untuk daftar dengan jumlah elemen yang sedikit maupun yang sangat banyak. Ketika panjang daftar relatif kecil, Timsort memanfaatkan algoritma insertion sort, yang memiliki kinerja sangat cepat untuk data berukuran kecil. Sebaliknya, untuk daftar yang lebih panjang, Timsort menggunakan metode penggabungan (merge) yang berasal dari algoritma merge sort sehingga proses pengurutan tetap berjalan secara efisien. Oleh karena itu, kombinasi kedua pendekatan tersebut membuat Timsort memiliki kemampuan adaptif dalam menangani berbagai ukuran data, sehingga algoritma ini sangat sesuai digunakan untuk mengurutkan array dengan panjang apa pun dalam berbagai kondisi penggunaan nyata.

Program algoritma timsort dapat dilihat di link https://colab.research.google.com/drive/1BcwMGJ5i8G4sUYgJ9hJ3Ep15GMIJdSPY?usp=sharing.

10.16 Introspective Sort (Introsort)

Introspective Sort atau Introsort merupakan algoritma pengurutan berbasis perbandingan yang diperkenalkan oleh David R. Musser tahun 1997. Algoritma ini dirancang sebagai pendekatan hibrida yang menggabungkan keunggulan beberapa metode pengurutan, yaitu Quicksort sebagai mekanisme utama, Heapsort sebagai algoritma cadangan ketika kedalaman rekursi telah melewati batas tertentu, serta Insertion Sort untuk menangani partisi berukuran kecil secara lebih efisien. Gagasan utama Introsort ialah mempertahankan kinerja rata-rata Quicksort yang sangat baik, tetapi sekaligus menghindari penurunan performa menuju kasus terburuk kuadratik sebagaimana dapat terjadi ketika pembagian partisi terus-menerus tidak seimbang. Oleh sebab itu, Introsort memonitor kedalaman rekursi, yang dalam implementasi umum ditetapkan sekitar $2\lfloor \log_2 n \rfloor$, lalu mengalihkan proses ke Heapsort apabila batas tersebut tercapai. Dengan strategi ini, Introsort memiliki kompleksitas waktu rata-rata dan kasus terburuk sebesar O(n log n), serta tetap menunjukkan performa praktis yang kompetitif untuk data umum. Dalam kajian yang lebih mutakhir, Introsort juga dijelaskan sebagai algoritma pengurutan generik yang tidak stabil (unstable sort) dan menjadi dasar bagi implementasi penyortiran dalam pustaka standar C++ (Marcellino & Margi, 2021).

Secara konseptual, Introsort lahir dari kebutuhan untuk mengatasi kelemahan mendasar Quicksort. Musser menunjukkan bahwa sekalipun Quicksort sangat cepat terhadap banyak masukan, algoritma tersebut tetap rentan terhadap kasus-kasus tertentu yang menghasilkan partisi buruk secara berulang, sehingga waktu eksekusinya dapat memburuk menjadi $O(n^2)$. Untuk itu, Introsort menerapkan mekanisme "introspeksi", yaitu mengevaluasi kedalaman rekursi selama proses pengurutan. Jika jumlah level rekursi dianggap terlalu dalam, algoritma tidak lagi melanjutkan pola Quicksort, melainkan beralih ke Heapsort yang memiliki jaminan kompleksitas O(n log n) untuk kasus terburuk. Di sisi lain, untuk subarray kecil, penyelesaian akhir dilakukan dengan Insertion Sort karena pendekatan ini lebih sederhana dan efisien untuk ukuran data yang terbatas. Kombinasi tersebut menjadikan Introsort sebagai algoritma hibrida yang menyeimbangkan kecepatan praktis, ketahanan terhadap kasus terburuk, dan efisiensi implementasi.

Berikut langkah-langkah kerja Introsort yaitu:

1. Mulai proses pengurutan.
   Algoritma diawali dengan data atau daftar elemen yang akan diurutkan.
2. Hitung batas kedalaman rekursi (**maxDepth**)
   Nilai **maxDepth** ditentukan dengan rumus:
   maxDepth = log(length(*A*)) × 2
   Nilai ini digunakan sebagai batas agar proses rekursi tidak terlalu dalam.

3. Panggil fungsi IntroSort(List<>, maxDepth)
   Setelah maxDepth diperoleh, algoritma memanggil prosedur IntroSort dengan parameter daftar data dan batas kedalaman tersebut.
4. Hitung panjang data (n)
   Di dalam fungsi IntroSort, terlebih dahulu dihitung banyaknya elemen dalam daftar: **n = length (List<>)**
5. Periksa apakah jumlah elemen sangat kecil (n <= 1)
   Jika jumlah elemen kurang dari atau sama dengan 1, maka data dianggap sudah terurut, sehingga proses selesai.
6. Periksa apakah batas kedalaman rekursi sudah habis (maxDepth == 0)
   Jika jumlah elemen lebih dari 1, algoritma memeriksa apakah maxDepth sudah mencapai 0.
   - Jika ya, maka algoritma beralih ke HeapSort(List<>).
   - Peralihan ini dilakukan untuk menghindari kemungkinan performa buruk dari Quicksort dalam kasus terburuk.
7. Lakukan partisi jika maxDepth belum habis
   Jika maxDepth belum 0, maka algoritma menjalankan: **p = Partition(List<>)**
   Proses partisi ini membagi data menjadi dua sublist berdasarkan posisi pivot:
   - Bagian kiri: List<>[0 : p-1]
   - Bagian kanan: List<>[p+1 : n]
8. Kurangi nilai maxDepth
   Setelah partisi dilakukan, nilai batas kedalaman dikurangi satu: **maxDepth - -**
   Hal ini menandakan bahwa algoritma telah masuk satu tingkat rekursi lebih dalam.
9. Urutkan sublist kiri dan kanan secara rekursif
   Kedua bagian hasil partisi, yaitu sublist kiri dan sublist kanan, diproses kembali dengan IntroSort menggunakan nilai maxDepth yang sudah dikurangi.
10. Ulangi proses sampai selesai
    Langkah pemeriksaan ukuran data, pengecekan maxDepth, partisi, dan rekursi akan terus dilakukan sampai:
    - sublist hanya memiliki 0 atau 1 elemen, atau
    - maxDepth habis sehingga algoritma beralih ke Heapsort.
11. Proses pengurutan selesai
    Setelah seluruh sublist selesai diproses, daftar akhir menjadi terurut.

Algoritma Introsort(A):
Input : array A
Output : array A terurut

1. maxDepth ← 2 × log(length(A))
2. IntroSort(A, maxDepth)

Prosedur IntroSort(List, maxDepth)
1. n ← length(List)

2. Jika n ≤ 1 maka
   return

3. Jika maxDepth = 0 maka
   HeapSort(List)
   return

4. p ← Partition(List)

5. maxDepth ← maxDepth - 1

6. IntroSort(List kiri, maxDepth)
   dengan List kiri = List[0 : p-1]

7. IntroSort(List kanan, maxDepth)
   dengan List kanan = List[p+1 : n-1]

8. return

Program introsort dapat dilihat di link https://colab.research.google.com/drive/1mGdDfipZXbuh7iEJXtbyurMrb0fGnZu3?usp=sharing.

10.17 Pigeonhole Sort

Pigeonhole Sort merupakan algoritma pengurutan yang memanfaatkan distribusi data melalui pengelompokan elemen ke sejumlah wadah (pigeonhole) sesuai nilai atau rentangnya. Algoritma ini tergolong sederhana dan efektif untuk pengorganisasian data secara sekuensial. Secara konseptual, Pigeonhole Sort membentuk sekumpulan wadah yang merepresentasikan nilai unik atau interval tertentu, lalu menempatkan setiap elemen ke wadah yang sesuai melalui fungsi pemetaan. Setelah seluruh elemen terkelompok, isi tiap wadah digabungkan kembali sehingga menghasilkan urutan data secara global. Mekanisme pemetaan tersebut dapat

dirancang secara sederhana, misalnya langsung berdasarkan nilai elemen, atau disesuaikan dengan karakteristik tipe data yang diolah. Dengan demikian, Pigeonhole Sort tidak terutama mengandalkan perbandingan langsung antarelemen, tetapi menekankan klasifikasi elemen ke kelompok yang terstruktur (Rohith et al., 2024).

Karakteristik utama Pigeonhole Sort ialah efektivitasnya terhadap dataset dengan rentang nilai terbatas. Algoritma ini sangat sesuai digunakan ketika jumlah rentang nilai tidak jauh melebihi jumlah elemen yang akan diurutkan. Dalam kondisi tersebut, setiap elemen dapat segera ditempatkan ke wadah yang tepat tanpa memerlukan perbandingan berulang yang kompleks, sehingga kinerjanya menjadi sangat baik. Algoritma ini juga bermanfaat untuk distribusi elemen yang tidak merata, tetapi masih berada di rentang nilai yang sempit, terutama ketika nilai tertentu muncul lebih sering daripada nilai lainnya. Ditinjau dari kompleksitas, Pigeonhole Sort dapat mencapai waktu linear dalam kondisi terbaik, yaitu ketika rentang nilai relatif dekat dengan jumlah elemen. Namun, efisiensinya menurun jika rentang nilai sangat besar karena kebutuhan wadah kosong meningkat dan penggunaan memori menjadi kurang hemat. Oleh sebab itu, meskipun efisien untuk karakteristik data tertentu, Pigeonhole Sort tidak selalu menjadi pilihan optimal untuk semua jenis dataset. Keunggulan utamanya terletak dalam logika yang sederhana, kemudahan implementasi, dan potensi pengembangannya untuk komputasi paralel.

Langkah-langkah algoritma pigeonhole sort yaitu:

1. **Menentukan nilai minimum dan maksimum data**
   Algoritma terlebih dahulu mencari nilai terkecil dan nilai terbesar dari seluruh elemen dalam array input.
2. **Menghitung jumlah pigeonhole**
   Setelah nilai minimum dan maksimum diperoleh, jumlah pigeonhole ditentukan dengan rumus:
   Jumlah pigeonhole = nilai maksimum − nilai minimum + 1
3. **Membuat pigeonhole kosong**
   Algoritma menyiapkan sejumlah wadah kosong sesuai jumlah pigeonhole yang telah dihitung.
4. **Menginisialisasi array pigeonhole**
   Seluruh wadah disusun dalam bentuk array atau list kosong agar siap menampung elemen dari data masukan.
5. **Menelusuri setiap elemen dalam array input**
   Setiap elemen diproses satu per satu untuk ditentukan wadah yang sesuai.
6. **Menghitung indeks pigeonhole untuk setiap elemen**
   Indeks pigeonhole ditentukan berdasarkan nilai elemen. Untuk data bilangan bulat, rumus yang umum digunakan adalah: **index = elemen − nilai minimum**
7. **Menempatkan elemen ke pigeonhole yang sesuai**
   Setelah indeks diperoleh, elemen dimasukkan ke pigeonhole yang bersesuaian.
8. **Mengulangi distribusi hingga seluruh elemen selesai ditempatkan**
   Proses perhitungan indeks dan penempatan elemen dilakukan terus sampai semua elemen selesai didistribusikan.
9. **Menggabungkan isi seluruh pigeonhole**

```
Algoritma PigeonholeSort(A):
Input  : A = array bilangan bulat
Output : SortedArray = array yang sudah terurut

1. min ← nilai terkecil dalam A
2. max ← nilai terbesar dalam A

3. jumlahPigeonhole ← max - min + 1

4. Buat array Pigeonholes[0 ... jumlahPigeonhole - 1]
5. Untuk i ← 0 sampai jumlahPigeonhole - 1, lakukan:
     Pigeonholes[i] ← list kosong

6. Untuk setiap elemen x dalam A, lakukan:
     a. index ← x - min
     b. Tambahkan x ke Pigeonholes[index]

7. SortedArray ← array kosong

8. Untuk i ← 0 sampai jumlahPigeonhole - 1, lakukan:
     Untuk setiap elemen x dalam Pigeonholes[i], lakukan:
         Tambahkan x ke SortedArray

9. Return SortedArray
```

Setelah semua elemen terkelompok, isi seluruh pigeonhole digabungkan kembali menjadi satu array.

10. Menyusun hasil gabungan sebagai array terurut

Penggabungan dilakukan sesuai urutan pigeonhole dari indeks terkecil hingga terbesar sehingga terbentuk susunan elemen yang terurut.

11. Menghasilkan array hasil sorting

Array gabungan tersebut dikembalikan sebagai hasil akhir pengurutan.

Program algoritma pigeonhole dapat dilihat di link https://colab.research.google.com/drive/1fRAqgMnGKSeXUuqExcT0y-mBfJrV5CEv?usp=sharing.

Gambar 10.16 memperlihatkan ilustrasi kerja algoritma Pigeonhole Sort secara bertahap menggunakan data awal 4, 3, 6, 1, 2, 7, 5. Langkah pertama adalah mengidentifikasi nilai minimum dan nilai maksimum dari array, yaitu 1 sebagai nilai terkecil dan 7 sebagai nilai terbesar. Berdasarkan kedua nilai tersebut, rentang data dihitung dengan rumus range = max − min + 1, sehingga diperoleh 7 − 1 + 1 = 7. Hasil perhitungan ini menunjukkan bahwa algoritma harus menyediakan 7 buah pigeonhole atau wadah. Setiap wadah merepresentasikan satu posisi nilai dalam rentang data, yaitu dari nilai 1 sampai 7. Setelah wadah disiapkan, setiap elemen terhadap array asli dipetakan ke indeks pigeonhole menggunakan rumus array[i] − min. Karena nilai minimum adalah 1, maka setiap elemen dikurangi 1 untuk menentukan letak wadahnya. Dengan demikian, elemen 4 ditempatkan di pigeonhole ke-3, elemen 3 di pigeonhole ke-2, elemen 6 di pigeonhole ke-5, elemen 1 di pigeonhole ke-0, elemen 2 di pigeonhole ke-1, elemen 7 di pigeonhole ke-6, dan elemen 5 di pigeonhole ke-4.

#1. Menentukan nilai minimum dan maksimum

```
Fungsi FindMinMax(A)
Input  : A = array bilangan bulat
Output : min, max

1. min ← A[0]
2. max ← A[0]

3. Untuk i ← 1 sampai panjang(A) - 1, lakukan:
   a. Jika A[i] < min, maka:
        min ← A[i]
   b. Jika A[i] > max, maka:
        max ← A[i]

4. Return min, max
```

#2. Membuat pigeonhole kosong

```
Fungsi CreateEmptyPigeonholes(jumlahPigeonhole)
Input  : jumlahPigeonhole
Output : Pigeonholes

1. Buat array Pigeonholes[0 ... jumlahPigeonhole - 1]
2. Untuk i ← 0 sampai jumlahPigeonhole - 1, lakukan:
     Pigeonholes[i] ← list kosong
3. Return Pigeonholes
```

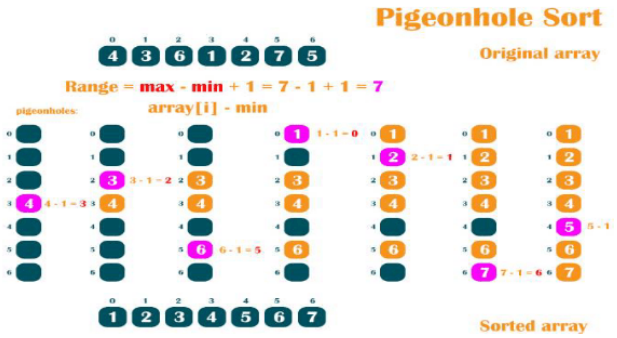
#3. Menghitung indeks pigeonhole

```
Fungsi CalculatePigeonholeIndex(element, min)
Input  : element, min
Output : index

1. index ← element - min
2. Return index
```

#4. Menggabungkan isi seluruh pigeonhole

```
Fungsi ConcatenatePigeonholes(Pigeonholes)
Input  : Pigeonholes
Output : SortedArray

1. SortedArray ← array kosong
2. Untuk i ← 0 sampai panjang(Pigeonholes) - 1,
lakukan:
     Untuk setiap elemen x dalam Pigeonholes[i],
lakukan:
          Tambahkan x ke SortedArray
3. Return SortedArray
```

Ilustrasi tersebut menunjukkan bahwa setelah seluruh elemen dimasukkan ke pigeonhole yang sesuai, wadah-wadah tersebut akan terisi mengikuti urutan nilai data. Karena data yang digunakan merupakan bilangan bulat berurutan tanpa duplikasi, setiap pigeonhole akhirnya berisi tepat satu elemen. Selanjutnya, algoritma membaca kembali isi pigeonhole mulai dari indeks terkecil hingga terbesar. Proses pembacaan berurutan inilah yang menghasilkan array terurut, yaitu 1, 2, 3, 4, 5, 6, 7. Melalui contoh ini, tampak bahwa Pigeonhole Sort tidak menekankan perbandingan langsung antar elemen seperti halnya banyak algoritma pengurutan lain, melainkan



Gambar 10.16: Contoh Algoritma Pigenhole Sort

memanfaatkan pengelompokan elemen berdasarkan rentang nilainya. Oleh karena itu, algoritma ini sangat efektif ketika data memiliki rentang nilai yang relatif sempit dan terdefinisi dengan jelas. Selain itu, juga sangat bergantung terhadap hubungan antara jumlah elemen dan rentang nilainya, karena jumlah pigeonhole ditentukan oleh selisih nilai maksimum dan minimum dalam data.

10.18 Smoothsort

Smoothsort merupakan algoritma pengurutan *in situ*, yaitu metode pengurutan yang bekerja langsung di dalam array tanpa memerlukan alokasi memori tambahan dalam jumlah besar. Algoritma ini memiliki kompleksitas waktu O(N log N) dalam kondisi terburuk, tetapi dapat mencapai O(N) dalam kondisi terbaik. Perubahan kinerja tersebut berlangsung secara bertahap sesuai tingkat keterurutan awal data, sehingga performanya bersifat adaptif. Karakteristik inilah yang menjadikan Smoothsort berbeda dari heapsort. Jika heapsort cenderung memperlakukan semua data dengan cara yang sama tanpa memanfaatkan keteraturan awal, Smoothsort justru dirancang untuk memperoleh keuntungan dari data yang sudah terurut atau hampir terurut (Dijkstra, 1981).

Secara konseptual, Smoothsort dapat dipahami sebagai pengembangan dari heapsort dengan struktur yang lebih adaptif terhadap susunan awal data. Setelah tahap persiapan, algoritma ini membentuk urutan terurut dari kanan ke kiri. Selama proses berlangsung, bagian akhir array (suffix) dipertahankan dalam keadaan terurut, sedangkan bagian awal (prefix) yang belum selesai diproses direpresentasikan sebagai traversal pascaurutan (postorder traversal) dari struktur pohon tertentu. Susunan ini penting karena menjamin bahwa elemen paling kanan dalam prefix yang belum terurut merupakan elemen maksimum, sehingga elemen tersebut dapat dipindahkan ke bagian terurut tanpa mengganggu keteraturan yang telah terbentuk sebelumnya.

Untuk merepresentasikan prefix yang belum selesai diproses, Smoothsort menggunakan konsep stretch dan standard concatenation. Stretch adalah subsekuens berurutan yang dipandang sebagai traversal pascaurutan dari suatu subpohon biner. Prefix yang belum terurut tidak selalu dianggap sebagai satu pohon besar, melainkan sebagai gabungan beberapa stretch dengan ukuran tertentu. Ukuran-ukuran tersebut ditentukan berdasarkan bilangan Leonardo. Pemilihan bilangan ini memungkinkan setiap simpul induk memiliki dua anak, sehingga proses pemeliharaan struktur menjadi lebih teratur dan efisien. Dengan demikian, Smoothsort tidak membangun pohon secara eksplisit seperti struktur simpul terhubung, melainkan merepresentasikan bentuknya secara ringkas melalui informasi panjang dan posisi.

Cara kerja Smoothsort mencakup dua gagasan utama, yaitu membangun struktur heap yang adaptif dan kemudian menguranginya kembali sambil menghasilkan urutan data yang terurut. Dalam fase pembentukan, prefix yang belum terurut disusun menjadi gabungan beberapa stretch yang dilacak melalui parameter internal (p,b,c). Parameter tersebut menunjukkan bentuk standard concatenation dan ukuran stretch yang sedang aktif. Ketika sebuah elemen baru dimasukkan, algoritma dapat menambahkan stretch baru atau menggabungkan stretch yang sudah ada sesuai pola tertentu. Perubahan bentuk struktur ini dikendalikan melalui operasi up dan down,

yang memperbarui pasangan panjang Leonardo (b,c). Dengan pendekatan tersebut, pengelolaan struktur dapat dilakukan secara padat tanpa harus membangun representasi pohon secara eksplisit.

Agar struktur yang terbentuk tetap valid, setiap stretch dibedakan ke dalam dua keadaan, yaitu trusty dan dubious. Sebuah stretch disebut trusty apabila akar dari kedua sub-stretch di bawahnya tidak melebihi akar stretch tersebut. Dalam keadaan ini, akar stretch menjadi elemen maksimum dari keseluruhan stretch. Sebaliknya, stretch yang belum memenuhi sifat tersebut disebut dubious. Untuk mengubah dubious stretch menjadi trusty stretch, digunakan operasi sift. Operasi ini bekerja dengan membandingkan akar terhadap anak terbesarnya. Jika anak tersebut lebih besar, keduanya ditukar, lalu proses dilanjutkan ke bawah sampai sifat heap kembali terpenuhi. Dengan demikian, sift berfungsi menjaga keteraturan lokal di dalam satu stretch.

Selain menjaga keteraturan di dalam masing-masing stretch, Smoothsort juga memelihara keteraturan antarakar stretch. Akar-akar stretch dalam standard concatenation harus tersusun menaik dari kiri ke kanan. Invarian ini menjamin bahwa elemen paling kanan dari prefix yang belum terurut benar-benar merupakan elemen maksimum global dari bagian tersebut. Untuk mempertahankan kondisi ini digunakan operasi trinkle. Secara umum, trinkle menyerupai sift, tetapi cakupannya lebih luas karena tidak hanya memulihkan heap di dalam satu stretch, melainkan juga menyesuaikan posisi akar terhadap urutan akar-akar stretch yang telah terbentuk sebelumnya. Dalam keadaan tertentu, digunakan pula operasi semitrinkle sebagai bentuk perbaikan parsial yang lebih efisien dibandingkan menjalankan trinkle penuh berulang kali.

Jika diringkas secara prosedural, Smoothsort bekerja dengan memindai array dan membangun kumpulan stretch berbasis bilangan Leonardo yang merepresentasikan prefix yang belum terurut. Selama proses pembentukan, setiap stretch dijaga agar sesegera mungkin memenuhi sifat trusty melalui operasi sift atau trinkle. Setelah struktur terbentuk, elemen paling kanan dari prefix yang belum terurut dikenali sebagai elemen maksimum. Elemen ini kemudian dipindahkan ke suffix yang telah terurut, sedangkan ukuran prefix dikurangi satu. Pengurangan tersebut biasanya memecah satu pohon menjadi beberapa subpohon yang lebih kecil, sehingga struktur harus disusun kembali melalui penggabungan atau perbaikan stretch agar seluruh invarian tetap terpenuhi. Siklus ini terus berulang hingga seluruh elemen selesai diproses dan array menjadi terurut menaik.

Keunggulan utama Smoothsort yaitu sifat adaptifnya. Algoritma ini tetap memiliki jaminan kompleksitas O(NlogN) dalam kasus terburuk, tetapi mampu menunjukkan performa yang jauh lebih baik ketika data telah terurut atau hampir terurut. Oleh karena itu, Smoothsort dapat dipandang sebagai varian heapsort yang lebih peka terhadap struktur awal data. Namun, kelebihan tersebut diimbangi oleh meningkatnya kerumitan representasi internal dan pengelolaan invarian. Bentuk struktur harus dilacak dengan cermat melalui parameter (p,b,c), serta dipelihara melalui operasi up, down, sift, trinkle, dan semitrinkle. Meskipun lebih kompleks dibandingkan heapsort biasa, Smoothsort menawarkan keseimbangan antara efisiensi teoretis dan kemampuan beradaptasi terhadap kondisi awal data.

Secara ringkas, Smoothsort merupakan algoritma pengurutan adaptif berbasis himpunan pohon dengan ukuran yang mengikuti bilangan Leonardo. Algoritma ini membangun struktur heap dari kanan ke kiri, menjaga validitas tiap pohon melalui sift, mempertahankan urutan antarakar melalui trinkle, lalu secara bertahap mengeluarkan elemen maksimum hingga seluruh array terurut. Dengan sifat tersebut, Smoothsort tetap aman dalam kasus terburuk, tetapi mampu bekerja lebih efisien ketika data telah memiliki tingkat keterurutan tertentu.

Algoritma Smoothsort(A):
Input : A = array data yang akan diurutkan
Output : A = array terurut menaik

Fungsi Up(b, c)
1. temp ← b
2. b ← b + c + 1
3. c ← temp
4. Return b, c

Fungsi Down(b, c)
1. temp ← c
2. c ← b - c - 1
3. b ← temp
4. Return b, c

Prosedur Sift(A, r, b, c)
1. Selama stretch berakar di r belum memenuhi sifat heap, lakukan:
   a. Tentukan anak terbesar dari stretch tersebut
   b. Jika A[r] ≥ nilai anak terbesar, maka:
      keluar dari perulangan
   c. Tukar A[r] dengan anak terbesar
   d. Pindahkan r ke posisi anak terbesar
   e. Perbarui ukuran stretch dengan operasi Down sesuai sub-stretch yang dipilih

Prosedur Trinkle(A, r, p, b, c)
1. Selama akar stretch di posisi r melanggar urutan terhadap akar-akar stretch sebelumnya, lakukan:
   a. Tentukan stretch pendahulu yang relevan
   b. Jika A[r] sudah berada di posisi yang benar, maka:
      keluar dari perulangan
   c. Tukar A[r] dengan akar stretch pendahulu yang lebih besar
   d. Perbarui parameter (p, b, c) agar sesuai dengan posisi baru
2. Setelah perpindahan selesai, panggil Sift(A, r, b, c) untuk memulihkan sifat heap lokal

Algoritma Utama Smoothsort(A)
1. n ← panjang(A)
2. Jika n ≤ 1, maka:
   Return A
3. p ← 1
4. b ← 1
5. c ← 1
6. q ← 0

7. // Fase 1: membangun struktur heap adaptif

8. Selama q < n, lakukan:
   a. Tambahkan elemen A[q] ke struktur aktif
   b. Tentukan apakah elemen tersebut:
      - membentuk stretch baru, atau
      - digabungkan dengan stretch yang sudah ada
   c. Perbarui bentuk standard concatenation dengan p
   d. Perbarui pasangan ukuran stretch (b, c) menggunakan Up atau Down
   e. Jika keteraturan lokal stretch terganggu, lakukan:
      Sift(A, q, b, c)
   f. Jika akar stretch juga harus disesuaikan terhadap urutan akar stretch lain, lakukan:
      Trinkle(A, q, p, b, c)
   g. q ← q + 1

9. // Fase 2: mengurangi struktur sambil membentuk array terurut
10. q ← n - 1
11. Selama q > 0, lakukan:
   a. Anggap A[q] sebagai elemen maksimum dari prefix aktif,
      sehingga elemen tersebut sudah berada di posisi akhirnya
   b. Jika stretch aktif berukuran 1, maka:
      pindah ke stretch sebelumnya
   c. Jika stretch aktif berukuran lebih dari 1, maka:
      - pecah stretch aktif menjadi dua sub-stretch
      - perbarui (b, c) dengan operasi Down
      - terapkan Trinkle atau Sift dalam sub-stretch yang muncul
        agar sifat heap dan urutan antarakar tetap valid
   d. Perbarui parameter p agar mencerminkan struktur baru
   e. q ← q - 1

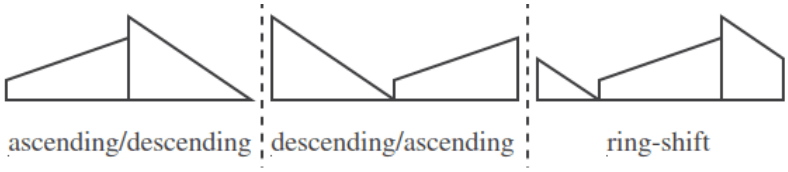
12. Return A

Program algoritma smoothsort dapat dilihat di link https://colab.research.google.com/drive/1RjhvfCSGbL-iVVNOvUjTme98Px9z-gTE?usp=sharing.

10.19 Bitonic Sort

Bitonic Sort adalah algoritma pengurutan paralel yang menggunakan pola perbandingan tetap dan teratur. Urutan, arah, serta jumlah perbandingannya sudah ditentukan sejak awal, sehingga proses pengurutan tidak bergantung nilai data yang diurutkan. Karena sifatnya terstruktur, algoritma ini cocok digunakan untuk sistem paralel, seperti GPU dan sistem komputasi dengan banyak prosesor. Oleh sebab itu, Bitonic Sort banyak digunakan sebagai dasar untuk pengembangan metode pengurutan paralel yang lebih lanjut (Peters et al., 2012).

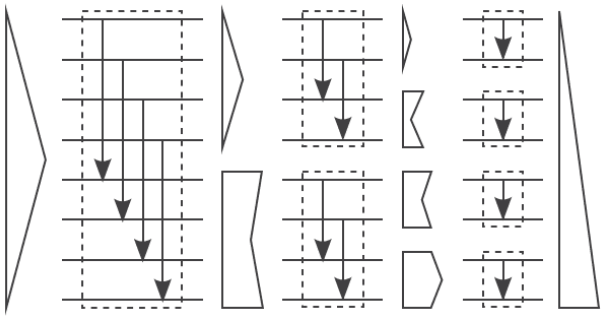
Secara konsep, bitonic sort menggunakan bitonic sequence, yaitu barisan data yang terdiri atas dua sub



Gambar 10.18: Bitonic sequences (Peters et al., 2012)

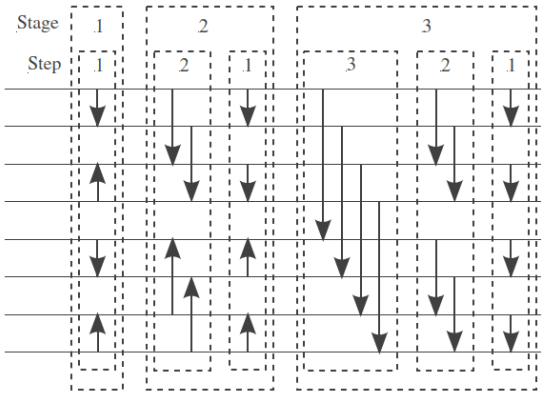
sequence dengan urutan berlawanan, satu ascending dan satu descending. Barisan ini juga dapat berupa hasil pergeseran dari susunan tersebut. Bitonic sequence disajikan di Gambar 10.17 yang memperlihatkan tiga bentuk barisan bitonik. Bagian pertama, barisan data mula-mula naik lalu turun (ascending/descending). Artinya, nilai elemen meningkat hingga mencapai titik puncak, kemudian menurun. Bentuk ini merupakan contoh paling umum dari bitonic sequence. Bagian kedua, barisan data mula-mula turun lalu naik (descending/ascending). Bentuk ini juga tetap termasuk barisan bitonik, karena masih terdiri atas dua bagian berurutan dengan arah yang berlawanan. Bagian ketiga, ditunjukkan bentuk ring-shift atau pergeseran melingkar. Dalam bentuk ini, susunan data sebenarnya masih berasal dari pola naik lalu turun atau turun lalu naik, tetapi titik awal pembacaannya digeser. Akibatnya, puncak atau lembah tidak selalu berada di tengah, melainkan dapat tampak berada di salah satu sisi. Meskipun tampilannya berbeda, susunan tersebut tetap termasuk bitonic sequence. Karena telah memiliki pola urutan tertentu, barisan bitonik dapat dipisahkan menjadi dua bagian, yaitu bagian yang memuat elemen bernilai kecil dan bagian yang memuat elemen bernilai besar. Kedua bagian itu tetap memiliki sifat bitonik, sehingga proses yang sama dapat diulang secara rekursif sampai seluruh data terurut. Proses ini menjadi dasar bitonic merge.

Cara kerja bitonic sort mengikuti pendekatan *divide-and-conquer* yang disajikan di Gambar 10.18. Ilustrasi



Gambar 10.17: *Divide-and-Conquer* Algoritma Bitonic Sort (Peters et al., 2012)

tersebut menunjukkan bahwa sekumpulan data mula-mula diproses melalui jaringan perbandingan (compare-exchange network) yang tersusun secara bertahap. Tahap awal, elemen-elemen dibandingkan dalam kelompok yang masih besar untuk memisahkan nilai yang relatif lebih kecil dan lebih besar. Hasil pemisahan ini kemudian dibagi lagi menjadi dua subbagian yang lebih kecil, dan masing-masing subbagian diproses kembali dengan pola perbandingan yang sama. Proses pembagian berulang tersebut tampak dari susunan jaringan yang semakin



Gambar 10.19: Algoritma Bitonic Sort (Peters et al., 2012)

menyempit ke arah kanan, yang menandakan bahwa masalah pengurutan besar dipecah menjadi submasalah yang lebih sederhana hingga mencapai unit-unit yang lebih kecil dan lebih mudah diurutkan. Secara konseptual, gambar tersebut menegaskan bahwa bitonic sort tidak mengurutkan data secara langsung dalam satu langkah, melainkan melalui serangkaian proses rekursif yang terstruktur. Setiap tahap berfungsi membentuk dan menggabungkan bitonic sequence, yaitu susunan data yang memiliki pola naik dan turun, sehingga dapat dipisahkan menjadi bagian bernilai lebih kecil dan bagian bernilai lebih besar. Setelah itu, kedua bagian tersebut diolah kembali dengan mekanisme yang sama sampai diperoleh urutan akhir yang teratur. Dengan demikian, gambar ini menunjukkan bahwa efisiensi bitonic sort dalam lingkungan paralel berasal dari pola kerjanya yang tetap, sistematis, dan dapat dibagi ke dalam beberapa tahap pemrosesan yang saling independen. Struktur seperti ini menjadikan Bitonic Sort sangat sesuai untuk implementasi paralel, karena banyak operasi perbandingan dapat dilakukan secara bersamaan untuk setiap tingkat pembagian.

Gambar 10.19 menampilkan jaringan pengurutan bitonic sort yang tersusun ke dalam beberapa stage dan step. Setiap garis horizontal merepresentasikan aliran elemen data, sedangkan garis vertikal yang disertai panah menunjukkan operasi compare-exchange, yaitu proses membandingkan dua elemen dan menempatkan hasilnya sesuai arah pengurutan yang ditentukan. Pembagian ke dalam Stage 1, Stage 2, dan Stage 3 menunjukkan tingkat utama pengurutan, sementara step di dalam setiap stage menunjukkan urutan operasi perbandingan yang harus dilakukan. Tahap awal, elemen dibandingkan dalam pasangan-pasangan sederhana. Tahap berikutnya menggabungkan hasil pengurutan tersebut ke dalam kelompok yang lebih besar, kemudian memprosesnya kembali melalui pola perbandingan yang terstruktur. Dengan demikian, jaringan ini memperlihatkan bahwa Bitonic Sort bekerja secara bertahap dari susunan kecil menuju susunan yang semakin besar hingga seluruh elemen berada dalam urutan yang benar. Bitonic sort menerapkan

Algoritma BitonicSort(A):
Input : A = array data yang akan diurutkan
Output : A = array terurut menaik

Konstanta:
ASC = 1
DESC = 0

Prosedur CompareExchange(A, i, j, direction)
1. Jika direction = ASC dan A[i] > A[j], maka:
   tukar A[i] dengan A[j]
2. Jika direction = DESC dan A[i] < A[j], maka:
   tukar A[i] dengan A[j]

Prosedur BitonicMerge(A, low, count, direction)
1. Jika count > 1, maka:
   a. k ← count / 2
   b. Untuk i ← low sampai low + k - 1, lakukan:
      CompareExchange(A, i, i + k, direction)
   c. BitonicMerge(A, low, k, direction)
   d. BitonicMerge(A, low + k, k, direction)

Prosedur BitonicSortRec(A, low, count, direction)
1. Jika count > 1, maka:
   a. k ← count / 2
   b. BitonicSortRec(A, low, k, ASC)
   c. BitonicSortRec(A, low + k, k, DESC)
   d. BitonicMerge(A, low, count, direction)

Algoritma Utama
1. n ← panjang(A)
2. BitonicSortRec(A, 0, n, ASC)
3. Return A

pola pengurutan yang tetap dan deterministik. Setiap stage, data diproses melalui serangkaian langkah yang membentuk dan menggabungkan bitonic sequence, yaitu susunan data yang terdiri atas dua bagian dengan arah urutan berlawanan. Panah ke atas dan ke bawah mengindikasikan arah perpindahan hasil perbandingan, sehingga elemen bernilai lebih kecil dan lebih besar diposisikan ke jalur yang sesuai. Seiring bertambahnya stage, ukuran subsekuens yang diproses juga semakin besar, sampai akhirnya seluruh data tersusun terurut. Struktur jaringan seperti ini sangat mendukung komputasi paralel karena operasi compare-exchange dalam satu step dapat dijalankan secara serentak. Keunggulan utama bitonic sort terletak yaitu pola kerja yang memudahkan pembagian tugas ke banyak unit pemroses.

Ditinjau dari kompleksitas, bitonic sort memiliki biaya komputasi sebesar O (N $\log^2$ N). Kompleksitas tersebut muncul karena setiap langkah terdapat sekitar N/2 operasi compare-exchange, sedangkan jumlah total langkah mengikuti akumulasi bertingkat sepanjang log N tahap. Walaupun belum seoptimal algoritma pengurutan berbasis perbandingan terbaik yang dapat mencapai O (N log N), bitonic sort tetap sangat menarik dalam komputasi paralel karena prosedurnya bersifat deterministik, teratur, dan mudah dipetakan ke banyak unit pemroses secara serentak. Dalam praktiknya, hal ini membuat bitonic sort sangat sesuai untuk GPU, meskipun CPU algoritma seperti quick sort sering kali menunjukkan performa yang lebih unggul.

Program bitonic sort dapat dilihat di link https://colab.research.google.com/drive/13E2NvCGALpBt3Ei62-i2VLeei2JcHAbT?usp=sharing.

10.20 Pancake Sort

Pancake Sorting merupakan permasalahan pengurutan yang memodelkan suatu tumpukan pancake dengan ukuran berbeda, dengan ketentuan bahwa satu-satunya operasi yang diperbolehkan adalah menyisipkan spatula di bawah pancake ke-i dari atas, kemudian membalik seluruh bagian tumpukan di atas spatula tersebut. Dalam representasi matematis, operasi ini dikenal sebagai prefix reversal, yaitu pembalikan urutan elemen dalam suatu prefiks permutasi. Tujuan pengurutan adalah menyusun pancake berdasarkan ukuran secara meningkat dari atas ke bawah. Jika π menyatakan suatu susunan pancake, maka $f(\pi)$ menyatakan jumlah minimum flip yang diperlukan untuk mengurutkan susunan tersebut, sedangkan $f(n)$ menyatakan nilai maksimum dari jumlah minimum flip untuk seluruh susunan berukuran n (Cibulka, 2011).

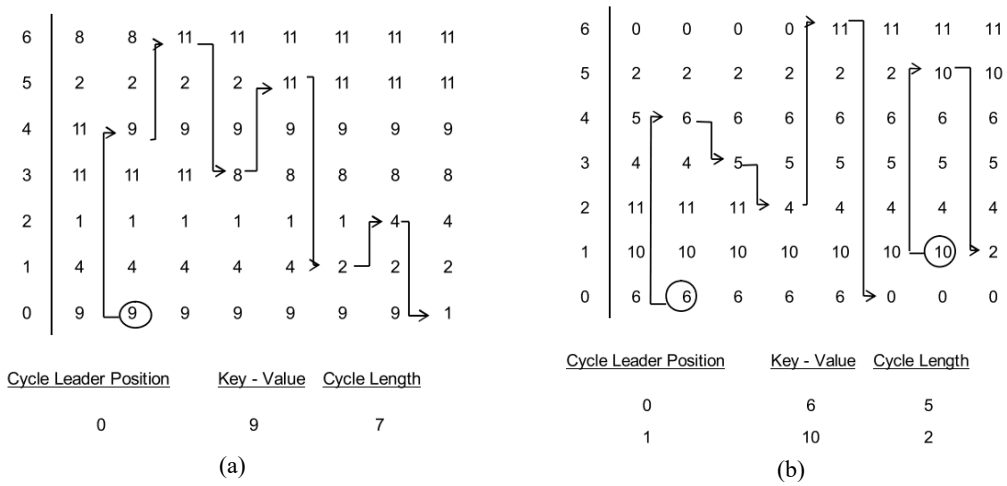
Pancake sorting bekerja dengan prinsip bahwa pengurutan hanya dapat dilakukan dari bagian atas tumpukan dan tidak melalui pertukaran elemen secara bebas. Oleh karena itu, setiap langkah selalu berbentuk prefix reversal. Dalam analisis algoritmik, dua pancake dikatakan membentuk adjacency apabila ukurannya berurutan. Susunan pancake yang telah membentuk adjacency dapat dipandang sebagai satu unit melalui proses contraction, sehingga ukuran masalah berkurang dan proses analisis menjadi lebih sederhana. Selain itu, dikenal pula konsep block, yaitu subsekuens maksimal yang elemennya sudah tersusun bersebelahan secara benar. Konsep adjacency, block, dan contraction sangat penting karena memberikan dasar struktural untuk memahami bagaimana sebuah susunan besar dapat direduksi secara bertahap menjadi susunan yang lebih sederhana.

Cara kerja pancake sorting dapat dipahami sebagai proses membentuk adjacency baru secara bertahap hingga seluruh tumpukan menjadi terurut. Salah satu gagasan utamanya adalah bahwa pancake berukuran terbesar dapat dipindahkan ke posisi paling bawah dengan paling banyak dua flip. Setelah elemen terbesar berada di posisi akhirnya, elemen tersebut tidak perlu diproses kembali. Dengan demikian, persoalan berukuran n direduksi menjadi persoalan berukuran n−1. Pola ini menunjukkan bahwa pancake sorting memiliki sifat recursive, karena setiap langkah menyelesaikan satu bagian masalah, lalu menerapkan prosedur yang sama terhadap sisa tumpukan di atasnya. Inti mekanismenya bukan sekadar membalik tumpukan secara acak, melainkan menempatkan elemen terbesar ke posisi akhir secara sistematis, kemudian mengulangi proses tersebut sampai seluruh susunan selesai diurutkan.

Dari sudut pandang analisis, pancake sorting juga dapat dipelajari melalui pembentukan adjacency sebagai ukuran kemajuan. Semakin banyak adjacency yang berhasil dibentuk, semakin dekat susunan tersebut terhadap keadaan terurut. Berdasarkan gagasan ini, lower bound jumlah flip dapat ditaksir dari banyaknya adjacency yang belum terbentuk. Jika dalam suatu susunan tidak tersedia urutan flip yang selalu menghasilkan adjacency baru untuk setiap langkah, maka jumlah adjacency

Algoritma PancakeSort(A):
Input : A = array data yang akan diurutkan
Output : A = array terurut menaik

```
Prosedur Flip(A, k)
1. left ← 0
2. right ← k
3. Selama left < right, lakukan:
    a. tukar A[left] dengan A[right]
    b. left ← left + 1
    c. right ← right - 1

Fungsi FindMaxIndex(A, n)
1. maxIndex ← 0
2. Untuk i ← 1 sampai n - 1, lakukan:
    a. jika A[i] > A[maxIndex], maka:
        maxIndex ← i
3. Return maxIndex

Prosedur PancakeSortRec(A, n)
1. Jika n ≤ 1, maka:
    Return

2. maxIndex ← FindMaxIndex(A, n)

3. Jika maxIndex = n - 1, maka:
    PancakeSortRec(A, n - 1)
    Return

4. Jika maxIndex ≠ 0, maka:
    Flip(A, maxIndex)

5. Flip(A, n - 1)

6. PancakeSortRec(A, n - 1)

Algoritma Utama
1. n ← panjang(A)
2. PancakeSortRec(A, n)
3. Return A
```



Gambar 10.20: Permutasi a) Satu Siklus b) Dua Siklus

yang masih harus dibentuk menjadi dasar untuk memperkirakan batas bawah jumlah operasi. Dengan demikian, efisiensi Pancake Sorting tidak hanya ditentukan oleh jumlah flip yang dilakukan, tetapi juga oleh kemampuan setiap flip dalam menciptakan struktur lokal yang semakin mendekati urutan akhir.

Secara ringkas, pancake sorting adalah algoritma pengurutan yang mengandalkan flip sebagai satu-satunya operasi dasar. Prosesnya dilakukan dengan memindahkan pancake terbesar ke posisi akhir, membentuk adjacency baru, menerapkan contraction untuk bagian yang sudah benar, lalu mengulangi prosedur tersebut secara recursive hingga seluruh tumpukan terurut. Karakteristik ini menjadikan pancake sorting menarik, baik sebagai model teoretis dalam kajian prefix reversal, maupun sebagai contoh algoritma pengurutan yang menekankan reduksi masalah secara bertahap melalui operasi yang sangat terbatas.

Program algoritma pancake sort disajikan di link https://colab.research.google.com/drive/1nNZuDa0D8h9oO_HRBiXLB1sC9sul5JzB?usp=sharing.

10.21 Flashsort

Flash sort merupakan algoritma pengurutan yang dibangun atas prinsip in situ permutation, yaitu proses pengurutan yang berlangsung langsung di dalam array utama tanpa memerlukan array keluaran terpisah. Algoritma ini dirancang untuk mencapai kompleksitas waktu linear, yaitu O(N), dengan kebutuhan memori bantu yang relatif kecil, berupa sebuah vektor yang panjangnya sebanding dengan jumlah key yang berbeda. Terdapat dua tahap besar yaitu mengelompokkan elemen ke dalam sejumlah class melalui proses **classification**, kemudian menata ulang elemen-elemen tersebut dengan **cyclic permutation** di array yang sama. Efisiensi metode ini juga ditopang oleh prosedur khusus untuk menemukan cycle leader baru, sehingga perpindahan elemen dapat berlangsung secara terarah dan terkontrol (Neubert, 1997).

Flash sort memandang pengurutan sebagai proses membalik permutasi yang membentuk susunan tidak terurut. Karena sebuah permutasi umumnya tersusun atas beberapa siklus, bukan satu siklus tunggal, muncul dua persoalan utama dalam in situ permutation. Persoalan pertama adalah menentukan posisi tujuan setiap elemen selama siklus permutasi berlangsung. Persoalan kedua adalah menemukan cycle leader baru yang benar setelah suatu siklus selesai. Untuk mengatasi persoalan pertama, Flash sort menggunakan class pointer vector, yaitu vektor penunjuk kelas yang selalu mengarah ke posisi kosong terkini dalam suatu class. Persoalan kedua diselesaikan dengan memanfaatkan struktur yang terbentuk selama proses pengurutan berlangsung, sehingga pencarian cycle leader baru dapat dilakukan secara efisien.

Gambar 10.20 memperlihatkan konsep *permutation cycle* yang menjadi dasar penting dalam mekanisme in situ permutation untuk proses pengurutan. Kedua bagian gambar sama-sama menggunakan *Number of Strings* N=7, tetapi menunjukkan dua struktur permutasi yang berbeda. Bagian (a) menggambarkan permutasi yang

terdiri atas satu siklus, sedangkan bagian (b) menunjukkan permutasi yang terdiri atas dua siklus. Setiap angka dalam baris-baris vertikal merepresentasikan nilai atau key yang berpindah dari satu posisi ke posisi lain, sedangkan garis panah menunjukkan urutan perpindahan elemen selama proses permutasi berlangsung. Lingkaran yang ditandai di salah satu nilai menunjukkan cycle leader, yaitu elemen awal yang menjadi titik masuk untuk menelusuri seluruh siklus perpindahan.

Gambar (a) menunjukkan seluruh elemen yang terlibat membentuk satu siklus tunggal. Artinya, jika perpindahan dimulai dari posisi awal yang ditandai sebagai cycle leader position = 0 dengan key-value = 9, maka perpindahan elemen berlangsung berantai melalui seluruh posisi lain hingga akhirnya kembali ke posisi awal. Karena semua elemen tergabung dalam satu lintasan tertutup yang sama, panjang siklusnya adalah 7, sesuai dengan jumlah elemen yang terlibat. Situasi ini menunjukkan bahwa seluruh permutasi dapat diselesaikan hanya dengan menelusuri satu siklus penuh dari satu cycle leader saja. Secara konseptual, gambar ini menegaskan bahwa suatu permutasi dapat berbentuk sederhana apabila semua elemen saling terhubung dalam satu rangkaian perpindahan yang utuh.

Sebaliknya, gambar (b) menjelaskan bahwa permutasi tidak lagi membentuk satu siklus tunggal, melainkan terbagi menjadi dua siklus yang terpisah. Siklus pertama dimulai dari cycle leader position = 0 dengan key-value = 6 dan memiliki cycle length = 5. Siklus kedua dimulai dari cycle leader position = 1 dengan key-value = 10 dan memiliki cycle length = 2. Hal ini menunjukkan bahwa tidak semua elemen saling terhubung dalam satu lintasan perpindahan yang sama. Setelah siklus pertama selesai ditelusuri, masih terdapat elemen lain yang belum berpindah ke posisi akhirnya, sehingga diperlukan cycle leader baru untuk memulai siklus berikutnya. Dengan demikian, gambar ini menjelaskan bahwa suatu permutasi umumnya dapat tersusun atas beberapa siklus, bukan selalu satu siklus tunggal.

Algoritma flash sort menyimpan data dalam array A(i). sebuah vektor L dengan panjang M digunakan sebagai class pointer vector. Tahap awal algoritma adalah CLASSIFY, yaitu menghitung jumlah elemen yang masuk ke setiap class. Setelah proses ini selesai, setiap komponen L(k) tidak lagi sekadar menunjukkan frekuensi suatu class, melainkan jumlah kumulatif elemen dari class 0 hingga class k. Dengan demikian, L(M−1) selalu bernilai N−1, tanpa bergantung terhadap distribusi data di dalam array. Tahap klasifikasi ini sangat penting karena memberikan gambaran global mengenai penyebaran data sekaligus menentukan batas akhir setiap class di dalam array.

Setelah tahap classification selesai, pengurutan dilanjutkan melalui dua prosedur inti, yaitu LEADER dan PERMUTE. Prosedur PERMUTE bertugas memindahkan elemen ke class yang sesuai melalui cyclic permutation. Dalam proses ini, suatu posisi array disebut empty jika elemen aslinya belum digantikan elemen lain, dan disebut occupied jika posisi tersebut sudah diisi oleh elemen hasil perpindahan. Ketika suatu elemen, yang sering disebut sebagai FLASH, dipindahkan ke class yang benar, penunjuk kelas L(KEY(FLASH)) dikurangi satu, lalu langsung menunjuk ke posisi kosong berikutnya dalam class tersebut. Melalui mekanisme ini, setiap elemen secara bertahap ditempatkan ke wilayah class yang sesuai sampai satu siklus permutasi selesai.

Bagian yang sangat penting dalam Flash Sort adalah menentukan kapan sebuah siklus berakhir dan bagaimana menemukan cycle leader berikutnya. Suatu siklus dianggap selesai ketika penunjuk kelas dari class yang menyediakan cycle leader telah bergerak ke satu posisi di bawah batas bawah class tersebut. Kondisi ini dinyatakan dengan $L(KEY(A(j)))<j$. Setelah satu siklus berakhir, algoritma harus mencari cycle leader baru. Secara operasional, cycle leader adalah elemen yang berada di posisi kosong terendah, atau elemen A(j) terendah yang masih memenuhi syarat $L(KEY(A(j)))\geq j$. Cycle leader pertama terletak di A(0), sedangkan cycle leader berikutnya ditemukan dengan menaikkan indeks j hingga diperoleh elemen yang memenuhi kondisi tersebut. Dengan pola ini, setiap siklus baru dapat dimulai tanpa harus memeriksa seluruh array dari awal.

Jika diringkas, cara kerja Flash Sort berlangsung melalui empat tahap yang saling terkait. Mula-mula, elemen-elemen dikelompokkan ke dalam sejumlah class melalui classification. Berikutnya, vektor L digunakan untuk mengetahui rentang posisi yang semestinya ditempati oleh setiap class di dalam array. Setelah itu, algoritma menjalankan cyclic permutation untuk memindahkan setiap elemen ke class yang sesuai sambil terus

memperbarui penunjuk kelas. Jika satu siklus telah selesai, algoritma mencari cycle leader baru dan memulai siklus berikutnya. Proses tersebut diulang sampai seluruh elemen berada di wilayah kelas yang benar. Dengan demikian, inti mekanisme Flash Sort tidak bertumpu terhadap perbandingan langsung antarelemen, melainkan classification yang diikuti in situ permutation secara sistematis.

Untuk mempercepat pencarian cycle leader, diperkenalkan prosedur opsional bernama LEAP. Tanpa prosedur ini, cycle leader baru dicari dengan memeriksa kandidat elemen satu per satu mulai dari posisi tertentu. Kendalanya, vektor L menunjuk ke elemen tertinggi suatu class, bukan ke elemen terendahnya, sehingga tidak dapat langsung digunakan untuk menemukan cycle leader baru. Prosedur LEAP mengatasi keterbatasan tersebut dengan terlebih dahulu menaikkan indeks class k, bukan indeks posisi j, sampai ditemukan class yang berada tepat di bawah class yang memuat cycle leader saat ini. Dari titik tersebut, pencarian dilanjutkan dengan menaikkan j. Pendekatan ini secara nyata mengurangi jumlah langkah yang diperlukan untuk menemukan cycle leader baru.

Salah satu karakteristik menarik dari flash sort adalah bahwa setiap elemen dipindahkan tepat satu kali, tanpa bergantung nomor class maupun posisi akhirnya. Karena itu, elemen yang sejak awal sudah berada dekat dengan posisi yang benar tidak menimbulkan masalah khusus. Bahkan ketika hanya tersisa satu posisi kosong dalam suatu class dan elemen yang berkaitan menjadi cycle leader, perpindahannya dapat tereduksi menjadi elemen yang diambil lalu dikembalikan ke tempat semula. Meskipun terlihat seperti perpindahan yang sia-sia, penunjuk kelas dan penghitung perpindahan tetap diperbarui secara konsisten, sehingga algoritma tetap berjalan dengan benar.

Dari sisi kinerja, flash sort menunjukkan perilaku yang mendekati linear. Jumlah siklus biasanya relatif kecil, sehingga pengaruhnya terhadap waktu total pengurutan tidak terlalu besar. Faktor yang lebih menentukan justru adalah biaya pencarian cycle leader terakhir. Tanpa LEAP, biaya ini dapat mengambil porsi yang cukup besar dari total runtime. Namun, setelah LEAP digunakan, biaya pencarian cycle leader menurun tajam hingga hampir tidak berpengaruh terhadap waktu keseluruhan. Hasil ini menunjukkan bahwa efisiensi Flash Sort tidak hanya ditentukan oleh mekanisme klasifikasi dan permutasi, tetapi juga oleh strategi pencarian cycle leader yang tepat.

Flash sort juga dapat digeneralisasi dari contoh data sederhana menjadi bentuk yang lebih umum. Algoritma ini dapat diterapkan untuk mengurutkan string dengan panjang berapa pun, jumlah key berapa pun, serta urutan kolasi yang dapat dipilih secara independen untuk setiap kolom. Generalisasi tersebut memang menambah overhead akses data dibandingkan versi dasar, tetapi tetap mempertahankan struktur kerja yang sama, yaitu classification, cyclic permutation, dan pencarian cycle leader. Dengan demikian, Flash Sort tidak hanya relevan untuk data numerik sederhana, tetapi juga untuk data yang lebih kompleks sepanjang fungsi key dan pembentukan class dapat didefinisikan dengan jelas.

Jika dibandingkan dengan Counting Sort, Flash sort menunjukkan perbedaan penting dalam penggunaan memori. Counting sort memakai vektor penunjuk kelas juga, tetapi hasil pengurutan dituliskan ke array target terpisah, lalu disalin kembali ke array sumber. Flash sort tidak memerlukan array tambahan semacam itu karena seluruh proses berlangsung in place. Konsekuensinya, biaya tambahannya berpindah ke pencarian cycle leader. Walaupun demikian, biaya ini relatif kecil, terutama ketika $N > M$. Keuntungan yang diperoleh adalah penghematan memori yang signifikan, sehingga kapasitas data yang dapat diurutkan menjadi lebih besar.

Flash sort dapat dipahami sebagai algoritma pengurutan linear berbasis classification yang memanfaatkan class pointer vector, cycle leader, dan in situ permutation untuk menempatkan elemen ke wilayah yang benar secara efisien. Keunggulan utamanya yaitu kombinasi kompleksitas waktu linear, kebutuhan memori tambahan yang rendah, dan kemampuan bekerja langsung di dalam array asal. Mekanisme kerjanya menunjukkan bahwa proses pengurutan tidak selalu harus bertumpu terhadap perbandingan berulang antarelemen, tetapi juga dapat dibangun melalui klasifikasi global yang dilanjutkan dengan permutasi siklik yang terarah. Dalam konteks itu, flash sort menempati posisi penting sebagai contoh sorting by classification yang efisien sekaligus hemat memori.

Algoritma FlashSort(A, N, M):
Input : A = array data yang akan diurutkan
N = jumlah elemen
M = jumlah class
Output : A = array terurut

Variabel:
L[0..M-1] = class pointer vector
JJ = indeks cycle leader
NMove = jumlah elemen yang belum dipindahkan
K = indeks class
Fungsi KeyValue(x)
1. Tentukan key atau class dari elemen x
2. Return nilai key
Prosedur Classify(A, N, L, M)
1. Untuk k ← 0 sampai M-1, lakukan:
L[k] ← 0
2. Untuk i ← 0 sampai N-1, lakukan:
k ← KeyValue(A[i])
L[k] ← L[k] + 1
Prosedur BuildLVector(L, M)
1. Untuk k ← 1 sampai M-1, lakukan:
L[k] ← L[k] + L[k-1]
Prosedur Leap(A, L, M, JJ, K)
1. Jika JJ > 0, maka:
a. K ← K - 1
b. Selama K < M-1 dan L[K] < JJ, lakukan:
K ← K + 1
c. Jika K < M, maka:
JJ ← L[K]

Prosedur Leader(A, L, M, JJ, K)
1. Panggil Leap(A, L, M, JJ, K)
2. Ulangi:
a. JJ ← JJ + 1
sampai L[KeyValue(A[JJ])] ≥ JJ
3. Return JJ

Prosedur Permute(A, L, JJ, NMove)
1. FLASH ← A[JJ]
2. Ulangi:
a. K ← KeyValue(FLASH)
b. target ← L[K]
c. HOLD ← A[target]
d. A[target] ← FLASH
e. FLASH ← HOLD
f. L[K] ← L[K] - 1
g. NMove ← NMove - 1
sampai L[KeyValue(A[JJ])] < JJ

Algoritma Utama
1. Classify(A, N, L, M)
2. BuildLVector(L, M)
3. JJ ← -1
4. NMove ← N
5. K ← M - 1
6. Selama NMove > 0, lakukan:
a. JJ ← Leader(A, L, M, JJ, K)
b. Permute(A, L, JJ, NMove)
7. Return A

Latihan:
Soal 1:
Diketahui sebuah array data sebagai berikut: [72, 15, 94, 38, 61, 47, 29, 83, 56]

Buatkan program untuk mengurutkan data di atas secara ascending dan descending menggunakan algoritma cycle sort, gnome sort, timsort, introsort, pancake sort, pigeonhole sort, bitonic sort, smoothsort, dan flashsort.

Program algoritma flash sort disajikan di link https://colab.research.google.com/drive/16taq2Y8W0Rqi6Zq0lsDBB5jBI_Uh_g0M?usp=sharing.

Berikut pseudocode algoritma flash sort.

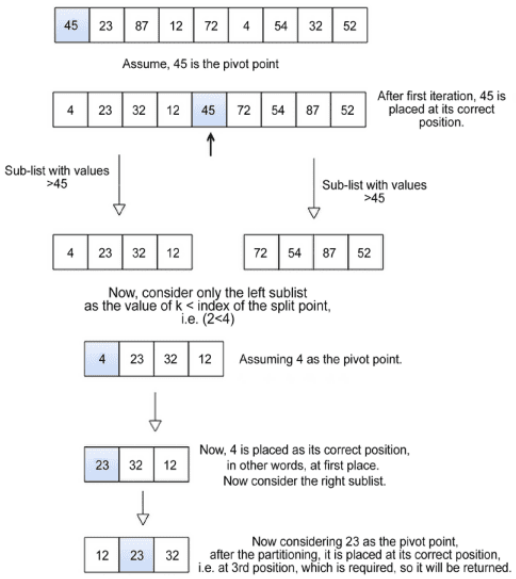
10.22 Randomize Selection atau Quick Selection

Randomized selection merupakan algoritma yang digunakan untuk memperoleh k^{th} smallest number, dan algoritma ini didasarkan algoritma quicksort. Algoritma randomized selection juga dikenal dengan nama quickselect. Dalam pembahasan tentang Sorting, quicksort algorithm telah dijelaskan sebagai algoritma yang efisien untuk mengurutkan unordered list of items. Secara ringkas, quicksort algorithm bekerja melalui tiga langkah utama, yaitu memilih sebuah pivot, melakukan partitioning terhadap unsorted list di sekitar pivot, lalu mengurutkan dua bagian hasil partitioned list secara recursively dengan mengulangi kedua langkah sebelumnya. Salah satu fakta yang sangat penting dalam quicksort adalah bahwa sesudah setiap langkah partitioning, index dari pivot tidak berubah, bahkan ketika list akhirnya menjadi terurut. Hal ini berarti bahwa sesudah setiap iterasi,

nilai pivot yang dipilih akan menempati posisi yang benar dalam list. Sifat inilah yang memperoleh k^{th} smallest number tanpa harus mengurutkan seluruh list secara lengkap (Agarwal, 2022).

Dengan memanfaatkan sifat tersebut, metode randomized selection atau algoritma quickselect dapat digunakan untuk menemukan k^{th} smallest element dari suatu list yang berisi n data items secara lebih efisien. Meskipun algoritma quick sort dasarnya dirancang untuk mengurutkan *unordered list of items*, algoritma ini memiliki mekanisme yang mempertahankan index elemen-elemen tertentu selama proses pengurutan berlangsung. Karena pivot selalu berakhir di posisi yang benar sesudah setiap partitioning step, maka pencarian i^{th}-smallest number dapat dilakukan hanya dengan memfokuskan proses bagian list yang relevan, tanpa perlu menyelesaikan pengurutan seluruh elemen. Oleh sebab itu, karena **randomized selection berlandaskan prinsip kerja quick sort algorithm, algoritma ini secara umum disebut quick select**.

Dalam algoritma quickselect, index dari pivot point dibandingkan dengan nilai k untuk memperoleh k^{th}



Gambar 10.21: Ilustrasi Algoritma Quick Selection **(Agarwal, 2022)**

elemen paling kecil dari data yang belum terurut yang diberikan. Terdapat tiga kemungkinan kondisi dalam algoritma ini, yaitu:

- Pertama, jika index dari pivot point lebih kecil daripada k, maka k^{th} nilai terkecil dipastikan berada di right-hand sublist dari pivot point, sehingga fungsi quickselect hanya dipanggil secara recursively dalam right sublist.
- Kedua, jika index dari pivot point lebih besar daripada k, maka k^{th} elemen terkecil pasti berada di sisi kiri pivot point, sehingga pencarian secara recursively hanya dilakukan di left sublist.
- Ketiga, jika index dari pivot point sama dengan k, maka hal itu menunjukkan bahwa k^{th} nilai paling kecil telah ditemukan dan nilai tersebut langsung dikembalikan.

Gambar 10.21 memperlihatkan ilustrasi Algoritma Quick Selection untuk menemukan elemen ke-3 terkecil dari sebuah list yang belum terurut, yaitu {45, 23, 87, 12, 72, 4, 54, 32, 52}. Proses diawali dengan memilih 45 sebagai pivot point. Sesudah dilakukan proses partitioning, elemen-elemen yang nilainya lebih kecil daripada 45 ditempatkan di sisi kiri, sedangkan elemen-elemen yang nilainya lebih besar ditempatkan di sisi kanan. Hasil partisi pertama menghasilkan susunan 4, 23, 32, 12, 45, 72, 54, 87, 52, sehingga 45 berpindah ke posisi yang benar, yaitu index 4 jika pengindeksan dimulai dari 0. Dalam gambar, kondisi ini dijelaskan sebagai "After first iteration, 45 is placed at its correct position." Dari hasil tersebut, list terbagi menjadi dua sub-list, yaitu sub-list with values < 45 yang berisi 4, 23, 32, 12, dan sub-list with values > 45 yang berisi 72, 54, 87, 52. Karena

elemen yang dicari adalah elemen ke-3 terkecil, maka nilai k = 2 dalam pengindeksan berbasis nol. Karena 2 < 4, elemen yang dicari pasti berada di bagian kiri dari pivot point, sehingga hanya left sublist yang dilanjutkan ke tahap berikutnya.

Tahap berikutnya memfokuskan pencarian left sublist, yaitu 4, 23, 32, 12, lalu dipilih 4 sebagai pivot point. Setelah proses partisi dilakukan, 4 langsung menempati posisi yang benar, yaitu 0th index. Angka 4 telah berada dalam posisi yang tepat, yaitu posisi pertama. Karena index dari pivot sekarang adalah 0, sedangkan elemen yang dicari tetap berada di index 2, maka pencarian tidak dilakukan di sisi kiri, melainkan dilanjutkan ke right sublist dari partisi tersebut, yaitu bagian yang masih mungkin memuat elemen ke-3 terkecil. Dengan demikian, gambar menegaskan prinsip penting quickselect, yaitu sesudah pivot ditempatkan di posisi yang benar, hanya satu sisi list yang tetap dipertimbangkan, sedangkan sisi lain diabaikan karena dipastikan tidak mengandung elemen yang dicari.

Langkah terakhir memperlihatkan pemilihan 23 sebagai pivot point dalam sublist yang tersisa. Setelah partitioning, diperoleh susunan 12, 23, 32, dan 23 ditempatkan di posisi yang benar, yaitu 3rd position dalam pengertian urutan elemen terkecil, atau index 2 sesuai target pencarian. Karena index dari pivot point sama dengan nilai k, maka proses berhenti dan 23 dikembalikan sebagai hasil. Algoritma Quick Selection tidak mengurutkan seluruh list secara lengkap, tetapi hanya memanfaatkan sifat partitioning untuk menempatkan pivot di posisi yang benar, lalu membatasi pencarian ke sublist yang relevan saja. Secara konseptual, ilustrasi tersebut menunjukkan efisiensi Quick Selection dalam menemukan elemen ke-k terkecil, karena algoritma hanya menelusuri bagian data yang benar-benar berpotensi mengandung elemen target, bukan keseluruhan list.

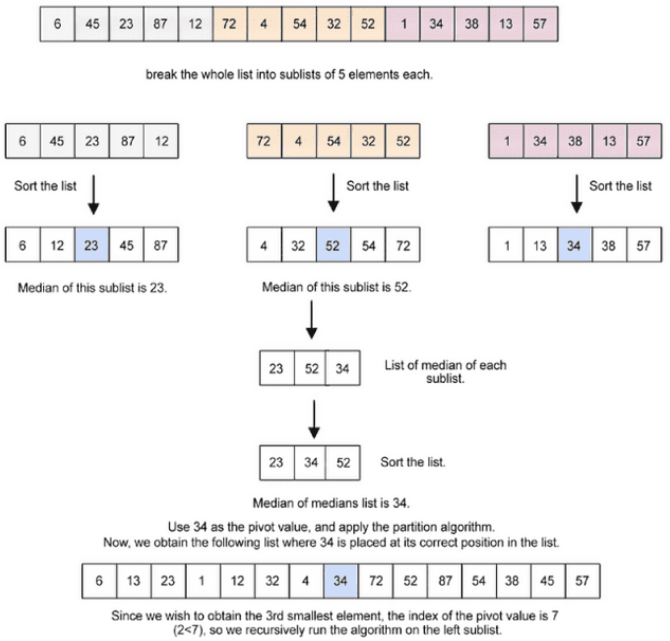
Program algoritma Quick Selection dapat dilihat di https://colab.research.google.com/drive/1m4y1H-6csCNPIX7ItbYzsvxO_PLKcOZn?usp=sharing.

10.23 Deterministic Selection

Deterministic selection merupakan algoritma yang digunakan untuk menentukan kth item dalam sebuah unordered list of elements. Sebagaimana terlihat dalam quickselect algorithm, proses pencarian dilakukan dengan memilih sebuah elemen pivot secara acak, lalu elemen tersebut digunakan untuk mempartisi list menjadi dua sublists, setelah itu algoritma memanggil dirinya sendiri secara recursively hanya untuk salah satu dari dua sublists tersebut. Berbeda dari pendekatan itu, dalam deterministic selection algorithm, elemen pivot dipilih dengan cara yang lebih efisien daripada sekadar mengambil random pivot element.

Gagasan utama dari algoritma deterministic adalah memilih pivot element yang mampu menghasilkan good split terhadap list, dan good split adalah pembagian yang memisahkan list menjadi dua bagian yang relatif seimbang. Salah satu cara yang ideal untuk memilih pivot element adalah mengambil median dari seluruh nilai. Akan tetapi, untuk memperoleh median tersebut, elemen-elemen harus diurutkan terlebih dahulu, dan langkah ini tidak efisien. Oleh sebab itu, digunakan pendekatan lain untuk memilih pivot element yang mampu membagi list kurang lebih di sekitar titik tengah. Dalam konteks ini, median of medians merupakan metode yang menyediakan nilai approximate median, yaitu nilai yang mendekati actual median untuk sebuah unsorted list of elements. Metode ini membagi list sedemikian rupa sehingga, dalam worst case, sedikitnya 3 out of 10 (3/10) elemen akan berada di bawah pivot element, dan sedikitnya 3 out of 10 elemen akan berada di atasnya.

Sebagai ilustrasi, misalkan tersedia list yang berisi 15 elements, yaitu {11, 13, 12, 111, 110, 15, 14, 16, 113, 112, 19, 18, 17, 114, 115}. List tersebut kemudian dibagi menjadi kelompok-kelompok yang masing-masing berisi 5 elements, lalu setiap kelompok diurutkan menjadi {{11, 12, 13, 110, 111}, {14, 15, 16, 112, 113}, {17, 18, 19, 114, 115}}. Sesudah itu, median dari masing-masing kelompok dihitung, yaitu 13, 16, dan 19. Langkah berikutnya adalah menentukan median dari kumpulan nilai median tersebut, yaitu dari {13, 16, 19}, sehingga diperoleh nilai 16. Nilai inilah yang disebut median of medians untuk list yang diberikan. Dari contoh tersebut terlihat bahwa terdapat 5 elements yang lebih kecil dan 9 elements yang lebih besar daripada pivot element. Ketika



Gambar 10.22: Algoritma Deterministic selection (Agarwal, 2022)

median of medians ini dipilih sebagai pivot element, list yang berisi n elements dibagi sedemikian rupa sehingga sedikitnya 3n/10 elements berada di bawah pivot element.

Secara umum, algoritma deterministic untuk memilih kth smallest element bekerja melalui beberapa tahap. Mula-mula, list yang berisi unordered items dibagi ke dalam kelompok-kelompok yang masing-masing terdiri atas lima elemen; jumlah 5 ini sebenarnya tidak bersifat wajib dan dapat diganti dengan jumlah lain, misalnya 8. Sesudah itu, setiap kelompok diurutkan, dan secara umum digunakan insertion sort untuk keperluan ini, lalu ditentukan median dari setiap kelompok. Langkah berikutnya adalah mencari secara recursively median dari seluruh nilai median yang telah diperoleh dari kelompok-kelompok tersebut; misalkan nilai itu adalah titik p. Kemudian, titik p ini digunakan sebagai pivot element, lalu algoritma partition yang serupa dengan quickselect dipanggil secara recursively untuk menemukan kth smallest element.

Gambar 10.22 memperlihatkan cara kerja algoritma Deterministic selection untuk menentukan elemen ke-k terkecil melalui pendekatan median of medians. Di bagian paling atas, ditampilkan sebuah list yang belum terurut, yaitu 6, 45, 23, 87, 12, 72, 4, 54, 32, 52, 1, 34, 38, 13, 57. Langkah pertama yang dilakukan algoritma adalah memecah seluruh list menjadi beberapa sublists yang masing-masing berisi 5 elements. Dari pemecahan tersebut dihasilkan tiga kelompok, yaitu {6, 45, 23, 87, 12}, {72, 4, 54, 32, 52}, dan {1, 34, 38, 13, 57}. Tahap ini sangat penting karena Deterministic selection tidak memilih pivot secara acak, melainkan berusaha memperoleh pivot yang lebih representatif melalui proses yang sistematis.

Tahap berikutnya adalah mengurutkan setiap sublist secara terpisah. Setelah diurutkan, kelompok pertama menjadi {6, 12, 23, 45, 87}, kelompok kedua menjadi {4, 32, 52, 54, 72}, dan kelompok ketiga menjadi {1, 13, 34, 38, 57}. Dari masing-masing sublist yang telah terurut tersebut kemudian diambil nilai tengahnya, yaitu 23, 52, dan 34. Ketiga nilai ini merupakan median of each sublist. Sesudah itu, daftar median tersebut, yaitu {23, 52, 34}, kembali diurutkan menjadi {23, 34, 52}. Nilai tengah dari daftar median ini adalah 34, dan nilai inilah yang disebut sebagai median of medians. Dalam gambar, 34 diberi penekanan khusus karena dipilih sebagai pivot value untuk seluruh list.

Sesudah pivot diperoleh, algoritma menerapkan proses partition terhadap list semula dengan menggunakan 34 sebagai titik pemisah. Hasil partisi yang ditunjukkan dalam gambar adalah 6, 13, 23, 1, 12, 32, 4, 34, 72, 52, 87, 54, 38, 45, 57. Dari susunan tersebut terlihat bahwa 34 telah ditempatkan di posisi yang benar dalam list, yaitu index 7 jika pengindeksan dimulai dari 0. Seluruh elemen di sebelah kiri 34 bernilai lebih kecil daripada 34,

sedangkan seluruh elemen di sebelah kanan 34 bernilai lebih besar. Inilah sifat utama dari proses partition dalam Deterministic selection, yaitu menempatkan pivot element langsung ke posisi finalnya meskipun seluruh list belum sepenuhnya terurut.

Pencarian dilanjutkan ke sisi kiri list untuk mencari 3rd smallest element, maka nilai k dalam pengindeksan berbasis nol adalah 2. Sementara itu, pivot value 34 berada di index 7. Karena 2 < 7, maka elemen ke-3 terkecil dipastikan berada di left sublist, bukan di sisi kanan. Oleh sebab itu, algoritma akan dijalankan kembali secara recursively hanya terhadap left sublist. Penjelasan ini menunjukkan bahwa Deterministic selection bekerja dengan prinsip yang mirip dengan quickselect, tetapi pemilihan pivot dilakukan secara lebih terarah melalui median of medians agar pembagian list menjadi lebih seimbang.

Keunggulan Deterministic selection dibandingkan pemilihan pivot acak. Dengan memilih median of medians sebagai pivot, algoritma berusaha menghasilkan good split, yaitu pembagian list yang relatif seimbang. Pembagian yang baik akan memperkecil ruang pencarian secara lebih efektif, sehingga proses penentuan kth smallest element menjadi lebih stabil dan efisien. Deterministic selection tidak bertujuan mengurutkan seluruh list, melainkan menempatkan pivot ke posisi yang benar, lalu mempersempit pencarian hanya ke bagian list yang benar-benar memuat elemen ke-k yang dicari.

Program algoritma Deterministic selection dapat dilihat di https://colab.research.google.com/drive/1oD6kUdgL06cytJSdOw51A4soUTNWyc_P?usp=sharing.

BAB 11
Searching

Operasi yang sangat penting dalam seluruh data structures adalah pencarian elemen dari suatu kumpulan data. Terdapat beragam metode untuk mencari sebuah elemen dalam struktur data structures. Operasi pencarian memiliki peran yang sangat penting dalam banyak aplikasi, terutama ketika diperlukan kepastian apakah suatu data element tertentu terdapat di dalam daftar item data yang sudah tersedia. Agar informasi dapat diperoleh secara efisien, dibutuhkan search algorithm yang efisien pula.

Efficient searching sangat penting karena memungkinkan lokasi *desired data item* diperoleh secara cepat dari *daftar stored data items*. Sebagai contoh, jika tersedia daftar nilai data yang panjang, seperti {1, 45, 65, 23, 65, 75, 23}, dan ingin diketahui apakah 75 terdapat di dalam daftar tersebut, maka penggunaan algoritma pencarian yang efisien menjadi sangat penting, khususnya ketika jumlah data items semakin besar. Kinerja algoritma pencarian juga dipengaruhi oleh cara data diorganisasikan, (Agarwal, 2022). Secara umum, data dapat disusun dalam dua bentuk, yaitu:

- Pertama, algoritma pencarian diterapkan dalam daftar item yang telah sorted, yaitu *ordered set of items*, misalnya [1, 3, 5, 7, 9, 11, 13, 15, 17].
- Kedua, algoritma pencarian diterapkan dalam *unordered set of items* yang belum sorted, misalnya [11, 3, 45, 76, 99, 11, 13, 35, 37].

11.1 Algoritma Linear Search

Linear search merupakan operasi pencarian yang digunakan untuk menemukan *index position* dari suatu data item tertentu di dalam daftar data items. Jika item yang dicari tersedia dalam daftar tersebut, maka search algorithm akan mengembalikan index position tempat item itu berada; sebaliknya, apabila item tersebut tidak ditemukan, algoritma akan mengembalikan informasi bahwa item tidak ditemukan. Dalam hal ini, index position menyatakan lokasi dari desired item di dalam daftar yang diberikan (Agarwal, 2022).

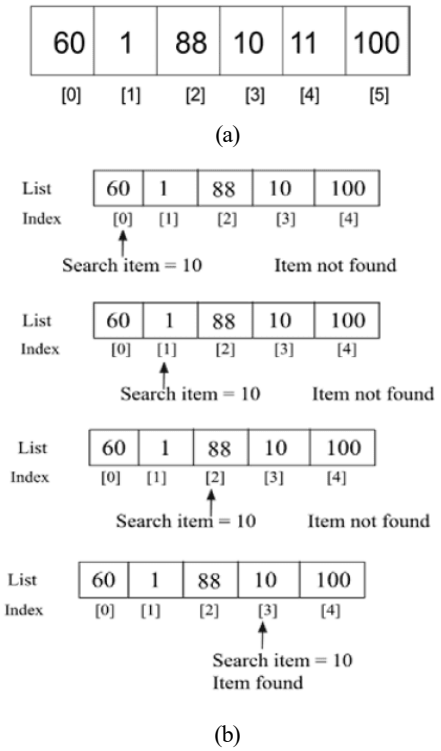
Pendekatan paling sederhana untuk mencari item dalam sebuah daftar adalah melakukan pencarian secara linear, yaitu memeriksa item satu per satu di seluruh daftar. Sebagai ilustrasi, misalkan terdapat enam list items, yaitu {60, 1, 88, 10, 11, 100}, untuk memahami cara kerja linear search algorithm. Daftar tersebut memiliki elemen-elemen yang dapat diakses melalui index. Untuk menemukan suatu elemen di dalam daftar, elemen yang dicari diperiksa secara linear satu demi satu. Teknik ini menelusuri daftar elemen dengan memanfaatkan index untuk bergerak dari awal daftar sampai ke akhir daftar. Setiap elemen diperiksa, dan jika elemen tersebut tidak cocok dengan search item, maka pemeriksaan dilanjutkan ke item berikutnya. Dengan berpindah dari satu item ke item lainnya, daftar ditelusuri secara berurutan. Dalam pembahasan ini digunakan list items bernilai integer agar konsep lebih mudah dipahami, karena integer dapat dibandingkan dengan lebih sederhana, meskipun praktiknya sebuah list item dapat menyimpan tipe data lain.

Pendekatan linear search sangat bergantung terhadap cara list items disimpan di dalam memori, yaitu apakah item-item tersebut telah sorted atau belum sorted. Oleh sebab itu, langkah awal yang perlu dipahami adalah bagaimana linear search algorithm bekerja ketika daftar item yang diberikan belum sorted.

Pencarian Tak Terurut dengan Algoritma Linear Search

Unordered linear search adalah algoritma linear search yang digunakan ketika daftar data items yang diberikan belum terurut. Dalam metode ini, desired data item dicocokkan secara linear dengan data items di dalam daftar satu per satu hingga mencapai akhir daftar atau sampai desired data item berhasil ditemukan. Sebagai contoh, misalkan terdapat daftar yang berisi elemen 60, 1, 88, 10, dan 100, yang merupakan unordered list. Untuk melakukan search operation dalam daftar semacam ini, proses dimulai dari item pertama, lalu item tersebut dibandingkan dengan search item. Jika search item tidak cocok, maka pemeriksaan dilanjutkan ke elemen

berikutnya dalam daftar. Proses ini berlangsung terus sampai elemen terakhir dicapai atau sampai ditemukan kecocokan.



Gambar 11.1: (a) Contoh Algoritma Liniear Search (b) Contoh Pencarian Tak Terurut dengan Algoritma Liniear Search (Agarwal, 2022)

Gambar 11.1 memperlihatkan ilustrasi Algoritma Linier Search beserta penerapannya untuk pencarian data dalam daftar yang tidak terurut. Bagian (a) menampilkan susunan awal elemen dalam list, yaitu 60, 1, 88, 10, 11, dan 100, disertai posisi indeks [0], [1], [2], [3], [4], dan [5]. Setiap elemen memiliki lokasi tertentu di dalam memori logis list, sehingga hasil pencarian tidak hanya berupa kepastian bahwa elemen ditemukan, melainkan juga berupa index position tempat elemen itu berada.

Bagian (b) memperlihatkan proses pencarian tak terurut dengan Algoritma Linier Search untuk menemukan search item = 10. Mekanisme pencarian dilakukan secara berurutan dari elemen pertama menuju elemen berikutnya. Langkah pertama dimulai dengan memeriksa elemen 60 di indeks [0]. Karena nilai tersebut tidak sama dengan 10, hasil yang ditunjukkan ialah item not found, lalu pemeriksaan dilanjutkan ke elemen berikutnya. Langkah kedua membandingkan search item = 10 dengan elemen 1 di indeks [1], tetapi hasilnya tetap tidak cocok. Langkah ketiga memperlihatkan perbandingan dengan elemen 88 di indeks [2], dan keluaran yang diperoleh masih item not found. Sesudah tiga kali pemeriksaan yang gagal, langkah keempat membandingkan 10 dengan elemen 10 di indeks [3]. Dalam tahap inilah kecocokan ditemukan, sehingga keluaran berubah menjadi item found. Rangkaian ilustrasi anak panah yang berpindah dari indeks [0] ke [1], lalu ke [2], dan akhirnya ke [3] menunjukkan secara jelas bahwa algoritma bergerak satu per satu tanpa melompati elemen mana pun.

Pencarian algoritma Linier Search dilakukan dengan pemeriksaan sekuensial terhadap seluruh elemen list sampai elemen target ditemukan atau hingga seluruh elemen habis diperiksa. Teknik ini sangat sederhana karena tidak memerlukan data yang telah diurutkan terlebih dahulu. Akan tetapi, kesederhanaan tersebut menyebabkan efisiensi pencarian sangat bergantung posisi elemen yang dicari. Dalam contoh ini, elemen 10 ditemukan setelah algoritma membandingkan empat elemen, yaitu 60, 1, 88, dan 10. Hal ini menunjukkan bahwa semakin jauh letak elemen target dari awal list, semakin banyak langkah perbandingan yang harus dilakukan. Algoritma Linier Search cocok digunakan untuk memahami konsep dasar pencarian data, terutama dalam list yang tidak terurut,

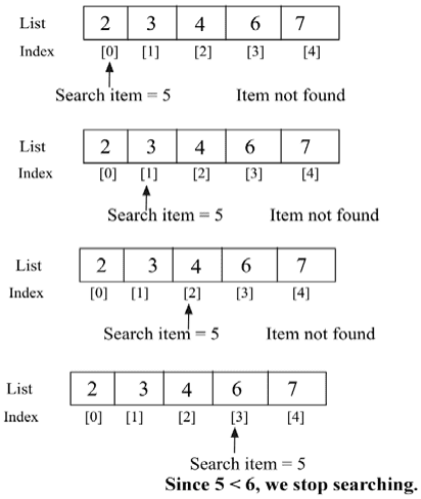
meskipun untuk data berukuran besar metode ini cenderung kurang efisien dibandingkan algoritma pencarian yang lebih canggih.

Ketika mencari elemen dalam data yang tidak terurut, kondisi terburuk dapat terjadi jika desired item berada di posisi terakhir atau bahkan tidak terdapat di dalam daftar. Dalam situasi seperti ini, search item harus dibandingkan dengan seluruh elemen daftar, yaitu sebanyak n kali apabila jumlah total data items dalam daftar adalah n. Oleh sebab itu, unordered linear search memiliki worst-case running time sebesar O(n). Seluruh elemen mungkin harus dikunjungi terlebih dahulu sebelum search term ditemukan. Worst-case scenario terjadi ketika search term berada di posisi paling akhir dalam daftar.

Pencarian Terurut dengan Algoritma Linear Search

Ordered linear search digunakan ketika data elements telah tersusun dalam keadaan sorted order, sehingga linear search algorithm dapat dibuat lebih efisien. Dalam daftar elemen yang telah terurut, proses pencarian dilakukan dengan bergerak melalui daftar secara sekuensial. Akan tetapi, terdapat kondisi tambahan yang membedakannya dari unordered linear search, yaitu jika nilai search item lebih kecil daripada current item yang sedang diperiksa dalam perulangan, maka pencarian dapat segera dihentikan dan algoritma mengembalikan None. Dengan kata lain, selama proses iterasi, jika nilai search term lebih kecil daripada item yang sedang diperiksa, maka tidak ada alasan untuk melanjutkan pencarian karena search term tersebut tidak mungkin muncul lagi di posisi sesudahnya dalam daftar yang telah diurutkan.

Gambar 11.2 memperlihatkan proses pencarian terurut dengan Algoritma Linier Search dalam sebuah list



Gambar 11.2: Contoh Pencarian Terurut dengan Algoritma Liniear Search (Agarwal, 2022)

yang sudah disusun secara menaik, yaitu 2, 3, 4, 6, dan 7. Di bagian atas, list ditampilkan bersama index masing-masing, yaitu [0], [1], [2], [3], dan [4], sehingga setiap elemen memiliki posisi yang jelas di dalam struktur data. Search item yang ingin ditemukan adalah 5. Berbeda dari pencarian linier dalam list tak terurut, urutan data yang sudah tersusun memberikan keuntungan tambahan, karena proses pencarian dapat dihentikan lebih awal ketika elemen yang sedang diperiksa telah melampaui nilai search item. Artinya, algoritma tidak perlu lagi memeriksa elemen 7 di index [4], karena secara logis 5 tidak akan ditemukan di posisi yang lebih belakang.

Urutan pencarian dimulai dari elemen pertama, yaitu 2 di index [0]. Nilai 2 dibandingkan dengan search item = 5, tetapi hasilnya tidak cocok, sehingga muncul keterangan item not found dan algoritma bergerak ke elemen berikutnya. Langkah kedua memeriksa elemen 3 di index [1]. Hasil perbandingan kembali menunjukkan ketidakcocokan, sehingga proses diteruskan ke elemen ketiga, yaitu 4 di index [2]. Sesudah itu, algoritma memeriksa elemen 6 di index [3]. Sampai tahap ini, elemen 5 memang belum ditemukan. Namun, terdapat perbedaan penting dibandingkan pencarian linier biasa, karena nilai 6 ternyata lebih besar daripada search item 5.

Keadaan ini memberi informasi bahwa elemen 5 tidak mungkin lagi muncul di bagian sesudahnya, sebab list telah diurutkan secara menaik.

Dalam *worst-case scenario*, *desired search item* dapat berada di posisi terakhir dalam daftar atau bahkan tidak terdapat sama sekali di dalam daftar. Dalam situasi seperti ini, algoritma tetap harus menelusuri seluruh daftar, misalnya sebanyak n elements. Karena itu, *worst-case time complexity* dari *ordered linear search* tetap bernilai O(n). Artinya, dalam kondisi terburuk, algoritma ordered linear search masih bisa harus memeriksa elemen satu per satu sampai hampir seluruh atau seluruh list selesai diperiksa. Notasi O(n) menunjukkan bahwa waktu pencarian bertambah sebanding dengan jumlah data di dalam list. Jika jumlah elemen sedikit, waktu pencarian juga sedikit. Jika jumlah elemen menjadi dua kali lebih banyak, maka jumlah pemeriksaan yang mungkin dilakukan juga kira-kira menjadi dua kali lebih banyak.

Program algoritma linear search dapat dilihat di link https://colab.research.google.com/drive/1ImzzUxKl8TMHFZhX6OFucpm7dT4tluTI?usp=sharing.

11.2 Algoritma Jump Search

Jump search merupakan algoritma pencarian yang dikembangkan sebagai perbaikan dari linear search untuk menemukan elemen tertentu dalam list yang bersifat ordered atau sorted. Algoritma ini memanfaatkan strategi divide-and-conquer dengan cara tidak memeriksa setiap elemen satu per satu seperti linear search, melainkan membandingkan search value dalam interval tertentu dalam list sehingga jumlah perbandingan dapat dikurangi (Agarwal, 2022). Mekanisme kerjanya dimulai dengan:

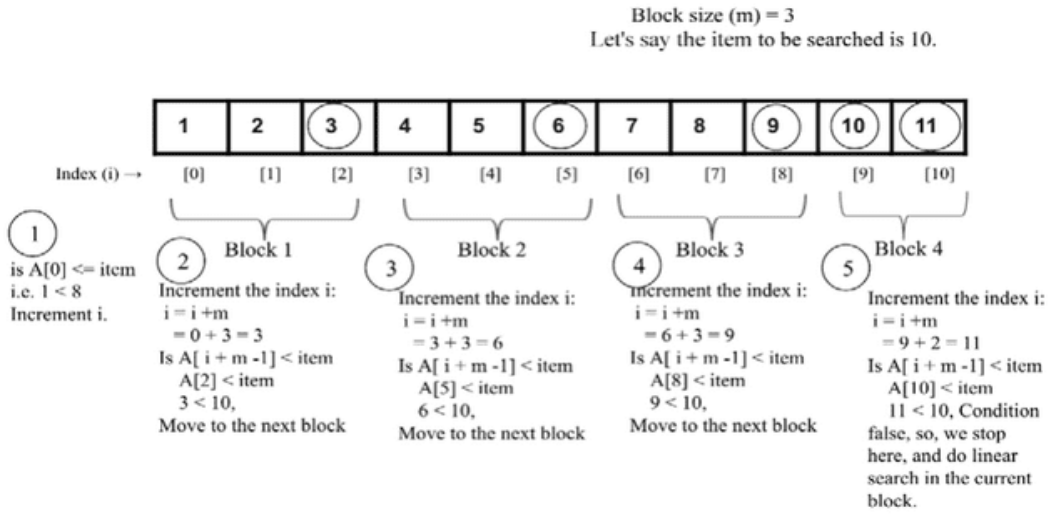
1) Membagi list terurut ke dalam beberapa subset yang disebut blocks. Karena array telah tersusun secara menaik, nilai terbesar dalam setiap block akan berada di elemen terakhir block tersebut.
2) Selanjutnya, search value dibandingkan dengan elemen terakhir dari tiap block. Jika search value lebih besar daripada elemen terakhir suatu block, pencarian dilanjutkan ke block berikutnya.
3) Jika search value lebih kecil daripada elemen terakhir block, hal itu menunjukkan bahwa elemen yang dicari, apabila memang ada, harus berada di block tersebut, sehingga linear search diterapkan di dalam block itu untuk memperoleh index position.
4) Jika search value sama dengan elemen pembanding di ujung block, algoritma langsung mengembalikan index position elemen tersebut sebagai candidate. Jump search bekerja melalui dua tahap utama, yaitu menemukan block yang paling mungkin memuat elemen target, lalu melakukan linear search di dalam block itu.

Secara konseptual, ukuran block umumnya diambil sebesar √n, karena pilihan ini memberikan kinerja terbaik untuk array dengan panjang n. Dalam worst-case situation, algoritma dapat melakukan n/m kali lompatan, dengan m menyatakan ukuran block, sebelum menemukan block yang relevan, lalu masih memerlukan hingga m - 1 kali perbandingan tambahan melalui linear search di dalam block terakhir tersebut. Oleh sebab itu, total jumlah perbandingan dapat dinyatakan sebagai (n/m) + m - 1. Nilai ini mencapai kondisi minimum ketika m = √n, sehingga ukuran block yang paling efisien dipilih sebesar √n. Konsekuensinya, worst-case time complexity dari Jump search adalah O(√n). Sebagai ilustrasi, jika digunakan list {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11} untuk mencari elemen 10, maka algoritma akan melompat dari satu block ke block lain sampai menemukan rentang yang memuat 10, lalu melakukan linear search hanya di dalam block tersebut. Cara kerja ini menjadikan Jump search lebih efisien daripada linear search biasa untuk list yang telah sorted.

Gambar 11.3 memperlihatkan cara kerja Jump search untuk mencari elemen 10 dalam sebuah array terurut yang berisi nilai 1 sampai 11. Ilustrasi menunjukkan bahwa array dibagi ke dalam beberapa block dengan ukuran m = 3, sehingga terbentuk Block 1 yang memuat elemen [1, 2, 3], Block 2 yang memuat [4, 5, 6], Block 3 yang memuat [7, 8, 9], dan Block 4 yang memuat [10, 11]. Pembagian ini menegaskan prinsip utama Jump search, yaitu tidak memeriksa seluruh elemen satu per satu dari awal, melainkan melompat dari satu block ke block berikutnya dengan membandingkan search item terhadap elemen terakhir di setiap block. Elemen-elemen yang dilingkari dalam gambar, yaitu 3, 6, 9, dan 10, menandai titik-titik pemeriksaan yang digunakan algoritma untuk

menentukan apakah pencarian harus dilanjutkan ke block berikutnya atau dihentikan untuk kemudian dilakukan pencarian lebih rinci di dalam block tertentu.

Urutan proses dalam gambar dijelaskan melalui lima langkah. Langkah pertama dimulai dari indeks awal i = 0, kemudian algoritma membandingkan item = 10 dengan elemen terakhir Block 1, yaitu A[2] = 3. Karena 3



Gambar 11.3 Ilustrasi Algoritma Jump Search (Agarwal, 2022)

< 10, elemen target belum mungkin ditemukan di Block 1, sehingga pencarian dilanjutkan ke block berikutnya. Langkah kedua menaikkan indeks menjadi i = i + m = 0 + 3 = 3, lalu algoritma memeriksa elemen terakhir Block 2, yaitu A[5] = 6. Karena 6 < 10, keputusan yang diambil tetap sama, yaitu bergerak ke block selanjutnya. Langkah ketiga menaikkan indeks lagi menjadi i = 6, kemudian diperiksa elemen terakhir Block 3, yaitu A[8] = 9. Hasil perbandingan menunjukkan 9 < 10, sehingga algoritma kembali melompat ke block berikutnya. Langkah keempat menaikkan indeks menjadi i = 9, lalu memeriksa elemen terakhir rentang yang sedang diamati, yaitu A[10] = 11. Kali ini kondisi A[i + m - 1] < item bernilai salah, karena 11 < 10 tidak terpenuhi. Hasil tersebut memberi informasi penting bahwa elemen 10 tidak mungkin berada di block sebelumnya, tetapi sangat mungkin berada di block saat ini, yaitu rentang yang mencakup 10 dan 11.

Setelah algoritma berhasil menentukan block kandidat, proses Jump search tidak lagi melakukan lompatan ke block lain, melainkan beralih ke linear search di dalam block yang sedang diperiksa. Dalam contoh ini, linear search diterapkan di Block 4 untuk memeriksa elemen satu per satu sampai ditemukan nilai 10. Dengan demikian, gambar ini menjelaskan dua tahap utama Jump search, yaitu tahap pertama berupa lompatan antar-block untuk mempersempit rentang pencarian, dan tahap kedua berupa linear search di dalam block yang paling mungkin memuat elemen target.

Program algoritma jump search dapat dilihat di link https://colab.research.google.com/drive/1gprT27reHhedqq2OynGMUj4qEWoJBmoP?usp=sharing.

11.3 Algoritma Binary Search

Binary search adalah algoritma yang digunakan untuk menemukan suatu item tertentu dari daftar item yang diurutkan. Algoritma ini dikenal sebagai metode pencarian yang cepat dan efisien. Namun, salah satu keterbatasannya adalah bahwa algoritma ini hanya dapat diterapkan jika daftar data sudah berada dalam keadaan sorted. Dari segi kinerja, worst-case running time complexity dari algoritma binary search adalah O(log n), sedangkan linear search memiliki worst-case running time complexity sebesar O(n). Hal ini menunjukkan bahwa binary search mampu melakukan pencarian dengan jumlah langkah yang jauh lebih sedikit dibandingkan linear search, terutama ketika ukuran data sangat besar (Agarwal, 2022).

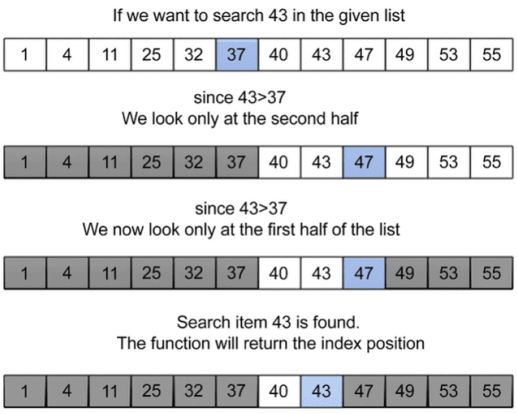
Cara kerja binary search algorithm dimulai dengan:

1. Membagi daftar yang diberikan menjadi dua bagian.
2. Kemudian, nilai search item dibandingkan dengan nilai tengah dari daftar tersebut.
3. Jika search item lebih kecil daripada middle value, maka pencarian hanya dilanjutkan ke bagian pertama dari daftar.
4. Sebaliknya, jika search item lebih besar daripada middle value, maka pencarian hanya diteruskan ke bagian kedua dari daftar.
5. Proses ini diulangi terus-menerus hingga search item ditemukan atau hingga seluruh kemungkinan bagian daftar telah diperiksa.
6. Prinsip ini juga berlaku untuk data nonnumerik.

Sebagai contoh, jika data items berupa string, maka data harus terlebih dahulu diurutkan secara alfabetis, sebagaimana daftar kontak dalam telepon disimpan berdasarkan urutan huruf.

Contoh untuk memahami mekanisme kerja binary search adalah pencarian item 43 dari sebuah daftar yang berisi 12 items, sebagaimana ditunjukkan dalam ilustrasi Gambar 11.4.

Gambar 11.4 memperlihatkan cara kerja algoritma Binary Search untuk menemukan nilai 43 di dalam sebuah daftar yang telah tersusun secara menaik, yaitu 1, 4, 11, 25, 32, 37, 40, 43, 47, 49, 53, dan 55. Ilustrasi



Gambar 11.4: Ilustrasi Algoritma Binary Search (Agarwal, 2022)

disusun dalam beberapa tahap untuk menunjukkan bahwa Binary Search tidak memeriksa elemen satu per satu seperti Linear Search, melainkan langsung memusatkan perhatian dalam elemen tengah dari rentang pencarian. Tahap pertama, elemen tengah dari seluruh daftar adalah 37, yang ditandai dengan warna berbeda. Karena nilai yang dicari, yaitu 43, lebih besar daripada 37, maka bagian kiri daftar tidak perlu diperiksa lagi. Oleh sebab itu, pencarian dilanjutkan hanya ke second half dari daftar, yaitu bagian yang memuat elemen-elemen di sebelah kanan 37.

Tahap berikutnya menunjukkan bahwa setelah ruang pencarian dipersempit ke separuh kanan, algoritma kembali menentukan elemen tengah dari bagian yang tersisa. Dalam rentang baru tersebut, elemen yang dijadikan titik pemeriksaan adalah 47, yang juga ditandai dengan warna berbeda. Karena 43 lebih kecil daripada 47, maka bagian kanan dari rentang itu dapat dieliminasi, sehingga pencarian dipusatkan hanya first half dari subdaftar tersebut. Langkah ini menegaskan prinsip utama Binary Search, yaitu setiap kali satu elemen tengah diperiksa, ruang pencarian langsung diperkecil menjadi separuh dari ukuran sebelumnya. Dengan demikian, jumlah perbandingan yang diperlukan menjadi jauh lebih sedikit dibandingkan pencarian berurutan biasa.

Tahap terakhir, ruang pencarian telah menyempit hingga mencapai elemen 43, yang kemudian ditandai sebagai elemen yang berhasil ditemukan. Keterangan di bawah ilustrasi menjelaskan bahwa search item 43 is found dan fungsi akan mengembalikan index position dari elemen tersebut. Secara konseptual, gambar ini menegaskan bahwa Binary Search bekerja sangat efisien karena selalu membagi ruang pencarian menjadi dua bagian dan hanya melanjutkan pencarian ke bagian yang masih mungkin memuat elemen target. Oleh karena itu,

ilustrasi tersebut menunjukkan dengan jelas bahwa keberhasilan Binary Search sangat bergantung kondisi daftar yang telah terurut, karena hanya dengan susunan data yang teratur algoritma dapat menentukan apakah pencarian harus diarahkan ke bagian kiri atau ke bagian kanan daftar.

Program algoritma Binary Search dapat dilihat di https://colab.research.google.com/drive/1VgqeFbQkR4cqEIHXnc0YpfdX-S_0lFNW?usp=sharing.

11.4 Algoritma Interpolation Search

Interpolation search merupakan algoritma pencarian yang dikembangkan sebagai penyempurnaan dari binary search. Binary search dikenal sebagai algoritma pencarian yang efisien karena selalu mempersempit search space menjadi setengah bagian dengan mengabaikan salah satu bagian berdasarkan nilai search item. Apabila search item lebih kecil daripada nilai yang berada di tengah list, maka separuh bagian kedua dari search space akan diabaikan. Dengan demikian, binary search selalu mengurangi search space sebesar setengah secara tetap. Berbeda dari mekanisme tersebut, algoritma interpolation search menggunakan pendekatan yang lebih adaptif sehingga search space dapat diperkecil dengan cara yang lebih efisien, bahkan dapat lebih dari setengah dalam setiap iterasi tertentu (Agarwal, 2022).

Kinerja algoritma interpolation search akan sangat baik jika elemen-elemen dalam sorted list memiliki sebaran yang seragam atau *uniformly distributed*. Dalam binary search, proses pencarian selalu dimulai dari posisi tengah list. Sebaliknya, dalam interpolation search, posisi awal pencarian dihitung berdasarkan nilai item yang dicari. Oleh sebab itu, posisi awal pencarian dalam interpolation search algorithm cenderung berada lebih dekat ke awal atau ke akhir list, bergantung letak nilai search item. Jika search item mendekati elemen pertama dalam list, maka posisi awal pencarian umumnya akan berada dekat awal list. Sebaliknya, jika search item mendekati elemen terakhir, maka posisi awal pencarian cenderung berada dekat akhir list.

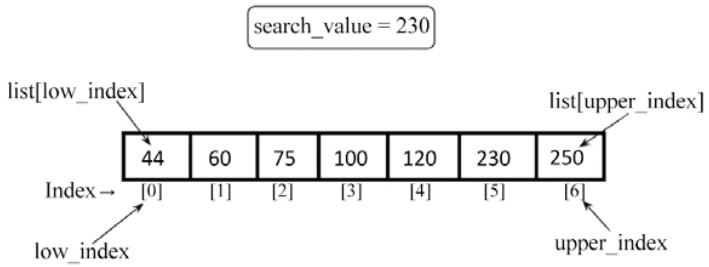
Secara konseptual, interpolation search menyerupai cara manusia melakukan pencarian dalam suatu daftar, yaitu dengan memperkirakan terlebih dahulu posisi indeks yang paling mungkin memuat item yang dicari dalam sorted list. Mekanismenya mirip dengan binary search algorithm, tetapi berbeda dalam menentukan kriteria pembagian data untuk mengurangi jumlah perbandingan. Jika binary search membagi data menjadi dua bagian yang sama besar, maka interpolation search membagi data berdasarkan rumus: mid = low_index + ((upper_index - low_index) / (list[upper_index] - list[low_index])) * (search_term - list[low_index]). Dalam rumus tersebut, low_index menyatakan indeks batas bawah dari list, yaitu indeks yang memuat nilai terkecil, sedangkan upper_index menunjukkan posisi indeks dengan nilai tertinggi dalam list. Sementara itu, list[low_index] dan list[upper_index] masing-masing merepresentasikan nilai terendah dan nilai tertinggi dalam list. Variabel search_value berisi nilai item yang akan dicari.

Proses kerja interpolation search algorithm dimulai dengan:

1. mencari midpoint menggunakan rumus tersebut.
2. Selanjutnya, nilai search value dibandingkan dengan nilai yang tersimpan dalam indeks midpoint.
3. Jika keduanya sama, maka posisi indeks tersebut dikembalikan sebagai hasil pencarian.
4. Namun, jika nilainya tidak sama, maka list dibagi menjadi dua bagian, yaitu higher sublist dan lower sublist.
5. Higher sublist memuat elemen-elemen dengan nilai indeks lebih tinggi daripada midpoint, sedangkan lower sublist memuat elemen-elemen dengan nilai indeks lebih rendah.
6. Apabila search value lebih besar daripada nilai di midpoint, pencarian dilanjutkan ke higher sublist dan lower sublist diabaikan.
7. Sebaliknya, jika search value lebih kecil daripada nilai di midpoint, pencarian diteruskan ke lower sublist dan higher sublist tidak digunakan.
8. Proses ini dilakukan secara berulang hingga ukuran sublist menjadi nol.

Secara kompleksitas, algoritma interpolation search sangat efektif ketika data set telah terurut dan bersifat *uniformly distributed*. Dalam kondisi demikian, average case time complexity-nya adalah O(log(log n)), dengan n menyatakan panjang array. Akan tetapi, jika dataset tidak memiliki distribusi yang seragam atau bersifat acak, maka *worst-case time complexity* dari algoritma interpolation search dapat menjadi O(n). Oleh karena itu, interpolation search berpotensi bekerja lebih baik daripada binary search ketika data yang digunakan memiliki distribusi yang seragam.

Gambar 11.5 memperlihatkan proses kerja Interpolation Search dalam mencari search_value = 230 di dalam sebuah list yang telah diurutkan secara menaik, yaitu 44, 60, 75, 100, 120, 230, dan 250. Setiap elemen



Gambar 11.5: Ilustrasi Algoritma Interpolation Search (Agarwal, 2022)

memiliki index berurutan dari [0] hingga [6]. Batas bawah pencarian ditetapkan melalui low_index = 0 yang menunjuk ke nilai list[low_index] = 44, sedangkan batas atas pencarian ditetapkan melalui upper_index = 6 yang menunjuk ke nilai list[upper_index] = 250. Dua batas ini tidak hanya menandai rentang pencarian, tetapi juga menjadi dasar perhitungan untuk memperkirakan posisi elemen yang dicari. Berbeda dari binary search yang langsung memeriksa elemen tengah, Interpolation Search terlebih dahulu menghitung posisi taksiran dengan mempertimbangkan sebaran nilai data dan kedekatan search_value terhadap nilai batas bawah maupun batas atas.

Proses pencarian dimulai dengan menghitung nilai mid menggunakan rumus interpolasi, yaitu mid = low_index + ((upper_index - low_index) / (list[upper_index] - list[low_index])) × (search_value - list[low_index]). Jika nilai-nilai dalam gambar disubstitusikan ke rumus tersebut, maka diperoleh mid = 0 + ((6 - 0) / (250 - 44)) × (230 - 44). Hasil perhitungan ini menjadi mid = 0 + (6 / 206) × 186, sehingga diperoleh nilai sekitar 5,42. Karena index harus berupa bilangan bulat, hasil tersebut diambil ke posisi terdekat yang digunakan dalam pencarian, yaitu index [5]. Langkah ini menunjukkan bahwa algoritma tidak memilih posisi tengah daftar secara tetap, melainkan langsung mengarahkan pencarian ke lokasi yang diperkirakan paling dekat dengan search_value. Dalam kasus ini, karena nilai 230 jauh lebih dekat ke 250 daripada ke 44, maka hasil interpolasi secara logis bergerak ke sisi kanan list.

Setelah posisi mid diperoleh, algoritma membandingkan nilai elemen di index [5] dengan search_value. Nilai di list[5] adalah 230, sedangkan nilai yang dicari juga 230. Karena kedua nilai tersebut sama, proses pencarian langsung berhenti dan index [5] dinyatakan sebagai lokasi ditemukannya elemen target. Dengan demikian, gambar tersebut sebenarnya memperlihatkan bahwa pencarian selesai hanya dalam satu iterasi. Kondisi ini menegaskan efisiensi Interpolation Search ketika data dalam list tersusun terurut dan memiliki distribusi yang relatif seragam. Algoritma mampu memperkirakan posisi elemen dengan sangat dekat, sehingga tidak perlu memeriksa elemen lain seperti 44, 60, 75, 100, atau 120 satu per satu.

Algoritma memanfaatkan hubungan antara index dan nilai data. Nilai search_value = 230 berada dekat dengan elemen terbesar, yaitu 250, sehingga hasil perhitungan mid juga bergeser mendekati upper_index. Inilah alasan mengapa pencarian langsung tertuju ke index [5], bukan ke index [3] atau titik tengah daftar. Interpolation Search bekerja dengan pendekatan estimasi posisi, bukan pembagian rentang secara simetris.

Program algoritma Interpolation Search disajikan di https://colab.research.google.com/drive/1Nw0pF9jrzZQyRrF1mpGPP6xyVXpS5JgG?usp=sharing.

11.5 Algoritma Exponential Search

Exponential search merupakan salah satu algoritma pencarian yang umum digunakan ketika jumlah elemen dalam sebuah list sangat besar, (Agarwal, 2022). Algoritma ini juga dikenal dengan sebutan **galloping search dan doubling search**. Secara umum, algoritma exponential search bekerja melalui dua tahap utama, yaitu:

1. Tahap pertama bertujuan menentukan subrange dalam sorted array asal yang diduga memuat search item yang diinginkan.
2. Tahap kedua memanfaatkan algoritma binary search untuk menemukan search value di dalam subrange yang telah diidentifikasi di tahap pertama.

Dengan demikian, exponential search tidak langsung menelusuri seluruh data, melainkan lebih dahulu mempersempit area pencarian sebelum menerapkan proses pencarian yang lebih terfokus.

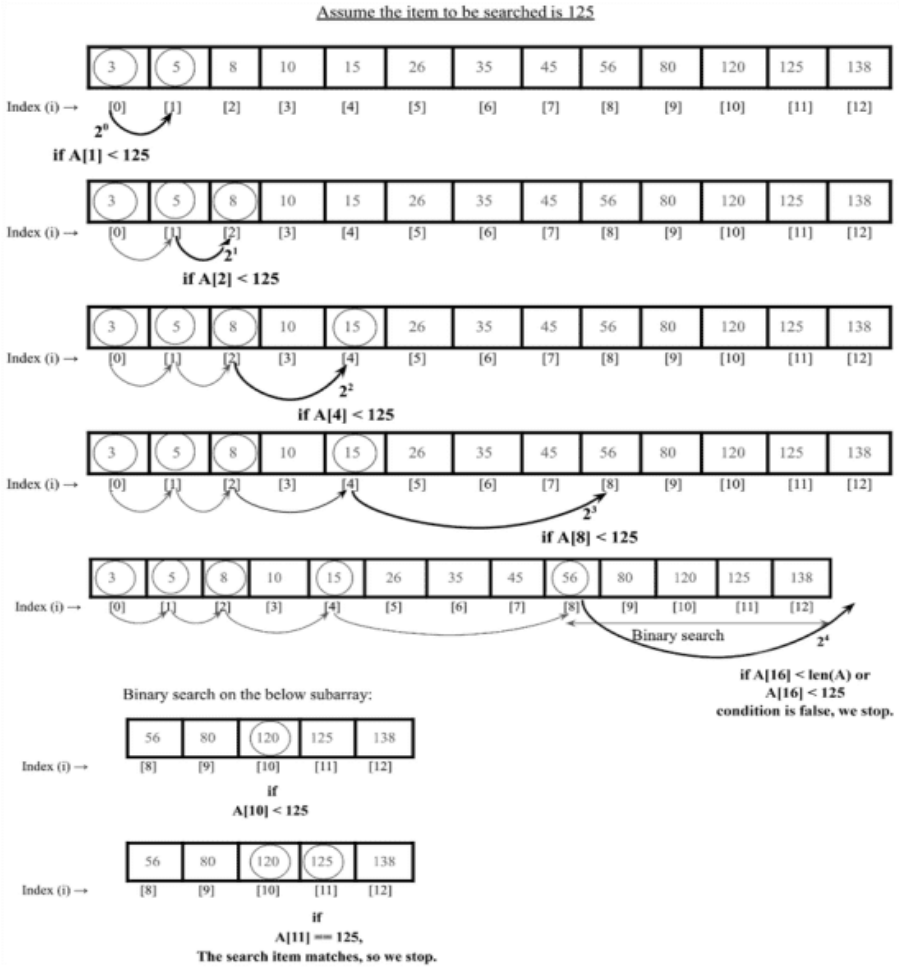
Untuk menentukan subrange elemen data, proses pencarian dimulai dengan melompat sejauh 2^i elemen dalam setiap iterasi di dalam *sorted array* yang diberikan, dengan i menyatakan nilai index array. Setelah setiap lompatan dilakukan, sistem memeriksa apakah search item berada di antara lompatan sebelumnya dan lompatan saat ini. Jika search item ditemukan berada dalam rentang tersebut, maka algoritma binary search diterapkan dalam subarray itu. Sebaliknya, apabila elemen yang dicari belum berada dalam rentang tersebut, maka index digeser ke lokasi berikutnya. Oleh karena itu, langkah awal algoritma ini berfokus dalam pencarian kemunculan pertama dari eksponen i sedemikian rupa sehingga nilai index 2^i lebih besar daripada search value. Dalam kondisi tersebut, 2^(i-1) menjadi batas bawah dan 2^i menjadi batas atas bagi rentang elemen data yang diperkirakan memuat search value.

Secara prosedural, exponential search algorithm diawali dengan:

1. Membandingkan elemen pertama, yaitu A[0], dengan elemen pencarian.
2. Posisi index diinisialisasi dengan nilai i = 1.
3. Algoritma memeriksa dua kondisi, yaitu apakah pencarian telah mencapai akhir array atau belum, yang dinyatakan dengan 2^i < len(A), serta apakah A[i] <= search_value. Kondisi pertama digunakan untuk memastikan apakah seluruh list telah ditelusuri; proses dihentikan ketika ujung list telah tercapai. Kondisi kedua digunakan untuk menghentikan pencarian saat ditemukan elemen yang nilainya lebih besar daripada search value, karena hal itu menunjukkan bahwa elemen yang dicari harus terletak sebelum posisi index tersebut, mengingat data telah terurut. Selama proses masih berlanjut, perpindahan ke index berikutnya dilakukan dengan kenaikan berbasis pangkat dua.
4. Proses tersebut dihentikan ketika salah satu kondisi penghentian terpenuhi.
5. Setelah rentang pencarian berhasil ditentukan, binary search algorithm diterapkan dalam rentang 2^i // 2 hingga min(2^i, len(A)).

Melalui strategi ini, exponential search menjadi efisien untuk menangani data berukuran besar karena area pencarian dapat dipersempit secara cepat sebelum pencarian rinci dilakukan.

Gambar 11.6 memperlihatkan ilustrasi kerja Exponential Search untuk menemukan item bernilai 125 di dalam sebuah array terurut menaik, yaitu 3, 5, 8, 10, 15, 26, 35, 45, 56, 80, 120, 125, 138. Setiap elemen disertai index yang dimulai dari [0] hingga [12]. Susunan visual dalam gambar menunjukkan dua tahap utama algoritma, yaitu tahap penentuan rentang pencarian melalui lompatan eksponensial dan tahap pencarian rinci menggunakan Binary Search. Lingkaran-lingkaran dalam beberapa elemen menandai elemen yang diperiksa secara bertahap, sedangkan panah-panah lengkung menunjukkan pola perpindahan index yang bertambah menurut pangkat dua, yaitu 2^0, 2^1, 2^2, 2^3, dan seterusnya.



Gambar 11.6: Ilustrasi Algoritma Exponential Search (Agarwal, 2022)

Proses pencarian dalam gambar dimulai dengan asumsi bahwa item yang dicari adalah 125. Langkah awal algoritma memeriksa elemen pertama, yaitu A[0] = 3. Karena nilai tersebut tidak sama dengan 125, pencarian dilanjutkan dengan lompatan berbasis eksponen dua. Pemeriksaan berikutnya dilakukan di A[1] = 5, lalu dibandingkan dengan 125. Karena 5 < 125, algoritma melanjutkan ke lompatan berikutnya, yaitu A[2] = 8. Hasil perbandingan kembali menunjukkan bahwa 8 < 125, sehingga pencarian diteruskan ke A[4] = 15. Nilai ini juga masih lebih kecil daripada 125, maka algoritma kembali memperbesar rentang dengan melompat ke A[8] = 56. Karena 56 < 125, proses belum berhenti. Setelah itu, algoritma hendak melompat ke A[16], tetapi index 16 telah melebihi panjang array karena elemen terakhir hanya berada di index [12]. Kondisi ini menandakan bahwa rentang yang mungkin memuat item 125 telah berhasil ditemukan, yaitu antara index [8] dan batas akhir array, yakni index [12].

Tahap berikutnya adalah penerapan Binary Search dalam subarray hasil identifikasi rentang, yaitu [56, 80, 120, 125, 138], yang terletak dalam index [8] hingga [12]. Dalam gambar, tahap ini dijelaskan di bagian bawah. Elemen tengah dari rentang tersebut berada di index [10], yaitu A[10] = 120. Karena 120 < 125, maka bagian kiri dari subarray, yaitu elemen 56 dan 80, dapat diabaikan. Pencarian kemudian dipersempit ke bagian kanan, yaitu rentang index [11] hingga [12]. Dari rentang baru ini, elemen yang diperiksa adalah A[11] = 125. Hasil perbandingan menunjukkan bahwa nilai tersebut sama dengan item yang dicari, sehingga pencarian dihentikan. Search item berhasil ditemukan di index [11].

Secara algoritmik, Exponential Search bekerja dengan strategi dua fase. Fase pertama menggunakan lompatan eksponensial untuk menemukan batas bawah dan batas atas yang lebih sempit. Dalam contoh ini,

lompatan dilakukan secara berurutan ke index 1, 2, 4, dan 8. Ketika lompatan berikutnya tidak lagi memenuhi syarat karena index melebihi panjang array, algoritma menyimpulkan bahwa target terletak dalam rentang terakhir yang valid. Fase kedua memanfaatkan Binary Search agar pencarian dalam rentang tersebut berlangsung cepat dan efisien. Strategi ini sangat bermanfaat untuk array berukuran besar karena algoritma tidak perlu memeriksa seluruh elemen satu per satu. Sebaliknya, algoritma langsung memperkirakan wilayah pencarian melalui pertumbuhan rentang yang sangat cepat, lalu menyempurnakan pencarian dengan mekanisme pembelahan rentang.

Program algoritma Exponential Search dapat dilihat di https://colab.research.google.com/drive/1cbJ5dB9h5bDuH1NnBXnhzWITcPYZ_sDR?usp=sharing.

11.6 Pemilihan Algoritma Search

Pemilihan algoritma Search perlu mempertimbangkan karakteristik data dan kebutuhan pencarian. Berbagai jenis search algorithms dibahas, terlihat bahwa setiap algoritma memiliki konteks penggunaan yang berbeda. Binary search dan interpolation search algorithms menawarkan performa yang lebih baik dibandingkan ordered maupun unordered linear search functions. Sebaliknya, linear search algorithm cenderung lebih lambat karena proses pencarian berlangsung melalui pemeriksaan elemen secara berurutan atau sequential probing hingga search term ditemukan. Oleh karena itu, efisiensi algoritma sangat ditentukan oleh susunan data, ukuran daftar, dan pola distribusi nilai di dalam list.

Ditinjau dari kompleksitas waktu, linear search memiliki time complexity sebesar O(n). Kondisi ini menunjukkan bahwa linear search algorithm kurang efektif ketika daftar data elements berukuran besar, karena jumlah pemeriksaan meningkat sebanding dengan banyaknya elemen. Sebaliknya, binary search operation selalu membagi list menjadi dua bagian setiap kali pencarian dilakukan. Di setiap iterasi, algoritma ini bergerak menuju search term jauh lebih cepat dibandingkan strategi linear, sehingga time complexity-nya menjadi O(log n). Meskipun demikian, binary search algorithm memiliki keterbatasan, yaitu hanya dapat digunakan jika list sudah berada dalam keadaan sorted. Karena itu, jika data elements yang tersedia berjumlah sedikit dan masih unsorted, maka linear search algorithm lebih tepat digunakan.

Sementara itu, interpolation search mampu membuang lebih dari separuh bagian list dari search space, sehingga algoritma ini dapat mencapai bagian list yang memuat search term secara lebih efisien. Dalam interpolation search algorithm, midpoint dihitung dengan cara yang memberikan probabilitas lebih tinggi untuk menemukan search term secara lebih cepat. Dalam kondisi rata-rata, average-case time complexity dari interpolation search adalah O(log(log n)), sedangkan worst-case time complexity-nya adalah O(n). Kenyataan tersebut menunjukkan bahwa interpolation search dapat bekerja lebih baik daripada binary search dan sering kali menghasilkan pencarian yang lebih cepat, terutama ketika data memiliki distribusi yang seragam.

Dengan demikian, pemilihan search algorithm sebaiknya disesuaikan dengan sifat list yang digunakan. Jika list berukuran kecil dan unsorted, maka linear search algorithm merupakan pilihan yang tepat. Jika list sudah sorted dan ukurannya tidak terlalu besar, maka binary search algorithm dapat digunakan. Selanjutnya, interpolation search algorithm cocok diterapkan apabila data elements di dalam list terdistribusi secara seragam. Untuk list yang sangat besar, exponential search algorithm dan jump search algorithm dapat dipertimbangkan sebagai alternatif yang lebih sesuai.

BAB 12
Algoritma Pencocokan String

Algoritma pencocokan string mencakup berbagai metode yang sangat populer dalam bidang ilmu komputer. Algoritma ini memiliki peran yang sangat penting dalam banyak aplikasi, seperti pencarian elemen dalam dokumen teks, deteksi plagiarisme, program penyuntingan teks, dan berbagai kebutuhan pemrosesan teks lainnya. Algoritma pencocokan pola digunakan untuk menemukan posisi kemunculan suatu pola atau substring tertentu di dalam teks yang diberikan. Dalam kajian ini, pembahasan difokuskan untuk algoritma pencocokan pola yang umum digunakan, yaitu algoritma brute force, Rabin-Karp, Knuth-Morris-Pratt (KMP), dan Boyer-Moore. Secara umum, pembahasan mengenai algoritma pencocokan string diarahkan untuk pemahaman prinsip kerja setiap metode serta penerapannya dalam penyelesaian masalah pencarian pola teks.

12.1 Notasi Dan Konsep String

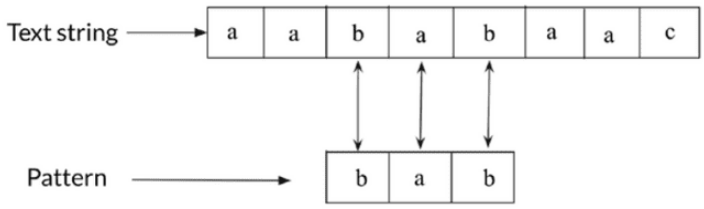
Dalam ilmu komputer, string merupakan urutan karakter yang membentuk data tekstual. Python menyediakan beragam operasi dan fungsi untuk tipe data string, sehingga pengolahan teks dapat dilakukan secara efisien. Sebagai bentuk data tekstual, string memiliki peran penting dalam berbagai aplikasi pemrograman. Contoh sederhana string adalah "algoritma pencarian".

Beberapa konsep dasar yang berkaitan dengan string meliputi **substring, subsequence, prefix**, dan **suffix**. **Substring** adalah rangkaian karakter yang menjadi bagian dari suatu string dan muncul secara berurutan di posisi tertentu. Sebagai contoh, "algoritma" merupakan substring dari string "algoritma pencarian". **Subsequence** adalah rangkaian karakter yang diperoleh dari string asal dengan menghapus beberapa karakter tanpa mengubah urutan kemunculan karakter yang tersisa. Misalnya, "agrtma pcarian" merupakan subsequence dari "algoritma pencarian" karena diperoleh dengan menghilangkan beberapa karakter tertentu. **Prefix** merupakan substring yang terletak di awal string. Sebagai contoh, "algo" dapat dianggap sebagai prefix dari string "algoritma pencarian" karena muncul di bagian awal string tersebut. **Suffix** adalah substring yang terletak di akhir string. Sebagai contoh, "arian" merupakan salah satu suffix dari string "algoritma pencarian". Dalam Python, tersedia fungsi bawaan yang memungkinkan pemeriksaan apakah suatu string diawali atau diakhiri oleh substring tertentu, sehingga identifikasi prefix dan suffix dapat dilakukan dengan mudah. Program yang menjelaskan notasi dan konsep string dapat dilihat di https://colab.research.google.com/drive/1DYyQI2oG1oEWANbSNE864WsQ_a57XvTk?usp=sharing.

12.2 Algoritma Pencocokan Pola

Algoritma pencocokan pola digunakan untuk menentukan posisi indeks tempat suatu string pola (P) cocok dengan string teks (T). Dengan demikian, algoritma ini berfungsi untuk mencari dan mengembalikan indeks kemunculan pola tertentu di dalam sebuah teks. Apabila pola yang dicari tidak ditemukan, maka algoritma akan memberikan hasil bahwa pola tersebut tidak terdapat dalam teks. Sebagai contoh, jika diberikan string teks "algoritma pencarian" dan string pola "pencarian", maka algoritma pencocokan pola akan mengembalikan posisi indeks tempat pola tersebut muncul di dalam string teks (Agarwal, 2022).

Gambar 12.1 memperlihatkan contoh sederhana masalah pencocokan string, yaitu proses menemukan kemunculan suatu pattern di dalam text string. string teks tersusun dari deretan karakter a, a, b, a, b, a, a, c, sedangkan pola yang dicari adalah b, a, b. Panah vertikal menunjukkan bahwa algoritma pencocokan membandingkan karakter-karakter dalam pola dengan karakter-karakter di bagian tertentu



Gambar 12.1: Contoh Pencocokan String (Agarwal, 2022)

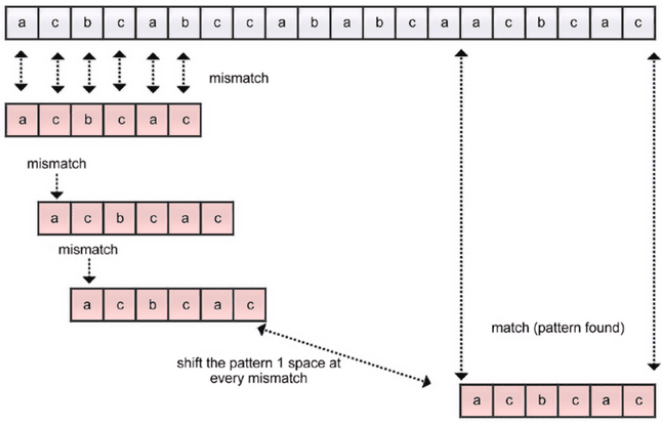
dari teks secara berurutan. Contoh, pola bab cocok dengan sebagian substring terhadap teks, yaitu segmen yang terdiri atas karakter b, a, b. Tujuan utama algoritma pencocokan string adalah menentukan posisi dalam teks tempat pola muncul secara tepat. Pencocokan string dilakukan dengan menyelaraskan pola terhadap teks, lalu memeriksa kesesuaian setiap karakter dengan kedua susunan tersebut. Jika seluruh karakter dalam pola sesuai dengan karakter segmen teks yang dibandingkan, maka pola dinyatakan ditemukan. Sebaliknya, jika terdapat ketidaksesuaian salah satu posisi, algoritma akan menggeser pola ke posisi lain dan mengulangi proses perbandingan. Prinsip dasar pencarian pola yaitu membandingkan karakter demi karakter untuk menemukan lokasi kemunculan pola di dalam string teks.

12.3 Algoritma Brute Force

Algoritma brute force sering disebut juga sebagai pendekatan naif dalam pencocokan pola. Istilah naif menunjukkan bahwa algoritma ini menggunakan cara yang paling dasar dan sederhana. Cara kerja dan pseudocode algoritma bruce forte adalah seluruh kemungkinan kecocokan pola terhadap teks diperiksa satu per satu untuk menemukan posisi kemunculan pola. Karena sifatnya yang sangat sederhana, algoritma ini kurang efisien apabila diterapkan dalam teks yang sangat panjang. Proses kerjanya dimulai dengan membandingkan karakter-karakter dalam pola dan teks secara berurutan. Jika seluruh karakter dalam pola sesuai dengan karakter teks, maka algoritma akan mengembalikan posisi indeks dalam teks tempat karakter pertama pola ditemukan. Namun, apabila ditemukan satu saja karakter yang tidak cocok, pola akan digeser satu posisi ke kanan, lalu proses perbandingan diulang kembali. Proses ini terus berlangsung dengan menggeser pola satu indeks demi satu indeks sampai pola ditemukan atau seluruh kemungkinan posisi telah diperiksa.

```
Algoritma BruteForceStringMatching(T, P):
Input  : T = string teks
         P = string pola
Output : indeks kemunculan pertama pola dalam teks
         atau "pattern not found"

1. n ← panjang(T)
2. m ← panjang(P)

3. Untuk i ← 0 sampai n - m, lakukan:
     a. j ← 0

     b. Selama j < m dan T[i + j] = P[j], lakukan:
          j ← j + 1

     c. Jika j = m, maka:
          Return i

4. Return "pattern not found"
```

Gambar 12.2 memperlihatkan cara kerja algoritma brute force dalam proses pencocokan string. String teks yang digunakan adalah “acbcabccababcaacbcac”, sedangkan pola yang dicari adalah “acbcac”. Algoritma brute force bekerja dengan menempatkan pola di bawah teks mulai dari posisi paling kiri, kemudian membandingkan karakter dalam pola dengan karakter teks satu per satu secara berurutan. Panah-panah vertikal menunjukkan proses perbandingan antar karakter. Selama karakter-karakter yang dibandingkan masih sesuai, proses pemeriksaan dilanjutkan ke karakter berikutnya. Namun, ketika ditemukan karakter yang tidak sesuai (*mismatch*), proses pencocokan terhadap posisi tersebut langsung dihentikan. Setelah terjadi mismatch, pola tidak dipindahkan secara jauh, melainkan hanya digeser satu



Gambar 12.2: Contoh Algoritma Brute Force untuk Pencocokan String (Agarwal, 2022)

posisi ke kanan, lalu proses perbandingan diulang kembali dari awal pola. Mekanisme ini terus dilakukan berulang-ulang untuk setiap kemungkinan posisi dalam teks. Terjadi beberapa kali mismatch sebelum akhirnya pola ditempatkan untuk posisi yang menghasilkan kesesuaian penuh dengan substring dalam teks. Tahap akhir, seluruh karakter dalam pola cocok dengan karakter dalam teks, sehingga diperoleh kondisi match atau pola ditemukan. Algoritma brute force memeriksa seluruh kemungkinan kecocokan pola terhadap teks secara bertahap dan sistematis, tetapi karena pola selalu digeser satu posisi setiap kali terjadi ketidaksesuaian, algoritma ini cenderung kurang efisien untuk teks yang panjang. Program algoritma brute force disajikan di https://colab.research.google.com/drive/1V5GOi4QTN20kIT3J5ry92qboZG0ouh9m?usp=sharing.

12.4 Algoritma Rabin-Karp

Algoritma Rabin-Karp merupakan pengembangan dari pendekatan brute force untuk menemukan posisi suatu pola di dalam string teks. Peningkatan kinerja algoritma ini dicapai dengan mengurangi jumlah perbandingan karakter melalui penggunaan hashing. Fungsi hashing menghasilkan suatu nilai numerik untuk string tertentu, sehingga pencarian pola dapat dilakukan dengan membandingkan nilai hash pola terhadap nilai hash substring teks. Jika nilai hash keduanya tidak sama, pola segera digeser satu posisi ke depan tanpa perlu membandingkan seluruh karakter secara langsung. Dengan cara ini, algoritma Rabin-Karp lebih efisien daripada brute force karena mampu menghindari banyak perbandingan yang tidak diperlukan.

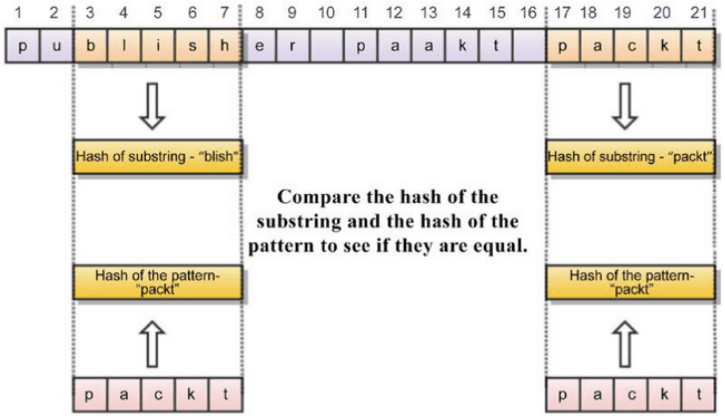
Prinsip dasar algoritma ini adalah bahwa dua string yang memiliki nilai hash sama diasumsikan identik. Namun, dalam praktiknya, dua string yang berbeda dapat menghasilkan nilai hash yang sama. Keadaan ini dikenal sebagai spurious hit dan terjadi akibat collision dalam proses hashing. Oleh karena itu, algoritma Rabin-Karp tidak hanya mengandalkan kecocokan nilai hash. Jika nilai hash pola dan substring teks yang sama, langkah selanjutnya adalah melakukan verifikasi dengan membandingkan karakter-karakter dalam pola dan substring satu per satu. Verifikasi ini bertujuan memastikan bahwa pola benar-benar cocok dengan bagian teks yang diperiksa. Dengan demikian, Rabin-Karp memadukan efisiensi hashing dan ketelitian pencocokan langsung untuk memperoleh hasil pencarian pola yang lebih cepat sekaligus tetap akurat. Cara kerja dan pseudocode algoritma Rabin-Karp yaitu:

1. Algoritma Rabin-Karp diawali dengan tahap praproses sebelum pencarian pola dilakukan.
2. Hitung nilai hash dari pola yang memiliki panjang m, serta nilai hash dari seluruh substring teks yang juga memiliki panjang m.
3. Jika panjang teks dinyatakan dengan n, maka jumlah substring yang mungkin diperiksa adalah sebanyak (n – m + 1).

4. Setelah nilai hash pola dan seluruh substring diperoleh, nilai hash pola dibandingkan dengan nilai hash masing-masing substring teks secara berurutan.
5. Jika nilai hash pola dan nilai hash substring tidak sama, maka pola dianggap tidak cocok dalam posisi tersebut dan pencarian dilanjutkan dengan menggeser pola satu posisi ke kanan.
6. Jika nilai hash pola dan nilai hash substring sama, maka dilakukan verifikasi lanjutan dengan membandingkan karakter-karakter dalam pola dan substring satu per satu.
7. Perbandingan karakter demi karakter tersebut bertujuan memastikan bahwa kecocokan nilai hash benar-benar menunjukkan kecocokan pola dalam teks, bukan sekadar akibat collision.
8. Proses perbandingan hash, pergeseran pola, dan verifikasi karakter dilakukan terus-menerus hingga seluruh bagian teks selesai diperiksa.

```
Algoritma RabinKarp(T, P):
Input  : T = teks
         P = pola
Output : posisi kemunculan pola atau "pattern not found"

Fungsi HitungHash(S)
1. hash ← 0
2. Untuk setiap karakter c dalam S, lakukan:
     hash ← hash + nilai(c)
3. Return hash

Fungsi HitungHashBaru(T, i, m, hashLama)
1. hashBaru ← hashLama - nilai(T[i]) + nilai(T[i + m])
2. Return hashBaru

Algoritma Utama
1. n ← panjang(T)
2. m ← panjang(P)
3. Jika m > n, maka Return "pattern not found"
4. hashPola ← HitungHash(P)
5. hashSubstring ← HitungHash(T[0 .. m-1])

6. Untuk i ← 0 sampai n - m, lakukan:
     a. Jika hashPola = hashSubstring, maka:
          j ← 0
          Selama j < m dan P[j] = T[i + j], lakukan:
               j ← j + 1
          Jika j = m, maka:
               Return i
     b. Jika i < n - m, maka:
          hashSubstring ← HitungHashBaru(T, i, m, hashSubstring)

7. Return "pattern not found"
```

Dengan demikian, algoritma Rabin-Karp bekerja dengan mengombinasikan pencocokan berbasis hash dan verifikasi langsung, sehingga pencarian pola dapat dilakukan dengan lebih efisien.



Gambar 12.3: Contoh Algoritma Rabin-Karp untuk Pencocokan String (Agarwal, 2022)

Gambar 12.3 memperlihatkan prinsip kerja algoritma Rabin-Karp dalam pencocokan string melalui pemanfaatan nilai hash. Pola yang dicari adalah “packt”, sedangkan teks yang diperiksa memuat beberapa substring yang panjangnya sama dengan pola. Algoritma Rabin-Karp tidak langsung membandingkan seluruh karakter pola dengan karakter teks secara berulang, melainkan terlebih dahulu menghitung hash dari pola dan hash dari substring teks yang sedang diperiksa. Perbandingan antara hash of substring “blish” dengan hash of the pattern “packt”, serta antara hash of substring “packt” dengan hash of the pattern “packt”. Tahap awal pencarian dilakukan melalui pencocokan nilai hash, bukan melalui pencocokan karakter secara langsung.

Apabila nilai hash substring dan nilai hash pola tidak sama, maka substring tersebut dapat langsung dianggap tidak cocok, sehingga algoritma dapat melanjutkan pencarian ke posisi berikutnya tanpa perlu memeriksa karakter satu per satu. Sebaliknya, apabila nilai hash keduanya sama, maka ada kemungkinan bahwa pola benar-benar ditemukan di dalam teks. Dalam contoh yang ditampilkan, substring “blish” menghasilkan hash yang berbeda dari pola “packt”, sehingga tidak dianggap sebagai kecocokan. Adapun substring “packt” menghasilkan hash yang sama dengan pola “packt”, sehingga menjadi kandidat kecocokan. Kondisi seperti inilah algoritma Rabin-Karp biasanya melanjutkan proses dengan verifikasi karakter demi karakter untuk memastikan bahwa kesamaan hash tersebut benar-benar menunjukkan kemunculan pola dalam teks, bukan sekadar akibat collision.

Keunggulan utama algoritma Rabin-Karp yaitu mampu mengurangi jumlah perbandingan langsung antara pola dan teks. Dengan membandingkan nilai hash terlebih dahulu, banyak substring yang tidak relevan dapat dieliminasi secara cepat. Strategi ini menjadikan Rabin-Karp lebih efisien daripada pendekatan brute force, terutama ketika teks yang diperiksa cukup panjang dan jumlah substring yang harus dianalisis sangat banyak. Algoritma Rabin-Karp bekerja melalui dua tahapan utama yaitu pencocokan hash sebagai penyaring awal dan verifikasi karakter sebagai tahap konfirmasi akhir.

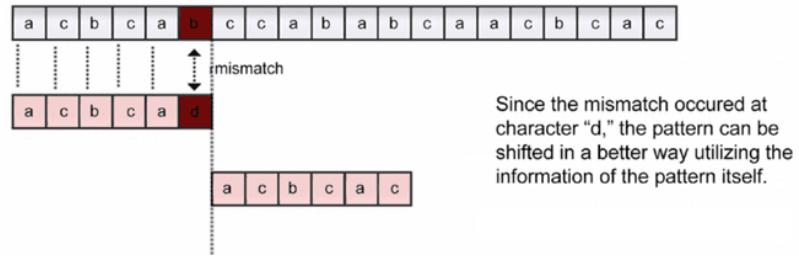
Program algoritma Rabin-Karp disajikan di https://colab.research.google.com/drive/1WbJpXBw6PYSpZNs0QD5Bp-sR0N9SloZA?usp=sharing.

12.5 Algoritma Knuth-Morris-Pratt (KMP)

Algoritma Knuth-Morris-Pratt atau KMP merupakan algoritma pencocokan pola, dimana bagian pola yang saling bertumpang tindih dapat dimanfaatkan untuk segera menentukan seberapa jauh pola harus digeser ketika terjadi ketidaksesuaian. Awalnya, dihitung sebuah prefix function yang menunjukkan banyaknya pergeseran pola yang diperlukan setiap kali ditemukan mismatch. Dengan demikian, KMP melakukan tahap praproses terhadap pola untuk menghindari perbandingan yang tidak perlu. Prefix function dimanfaatkan untuk memperkirakan pergeseran pola yang tepat saat pencarian berlangsung, sehingga pola dapat dicocokkan kembali dengan teks tanpa harus memulai pemeriksaan dari awal secara keseluruhan. Karena itulah, algoritma KMP dikenal efisien, sebab mampu meminimalkan jumlah perbandingan antara pola dan string teks.

Motivasi penggunaan algoritma KMP dapat dipahami dari situasi ketika mismatch terjadi setelah beberapa karakter awal pola berhasil dicocokkan dengan teks. Dalam kondisi seperti itu, informasi yang tersimpan dalam prefix function memungkinkan algoritma mengetahui bahwa sebagian karakter yang sudah cocok sebelumnya tidak perlu diperiksa kembali. Sebagai contoh, jika ketidaksesuaian terjadi di posisi keenam setelah lima karakter pertama cocok, maka berdasarkan nilai prefix function dapat ditentukan seberapa jauh pola perlu digeser. Apabila karakter yang menyebabkan mismatch belum pernah muncul sebelumnya dalam pola, maka pola dapat langsung digeser beberapa posisi sekaligus tanpa membandingkan ulang karakter-karakter yang sudah dipastikan tidak mungkin cocok. Dengan strategi ini,

KMP mampu melewati banyak perbandingan yang tidak diperlukan, sehingga proses pencarian pola menjadi lebih cepat dan lebih efisien dibandingkan algoritma brute force.



Gambar 12.4: Contoh Algoritma KMP (Agarwal, 2022)

Gambar 12.4 memperlihatkan prinsip dasar kerja algoritma KMP dalam menyelesaikan masalah pencocokan string secara lebih efisien. Pola mula-mula disejajarkan dengan bagian awal string teks, kemudian karakter-karakter dalam pola dibandingkan dengan karakter-karakter teks secara berurutan dari kiri ke kanan. Garis putus-putus vertikal menunjukkan pasangan karakter yang sedang dibandingkan, sedangkan blok berwarna menegaskan posisi karakter yang menjadi pusat perhatian dalam proses pencocokan. Lima karakter awal pola telah berhasil cocok dengan lima karakter teks. Namun, karakter berikutnya terjadi mismatch, yaitu ketika karakter "d" dalam pola dibandingkan dengan karakter "b" di teks. Titik ketidaksesuaian ini ditandai secara jelas untuk menunjukkan bahwa proses pencocokan dalam posisi tersebut tidak dapat dilanjutkan.

Keunikan yang perlu diperhatikan adalah saat algoritma KMP merespons terjadinya mismatch. Jika digunakan pendekatan sederhana seperti brute force, pola umumnya hanya digeser satu posisi ke kanan, lalu proses perbandingan dimulai kembali dari awal pola. Akan tetapi, ilustrasi ini menunjukkan bahwa KMP tidak melakukan langkah tersebut secara naif. Sebaliknya, algoritma memanfaatkan informasi internal pola itu sendiri, khususnya hubungan antara prefix dan suffix yang telah diproses sebelumnya, untuk menentukan pergeseran yang lebih tepat. Dengan kata lain, karakter-karakter awal yang telah terbukti cocok tidak seluruhnya dibandingkan ulang, karena sebagian informasi kecocokan sudah tersimpan dalam struktur pola. Pernyataan di sisi kanan gambar menegaskan bahwa pola dapat digeser dengan cara yang lebih baik karena mismatch terjadi di karakter "d", dan karakter tersebut tidak memberikan dukungan untuk mempertahankan kecocokan dengan bagian pola sebelumnya.

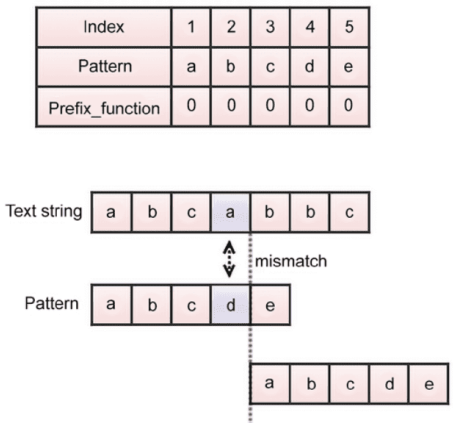
Efisiensi KMP yaitu kemampuannya menghindari perbandingan berulang yang tidak perlu. Ketika mismatch terjadi setelah beberapa karakter awal berhasil cocok, KMP tidak mengulang pencocokan dari posisi pertama pola, melainkan langsung menggeser pola ke posisi yang masih mungkin menghasilkan kecocokan berdasarkan prefix function atau failure function. Dengan mekanisme ini, pergeseran pola menjadi lebih besar dan lebih terarah dibandingkan brute force. Akibatnya, karakter-karakter teks yang sudah pernah dibandingkan tidak diperiksa kembali secara sia-sia. Kekuatan utama algoritma KMP terletak dalam pemanfaatan struktur pola untuk mempercepat pencarian, sehingga proses pencocokan string dapat berlangsung lebih efisien, terutama ketika teks yang diproses berukuran panjang.

⌘ Funtion Prefix

Prefix function, yang juga dikenal sebagai failure function, merupakan komponen penting dalam algoritma pencocokan pola karena berfungsi menemukan pola di dalam pola itu sendiri. Fungsi ini menentukan sejauh mana hasil perbandingan sebelumnya masih dapat dimanfaatkan kembali ketika terjadi mismatch, khususnya apabila di dalam pola terdapat pengulangan tertentu. Untuk setiap posisi yang mengalami ketidaksesuaian, prefix function menghasilkan suatu nilai yang menunjukkan seberapa jauh pola harus digeser. Dengan demikian, pergeseran pola tidak dilakukan secara sembarang, melainkan berdasarkan informasi internal yang sudah terkandung dalam pola tersebut.

Cara kerja prefix function dapat dipahami melalui contoh pola yang seluruh karakternya berbeda. Nilai prefix function untuk setiap posisi adalah 0, karena tidak terdapat bagian awal pola yang juga muncul kembali sebagai bagian akhir dari segmen yang telah diperiksa. Akibatnya, ketika terjadi mismatch, tidak ada hasil perbandingan sebelumnya yang dapat digunakan kembali. Pola harus langsung digeser sejauh jumlah karakter yang telah dibandingkan sampai titik ketidaksesuaian tersebut. Sebagai ilustrasi, jika digunakan pola abcde yang terdiri atas karakter-karakter berbeda seluruhnya, lalu mismatch terjadi di karakter keempat, maka karena nilai prefix function bernilai 0, pola harus digeser sejauh banyaknya karakter yang telah dibandingkan hingga posisi itu. Hal ini menunjukkan bahwa semakin kecil nilai prefix function, semakin sedikit informasi pencocokan sebelumnya yang dapat dimanfaatkan kembali.

Gambar 12.5 menjelaskan peran prefix function dalam algoritma KMP ketika terjadi



Gambar 12.5: Contoh Function Prefix dalam Algoritma KMP (Agarwal, 2022)

ketidaksesuaian antara pola dan teks. Bagian atas gambar ditampilkan tabel yang memuat indeks, susunan karakter pola a b c d e, serta nilai prefix_function untuk setiap posisi. Seluruh nilai prefix function bernilai 0, yang menunjukkan bahwa pola abcde tidak memiliki pengulangan struktur internal antara awalan dan akhiran untuk bagian-bagian yang telah diperiksa. Dengan kata lain, tidak ada bagian awal pola yang dapat dimanfaatkan kembali ketika proses pencocokan mengalami mismatch. Informasi inilah yang menjadi dasar dalam menentukan besar pergeseran pola.

Bagian tengah gambar ditunjukkan proses pencocokan antara text string dan pattern. Teks yang digunakan adalah a b c a b b c, sedangkan pola yang dibandingkan adalah a b c d e. Tiga karakter awal, yaitu a, b, dan c, berhasil dicocokkan dengan benar. Namun, karakter berikutnya terjadi mismatch, yaitu ketika karakter d dalam pola dibandingkan dengan karakter a dalam teks. Titik ketidaksesuaian ini ditandai dengan garis putus-putus vertikal dan label mismatch. Kondisi tersebut menandakan bahwa pencocokan di posisi tersebut tidak dapat diteruskan.

Karena nilai prefix function untuk posisi tersebut adalah 0, algoritma mengetahui bahwa tidak ada kecocokan parsial sebelumnya yang dapat digunakan kembali. Akibatnya, pola harus digeser sejauh jumlah karakter yang telah dibandingkan hingga titik mismatch tersebut. Pola abcde dipindahkan ke posisi baru yang lebih ke kanan. Pergeseran ini dilakukan tanpa mencoba mempertahankan sebagian kecocokan yang telah diperoleh, sebab struktur pola memang tidak menyediakan overlap yang dapat dimanfaatkan. Nilai prefix function bernilai 0 menyebabkan pola digeser secara penuh sesuai banyaknya karakter yang telah diperiksa sampai titik ketidaksesuaian.

Gambar 12.6 memperlihatkan proses pembentukan prefix function untuk algoritma KMP untuk pola a b c a b b c a b. Ilustrasi ini disusun dalam empat tabel yang menunjukkan perkembangan nilai prefix function secara bertahap dari indeks awal hingga indeks akhir pola. Baris pertama untuk setiap tabel memuat indeks, baris kedua memuat karakter pola, dan baris ketiga menunjukkan nilai prefix function yang dihitung untuk setiap posisi. Prefix function sendiri menyatakan panjang awalan terpanjang dari pola

Index	1	2	3	4	5	6	7	8	9
Pattern	a	b	c	a	b	b	c	a	b
Prefix_function	0	0	0						

Index	1	2	3	4	5	6	7	8	9
Pattern	a	b	c	a	b	b	c	a	b
Prefix_function	0	0	0	1					

Index	1	2	3	4	5	6	7	8	9
Pattern	a	b	c	a	b	b	c	a	b
Prefix_function	0	0	0	1	2				

Index	1	2	3	4	5	6	7	8	9
Pattern	a	b	c	a	b	b	c	a	b
Prefix_function	0	0	0	1	2	0	0	1	2

Gambar 12.6: Contoh Lain Function Prefix dalam Algoritma KMP (Agarwal, 2022)

yang sekaligus juga menjadi akhiran dalam bagian pola yang telah diperiksa sampai posisi tertentu. Dengan demikian, nilai ini menggambarkan sejauh mana kecocokan parsial di dalam pola dapat dimanfaatkan kembali apabila terjadi mismatch saat pencocokan dengan teks.

Tahap awal, nilai prefix function untuk tiga karakter pertama, yaitu a, b, dan c, seluruhnya bernilai 0. Hal ini menunjukkan bahwa segmen awal pola belum terdapat bagian awalan yang sama dengan bagian akhir segmen tersebut. Dengan kata lain, untuk substring pendek seperti a, ab, dan abc, belum ada overlap internal yang dapat dimanfaatkan. Selanjutnya, tabel kanan atas, ketika karakter keempat a dipertimbangkan, nilai prefix function berubah menjadi 1. Nilai ini menunjukkan bahwa substring abca, terdapat satu karakter awalan yang juga muncul sebagai akhiran, yaitu a. Jadi, awalan terpanjang yang sama dengan akhiran segmen tersebut memiliki panjang satu.

Perkembangan berikutnya ditunjukkan tabel kiri bawah. Ketika karakter kelima b diproses, nilai prefix function menjadi 2. Hal ini berarti substring abcab memiliki overlap internal berupa awalan ab yang juga muncul sebagai akhiran. Dengan demikian, dua karakter awal pola dapat dianggap berulang kembali di bagian akhir segmen yang telah diperiksa. Akan tetapi, karakter keenam b, nilai prefix function kembali menjadi 0. Kondisi ini menunjukkan bahwa setelah karakter tersebut dimasukkan, tidak lagi ditemukan awalan pola yang sesuai dengan akhiran segmen sampai posisi itu. Dengan kata lain, overlap yang sebelumnya terbentuk terputus di posisi tersebut.

Tabel kanan bawah, proses perhitungan dilanjutkan hingga karakter terakhir pola. Untuk karakter ketujuh c, nilai prefix function tetap 0, yang berarti belum terbentuk kembali overlap baru. Namun, karakter kedelapan a, nilai prefix function menjadi 1, dan karakter kesembilan b, nilainya meningkat menjadi 2. Perubahan ini menunjukkan bahwa bagian akhir pola kembali muncul kecocokan parsial dengan awalan pola, yaitu awalan a lalu ab. Dengan demikian, pola abcabbcab memiliki struktur internal yang memungkinkan sebagian awalan pola berulang sebagai akhiran dalam beberapa posisi tertentu, meskipun tidak secara konsisten di seluruh posisi.

Fungsi utama prefix function dalam algoritma KMP yaitu merekam informasi tentang struktur pengulangan di dalam pola. Informasi tersebut sangat penting ketika terjadi mismatch dalam proses pencocokan string, karena algoritma tidak perlu kembali membandingkan pola dari awal. Sebaliknya, nilai prefix function dapat langsung digunakan untuk menentukan posisi baru pola berdasarkan panjang overlap yang sudah diketahui. Oleh karena itu, semakin kaya pola internal yang dimiliki suatu string, semakin besar peluang algoritma KMP untuk menghindari perbandingan ulang yang tidak perlu. Dalam contoh yang ditampilkan, nilai prefix function yang bervariasi antara 0, 1, dan 2 menunjukkan adanya kecocokan parsial tertentu di dalam pola, yang nantinya dapat dimanfaatkan untuk mempercepat proses pencarian pola di dalam teks.

Pergeseran pola dalam algoritma KMP ditentukan berdasarkan overlap yang terdapat di dalam pola itu sendiri. Cara kerja dan pseudocode algoritma KMP yaitu:

1. Melakukan praproses terhadap pola dengan menghitung prefix function terlebih dahulu.

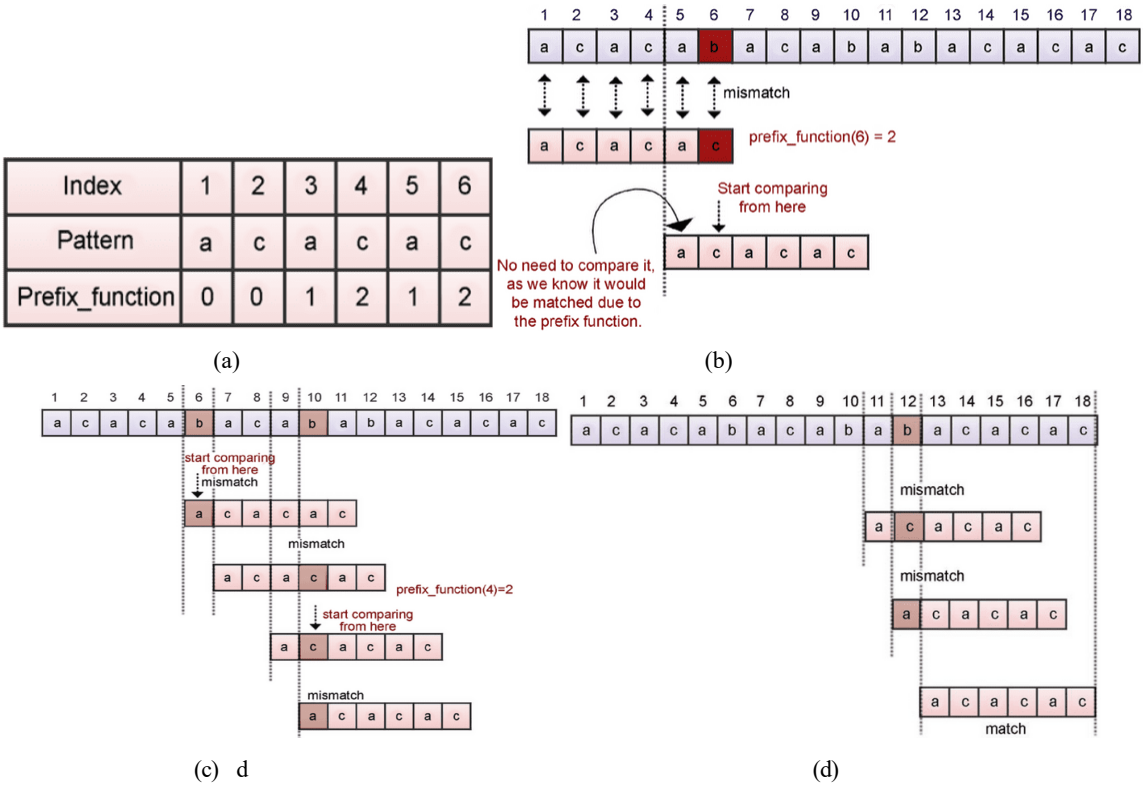
2. Setelah itu, diinisialisasi sebuah penghitung q yang menyatakan jumlah karakter pola yang telah berhasil dicocokkan.
3. Proses pencocokan dimulai dengan membandingkan karakter pertama dalam pola dengan karakter pertama dalam string teks.
4. Jika kedua karakter tersebut cocok, maka nilai penghitung q dinaikkan, demikian pula posisi teks, lalu pencocokan dilanjutkan ke karakter berikutnya.
5. Jika terjadi mismatch, maka nilai q tidak dikembalikan ke nol secara langsung, tetapi diperbarui menggunakan nilai prefix function yang telah dihitung sebelumnya.
6. Dengan cara tersebut, algoritma dapat menentukan posisi pergeseran pola yang tepat tanpa harus mengulang seluruh perbandingan dari awal.
7. Proses pencarian terus dilanjutkan sampai seluruh karakter teks selesai diperiksa atau sampai pola berhasil ditemukan.
8. Jika seluruh karakter dalam pola berhasil dicocokkan dengan bagian tertentu dari teks, maka algoritma mengembalikan posisi kemunculan pola tersebut dalam teks.
9. Setelah satu kecocokan ditemukan, pencarian dapat tetap dilanjutkan untuk menemukan kemungkinan kemunculan pola berikutnya.

```
Algoritma KMP(T, P):
Input  : T = string teks
         P = string pola
Output : posisi kemunculan pola dalam teks

Fungsi ComputePrefixFunction(P)
1. m ← panjang(P)
2. Buat array prefix[0 .. m-1]
3. prefix[0] ← 0
4. k ← 0
5. Untuk q ← 1 sampai m-1, lakukan:
   a. Selama k > 0 dan P[k] ≠ P[q], lakukan:
      k ← prefix[k-1]
   b. Jika P[k] = P[q], maka:
      k ← k + 1
   c. prefix[q] ← k
6. Return prefix

Algoritma Utama
1. n ← panjang(T)
2. m ← panjang(P)
3. prefix ← ComputePrefixFunction(P)
4. q ← 0

5. Untuk i ← 0 sampai n-1, lakukan:
   a. Selama q > 0 dan P[q] ≠ T[i], lakukan:
      q ← prefix[q-1]

   b. Jika P[q] = T[i], maka:
      q ← q + 1

   c. Jika q = m, maka:
      cetak atau simpan posisi kemunculan pola, yaitu i - m + 1
      q ← prefix[q-1]

6. Jika tidak ada kecocokan, kembalikan "pattern not found"
```

Gambar 12.7 menampilkan ilustrasi algoritma KMP dalam melakukan pencocokan pola string teks. Ilustrasi ini dibagi menjadi empat bagian, yaitu (a), (b), (c), dan (d), yang bersama-sama menunjukkan bagaimana prefix function dimanfaatkan untuk menghindari perbandingan ulang yang tidak diperlukan. Pola yang digunakan adalah acacac, sedangkan teks yang diperiksa terdiri atas deretan karakter yang lebih panjang. Inti dari gambar ini adalah menunjukkan bahwa ketika terjadi mismatch, algoritma KMP tidak menggeser pola secara naif satu posisi ke kanan, melainkan memanfaatkan struktur internal pola untuk menentukan posisi pergeseran yang lebih tepat.

Bagian (a) memperlihatkan tabel pembentukan prefix function untuk pola acacac. Tabel tersebut memuat indeks 1 sampai 6, karakter pola, dan nilai prefix function yang dihasilkan, yaitu 0, 0, 1, 2, 1, 2. Nilai ini menunjukkan panjang awalan terpanjang dari pola yang sekaligus juga menjadi akhiran dalam bagian pola hingga posisi tertentu. Sebagai contoh, posisi ke-3 nilai prefix function bernilai 1 karena substring aca memiliki awalan dan akhiran yang sama-sama berupa a. Posisi ke-4 nilainya menjadi 2

karena substring acac memiliki overlap internal berupa ac. Tabel ini sangat penting karena menjadi dasar bagi algoritma KMP dalam menentukan berapa jauh pola harus digeser ketika terjadi ketidaksesuaian.



Gambar 12.7: Ilustrasi Algoritma KMP (Agarwal, 2022)

Bagian (b) menunjukkan tahap awal pencocokan antara pola dan teks. Dari gambar terlihat bahwa lima karakter awal pola telah berhasil cocok dengan segmen teks yang bersesuaian. Ketidaksesuaian terjadi di karakter keenam, yaitu ketika karakter c dalam pola dibandingkan dengan karakter b di teks. Titik inilah nilai prefix_function(6) = 2 digunakan. Nilai tersebut menunjukkan bahwa setelah lima karakter awal berhasil dicocokkan, masih terdapat bagian pola sepanjang dua karakter yang sebenarnya tetap relevan untuk dipertahankan. Oleh sebab itu, pola tidak perlu digeser kembali ke awal. Perbandingan dapat langsung dimulai dari posisi baru tertentu, dan beberapa karakter tidak perlu diperiksa ulang karena hasilnya sudah dapat dipastikan dari informasi prefix function. Inilah inti efisiensi KMP yaitu hasil kecocokan parsial sebelumnya tidak dibuang, melainkan dimanfaatkan kembali.

Bagian (c) memperlihatkan kelanjutan proses ketika setelah pergeseran pertama masih terjadi mismatch lagi. Pola kembali ditempatkan di posisi baru yang ditentukan bukan oleh pergeseran satu langkah, tetapi oleh nilai prefix function dari bagian pola yang telah cocok sebelumnya. Tahap ini ditunjukkan bahwa prefix_function(4) = 2, sehingga pola kembali digeser ke posisi yang lebih tepat dan pencocokan dimulai dari titik yang relevan. Gambar ini menegaskan bahwa KMP bekerja secara berlapis: setiap kali terjadi mismatch, algoritma memeriksa berapa banyak bagian pola yang masih mungkin cocok berdasarkan nilai prefix function, lalu melanjutkan pencocokan dari sana. Algoritma tidak mengulang seluruh pemeriksaan dari karakter pertama pola.

Bagian (d) memperlihatkan tahap akhir ketika pola akhirnya berhasil dicocokkan seluruhnya dengan bagian teks tertentu. Setelah beberapa kali pergeseran yang ditentukan oleh prefix function, pola acacac akhirnya sejajar dengan segmen teks yang sesuai, sehingga seluruh karakter dalam pola cocok dengan karakter teks. Kondisi ini ditandai dengan label match di bagian bawah ilustrasi. Tahap ini menunjukkan bahwa meskipun sebelumnya sempat terjadi beberapa mismatch, algoritma tetap mampu

menemukan kecocokan penuh tanpa harus membandingkan ulang seluruh karakter dari awal setiap kali terjadi ketidaksesuaian. Program algoritma KMP disajikan di https://colab.research.google.com/drive/1mlcL_dwJVFBiJkQEO-5GLLZWJ1ZShjB_?usp=sharing.

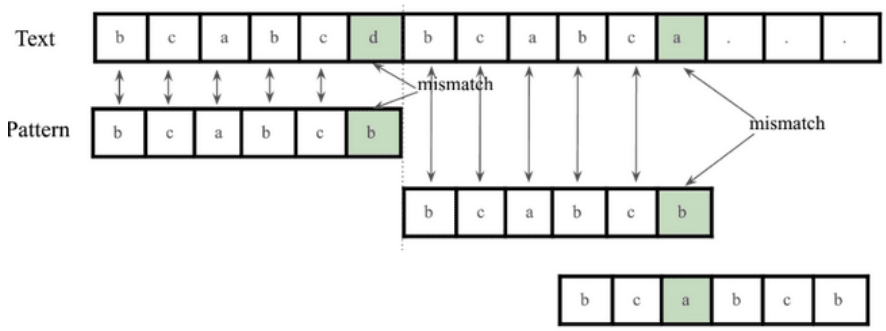
12.6 Algoritma Boyer-Moore

Algoritma Boyer-Moore merupakan salah satu algoritma pencocokan pola yang dirancang untuk mempercepat proses pencarian pola di dalam teks. Seperti algoritma KMP, metode ini bertujuan mengurangi perbandingan yang tidak perlu agar pencarian menjadi lebih efisien. Pola tetap digeser dari kiri ke kanan sepanjang teks, tetapi proses perbandingan karakter dilakukan dari kanan ke kiri. Cara ini berbeda dari KMP, yang membandingkan karakter dari kiri ke kanan. Untuk menentukan seberapa jauh pola harus digeser ketika terjadi ketidaksesuaian, Boyer-Moore menggunakan dua aturan utama, yaitu good suffix dan bad character shift. Kedua aturan tersebut membantu algoritma melewati bagian-bagian teks yang diperkirakan tidak mungkin menghasilkan kecocokan. Pergeseran pola dipilih berdasarkan nilai yang paling besar dari kedua aturan tersebut, sehingga pencarian dapat berlangsung lebih cepat. Oleh karena itu, algoritma Boyer-Moore dikenal sangat efektif, terutama ketika digunakan dalam teks yang panjang, karena mampu menghindari banyak perbandingan yang tidak diperlukan.

Memahami Algoritma Boyer-Moore

Algoritma Boyer-Moore melakukan pencocokan pola terhadap teks dengan cara membandingkan karakter dari kanan ke kiri. Artinya, pemeriksaan dimulai dari karakter terakhir dalam pola yang disejajarkan dengan karakter tertentu dalam teks. Jika karakter di ujung pola tidak sesuai dengan karakter teks, proses pencocokan tidak perlu dilanjutkan ke seluruh karakter lainnya. Dalam kondisi tersebut, pola dapat langsung digeser ke posisi baru yang lebih memungkinkan untuk menghasilkan kecocokan. Besarnya pergeseran ditentukan oleh karakter yang menyebabkan ketidaksesuaian. Jika karakter teks yang tidak cocok itu tidak muncul sama sekali di dalam pola, maka pola dapat digeser sejauh panjang pola secara penuh. Sebaliknya, jika karakter tersebut masih terdapat di bagian lain pola, maka pola digeser

```
Algoritma BoyerMoore(T, P):

Input :
      T = teks dengan panjang n
      P = pola dengan panjang m
Output :
      posisi kemunculan pola dalam teks, atau -1 jika tidak ditemukan

1. Bangun tabel BadCharacter untuk pola P
2. Bangun tabel GoodSuffix untuk pola P

3. s ← 0
  // s adalah posisi awal pergeseran pola terhadap teks

4. while s ≤ n - m do

5.    j ← m - 1
     // mulai membandingkan dari karakter terakhir pola

6.    while j ≥ 0 dan P[j] = T[s + j] do
7.       j ← j - 1
8.    end while

9.    if j < 0 then
10.      return s
         // pola ditemukan di posisi s
11.   else
12.      bcShift ← nilai pergeseran dari tabel BadCharacter
13.      gsShift ← nilai pergeseran dari tabel GoodSuffix
14.      s ← s + maksimum(bcShift, gsShift)
15.   end if

16. end while

17. return -1
```



Gambar 12.8: Contoh Algoritma Boyer-Moore (Agarwal, 2022)

sehingga karakter yang tidak cocok itu disejajarkan dengan kemunculan karakter yang sama di dalam pola.

Algoritma Boyer-Moore juga memperhatikan bagian pola yang telah berhasil cocok, khususnya suffix yang sudah sesuai dengan teks. Informasi ini digunakan untuk menentukan pergeseran pola yang lebih tepat, sehingga perbandingan yang tidak perlu dapat dihindari. Dengan demikian, inti utama algoritma Boyer-Moore adalah membuat pola "melompat" di sepanjang teks berdasarkan informasi yang diperoleh dari ketidaksesuaian dan kecocokan parsial sebelumnya. Pendekatan ini jauh lebih efisien daripada membandingkan setiap karakter pola dengan teks secara berurutan untuk setiap posisi, karena jumlah perbandingan dapat dikurangi secara signifikan, terutama teks yang panjang.

Gambar 12.8 memperlihatkan cara kerja algoritma Boyer-Moore dalam mencocokkan pola terhadap teks dengan memulai perbandingan dari karakter paling kanan pola. Pola yang dicari adalah "bcabcb", sedangkan teks memuat deretan karakter yang lebih panjang. Tahap pertama ditunjukkan ketika pola disejajarkan dengan bagian awal teks, lalu karakter-karakter dibandingkan dari kanan ke kiri. Beberapa karakter terakhir pola tampak sesuai dengan karakter teks, tetapi di titik tertentu terjadi mismatch, yaitu ketika karakter terakhir pola yang berwarna hijau, "b", dibandingkan dengan karakter teks "d" yang juga diberi penanda warna. Ketidaksesuaian ini menunjukkan bahwa pola tidak mungkin cocok dalam penjajaran tersebut, sehingga algoritma tidak perlu melanjutkan perbandingan ke seluruh karakter lainnya.

Sesudah mismatch ditemukan, Boyer-Moore tidak menggeser pola satu posisi demi satu posisi seperti pendekatan brute force. Sebaliknya, pola langsung digeser beberapa langkah ke kanan. Pergeseran ini ditentukan oleh informasi tentang karakter yang menyebabkan mismatch. Karakter "d" dalam teks tidak memberikan kecocokan dengan karakter terakhir pola, sehingga pola dapat dipindahkan melewati beberapa posisi yang diperkirakan tidak mungkin menghasilkan kecocokan. Hasil pergeseran pertama ditunjukkan pola kedua di bagian tengah gambar. Setelah pola berada di posisi baru, perbandingan kembali dilakukan dari kanan ke kiri. Akan tetapi, saat penjajaran ini masih terjadi mismatch, sehingga pola harus digeser lagi.

Pola ketiga, pola kembali dipindahkan ke kanan berdasarkan informasi mismatch sebelumnya, bukan dengan mengulang pencocokan dari awal setiap posisi. Hal ini menunjukkan bahwa Boyer-Moore memanfaatkan karakter yang tidak cocok untuk menentukan besar pergeseran yang lebih efisien. Dengan demikian, algoritma dapat melewati sejumlah posisi teks tanpa harus membandingkan setiap karakter satu per satu. Strategi inilah yang menjadikan Boyer-Moore lebih cepat dibandingkan metode pencocokan sederhana, terutama ketika pola cukup panjang dan teks yang diperiksa berukuran besar.

Algoritma Boyer-Moore mempunyai dua ciri utama. Pertama, pencocokan dilakukan dari kanan ke kiri, sehingga mismatch di bagian akhir pola dapat segera digunakan untuk mengambil keputusan. Kedua, pola digeser berdasarkan informasi karakter yang tidak cocok, sehingga perbandingan yang tidak perlu dapat dihindari. Efisiensi Boyer-Moore yaitu membuat pola "melompat" di sepanjang teks secara terarah.

Mekanisme tersebut menjadikan Boyer-Moore sangat efektif untuk pencarian pola dalam teks yang panjang.

Algoritma Boyer-Moore menggunakan dua heuristik utama untuk menentukan pergeseran pola yang paling jauh ketika terjadi mismatch, yaitu **bad character heuristic** dan **good suffix heuristic**. Saat ketidaksesuaian ditemukan, masing-masing heuristik memberikan saran mengenai seberapa jauh pola dapat digeser. Selanjutnya, algoritma Boyer-Moore memilih nilai pergeseran yang paling besar di antara kedua saran tersebut. Dengan cara ini, pola dapat dipindahkan lebih jauh dalam satu langkah, sehingga jumlah perbandingan yang tidak perlu dapat dikurangi. Penjelasan mengenai kedua heuristik tersebut, baik bad character maupun good suffix, biasanya disertai dengan contoh agar mekanisme pergeseran pola dapat dipahami secara lebih jelas.

Bad Character Heuristic

Bad character heuristic merupakan salah satu teknik yang digunakan dalam algoritma Boyer–Moore untuk mempercepat proses pencarian pola dalam sebuah teks. Perbandingan antara pola (pattern) dan teks (text string) dilakukan dari arah kanan ke kiri. Proses pencocokan dimulai dari karakter terakhir dalam pola dengan karakter yang bersesuaian dengan teks. Apabila kedua karakter tersebut sama, maka perbandingan dilanjutkan ke karakter sebelumnya secara berurutan hingga seluruh karakter pola berhasil dicocokkan atau ditemukan ketidaksesuaian (mismatch).

Karakter teks yang menyebabkan terjadinya ketidaksesuaian tersebut disebut sebagai bad character. Ketika terjadi mismatch, algoritma Boyer–Moore akan menggeser posisi pola terhadap teks dengan memanfaatkan aturan bad character heuristic. Pergeseran pola dilakukan berdasarkan beberapa kondisi yaitu:

1. Apabila karakter teks yang menyebabkan mismatch tidak terdapat dalam pola, maka pola dapat langsung digeser sehingga posisinya berada setelah karakter tersebut.
2. Apabila karakter yang menyebabkan mismatch muncul satu kali dalam pola, maka pola digeser sehingga karakter tersebut dalam pola sejajar dengan karakter yang sama dengan teks.
3. Apabila karakter tersebut muncul lebih dari satu kali dalam pola, maka pola digeser dengan jarak minimum yang memungkinkan agar salah satu kemunculan karakter tersebut dalam pola sejajar dengan karakter teks yang menyebabkan mismatch.

Algoritma BuildBadCharacter(P):

```
Input  : P = pola dengan panjang m
Output : tabel BadCharacter

1. Untuk setiap karakter c dalam alfabet lakukan
2.    BadCharacter[c] ← -1
3. EndFor

4. Untuk i ← 0 sampai m - 1 lakukan
5.    BadCharacter[P[i]] ← i
6. EndFor

7. return BadCharacter
```

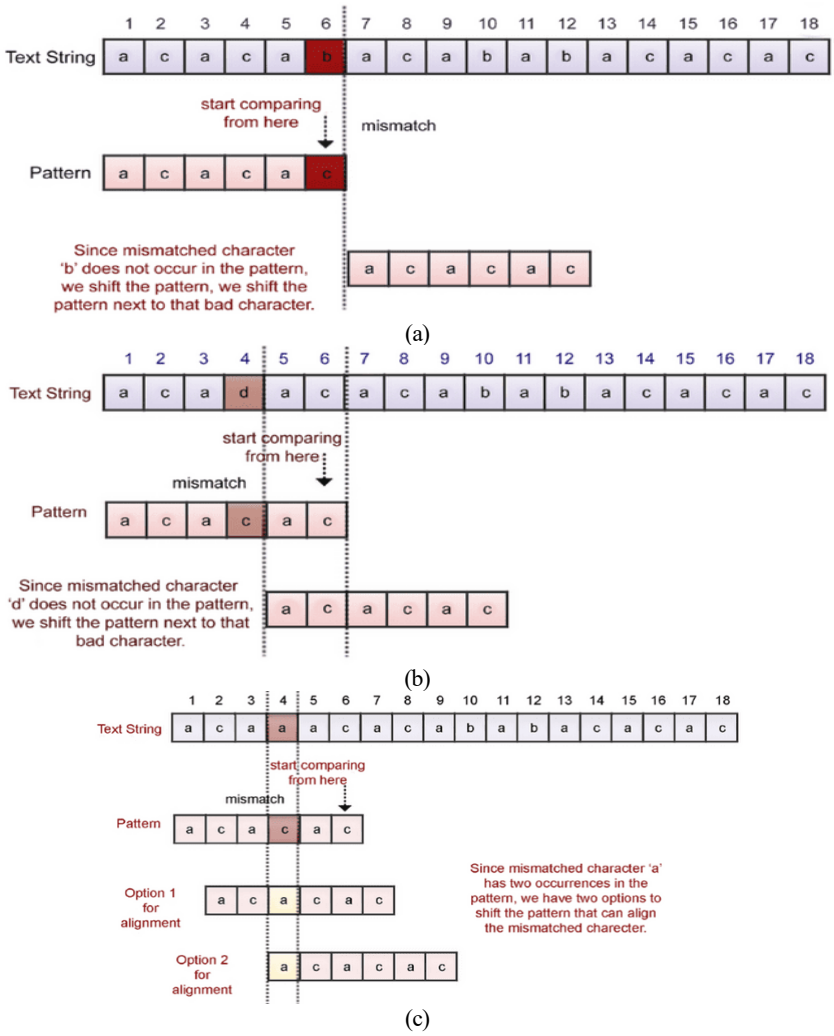
Pendekatan ini memungkinkan algoritma Boyer–Moore melakukan lompatan pergeseran yang lebih besar dibandingkan metode pencarian string sederhana, sehingga meningkatkan efisiensi proses pencarian pola dalam teks.

Gambar 12.9 menampilkan contoh penerapan heuristik bad character dalam algoritma Boyer–Moore untuk melakukan pencarian pola di dalam suatu teks. Secara umum, algoritma Boyer–Moore mencocokkan pola dengan teks dengan membandingkan karakter secara berurutan dari arah kanan ke kiri, dimulai dari karakter terakhir pola. Jika terjadi ketidaksesuaian (mismatch), algoritma tidak langsung memindahkan pola satu posisi seperti metode pencarian string yang lebih sederhana. Sebaliknya, pola digeser dengan mempertimbangkan karakter teks yang menyebabkan ketidaksesuaian tersebut. Ilustrasi ini menggambarkan tiga kondisi yang menunjukkan bagaimana pola digeser berdasarkan prinsip bad character heuristic.

Bagian (a) menggambarkan situasi ketika karakter teks yang memicu ketidaksesuaian tidak ditemukan dalam pola. Proses perbandingan dimulai dari karakter paling akhir pola dengan karakter teks yang sejajar dengannya. Ketika ketidaksesuaian terjadi dalam karakter 'b', sementara karakter tersebut tidak terdapat dalam pola "acacac", maka pola dapat langsung digeser melewati posisi karakter tersebut. Dengan demikian, posisi pola dipindahkan sehingga berada setelah karakter teks yang menyebabkan ketidaksesuaian. Pergeseran yang relatif jauh ini mempercepat proses pencarian karena beberapa kemungkinan posisi pemeriksaan dapat dilewati sekaligus.

Bagian (b) menunjukkan situasi yang hampir sama, yaitu ketika karakter teks yang menimbulkan ketidaksesuaian tidak muncul dalam pola. Dalam contoh ini, ketidaksesuaian terjadi di karakter 'd' dalam teks. Karena karakter tersebut tidak terdapat dalam pola, maka pola kembali digeser hingga posisinya berada tepat setelah karakter 'd' tersebut. Kondisi ini memperlihatkan bahwa algoritma Boyer–Moore mampu melakukan lompatan pergeseran yang cukup besar apabila karakter penyebab ketidaksesuaian tidak ditemukan dalam pola, sehingga jumlah perbandingan karakter dapat dikurangi secara signifikan.

Bagian (c) menggambarkan keadaan ketika karakter penyebab ketidaksesuaian muncul lebih dari satu kali dalam pola. Dalam contoh tersebut, karakter 'a' yang menimbulkan ketidaksesuaian terdapat beberapa kali dalam pola "acacac". Akibatnya, terdapat dua kemungkinan posisi pergeseran pola agar salah satu karakter 'a' dalam pola sejajar dengan karakter 'a' dalam teks. Dalam penerapannya, algoritma akan memilih pergeseran dengan jarak minimum yang masih memungkinkan penyelarasan karakter



Gambar 12.9: Contoh Bad Character Heuristic Algoritma Boyer–Moore (Agarwal, 2022)

tersebut. Pendekatan ini menjaga efisiensi proses pencocokan sehingga pola dapat dibandingkan kembali dengan teks secara optimal.

Good suffix heuristic

Good suffix heuristic merupakan salah satu strategi penting dalam algoritma Boyer-Moore untuk menentukan pergeseran pola ketika terjadi mismatch. Heuristik bad character tidak selalu memberikan pergeseran yang paling efektif, sehingga algoritma ini juga memanfaatkan good suffix heuristic berdasarkan bagian akhiran pola yang telah berhasil dicocokkan dengan teks. Dalam metode ini, pola digeser ke kanan sehingga akhiran yang telah cocok tersebut disejajarkan dengan kemunculan lain dari akhiran yang sama di dalam pola. Prosesnya dimulai dengan membandingkan pola dan teks dari kanan ke kiri. Jika kemudian ditemukan ketidaksesuaian, algoritma akan meninjau bagian akhiran pola yang sudah cocok sebelumnya, yang disebut sebagai good suffix, lalu menggunakan informasi tersebut untuk menentukan besar pergeseran pola yang lebih tepat.

Secara umum, good suffix heuristic bekerja dalam dua kondisi utama yaitu:

1. Ketika akhiran yang telah cocok memiliki satu atau lebih kemunculan lain di dalam pola, maka pola digeser sehingga kemunculan lain dari akhiran tersebut sejajar dengan bagian teks yang sudah cocok.
2. Apabila tidak ditemukan kemunculan lain secara utuh, tetapi sebagian dari akhiran yang telah cocok muncul di awal pola, maka pola digeser agar bagian akhiran tersebut sejajar dengan prefiks pola.

Dengan demikian, heuristik ini memungkinkan algoritma Boyer-Moore memanfaatkan kecocokan yang sudah diperoleh untuk melewati sejumlah perbandingan yang tidak perlu, sehingga proses pencocokan pola menjadi lebih efisien.

Gambar 2.10 menunjukkan contoh penerapan good suffix heuristic dalam algoritma Boyer–Moore dalam proses pencarian pola di dalam suatu teks. Algoritma Boyer–Moore melakukan pencocokan pola

Algoritma BuildGoodSuffix(P):

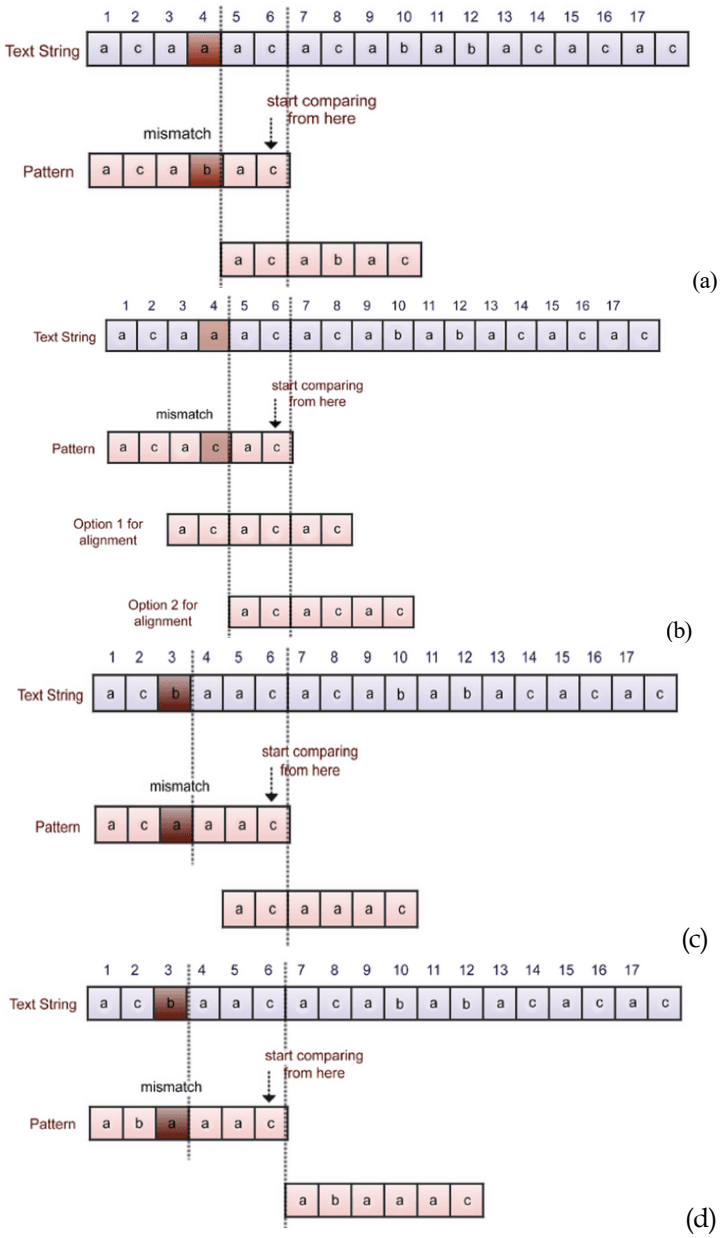
Input : P = pola dengan panjang m
Output : tabel GoodSuffix

1. Inisialisasi GoodSuffix dengan nilai default m
2. Tentukan semua suffix dari pola yang juga muncul di bagian lain pola
3. Untuk setiap suffix yang cocok, simpan besar pergeseran yang sesuai
4. Jika suffix tidak muncul lagi di pola, cari prefix pola yang cocok dengan suffix tersebut
5. Simpan nilai pergeseran minimum yang valid
6. return GoodSuffix

dengan teks dengan membandingkan karakter dari kanan ke kiri, dimulai dari karakter terakhir dalam pola. Ketika sebagian karakter bagian akhir pola berhasil dicocokkan dengan teks tetapi kemudian terjadi ketidaksesuaian (mismatch) di karakter sebelumnya, maka algoritma memanfaatkan good suffix heuristic untuk menentukan jarak pergeseran pola yang paling efisien. Prinsip utama dari heuristik ini adalah memanfaatkan suffix pola yang telah cocok agar dapat disejajarkan kembali dengan bagian lain dari pola atau dengan bagian teks yang sesuai.

Bagian (a) ditunjukkan kondisi awal ketika proses perbandingan dimulai dari karakter terakhir pola. Sebagian karakter di bagian akhir pola telah cocok dengan karakter teks, tetapi terjadi ketidaksesuaian di karakter sebelumnya. Dalam situasi ini, bagian pola yang telah cocok dianggap sebagai good suffix. Algoritma kemudian menggeser pola sehingga kemunculan suffix tersebut dalam pola dapat sejajar dengan bagian teks yang sesuai, sehingga proses pencocokan dapat dilanjutkan tanpa harus memeriksa kembali seluruh karakter dari awal.

Bagian (b) menggambarkan situasi ketika suffix yang telah cocok memiliki lebih dari satu kemungkinan posisi dalam pola. Dalam kondisi ini terdapat dua kemungkinan penyelarasan pola terhadap teks, yang disebut sebagai option 1 dan option 2. Kedua opsi tersebut menunjukkan posisi pergeseran pola agar suffix yang telah cocok dapat disejajarkan dengan kemunculan yang sama dalam pola. Algoritma Boyer–Moore akan memilih pergeseran yang paling sesuai agar proses pencocokan berikutnya dapat dilakukan secara efisien.



Gambar 12.10: Contoh Good Character Heuristic Algoritma Boyer–Moore (Agarwal, 2022)

Bagian (c) diperlihatkan kondisi ketika suffix yang telah cocok tidak memiliki kemunculan lain di dalam pola. Dalam keadaan ini, algoritma akan menggeser pola sehingga bagian awal pola sejajar dengan posisi setelah suffix yang telah cocok dalam teks. Pergeseran ini untuk melanjutkan proses pencarian tanpa perlu mengulangi perbandingan karakter yang sebelumnya sudah diketahui tidak sesuai.

Bagian (d) ditunjukkan kondisi lain ketika terjadi ketidaksesuaian setelah sebagian karakter pola cocok dengan teks. Algoritma kembali memanfaatkan informasi mengenai suffix yang telah cocok untuk menentukan posisi pergeseran pola yang paling tepat. Dengan memanfaatkan suffix yang telah cocok tersebut, algoritma dapat melakukan lompatan pergeseran yang lebih jauh dibandingkan metode pencarian string sederhana.

Program algoritma Boyer–Moore disajikan di https://colab.research.google.com/drive/12J91AB3MpBaUxPCA9TNt62QlgsphSgyT?usp=sharing.

BAB 13
Pendekatan dan Paradigma dalam Desain Algoritma

Studi tentang desain algoritma memiliki berbagai alasan yang sangat penting, dan motivasi untuk mempelajarinya sangat dipengaruhi oleh kebutuhan dan konteks masing-masing. Tentunya, ada alasan profesional yang kuat untuk mempelajari desain algoritma. **Algoritma adalah elemen dasar dalam seluruh proses komputasi**. Komputer sering dianggap sebagai perangkat keras yang terdiri dari komponen seperti hard drive, chip memori, prosesor, dan sebagainya. **Algoritma merupakan komponen paling penting, tanpa adanya algoritma, teknologi modern tidak akan dapat terwujud**.

Dasar teori algoritma, awalnya diterapkan melalui mesin Turing, berkembang beberapa dekade sebelum rangkaian logika digital mampu mewujudkan mesin tersebut. Mesin Turing merupakan model matematis yang, dengan aturan yang telah ditetapkan, mampu mengubah serangkaian input menjadi output tertentu. Mesin Turing pertama kali diimplementasikan secara mekanis, dan generasi berikutnya kemungkinan besar akan menggantikan sirkuit logika digital dengan sirkuit kuantum atau teknologi serupa. Tidak peduli platform yang digunakan, algoritma memiliki peran yang sangat penting dalam berbagai aspek.

Satu aspek penting lainnya adalah pengaruh algoritma terhadap inovasi teknologi. Sebagai contoh, pertimbangkan algoritma pencarian *page rank*, variasi dari algoritma pencarian yang digunakan dalam mesin pencari Google. Dengan algoritma ini dan algoritma serupa, para peneliti, ilmuwan, dan teknisi dapat dengan cepat mencari melalui sejumlah besar informasi. Hal ini berpengaruh besar terhadap kecepatan penemuan penelitian baru, penciptaan teknologi inovatif, dan pengembangan pengetahuan baru. Studi mengenai algoritma juga sangat penting karena membantu melatih cara berpikir secara lebih spesifik tentang masalah yang ada, serta meningkatkan kemampuan mental dan keterampilan pemecahan masalah dengan cara mengisolasi komponen-komponen dari suatu masalah dan mendefinisikan hubungan antar komponen tersebut. Secara ringkas, ada **empat alasan utama yang mendasari pentingnya mempelajari algoritma**:

1. Algoritma sangat penting bagi bidang ilmu komputer dan sistem *intelligent*.
2. Algoritma memiliki relevansi yang signifikan di berbagai bidang lainnya, seperti komputasi biologis, ekonomi, ekologi, komunikasi, dan fisika.
3. Algoritma memainkan peran kunci dalam inovasi teknologi.
4. Algoritma meningkatkan kemampuan pemecahan masalah serta keterampilan berpikir analitis.

Secara umum, algoritma dapat dipahami sebagai serangkaian instruksi atau langkah yang disusun dalam urutan tertentu. Algoritma merupakan sebuah konstruksi linier, seperti "lakukan x, kemudian lakukan y, lalu lakukan z, dan selesai". Namun, untuk memperluas fungsionalitasnya, sering kali ditambahkan kondisi seperti "lakukan x jika y terpenuhi", yang dalam bahasa pemrograman Python dapat diimplementasikan menggunakan pernyataan *if-else*. Dalam titik ini, urutan langkah yang dilakukan bergantung kondisi tertentu, misalnya status suatu struktur data. Oleh karena itu, dalam algoritma juga melibatkan operasi, iterasi, serta pernyataan *while* dan *for*. Untuk memperkaya pemahaman tentang algoritma, konsep rekursi juga diperkenalkan. Rekursi umumnya memberikan hasil yang sama seperti iterasi, meskipun keduanya memiliki perbedaan mendasar. Fungsi rekursif memanggil dirinya sendiri dan menerapkan fungsi yang sama terhadap input yang semakin mengecil, dengan setiap langkah rekursif menggunakan hasil dari langkah sebelumnya. Secara keseluruhan, **algoritma terdiri dari empat elemen utama** sebagai berikut:

- Operasi berurutan (*sequential*).
- Tindakan yang bergantung terhadap status suatu struktur data.
- Iterasi, yaitu pengulangan tindakan dalam beberapa kali proses.

- Rekursi, yang memanggil dirinya sendiri terhadap subset input tertentu.

Selain itu, terdapat beberapa **pendekatan utama dalam desain algoritma**, yaitu:

A. Algoritma *recursion*
B. Algoritma *Backtracking*
C. Algoritma *divide and conquer*
D. Algoritma *dynamic programming*
E. Algoritma *greedy*
F. Algoritma *brute force*
G. Algoritma *randomized*

Paradigma pembagian dan penaklukan, sebagaimana tercermin dalam namanya, melibatkan pemecahan masalah utama menjadi sub-masalah yang lebih kecil, yang kemudian diselesaikan secara terpisah dan hasilnya digabungkan untuk menghasilkan solusi keseluruhan. Pendekatan ini merupakan method yang sangat umum dan sering digunakan dalam desain algoritma untuk menyelesaikan berbagai jenis masalah komputasi.

13.1 Algoritma Recursion

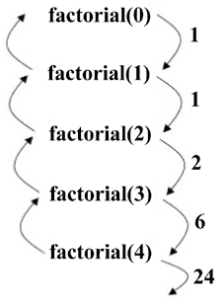
Rekursi merupakan suatu proses di mana sebuah fungsi memanggil dirinya sendiri untuk menyelesaikan suatu masalah. Konsep ini sering digunakan dalam matematika maupun pemrograman untuk menyelesaikan masalah secara bertahap hingga mencapai hasil akhir. Lebih jauh lagi, proses ini melibatkan penyelesaian masalah dengan memanfaatkan versi yang lebih kecil dari masalah tersebut untuk menyelesaikan versi yang lebih besar. Meskipun rekursi memerlukan waktu untuk dipahami sepenuhnya, teknik ini sering kali memberikan solusi yang sederhana dan elegan terhadap berbagai permasalahan yang ada (Goodrich et al., 2013).

Rekursi merupakan penerapan implisit dari STACK ADT (Thareja, 2014). Sebuah **fungsi rekursif** didefinisikan sebagai fungsi yang memanggil dirinya sendiri untuk menyelesaikan versi yang lebih kecil dari tugas yang dihadapi, hingga akhirnya mencapai panggilan terakhir yang tidak memerlukan pemanggilan lebih lanjut. Karena fungsi rekursif memanggil dirinya secara berulang, fungsi ini memanfaatkan stack sistem untuk menyimpan alamat pengembalian dan variabel lokal dari fungsi yang memanggilnya. Setiap solusi rekursif terdiri dari dua kasus utama, yaitu:

- *Base cases*, di mana masalah yang ada cukup sederhana untuk diselesaikan langsung tanpa memerlukan pemanggilan lebih lanjut ke fungsi yang sama. Atau kondisi yang menghentikan pemanggilan fungsi. Tanpa base case, fungsi akan terus memanggil dirinya sendiri dan menyebabkan error stack overflow.
- *Recursive cases*, di mana masalah yang dihadapi dibagi menjadi sub-masalah yang lebih sederhana. Fungsi kemudian memanggil dirinya sendiri untuk menyelesaikan sub-masalah tersebut, dan hasil yang diperoleh dari sub-masalah digabungkan untuk mendapatkan solusi masalah utama. Atau bagian yang memanggil fungsi itu sendiri dengan parameter yang telah dimodifikasi.

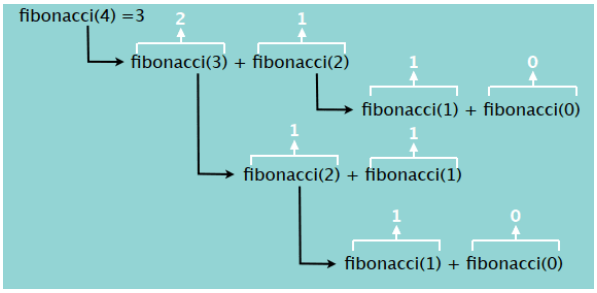
Gambar 13.1 memperlihatkan faktorial sebagai contoh proses rekursif, yaitu teknik pemrograman yang mendefinisikan suatu masalah melalui pemanggilan terhadap bentuk masalah yang lebih kecil tetapi sejenis. Dalam ilustrasi itu, fungsi factorial(4) tidak dihitung secara langsung, melainkan diuraikan terlebih dahulu menjadi factorial(3), kemudian factorial(2), selanjutnya factorial(1), dan akhirnya factorial(0). Urutan ini menunjukkan bahwa setiap pemanggilan fungsi bergantung hasil pemanggilan sebelumnya yang lebih kecil. Secara matematis, konsep tersebut mengikuti definisi faktorial, yaitu

n!=n×(n−1)!. Oleh karena itu, untuk memperoleh nilai factorial(4), sistem harus lebih dahulu mengetahui nilai factorial(3); untuk memperoleh factorial(3), sistem harus mengetahui factorial(2); demikian seterusnya hingga mencapai factorial(0).



Gambar 13.1: Faktorial dengan Rekursif (Agarwal, 2022)

Bagian terpenting dalam rekursi ini adalah keberadaan base case, yaitu kondisi penghentian agar pemanggilan fungsi tidak berlangsung tanpa akhir. Dalam contoh faktorial, factorial(0) = 1 menjadi titik dasar yang menghentikan proses pemanggilan rekursif. Sesudah base case tercapai, proses perhitungan bergerak berbalik ke atas melalui pengembalian nilai. Nilai factorial(1) diperoleh dari 1×factorial(0), sehingga hasilnya 1. Sesudah itu, factorial(2) dihitung sebagai 2×factorial(1), sehingga menghasilkan 2. Berikutnya, factorial(3) menjadi 3×2=6, dan akhirnya factorial(4) dihitung sebagai 4×6=24. Angka-angka 1, 1, 2, 6, dan 24 yang tertera di sisi kanan gambar menunjukkan urutan hasil pengembalian nilai dari setiap level rekursi.



Gambar 13.2: Fibonacci dengan Rekursif (Karimov, 2022)

Gambar 13.2 menjelaskan bagaimana nilai fibonacci(4) diperoleh melalui pemecahan masalah ke bentuk-bentuk yang lebih kecil tetapi sejenis. Di bagian paling atas terlihat bahwa fibonacci(4) = 3, lalu nilai itu diuraikan menjadi fibonacci(3) + fibonacci(2). Pemecahan semacam ini mencerminkan definisi rekursif deret Fibonacci, yaitu fibonacci(n) = fibonacci(n-1) + fibonacci(n-2) untuk n ≥ 2, dengan dua base case, yakni fibonacci(0) = 0 dan fibonacci(1) = 1. Dengan demikian, gambar menegaskan bahwa suatu nilai Fibonacci tidak dihitung secara langsung, melainkan diperoleh melalui penjumlahan dua nilai Fibonacci sebelumnya yang lebih kecil.

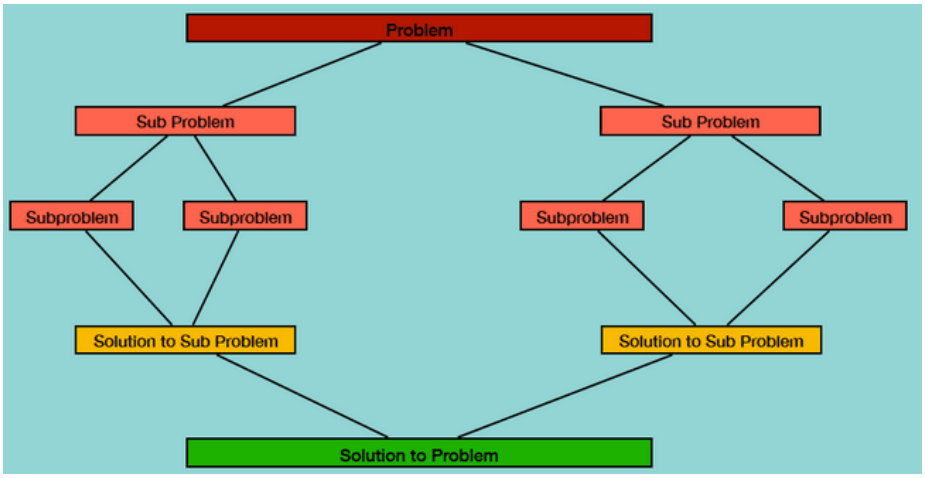
Secara rinci, cabang kiri menunjukkan bahwa fibonacci(3) masih harus diuraikan lagi menjadi fibonacci(2) + fibonacci(1). Selanjutnya, fibonacci(2) dipecah menjadi fibonacci(1) + fibonacci(0). Karena fibonacci(1) = 1 dan fibonacci(0) = 0, maka fibonacci(2) = 1. Nilai ini kemudian digunakan untuk menghitung fibonacci(3), yaitu 1 + 1 = 2. Sementara itu, cabang kanan dari bentuk awal menunjukkan bahwa fibonacci(2) juga muncul kembali sebagai bagian dari fibonacci(4) = fibonacci(3) + fibonacci(2), dan lagi-lagi dihitung sebagai fibonacci(1) + fibonacci(0) = 1 + 0 = 1. Sesudah kedua hasil antara tersebut tersedia, diperoleh fibonacci(4) = fibonacci(3) + fibonacci(2) = 2 + 1 = 3. Angka-angka kecil yang dituliskan di atas setiap cabang memperlihatkan nilai hasil dari subpemanggilan rekursif yang kemudian dikembalikan ke tingkat pemanggilan di atasnya.

Sebagai contoh rekursif, memperlihatkan dua unsur utama rekursi, yaitu base case dan recursive case. Base case tampak dalam fibonacci(0) dan fibonacci(1), yang nilainya sudah diketahui sehingga proses tidak perlu diuraikan lagi. Recursive case tampak dalam fibonacci(4), fibonacci(3), dan fibonacci(2), yang masing-masing masih harus dipecah menjadi pemanggilan fungsi yang lebih kecil. Ciri penting rekursi Fibonacci, yaitu adanya submasalah yang berulang, misalnya fibonacci(2) dihitung lebih dari satu kali.

Program faktorial dan fibonacci secara rekursif dapat dilihat di https://colab.research.google.com/drive/1Oh5gO0R86YOISw039L84Wcr4TNGwvQKA?usp=sharing.

13.2 Algoritma Divide and Conquer

Divide and conquer merupakan salah satu teknik yang penting dan efektif untuk menyelesaikan masalah yang kompleks. **Dalam divide-and-conquer paradigm, suatu masalah dipecah menjadi beberapa sub-problems yang lebih kecil, kemudian masing-masing sub-problems tersebut diselesaikan, dan akhirnya seluruh hasilnya digabungkan kembali untuk memperoleh global, optimal solution**. Secara lebih khusus, dalam rancangan divide-and-conquer, masalah dibagi menjadi dua smaller sub-problems, lalu masing-masing diselesaikan secara recursively. Sesudah itu, partial solutions yang dihasilkan digabungkan untuk memperoleh final solution. Teknik ini merupakan pendekatan pemecahan masalah yang sangat umum digunakan dan bahkan dapat dianggap sebagai salah satu pendekatan yang paling sering diterapkan dalam algorithm design. Beberapa contoh teknik divide and conquer antara lain Binary search, Merge sort, Quick sort, algorithm for fast multiplication, Strassen's matrix multiplication, serta closest pair of points (Agarwal, 2022).



Gambar 13.3: Ilustrasi Algoritma Divide and conquer (Karimov, 2022)

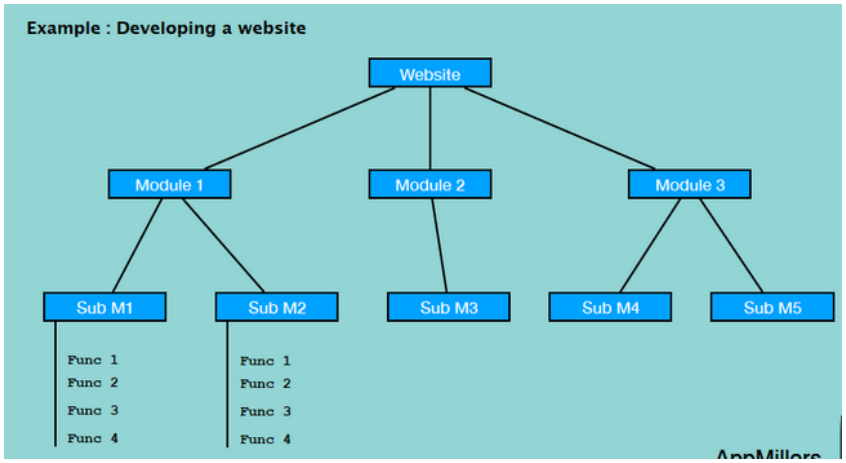
Gambar 13.3 memperlihatkan alur kerja algoritma Divide and Conquer secara konseptual melalui struktur bertingkat dari atas ke bawah. Di bagian paling atas merepresentasikan masalah utama yang ingin diselesaikan. Masalah utama ini kemudian tidak diselesaikan secara langsung, melainkan dipecah terlebih dahulu menjadi dua Sub Problem yang lebih kecil. Setiap Sub Problem selanjutnya kembali diuraikan menjadi Subproblem yang lebih rinci. Susunan bercabang ini menegaskan prinsip dasar Divide and Conquer, yaitu memecah masalah besar menjadi beberapa bagian yang lebih sederhana agar lebih mudah ditangani. Dengan cara tersebut, kompleksitas persoalan utama dikurangi melalui pembagian bertahap hingga diperoleh bagian-bagian yang cukup kecil untuk diselesaikan secara lebih efisien.

Bagian tengah gambar menunjukkan bahwa setiap cabang masalah kecil menghasilkan Solution to Sub Problem. Hal ini berarti setiap Subproblem yang telah dipecah sebelumnya diselesaikan terlebih dahulu secara terpisah. Sesudah solusi parsial diperoleh dari masing-masing cabang, hasil-hasil tersebut

tidak dibiarkan berdiri sendiri, melainkan digabungkan kembali. Proses penggabungan inilah yang menjadi tahap conquer, yaitu tahap ketika solusi-solusi parsial disatukan untuk membentuk jawaban yang lebih besar. Dalam ilustrasi ini, dua kotak Solution to Sub Problem berperan sebagai hasil antara dari dua kelompok submasalah yang berbeda, lalu keduanya diarahkan ke satu kotak terakhir di bagian bawah.

Kotak paling bawah bertuliskan Solution to Problem merepresentasikan hasil akhir dari seluruh proses. Keberadaan kotak ini menunjukkan bahwa solusi akhir diperoleh bukan dari penyelesaian langsung terhadap masalah utama, melainkan dari penggabungan beberapa solusi kecil yang telah diselesaikan sebelumnya. Dengan demikian, gambar ini menjelaskan tiga tahapan penting dalam Divide and Conquer, yaitu divide, solve, dan combine. Tahap divide terlihat dari pemecahan Problem menjadi Sub Problem dan Subproblem. Tahap solve tampak dari munculnya Solution to Sub Problem. Tahap combine terlihat dari penggabungan solusi-solusi parsial menjadi Solution to Problem.

Gambar 13.4 memperlihatkan contoh penerapan algoritma Divide and Conquer dalam konteks developing a website. Di bagian paling atas terdapat kotak Website yang merepresentasikan tujuan utama



Gambar 13.4: Contoh Algoritma Divide and conquer (Karimov, 2022)

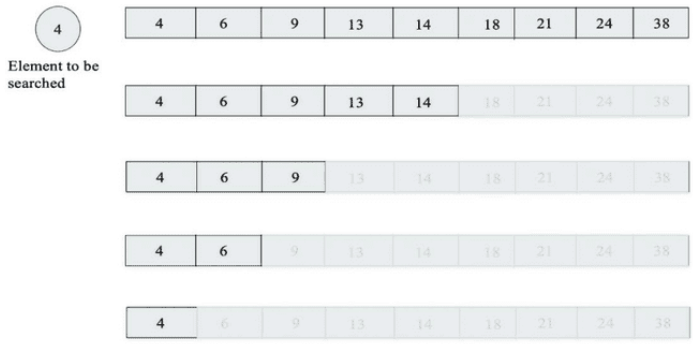
atau sistem yang ingin dibangun secara utuh. Sistem tersebut kemudian tidak dikerjakan sebagai satu kesatuan besar secara langsung, melainkan dipecah menjadi tiga bagian utama, yaitu Module 1, Module 2, dan Module 3. Pemecahan ini menunjukkan prinsip dasar Divide and Conquer, yakni membagi persoalan besar menjadi beberapa komponen yang lebih kecil agar proses perancangan dan pengembangannya menjadi lebih terstruktur, lebih mudah dikelola, dan lebih efisien. Dengan cara ini, pembangunan website tidak lagi dipandang sebagai satu pekerjaan besar yang kompleks, melainkan sebagai sekumpulan modul yang dapat ditangani secara lebih sistematis.

Cabang berikutnya menunjukkan bahwa setiap modul utama masih dapat dipecah lagi menjadi bagian yang lebih rinci. Module 1 diuraikan menjadi Sub M1 dan Sub M2, Module 2 diuraikan menjadi Sub M3, sedangkan Module 3 dipecah menjadi Sub M4 dan Sub M5. Struktur bertingkat ini menegaskan bahwa dalam Divide and Conquer, pemecahan masalah dapat berlangsung lebih dari satu level hingga diperoleh unit kerja yang lebih kecil dan lebih spesifik. Di bawah Sub M1 dan Sub M2, bahkan ditunjukkan lagi rincian berupa Func 1, Func 2, Func 3, dan Func 4, yang menggambarkan bahwa submodul masih dapat dijabarkan menjadi fungsi-fungsi operasional. Hal ini menunjukkan bahwa proses pembagian tidak berhenti di tingkat modul, tetapi dapat diteruskan sampai ke tingkat fungsi, yaitu bagian paling rinci yang benar-benar dapat diimplementasikan secara langsung dalam pengembangan sistem.

Binary Search

Binary search merupakan algoritma yang didasarkan teknik desain divide and conquer. Algoritma ini digunakan untuk menemukan elemen tertentu dari sebuah sorted list of elements. Prosesnya dimulai dengan membandingkan search element dengan elemen tengah dari list. Jika search element lebih kecil daripada elemen tengah tersebut, maka separuh list yang memuat elemen-elemen lebih besar dari elemen tengah akan dibuang. Proses ini kemudian diulangi secara recursively hingga search element ditemukan atau hingga pencarian mencapai akhir list. Hal yang penting untuk diperhatikan ialah bahwa dalam setiap iterasi, separuh dari search space dibuang. Mekanisme ini meningkatkan performa algoritma secara keseluruhan karena jumlah elemen yang masih harus diperiksa menjadi semakin sedikit.

Sebagai ilustrasi, misalkan ingin dicari elemen 4 dari sebuah sorted list of elements seperti yang ditunjukkan dalam contoh. Dalam setiap iterasi, list dibagi menjadi dua bagian. Melalui strategi divide



Gambar 13.5: Proses Pencarian Elemen Menggunakan Binary Search (Agarwal, 2022)

and conquer, pencarian elemen tersebut dapat dilakukan hanya dalam O(log n) kali langkah. Dengan demikian, binary search menjadi salah satu algoritma pencarian yang sangat efisien untuk data yang telah tersusun secara terurut.

Gambar 13.5 memperlihatkan proses pencarian elemen menggunakan Binary Search terhadap sebuah sorted list yang berisi nilai 4, 6, 9, 13, 14, 18, 21, 24, dan 38, dengan element to be searched adalah 4. Ilustrasi disusun secara bertahap dari baris atas ke baris bawah untuk menunjukkan bagaimana ruang pencarian terus dipersempit. Baris pertama, seluruh elemen masih dianggap sebagai ruang pencarian aktif. Karena daftar telah tersusun secara menaik, algoritma Binary Search dapat langsung memeriksa elemen tengah sebagai titik pembanding. Setelah elemen tengah dibandingkan dengan nilai target, bagian daftar yang tidak mungkin memuat elemen 4 dihilangkan dari ruang pencarian. Dalam gambar, bagian yang dieliminasi ditandai dengan warna yang semakin pudar, sedangkan bagian yang masih relevan tetap tampak lebih jelas. Teknik ini menunjukkan karakter utama Binary Search, yaitu tidak memeriksa seluruh elemen satu per satu, tetapi membagi ruang pencarian menjadi dua bagian dan hanya mempertahankan bagian yang masih mungkin mengandung elemen target.

Tahap kedua memperlihatkan bahwa setelah pemeriksaan pertama, pencarian hanya difokuskan separuh kiri daftar, yaitu 4, 6, 9, 13, dan 14, karena nilai target 4 lebih kecil daripada elemen tengah di rentang awal. Tahap berikutnya kembali memperkecil ruang pencarian menjadi 4, 6, dan 9, sebab elemen tengah dari rentang sebelumnya masih lebih besar daripada nilai yang dicari. Sesudah itu, ruang pencarian dipersempit lagi menjadi 4 dan 6, lalu akhirnya tinggal 4 saja. Ketika hanya satu elemen yang tersisa dan nilainya sama dengan target, proses pencarian selesai. Dengan demikian, gambar ini menjelaskan bahwa Binary Search bekerja melalui serangkaian eliminasi sistematis terhadap setengah bagian daftar yang tidak relevan, sehingga elemen target dapat ditemukan dengan jauh lebih efisien dibandingkan pencarian linear. Ilustrasi tersebut sekaligus menegaskan bahwa keberhasilan Binary Search bergantung kondisi data yang telah terurut, karena hanya dengan susunan semacam itu algoritma dapat menentukan secara logis bagian mana yang harus dipertahankan dan bagian mana yang dapat diabaikan.

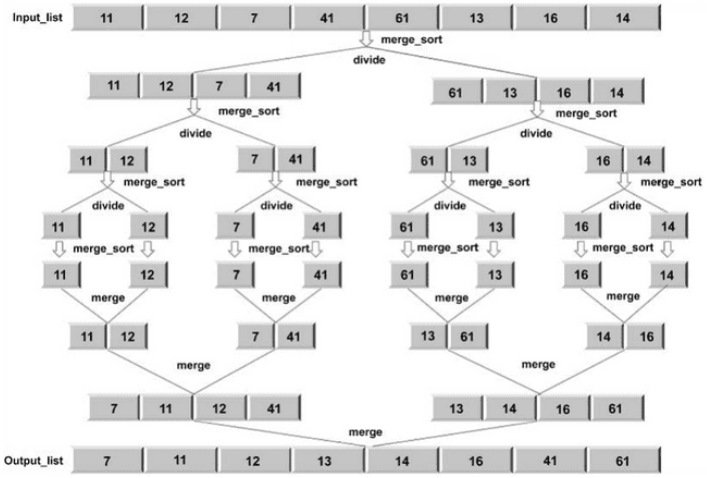
Merge Sort

Merge sort adalah algoritma untuk mengurutkan sebuah list yang berisi n natural numbers ke dalam urutan menaik. Prosesnya dimulai dengan membagi list elemen yang diberikan secara berulang menjadi bagian-bagian yang sama besar sampai setiap sublist hanya berisi satu elemen. Sesudah itu, seluruh sublist tersebut digabungkan kembali untuk membentuk list baru dalam keadaan sorted order. Pendekatan pemrograman untuk menyelesaikan masalah ini didasarkan metodologi divide-and-conquer, yang menekankan pentingnya memecah suatu masalah menjadi sub-problems yang lebih kecil dengan bentuk yang sama seperti masalah asal. Sub-problems tersebut diselesaikan secara terpisah, lalu hasilnya digabungkan kembali untuk memperoleh solusi dari masalah semula.

Dalam kasus ini, ketika diberikan list elemen yang masih unsorted, list tersebut dibagi menjadi dua bagian yang kira-kira sama besar. Proses pembagian itu terus dilakukan ke dalam dua bagian secara recursively. Setelah beberapa tahap, sublist yang dihasilkan dari pemanggilan rekursif hanya akan memuat satu elemen. Titik itulah proses berpindah ke tahap conquer atau merge, yaitu tahap ketika solusi-solusi kecil mulai digabungkan kembali menjadi list yang lebih besar dan terurut. Dengan demikian, Merge sort memperlihatkan dengan jelas hubungan antara tahap pemecahan masalah dan tahap penggabungan solusi dalam pendekatan divide-and-conquer.

Worst-case running time complexity dari Merge sort bergantung dalam beberapa langkah penting. Pertama, tahap divide memerlukan constant time, karena hanya menghitung midpoint, dan hal ini dapat dilakukan dalam O(1). Kedua, dalam setiap iterasi, list dibagi menjadi dua bagian secara recursively, sehingga proses ini memerlukan O(log n), serupa dengan mekanisme binary search algorithm. Ketiga, tahap combine/merge menggabungkan seluruh n elements kembali ke dalam original array, sehingga memerlukan waktu O(n). Oleh karena itu, Merge sort algorithm memiliki runtime complexity sebesar O(n log n), yang dinyatakan sebagai $T(n) = O(n) \times O(\log n) = O(n \log n)$.

Gambar 13.6 memperlihatkan Algoritma Merge Sort sebagai implementasi nyata dari strategi divide and conquer untuk mengurutkan sekumpulan data. Di bagian paling atas terlihat input_list yang berisi elemen 11, 12, 7, 41, 61, 13, 16, 14. Tujuan algoritma ini adalah menghasilkan output_list yang



Gambar 13.6: Algoritma Merge Sort (Agarwal, 2022)

tersusun menaik, yaitu 7, 11, 12, 13, 14, 16, 41, 61. Struktur gambar berbentuk pohon menegaskan bahwa proses pengurutan tidak dilakukan secara langsung terhadap seluruh list sekaligus, melainkan melalui dua tahap utama, yaitu divide dan merge. Tahap divide memecah list menjadi bagian-bagian yang lebih kecil secara berulang, sedangkan tahap merge menggabungkan kembali bagian-bagian kecil tersebut dalam urutan yang sudah teratur. Dengan demikian, gambar ini menunjukkan bahwa Merge Sort bekerja dengan

cara memecah persoalan besar menjadi subpersoalan yang lebih sederhana, lalu menyusun kembali hasil penyelesaiannya menjadi solusi akhir yang utuh.

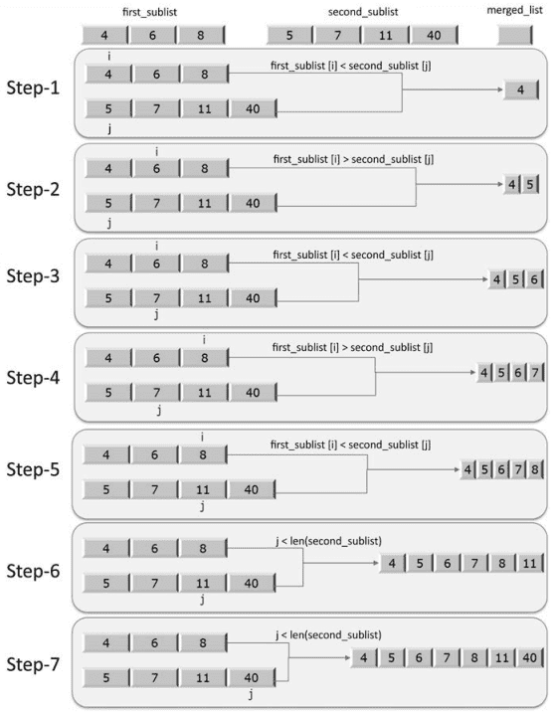
Dari sisi divide and conquer, tahap divide tampak jelas ketika input_list pertama kali dipecah menjadi dua sublist, yaitu 11, 12, 7, 41 di sisi kiri dan 61, 13, 16, 14 di sisi kanan. Masing-masing sublist kemudian kembali dipecah menjadi dua bagian yang lebih kecil. Sublist kiri menjadi 11, 12 dan 7, 41, sedangkan sublist kanan menjadi 61, 13 dan 16, 14. Proses pemecahan ini terus berlanjut hingga setiap sublist hanya berisi satu elemen, yaitu 11, 12, 7, 41, 61, 13, 16, dan 14. Dalam kerangka divide and conquer, kondisi ketika setiap sublist hanya memuat satu elemen merupakan titik dasar rekursi, karena sebuah list dengan satu elemen secara alami sudah berada dalam keadaan terurut. Oleh sebab itu, tahap divide berhenti ketika persoalan telah diperkecil hingga unit-unit terkecil yang tidak lagi memerlukan pengurutan tambahan.

Setelah tahap divide selesai, algoritma memasuki tahap conquer atau merge, yaitu tahap penggabungan sublist kecil menjadi sublist yang lebih besar dalam urutan menaik. Di cabang kiri, elemen 11 dan 12 digabungkan menjadi 11, 12, sedangkan elemen 7 dan 41 digabungkan menjadi 7, 41. Sesudah itu, dua sublist terurut tersebut digabungkan lagi dengan membandingkan elemen-elemen terkecilnya secara bertahap, sehingga dihasilkan sublist kiri yang telah terurut, yaitu 7, 11, 12, 41. Proses serupa terjadi di cabang kanan. Elemen 61 dan 13 digabungkan menjadi 13, 61, lalu elemen 16 dan 14 digabungkan menjadi 14, 16. Kedua sublist hasil penggabungan ini kemudian diproses lagi melalui mekanisme perbandingan berurutan, sehingga menghasilkan sublist kanan terurut, yaitu 13, 14, 16, 61. Tahap ini menunjukkan bahwa conquer dalam Merge Sort bukan berarti menyelesaikan setiap bagian secara terpisah tanpa hubungan, melainkan membangun keterurutan global melalui penggabungan terkontrol dari sublist-sublist yang sebelumnya sudah terurut.

Tahap terakhir memperlihatkan merge antara dua sublist besar yang sudah terurut, yaitu 7, 11, 12, 41 dan 13, 14, 16, 61. Penggabungan dilakukan dengan membandingkan elemen terdepan dari masing-masing sublist secara berulang. Elemen yang lebih kecil dipindahkan terlebih dahulu ke list hasil, lalu perbandingan dilanjutkan ke elemen berikutnya hingga seluruh elemen habis diproses. Melalui mekanisme ini dihasilkan output_list akhir, yaitu 7, 11, 12, 13, 14, 16, 41, 61. Bagian ini menegaskan inti keunggulan Merge Sort, yaitu kemampuan menjaga keterurutan lokal di setiap sublist kecil, lalu mengubahnya menjadi keterurutan global melalui proses merge yang sistematis.

Gambar 13.7 memperlihatkan the process of merging the two sublists sebagai salah satu bagian penting dari Merge Sort, yang merupakan contoh klasik dari algoritma Divide and Conquer. Dalam ilustrasi ini, dua sorted sublists, yaitu [4, 6, 8] sebagai first_sublist dan [5, 7, 11, 40] sebagai second_sublist, digabungkan secara bertahap untuk membentuk merged_list yang tetap berada dalam urutan menaik. Proses ini menunjukkan tahap conquer atau merge, yaitu saat dua hasil pengurutan kecil yang telah diperoleh dari tahap pemecahan sebelumnya digabungkan kembali menjadi satu list yang lebih besar dan tetap terurut.

Secara rinci, Step-1 dimulai dengan membandingkan elemen pertama dari kedua sublists, yaitu 4 dari first_sublist dan 5 dari second_sublist. Karena $4 < 5$, elemen 4 dipindahkan ke merged_list. Dalam Step-2, elemen terdepan berikutnya dari first_sublist, yaitu 6, dibandingkan dengan elemen terdepan dari second_sublist, yaitu 5. Karena 5 lebih kecil, maka 5 dipindahkan ke merged_list, sehingga urutan hasil sementara menjadi [4, 5]. Dalam Step-3, elemen 6 dibandingkan dengan 7, dan karena 6 lebih kecil, elemen 6 dipindahkan ke merged_list sehingga hasil sementara menjadi [4, 5, 6]. Dalam Step-4, elemen 8 dibandingkan dengan 7, dan karena 7 lebih kecil, elemen itu dipindahkan ke merged_list, membentuk [4, 5, 6, 7]. Dalam Step-5, elemen 8 dibandingkan dengan 11, sehingga 8 dipindahkan ke merged_list dan hasil sementara menjadi [4, 5, 6, 7, 8]. Sesudah langkah ini, first_sublist menjadi kosong. Karena salah satu sublist telah habis, maka dalam Step-6 dan Step-7 sisa elemen dari second_sublist, yaitu 11 dan 40,



Gambar 13.7: Proses Penggabungan Dua Sublist (Agarwal, 2022)

dipindahkan langsung ke merged_list tanpa perbandingan tambahan. Hasil akhirnya adalah [4, 5, 6, 7, 8, 11, 40].

Pogram algoritma devide and concuer dapat dilihat di https://colab.research.google.com/drive/1dqOm8vhDuNQd1xn8WPaLzw_fdHHcJN6p?usp=sharing.

13.3 Algoritma Dynamic Programming

Dynamic programming merupakan teknik desain yang sangat kuat untuk menyelesaikan optimization problems. Jenis masalah ini umumnya memiliki banyak kemungkinan solusi. Gagasan dasar dynamic programming berangkat dari intuisi teknik divide-and-conquer. Dalam pendekatan ini, ruang seluruh kemungkinan solusi dieksplorasi dengan cara memecah masalah menjadi serangkaian sub-problems, kemudian hasil-hasilnya digabungkan untuk memperoleh solusi yang benar bagi masalah yang lebih besar. Perbedaan penting adalah sifat sub-problems yang dihadapi. Algoritma Divide-and-conquer digunakan ketika sub-problems bersifat non-overlapping (disjoint), sedangkan dynamic programming digunakan ketika sub-problems bersifat overlapping, artinya beberapa sub-problems berbagi sub-sub-problems yang sama. Dengan demikian, meskipun dynamic programming mirip dengan divide and conquer karena sama-sama memecah masalah menjadi bagian-bagian yang lebih kecil, dynamic programming-based techniques memiliki keunggulan karena setiap sub-sub-problems diselesaikan hanya satu kali dan tidak dihitung ulang jika pernah ditemui sebelumnya. Sebagai gantinya, pendekatan ini memanfaatkan teknik remembering untuk menghindari re-computation.

Masalah dalam dynamic programming memiliki dua karakteristik penting yaitu:

- Karakteristik pertama adalah optimal substructure. Suatu masalah dikatakan memiliki optimal substructure apabila solusi masalah tersebut dapat diperoleh dengan menggabungkan solusi dari sub-problems-nya. Dengan kata lain, solusi optimal dari masalah utama dapat dibangun dari solusi optimal sub-problems yang menyusunnya. Sebagai contoh, bilangan Fibonacci ke-i dapat dihitung dari bilangan Fibonacci ke-(i-1) dan ke-(i-2); misalnya, fib(6) dapat diperoleh dari fib(5) dan fib(4).

- Karakteristik kedua adalah overlapping sub-problem. Kondisi ini terjadi apabila suatu algoritma harus berulang kali menyelesaikan sub-problem yang sama. Sebagai ilustrasi, fib(5) akan menghasilkan perhitungan berulang untuk fib(3) dan fib(2).

Jika suatu masalah memiliki kedua karakteristik tersebut, maka pendekatan dynamic programming menjadi sangat berguna, karena implementasinya dapat ditingkatkan dengan memanfaatkan kembali solusi yang telah dihitung sebelumnya. Dalam strategi dynamic programming, masalah dipecah menjadi independent sub-problems, lalu hasil-hasil antara disimpan agar dapat digunakan kembali dalam operasi berikutnya.

Dalam pendekatan dynamic, masalah dibagi menjadi smaller sub-problems. Hal yang sama juga terjadi dalam recursion, karena di sana masalah juga dipecah menjadi sub-problems. Akan tetapi, perbedaan mendasarnya adalah bahwa dalam recursion, similar sub-problems dapat diselesaikan berkali-kali, sedangkan dalam dynamic programming, semua *previously solved sub-problems* dilacak dan dijaga agar tidak dihitung ulang. Salah satu sifat yang menjadikan suatu masalah sangat cocok diselesaikan dengan dynamic programming adalah adanya ***overlapping set of sub-problems***. Ketika selama proses komputasi terlihat bahwa bentuk sub-problems yang sama muncul kembali, maka sub-problem tersebut tidak perlu dihitung ulang. Sebaliknya, algoritma cukup mengembalikan *pre computed result* dari *previously encountered sub-problem* tersebut. Dengan demikian, dynamic programming berangkat dari prinsip bahwa setiap sub-problem seharusnya cukup diselesaikan satu kali saja.

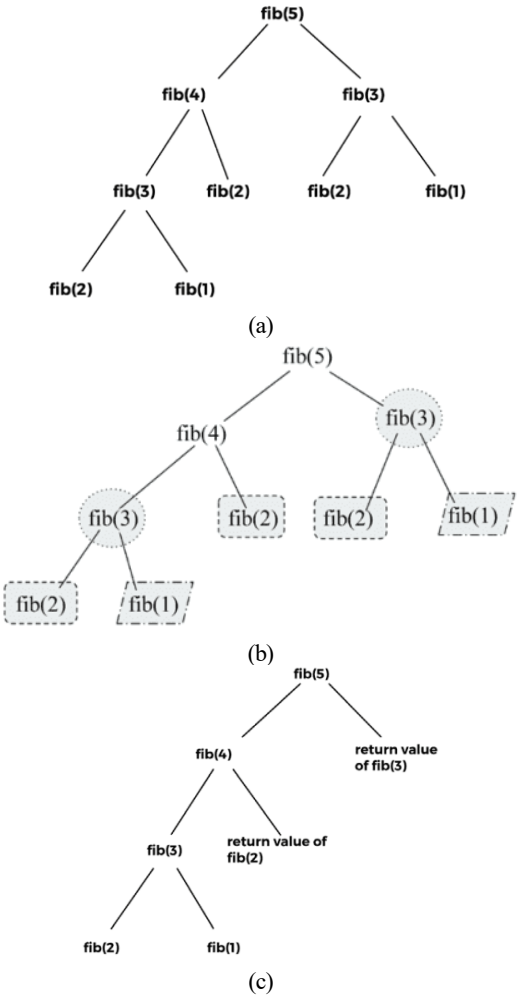
Agar tidak pernah mengevaluasi ulang sub-problem yang sama, dynamic programming membutuhkan cara yang efisien untuk menyimpan hasil dari setiap sub-problem. Terdapat dua teknik utama yang dapat digunakan, yaitu:

- Top-down with memoization. Proses dimulai dari masalah awal, kemudian dipecah menjadi small sub-problems. Setelah solusi untuk suatu sub-problem berhasil ditentukan, hasilnya disimpan. Ketika sub-problem yang sama muncul kembali di masa berikutnya, algoritma cukup mengembalikan pre computed result tersebut. Oleh sebab itu, jika solusi suatu masalah dapat dirumuskan secara recursively dengan memanfaatkan solusi dari sub-problems-nya, maka solusi untuk overlapping sub-problems dapat dengan mudah di-memoized. Memoization berarti menyimpan solusi dari sub-problem ke dalam array atau hash table. Setiap kali solusi terhadap suatu sub-problem dibutuhkan, algoritma terlebih dahulu memeriksa nilai yang telah disimpan. Jika hasilnya sudah tersedia, nilai itu langsung digunakan; jika belum tersedia, maka sub-problem dihitung dengan cara biasa. Prosedur ini disebut memoized, karena sistem "mengingat" hasil operasi yang pernah dihitung sebelumnya.
- Bottom-up approach bergantung ukuran dari sub-problems. Dalam pendekatan ini, smaller sub-problems diselesaikan terlebih dahulu. Dengan demikian, ketika suatu sub-problem tertentu akan diselesaikan, solusi dari smaller sub-problems yang menjadi prasyaratnya sudah tersedia. Setiap sub-problem hanya diselesaikan satu kali, dan ketika algoritma berusaha menyelesaikan suatu sub-problem, solusi dari seluruh prerequisite smaller sub-problems telah ada dan dapat langsung digunakan. Dalam pendekatan ini, masalah diselesaikan dengan memecahnya menjadi sub-problems secara recursively, lalu smallest possible sub-problems diselesaikan terlebih dahulu. Sesudah itu, solusi-solusi dari sub-problems tersebut digabungkan secara bottom-up untuk memperoleh solusi dari bigger sub-problem, dan proses ini berlanjut hingga mencapai final solution. Dengan demikian, Bottom-up approach dan Top-down with memoization sama-sama bertujuan menghindari re-computation, tetapi keduanya menempuh jalur yang berbeda dalam menyusun solusi akhir.

Gambar 13.8 memperlihatkan cara kerja Dynamic Programming melalui contoh perhitungan deret Fibonacci, sehingga pembaca dapat memahami bagaimana teknik ini mengatasi permasalahan overlapping sub-problems. Untuk menjelaskan mekanisme dynamic programming, digunakan persoalan menghitung nilai fib(5). Ilustrasi dibagi menjadi tiga bagian, yaitu (a), (b), dan (c), yang masing-masing menunjukkan

perbedaan antara pendekatan rekursif biasa dan pendekatan Dynamic Programming. Secara umum, gambar ini menegaskan bahwa perhitungan deret Fibonacci sangat cocok dijadikan contoh karena relasi fib(n) = fib(n-1) + fib(n-2) menghasilkan banyak pemanggilan sub-problems yang sama secara berulang.

Bagian (a) menampilkan pohon rekursi biasa untuk menghitung fib(5). Dari puncak pohon, fib(5)



Gambar 13.8: Fibonacci dengan Algoritma Dynamic Programming (Agarwal, 2022)

dipecah menjadi fib(4) dan fib(3). Selanjutnya, fib(4) kembali diuraikan menjadi fib(3) dan fib(2), sedangkan fib(3) di cabang kanan diuraikan lagi menjadi fib(2) dan fib(1). Di sisi kiri, fib(3) juga masih dipecah lagi menjadi fib(2) dan fib(1). Struktur ini menunjukkan bahwa fib(3) muncul lebih dari satu kali, demikian pula fib(2) dan fib(1). Artinya, rekursi biasa menghitung sub-problems yang sama berulang-ulang, sehingga terjadi pemborosan komputasi. Ilustrasi ini menegaskan karakteristik overlapping sub-problems, yaitu keadaan ketika beberapa cabang perhitungan membutuhkan hasil sub-problem yang identik.

Bagian (b) menyoroti inti persoalan tersebut dengan memberi penekanan visual terhadap sub-problems yang berulang. Simpul fib(3) di cabang kanan diberi tanda yang sama dengan fib(3) di cabang kiri, sehingga tampak bahwa keduanya sebenarnya mewakili perhitungan yang identik. Demikian pula, fib(2) dan fib(1) yang muncul berulang juga diberi penanda khusus. Penekanan visual ini menunjukkan bahwa dalam rekursi biasa, algoritma terus menghitung nilai yang sama meskipun hasilnya sebenarnya sudah pernah diperoleh sebelumnya. Di sinilah Dynamic Programming menawarkan perbaikan. Alih-alih menghitung ulang fib(3), fib(2), atau fib(1) setiap kali simpul tersebut muncul, algoritma cukup menyimpan hasil pertama yang telah diperoleh, lalu memanggil kembali nilai yang tersimpan itu ketika

dibutuhkan lagi. Dengan kata lain, gambar ini menjelaskan prinsip memoization, yaitu teknik "mengingat" hasil perhitungan sebelumnya agar re-computation dapat dihindari.

Bagian (c) memperlihatkan hasil penyederhanaan pohon rekursi sesudah prinsip Dynamic Programming diterapkan. Dalam ilustrasi ini, cabang kanan dari fib(5) tidak lagi dihitung sebagai pohon lengkap, melainkan digantikan dengan return value of fib(3). Hal yang sama juga terjadi di salah satu cabang dari fib(4), yang digantikan dengan return value of fib(2). Bentuk ini menunjukkan bahwa ketika nilai fib(3) dan fib(2) sudah pernah dihitung di cabang sebelumnya, algoritma tidak perlu lagi menurunkannya menjadi subcabang baru. Sebaliknya, hasil tersebut langsung diambil dari penyimpanan hasil antara. Melalui cara ini, jumlah simpul dalam pohon perhitungan berkurang secara signifikan, sehingga proses komputasi menjadi lebih ringkas dan lebih efisien. Ilustrasi tersebut menegaskan bahwa Dynamic Programming tidak mengubah definisi matematis deret Fibonacci, tetapi mengubah strategi perhitungannya agar setiap sub-problem hanya diselesaikan satu kali.

Secara konseptual, gambar ini juga memperlihatkan dua karakteristik penting Dynamic Programming, yaitu optimal substructure dan overlapping sub-problems. Optimal substructure tampak dari fakta bahwa nilai fib(5) diperoleh dari penggabungan hasil fib(4) dan fib(3), sedangkan fib(4) diperoleh dari fib(3) dan fib(2). Artinya, solusi masalah yang lebih besar dibangun dari solusi masalah yang lebih kecil. Sementara itu, overlapping sub-problems tampak dari kemunculan berulang fib(3), fib(2), dan fib(1) dalam beberapa cabang pohon rekursi. Karena kedua sifat ini hadir sekaligus, Fibonacci menjadi contoh klasik yang sangat sesuai untuk menjelaskan Dynamic Programming. Dengan demikian, gambar tersebut secara rinci menunjukkan bahwa menghitung Fibonacci dengan Algoritma Dynamic Programming berarti memecah persoalan menjadi sub-problems yang lebih kecil, menyimpan hasil-hasil antara yang sudah diperoleh, lalu menggunakan kembali hasil tersebut ketika diperlukan, sehingga perhitungan menjadi lebih efisien daripada pendekatan rekursif biasa.

Contoh program Dynamic Programming berupa deret fibonacci dapat dilihat di https://colab.research.google.com/drive/1OpIOUuAgh6YiIT6QgJOA9w-GTQn4otSb?usp=sharing.

13.4 Algoritma Greedy

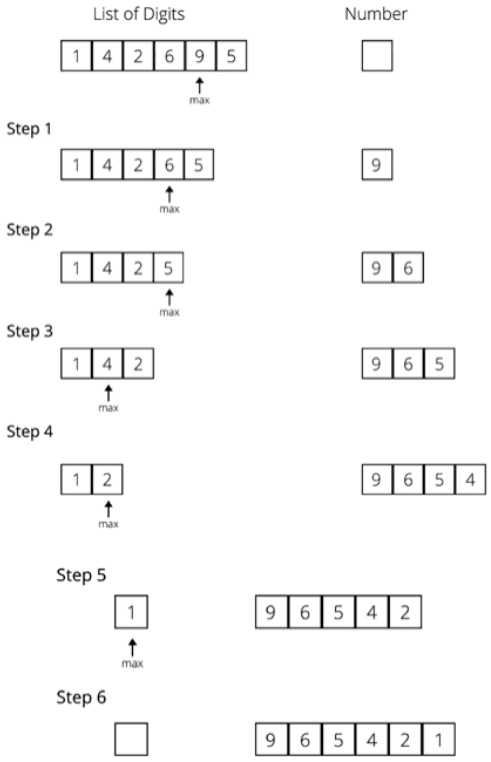
Greedy algorithms sering digunakan dalam optimization dan combinatorial problems. Dalam greedy algorithms, tujuan utamanya adalah memperoleh optimum solution dari sekian banyak kemungkinan solusi di setiap langkah. **Pendekatan yang digunakan adalah memilih local optimum solution, dengan harapan bahwa rangkaian pilihan lokal tersebut akhirnya akan mengarah ke global optimum solution**. Meskipun demikian, greedy strategy tidak selalu menghasilkan solusi yang benar-benar optimal secara global. Akan tetapi, urutan locally optimal solutions umumnya mampu memberikan pendekatan yang cukup dekat terhadap globally optimal solution.

Gambar 13.9 memperlihatkan ilustrasi Algoritma Greedy melalui persoalan membentuk bilangan terbesar dari sekumpulan digit yang tersedia. Di bagian atas, ditampilkan List of Digits yang berisi 1, 4, 2, 6, 9, dan 5, sedangkan di sisi kanan terdapat ruang Number yang awalnya masih kosong. Prinsip Greedy yang ditunjukkan gambar ini adalah memilih elemen terbaik di setiap langkah, yaitu digit terbesar yang masih tersedia, lalu menambahkannya ke bilangan hasil. Dengan demikian, algoritma tidak meninjau seluruh kemungkinan susunan digit secara menyeluruh, melainkan langsung mengambil keputusan lokal yang dianggap paling menguntungkan saat itu. Dalam konteks ini, keputusan lokal tersebut adalah selalu memilih digit maksimum dari daftar yang tersisa.

Proses dimulai Step 1, ketika digit terbesar dalam daftar awal, yaitu 9, dipilih sebagai max. Digit ini kemudian dipindahkan ke kolom Number, sehingga bilangan hasil sementara menjadi 9, sedangkan List of Digits tersisa 1, 4, 2, 6, dan 5. Step 2, dari daftar yang tersisa, digit terbesar adalah 6, sehingga digit ini ditambahkan ke kanan bilangan hasil dan membentuk 96. Setelah itu, daftar digit berubah menjadi 1, 4, 2, dan 5. Step 3, algoritma kembali mencari max dari digit yang tersisa, yaitu 5, lalu menambahkannya

ke bilangan hasil sehingga terbentuk 965, sedangkan daftar tersisa 1, 4, dan 2. Step 4, digit terbesar berikutnya adalah 4, sehingga bilangan hasil berubah menjadi 9654, dengan daftar tersisa 1 dan 2. Dalam Step 5, digit terbesar dari daftar yang tersisa adalah 2, sehingga bilangan hasil bertambah menjadi 96542, dan hanya tersisa digit 1. Terakhir, Step 6, digit 1 dipindahkan ke kolom Number, sehingga bilangan akhir yang terbentuk adalah 965421, sedangkan List of Digits menjadi kosong.

Greedy algorithm bekerja dengan menyusun solusi sedikit demi sedikit melalui pilihan lokal terbaik



Gambar 13.9: Ilustrasi Algoritma Greedy (Agarwal, 2022)

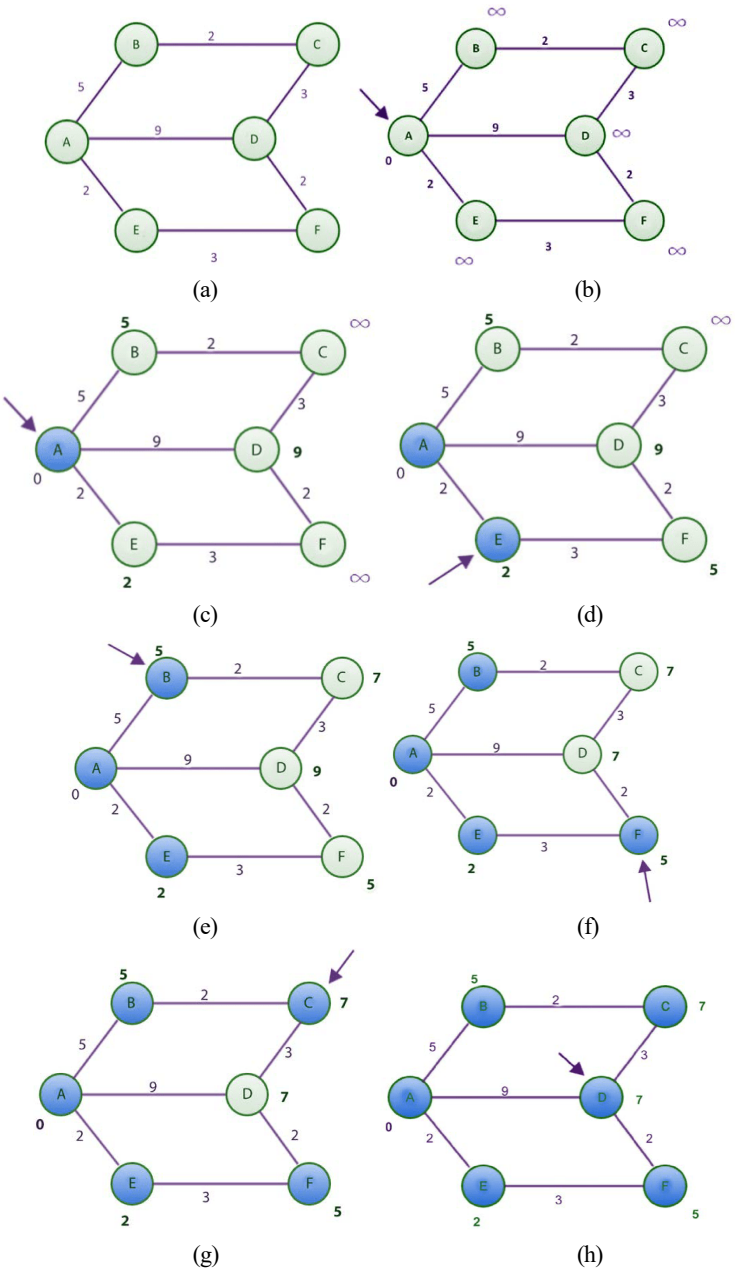
di setiap langkah. Dalam contoh ini, pilihan lokal terbaik adalah selalu memilih digit terbesar yang masih tersedia. Urutan pemilihan tersebut menghasilkan bilangan 965421, yang merupakan bilangan terbesar yang dapat dibentuk dari semua digit yang diberikan tanpa pengulangan. Ilustrasi ini juga menunjukkan bahwa Greedy algorithm sangat mudah dipahami dan diimplementasikan karena prosesnya berlangsung secara bertahap, terarah, dan tidak memerlukan pemeriksaan terhadap seluruh kemungkinan permutasi digit.

Contoh klasik lain dari penggunaan greedy algorithm adalah traveling salesperson problem, ketika pendekatan greedy selalu memilih tujuan terdekat lebih dahulu. Dalam persoalan ini, greedy approach selalu memilih kota terdekat yang belum dikunjungi dari kota saat ini. Dengan cara tersebut, tidak ada jaminan bahwa solusi terbaik secara global pasti diperoleh, tetapi pendekatan ini biasanya menghasilkan solusi yang cukup optimal. Strategi shortest-path semacam ini didasarkan upaya menemukan solusi terbaik untuk masalah lokal dengan harapan bahwa keputusan tersebut akan mengarah ke solusi global. Beberapa persoalan standar yang populer dan dapat diselesaikan menggunakan greedy algorithms antara lain Kruskal's minimum spanning tree, Dijkstra's shortest path problem, The Knapsack problem, Prim's minimal spanning tree algorithm, serta The traveling salesperson problem. Salah satu persoalan yang sangat terkenal dan dapat diselesaikan dengan pendekatan greedy adalah shortest path problem.

Algoritma Dijkstra's

Shortest path problem merupakan persoalan yang menuntut penentuan rute terpendek yang mungkin di antara nodes dalam suatu graph. Salah satu metode yang sangat populer untuk menyelesaikan persoalan ini dengan greedy approach adalah algoritma Dijkstra's. Algoritma ini digunakan untuk menentukan jarak terpendek dari sebuah source node menuju destination node dalam graph. Dijkstra's algorithm dapat diterapkan di weighted directed maupun undirected graphs. Keluaran algoritma ini berupa daftar shortest path dari suatu source node, misalnya A, dalam sebuah weighted graph.

Secara umum, Dijkstra's algorithm bekerja melalui beberapa tahapan. Mula-mula, seluruh nodes ditandai sebagai unvisited, lalu jarak dari source node ke semua nodes diinisialisasi dengan nilai tak hingga, kecuali jarak ke source node itu sendiri yang ditetapkan bernilai nol. Setelah itu, source node dijadikan current node. Dari current node tersebut, seluruh unvisited adjacent nodes dicari, kemudian jarak menuju masing-masing adjacent node dari source node melalui current node dihitung. Jarak baru yang diperoleh lalu dibandingkan dengan jarak yang sebelumnya telah tercatat. Jika jarak yang baru lebih kecil,



Gambar 13.10: Algoritma Dijkstra's (Agarwal, 2022)

maka nilai tersebut ditetapkan sebagai jarak yang baru. Setelah semua unvisited adjacent nodes dari current node selesai dipertimbangkan, current node tersebut ditandai sebagai visited.

Proses ini terus berlangsung sampai destination node telah ditandai sebagai visited, atau sampai daftar unvisited nodes menjadi kosong, yang berarti seluruh unvisited nodes telah dipertimbangkan. Dengan mekanisme tersebut, algoritma Dijkstra's membangun solusi shortest path secara bertahap melalui pemilihan langkah lokal terbaik sesuai prinsip greedy approach.

Gambar 13.10 memperlihatkan Algoritma Dijkstra's sebagai contoh nyata algoritma greedy untuk menentukan shortest path dari source node A ke seluruh node lain dalam sebuah weighted graph. Graf terdiri atas enam node, yaitu A, B, C, D, E, dan F, dengan bobot sisi A–B = 5, B–C = 2, C–D = 3, A–D = 9, A–E = 2, E–F = 3, dan D–F = 2. Gambar dibagi menjadi delapan tahap, yaitu (a) sampai (h), untuk menunjukkan bagaimana nilai jarak terpendek diperbarui secara bertahap. Dalam ilustrasi ini, node yang berwarna biru menunjukkan node yang telah dipilih atau telah dipastikan jarak terpendeknya dari A, sedangkan angka di dekat setiap node menunjukkan jarak sementara atau jarak final dari source node A. Simbol ∞ menandakan bahwa jarak ke node tersebut belum diketahui. Gambar ini sangat tepat dijadikan contoh greedy algorithm karena setiap langkah algoritma selalu memilih unvisited node dengan jarak sementara terkecil, lalu menggunakan node itu untuk memperbarui kemungkinan jarak ke node-node tetangganya.

Tahap (a), graf masih ditampilkan dalam keadaan awal, yaitu hanya memperlihatkan struktur hubungan antarnode beserta bobot setiap sisi. Belum ada node yang diproses, sehingga tahap ini berfungsi sebagai titik awal persoalan shortest path problem.

Tahap (b), A ditetapkan sebagai source node dan langsung diberi jarak 0, sedangkan seluruh node lain masih bernilai ∞.

Tahap (c), A dipilih sebagai current node, lalu seluruh tetangganya diperiksa. Dari A, diperoleh jarak ke B sebesar 5, ke D sebesar 9, dan ke E sebesar 2. Karena itu, label jarak di ketiga node tersebut diperbarui menjadi B = 5, D = 9, dan E = 2, sedangkan C dan F masih tetap ∞. Tahap ini menunjukkan prinsip dasar Dijkstra, yaitu menghitung jarak dari source node ke adjacent nodes melalui node yang sedang aktif.

Tahap (d), algoritma memilih E sebagai node berikutnya karena di antara seluruh unvisited nodes, E memiliki jarak sementara paling kecil, yaitu 2. Inilah keputusan greedy yang pertama: memilih node dengan nilai minimum lokal saat itu. Dari E, satu-satunya jalur baru yang relevan adalah menuju F dengan bobot 3, sehingga jarak dari A ke F melalui E menjadi 2 + 3 = 5. Oleh sebab itu, jarak F diperbarui dari ∞ menjadi 5.

Tahap (e), algoritma memilih B sebagai current node berikutnya. Meskipun F juga bernilai 5, ilustrasi ini memperlihatkan bahwa B dipilih lebih dahulu. Dari B, jalur ke C memiliki bobot 2, sehingga jarak C menjadi 5 + 2 = 7. Dengan demikian, label C diperbarui menjadi 7, sementara nilai D = 9 dan F = 5 tidak berubah dalam tahap ini.

Tahap (f), algoritma berpindah ke F, karena F merupakan unvisited node dengan jarak minimum berikutnya, yaitu 5. Dari F, terdapat jalur ke D dengan bobot 2. Jarak baru ke D melalui F menjadi 5 + 2 = 7, yang ternyata lebih kecil daripada jarak lama 9 yang sebelumnya diperoleh langsung dari A. Akibatnya, label D diperbarui dari 9 menjadi 7. Tahap ini memperlihatkan bahwa Dijkstra's algorithm tidak hanya menetapkan jarak awal, tetapi juga terus membandingkan jarak baru dengan jarak lama dan selalu menyimpan nilai yang lebih kecil. Tahap (g), C dipilih sebagai current node karena memiliki jarak sementara minimum, yaitu 7. Dari C, jalur ke D melalui sisi berbobot 3 menghasilkan jarak 7 + 3 = 10, tetapi nilai ini lebih besar daripada jarak D = 7 yang telah ada. Oleh sebab itu, tidak ada perubahan label dalam tahap ini. Tahap (h) menunjukkan langkah terakhir, yaitu pemilihan D dengan jarak 7. Karena semua node kini telah memperoleh jarak terpendek dari A, proses algoritma selesai. Pemilihan node oleh

Algoritma Dijkstra's adalah A → E → B → F → C → D. Hasil akhirnya adalah jarak terpendek dari A ke setiap node, yaitu A = 0, E = 2, B = 5, F = 5, C = 7, dan D = 7.

Program Algoritma Dijkstra's sebagai contoh algoritma greedy dapat dilihat di https://colab.research.google.com/drive/1ju2VJgVTKvKod-uGKhwzZt5jw8Ec-rk5?usp=sharing.

13.5 Algoritma Backtracking

Backtracking merupakan bentuk recursion yang sangat berguna untuk jenis persoalan tertentu, misalnya penelusuran tree structures, ketika setiap node menyediakan sejumlah pilihan dan satu di antaranya harus dipilih. Setelah suatu pilihan diambil, proses berlanjut ke himpunan pilihan berikutnya yang berbeda. Bergantung rangkaian keputusan yang dibuat, proses tersebut dapat berakhir di goal state atau justru mencapai dead end. Jika yang dicapai adalah dead end, maka proses harus melakukan backtrack ke *previous node* dan menelusuri *different branch*. Dalam pengertian ini, Backtracking dapat dipahami sebagai metode *divide and conquer* untuk pencarian menyeluruh. Ciri dari pendekatan ini yaitu kemampuannya melakukan memangkas cabang yang dipastikan tidak dapat menghasilkan solusi.

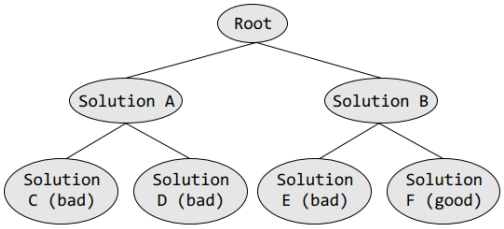
Secara lebih umum, Backtracking merupakan algoritma umum yang digunakan untuk menemukan seluruh solusi atau sebagian solusi dari suatu computational problem dengan cara membangun kandidat solusi secara bertahap. Di setiap langkah, valid candidates terus diperluas, sedangkan kandidat lain yang tidak layak segera dibuang. Dengan kata lain, setiap partial candidate c langsung ditinggalkan begitu diketahui bahwa c tidak mungkin menjadi solusi yang valid bagi masalah yang sedang diselesaikan. Dari sinilah istilah backtrack berasal, karena algoritma kembali ke keadaan sebelumnya untuk mencoba kemungkinan lain. Konsep Backtracking hanya relevan bagi masalah yang mendukung keberadaan partial candidate solutions serta memiliki pengujian yang relatif cepat untuk menentukan apakah partial candidate solution tersebut masih dapat dikembangkan menjadi valid solution.

Pendekatan Backtracking digunakan secara luas untuk menyelesaikan *constraint satisfaction problems*, seperti Sudoku, eight queen's problem, crosswords, verbal arithmetic, dan Solitaire. Selain itu, pendekatan ini juga diterapkan dalam combinational optimization problem, misalnya parsing dan Knapsack problem. Beberapa bahasa pemrograman logika yang secara internal menggunakan backtracking antara lain Icon, Planner, dan Prolog. Di samping itu, diff, yaitu version comparing engine untuk perangkat lunak MediaWiki, juga memanfaatkan pendekatan ini. Dengan cakupan penggunaan yang luas tersebut, Backtracking menempati posisi penting sebagai strategi penyelesaian masalah yang fleksibel di berbagai bidang komputasi.

Backtracking sering disebut meta-heuristic algorithm, bukan sebagai algoritma yang sangat spesifik, karena pendekatan ini dijamin dapat menemukan seluruh solusi untuk suatu masalah berhingga dalam waktu yang terbatas. Istilah meta-heuristic menunjukkan bahwa algoritma ini bekerja berdasarkan prosedur-prosedur yang diberikan pengguna untuk mendefinisikan masalah yang akan diselesaikan, sifat dari partial candidates, serta cara memperluasnya menjadi complete candidates. Oleh sebab itu, Backtracking tidak sekadar menjadi langkah mekanis yang kaku, melainkan merupakan kerangka umum yang dapat disesuaikan dengan struktur persoalan yang dihadapi.

Jika proses Backtracking dikonseptualisasikan sebagai search tree, maka partial candidates dapat dipandang sebagai nodes dalam tree tersebut. Setiap partial candidate memiliki child nodes yang merepresentasikan kandidat-kandidat baru yang berbeda darinya melalui satu langkah perluasan. Adapun leaf nodes merupakan partial candidates yang tidak dapat diperluas lebih lanjut. Dalam pelaksanaannya, backtracking algorithm menelusuri search tree secara depth-first order, dimulai dari root node. Di setiap node c, algoritma memeriksa apakah c masih dapat dikembangkan menjadi valid solution. Jika tidak, maka seluruh sub-tree yang berakar di c akan di-pruned. Sebaliknya, jika masih ada kemungkinan bahwa c dapat mengarah ke valid solution, maka algoritma terlebih dahulu memeriksa apakah c itu sendiri sudah

merupakan valid solution. Jika ya, maka c dinyatakan sebagai solusi yang valid. Jika belum, maka algoritma secara recursively menelusuri seluruh sub-trees dari c. Dengan cara inilah Backtracking bekerja secara sistematis untuk menjelajahi ruang solusi sekaligus membatasi pencarian hanya ke cabang-cabang yang masih berpotensi menghasilkan jawaban yang benar.



Gambar 13.11: Ilustrasi Algoritma (Thareja, 2014)

Proses Backtracking dapat dijelaskan secara bertahap dalam Gambar 13.11. Step 1, penelusuran dimulai dari root node. Dari titik ini tersedia dua pilihan, yaitu Solution A dan Solution B, lalu dipilih Solution A. Step 2, setelah mencapai Solution A, tersedia lagi dua pilihan, yaitu Solution C dan Solution D, kemudian dipilih Solution C. Step 3, karena Solution C bukan good (valid and possible) candidate, proses melakukan backtrack ke Solution A dan berikutnya memilih Solution D. Step 4, ternyata Solution D juga bukan good (valid and possible) candidate, sehingga proses kembali melakukan backtrack ke Solution A. Karena tidak ada lagi pilihan yang tersedia di Solution A, penelusuran dialihkan ke Solution B. Step 5, di Solution B terdapat dua pilihan, yaitu Solution E dan Solution f, lalu dipilih Solution E. Step 6, karena Solution E juga bukan good (valid and possible) candidate, proses kembali melakukan backtrack ke Solution B dan selanjutnya memilih Solution f. Step 7, Solution f terbukti merupakan good (valid and possible) candidate, sehingga algoritma berhenti di titik tersebut.

Keunggulan utama teknik Backtracking yaitu kemampuannya menghentikan penelusuran lebih awal dan tidak mengeksplorasi kandidat yang tidak mungkin mengarah ke *valid possible solutions*. Dengan strategi ini, jumlah nodes yang harus diperiksa menjadi lebih sedikit, sehingga ruang pencarian dapat dipersempit secara efektif. Oleh sebab itu, algoritma ini dapat digunakan untuk menyelesaikan exponential time problems dalam waktu yang masih masuk akal. Meskipun demikian, perlu ditegaskan bahwa Backtracking bukan optimization technique. Teknik ini tidak secara langsung mencari solusi paling optimal, melainkan hanya berfungsi mengurangi search space agar proses pencarian solusi menjadi lebih efisien (Thareja, 2014).

Pustaka

Agarwal, D. B. (2022). *Hands-On Data Structures and Algorithms with Python* (Third). Packt Publishing.

Alaa, M., Anzy, A., Mazhit, Z., Fadhil, A., Anzy, A., Algarni, A., Akhmedov, R., & Bauyrzhan, A. (2024). Comparative Analysis of Sorting Algorithms : A Review. *2024 11th International Conference on Soft Computing & Machine Intelligence (ISCMI)*, *11*, 88–100.

Aung, H. H. (2019). Analysis and Comparative of Sorting Algorithms. *International Journal of Trend in Scientific Research and Development (IJTSRD)*, *3*(5), 1049–1053.

Baka, B. (2017). *Python Data Structures and Algorithm* (First). Packt Publishing.

Cibulka, J. (2011). On average and highest number of flips in pancake sorting. *Theoretical Computer Science*, *412*(8–10), 822–834. https://doi.org/10.1016/j.tcs.2010.11.028

Das, U., Lawson, A., & Mayfield, C. (2024). *Introduction to Python Programming*.

Dijkstra, E. W. (1981). Smoothsort, An Alternative For Sorting In Situ. *Science of Computer Programming*, *1*, 223–233.

Fenyi, A., Fosu, M., & Bright Appiah. (2020). Comparative Analysis of Comparison and Non Comparison based Sorting Algorithms. *International Journal of Computer Applications (0975 – 8887)*, *175*(28), 22–25.

Gill, S. K. (2018). A Comparative Study of Various Sorting Algorithms. *Special Issue Based on Proceedings of 4th International Conference on Cyber Security (ICCS) 2018*, *4*.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). *Data Structures and Algorithms in Python*. Don Fowley.

Heinold, B. (2025). *An Intuitive Introduction to Data Structures , 2nd Edition* (Second).

Karimov, E. (2022). *Data Structures and Algorithms in Python*.

Marcellino, M., & Margi, K. (2021). Comparative of Advanced Sorting Algorithms (Quick Sort , Heap Sort , Merge Sort , Intro Sort , Radix Sort) Based on Time and Memory Usage. *2021 1st International Conference on Computer Science and Artificial Intelligence (ICCSAI)*, *1*(October), 154–160.

Neubert, K.-D. (1997). Flash-Sort: Sorting by in situ Permutation. *Proceedings of the EuroFORTH'97 - Conference, Oxford, England,Sept.26-28 1997*, *1*(09), 1–10.

Peters, H., Schulz-hildebrandt, O., Luttenberger, N., & Sort, A. B. (2012). A Novel Sorting Algorithm for Many-Core Architectures based on Adaptive Bitonic Sort. *IEEE 26th International Parallel and Distributed Processing Symposium A*, *1*(227–237).

Rahmawati, Y., & Setiawan, H. (2026). Analisis Perbandingan Algoritma Sorting Terhadap Data Numerik , Karakter , dan String di Python Berdasarkan Waktu, Memori Puncak , dan Perpindahan Data. *JATI (Jurnal Mahasiswa Teknik Informatika)*, *10*(2), 1–9.

Rohith, P., Datta, S., Kamath, C. D., Kini, N. G., & B, A. R. (2024). Parallelization of Pigeonhole Sort for Efficient Data Sorting. *Grenze International Journal of Engineering and Technology*, *1*(6), 6017–6021.

Rumapea, Y. Y. P. (2017). Analisis Perbandingan Metode Algoritma Quick Sort dan Merge Sort dalam Pengurutan Data terhadap Jumlah Langkah dan Waktu. *Jurnal METHODIKA*, *3*(2), 5–9.

Sundaramoorthy, S., & Karunanidhi, G. (2025). A Systematic Analysis on Performance and Computational Complexity of Sorting Algorithms. *Discover Computing*, *28*(250), 1–30.

Thareja, R. (2014). *Data Structures Using C* (Second). Oxford University Press.

https://www.w3schools.com/python/default.asp

Biodata Penulis

Yunianita Rahmawati lahir di Magetan. Penulis menyelesaikan pendidikan S-1 di STIKOM Surabaya (sekarang UNDIKA), S-2 Prodi Teknologi Informasi di Institut Sains dan Teknologi Terpadu Surabaya, dan S-3 Prodi Ilmu Komputer di ITS Surabaya dengan program beasiswa internal UMSIDA. Penulis merupakan dosen tetap di Program Studi Informatika, Universitas Muhammadiyah Sidoarjo. Mata kuliah yang diampu antara lain Algoritma Struktur Data, Data Mining ,dan Sistem Informasi. Penulis aktif terlibat dalam kegiatan penelitian dan pengabdian kepada masyarakat (abdimas). Daftar publikasi hasil penelitiandan abdimas penulis bersama rekan seprofesi dan mahasiswa tersedia di https://scholar.google.com/citations?user=QNx9MeMAAAAJ&hl=en.

Tanggal 30 April 1986 dilahirkan seorang anak laki-laki yang diberi nama Hamzah Setiawan, saat ini sebagai dosen di Universitas Muhammadiyah Sidoarjo Program Studi Informatika, yang telah menyelesaikanprogram sarjana strata 1 di Universitas Trunojoyo Madura Program Studi Teknik Informatika dan menyelesaikan program Pasca Sarjana di Institut Sains dan Teknologi Terpadu Surabaya di program studi teknologi informasi. Saat ini sedang menempuh S-3 di Prodi Ilmu Komputer ITS Surabaya dengan program beasiswa internal UMSIDA. Selain mempunyai kesibukan mengajar sebagai dosen penulis juga mempunyai penugasan struktural di Universitas Muhammadiyah Sidoarjo sebagai kepala bidang kerjasama dan PMB. penulis juga aktif diberbagai organisasi baik organisasi muhammadiyah juga organisasi kemasyarakatan, yang mempunyai motto hidup “hidup tidak boleh menyerah dan selalu berusaha untuk menjadi orang yang bermanfaat untuk orang lain”.

ISBN 978-623-464-141-7 (PDF)

UMSIDA PRESS
Universitas Muhammadiyah Sidoarjo
Jl. Mojopahit No. 666B
Sidoarjo, Jawa Timur