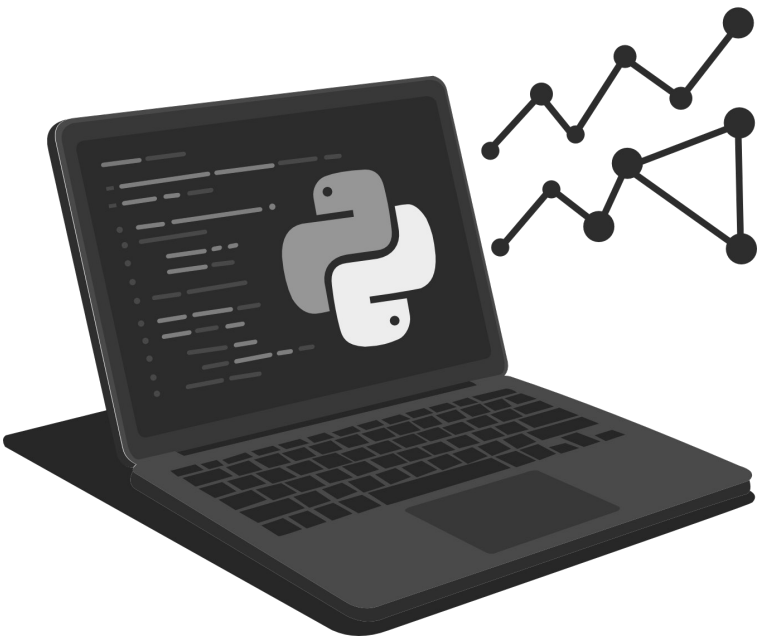
# STRUKTUR DATA dan ALGORITMA DALAM PYTHON

## Panduan Praktis

Heru Saputra, Shevti Arbekti Arman, Muhammad Fairuzabadi,
Faqihuddin Al Anshori, Ali Impron, Sugeng Winardi, Ester Lumba,
Firdiyan syah, Nurirwan Saputra, Marsellus Oton Kadang, Widi Hastomo

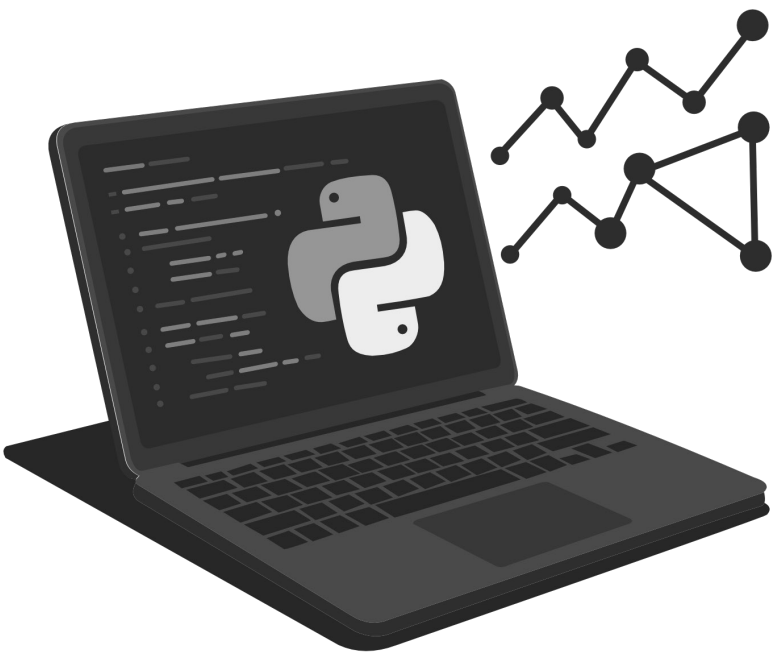
# STRUKTUR DATA dan ALGORITMA DALAM PYTHON

## Panduan Praktis



# STRUKTUR DATA dan ALGORITMA DALAM PYTHON

## Panduan Praktis



Heru Saputra, Shevti Arbekti Arman, Muhammad Fairuzabadi,
Faqihuddin Al Anshori, Ali Impron, Sugeng Winardi, Ester Lumba,
Firdiyan syah, Nurirwan Saputra, Marsellus Oton Kadang, Widi Hastomo

yash media.
IKAPI
IKATAN PENERBIT INDONESIA

Tata Letak : Tim Yash Media
Rancang Sampul : Mu`in
Editor : Muhammad Fairuzabadi
Halaman : xvi + 272 hlm.
Ukuran : 15 x 23 cm
Cetakan Pertama : Agustus 2025

**ISBN : 978-634-7327-39-0**


Diterbitkan pertama kali oleh :
**Yash Media**
Anggota IKAPI 205/DIY/2025
Jl. Imogiri Barat Rt 04, Tanjung, Bangunharjo, Kec. Sewon, Kab. Bantul,
Daerah Istimewa Yogyakarta 55188.
Email: yashmediaco@gmail.com"
**https://yashmedia.id/**

# Kata Pengantar

Puji syukur ke hadirat Tuhan Yang Maha Esa senantiasa kami panjatkan atas segala limpahan rahmat, karunia, dan petunjuk-Nya, sehingga buku ini yang berjudul "**Struktur Data dan Algoritma dalam Python: Panduan Praktis**" dapat diselesaikan dengan baik. Buku ini disusun untuk memberikan referensi yang komprehensif dan aplikatif bagi mahasiswa, dosen, dan praktisi teknologi informasi yang ingin memahami konsep dan implementasi struktur data dan algoritma dengan pendekatan praktis menggunakan Python.

Struktur data dan algoritma merupakan fondasi penting dalam pengembangan perangkat lunak dan sistem cerdas. Pemahaman terhadap struktur data memungkinkan pengelolaan informasi secara efisien, sementara algoritma menentukan bagaimana proses berjalan dengan optimal. Python dipilih karena sintaksisnya yang sederhana, fleksibel, dan sangat mendukung pembelajaran konsep algoritmik, baik untuk pemula maupun praktisi profesional.

Buku ini terdiri dari sebelas bab yang disusun secara sistematis, dimulai dari dasar-dasar algoritma dan Python, struktur data fundamental seperti array, list, set, dan dictionary, hingga implementasi struktur data lanjutan seperti stack, queue, dan berbagai bentuk linked list. Di bagian akhir, pembaca diajak mempelajari teknik pencarian dan pengurutan data, serta studi kasus nyata dalam bidang data science dan machine learning, sehingga relevansi praktis materi dapat langsung dirasakan.

Setiap bab dilengkapi dengan penjelasan teoritis, ilustrasi, serta contoh kode Python yang dapat dieksekusi langsung. Dengan demikian, pembaca tidak hanya memahami konsep, tetapi juga mampu mengimplementasikannya dalam berbagai konteks pengembangan sistem nyata. Pendekatan studi kasus yang digunakan bertujuan menjembatani kesenjangan antara teori akademik dan kebutuhan industri teknologi saat ini.

Penulis menyadari bahwa buku ini masih jauh dari sempurna. Oleh karena itu, kritik dan saran dari pembaca sangat kami harapkan untuk perbaikan pada edisi mendatang. Ucapan terima kasih kami sampaikan kepada semua pihak yang telah membantu dan mendukung proses penyusunan buku ini, baik secara langsung maupun tidak langsung.

Akhir kata, semoga buku ini dapat menjadi referensi bermanfaat dalam membangun dasar yang kuat di bidang algoritma dan struktur data, serta mendorong pembaca untuk terus belajar dan berinovasi dalam bidang teknologi informasi.

Yogyakarta, Agustus 2025

**Tim Penulis**

# Daftar Isi

# Daftar Gambar

# Daftar Tabel

# Bab 1
# Pengantar Algoritma dan Struktur Data

*"Algoritma adalah jiwa dari pemrograman, dan struktur data adalah tubuhnya. Tanpa keduanya, logika tidak bisa hidup."* — **Anonim**

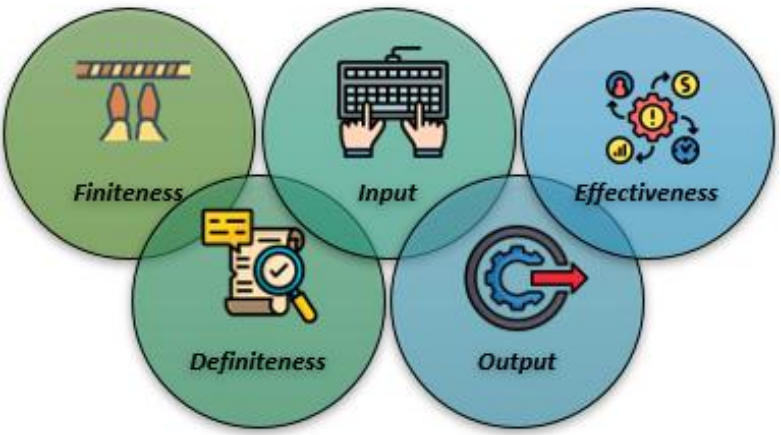
## 1.1 Definisi dan Konsep Dasar

Algoritma adalah sekumpulan instruksi yang disusun secara logis dan sistematis untuk menyelesaikan suatu permasalahan atau mencapai tujuan tertentu. Dalam ilmu komputer, algoritma menjadi pedoman utama dalam menuliskan program yang efisien dan efektif (Cormen et al., 2009). Penyusunan algoritma yang baik akan menghasilkan solusi yang dapat dijalankan oleh komputer dengan sumber daya yang optimal, baik dari segi waktu maupun memori. Sedangkan Struktur data, berperan penting dalam cara bagaimana data disimpan dan diakses di dalam program. Pemilihan struktur data yang tepat akan meningkatkan efisiensi proses pencarian, penyisipan, dan penghapusan data. Oleh karena itu, pemahaman yang mendalam tentang hubungan antara algoritma dan struktur data sangat penting dalam pengembangan perangkat lunak yang handal dan skalabel.

### 1.1.1 Karakteristik Algoritma

Algoritma yang baik harus memenuhi beberapa karakteristik penting agar dapat dijalankan secara efektif oleh komputer. Karakteristik pertama adalah ***Finiteness***, yaitu algoritma harus memiliki akhir atau batas yang jelas setelah sejumlah langkah tertentu. Algoritma yang tidak memiliki akhir akan menyebabkan proses berjalan tanpa henti. Kedua, ***Definiteness*** menekankan bahwa setiap langkah dalam algoritma harus didefinisikan secara jelas dan tidak menimbulkan kebingungan. Hal ini penting agar

algoritma dapat diinterpretasikan dan diimplementasikan dengan benar oleh mesin atau manusia. Karakteristik berikutnya adalah ***Input***, yaitu algoritma harus menerima nol atau lebih data masukan sebagai dasar pemrosesan. Kemudian, ***Output*** adalah hasil akhir dari algoritma yang harus dihasilkan minimal satu keluaran yang relevan terhadap masukan yang diberikan. Terakhir, ***Effectiveness*** berarti setiap langkah dalam algoritma harus cukup sederhana sehingga dapat dikerjakan secara manual dalam waktu yang wajar. Kelima karakteristik ini menjadi acuan dalam merancang algoritma yang tidak hanya logis, tetapi juga praktis dan dapat diimplementasikan secara efisien (Brassard & Bratley, 1996).




Gambar 1.1: Karakteristik Algoritma

### 1.1.2 Notasi dan Representasi Algoritma

Notasi dan representasi algoritma sangat penting dalam proses perancangan dan dokumentasi logika pemrograman. Representasi ini memudahkan programmer dalam memahami dan mengkomunikasikan solusi yang dirancang. Ada beberapa cara untuk merepresentasikan algoritma, masing-masing memiliki keunggulan dan digunakan sesuai kebutuhan.

## Deskripsi Naratif

Deskripsi naratif adalah metode representasi algoritma dengan menjelaskan langkah-langkah penyelesaian masalah secara verbal, menggunakan bahasa natural (seperti Bahasa Indonesia atau Inggris). Cara ini tidak menggunakan simbol khusus ataupun struktur formal.

**Kelebihan:**

- Mudah dipahami oleh audiens non-teknis.
- Cocok untuk dokumentasi awal atau pengantar.
- Sederhana dan tidak memerlukan perangkat bantu khusus.

**Kekurangan:**

- Kurang efisien untuk masalah yang kompleks.
- Sulit untuk digunakan dalam implementasi langsung ke dalam kode program.

**Contoh Deskripsi Naratif:**

Misalnya kita ingin mencari bilangan terbesar dari tiga bilangan:

```
"Pertama, bandingkan bilangan pertama dan kedua. Simpan bilangan yang lebih besar. Kemudian, bandingkan bilangan tersebut dengan bilangan ketiga. Bilangan yang lebih besar dari hasil perbandingan terakhir adalah bilangan terbesar."
```

## Pseudocode (Pseudosintaks)

Pseudocode adalah representasi algoritma dalam bentuk tulisan yang menyerupai bahasa pemrograman, tetapi tidak mengikuti aturan sintaks formal dari bahasa pemrograman tertentu. Pseudocode bertujuan untuk menggabungkan pemikiran logis dengan struktur yang mudah diterjemahkan ke dalam kode.

**Kelebihan:**

- Lebih dekat ke kode nyata dibanding deskripsi naratif.
- Mudah dipahami oleh programmer.
- Bahasa fleksibel, tidak terikat satu bahasa pemrograman tertentu.

**Kekurangan:**

- Tidak bisa langsung dijalankan oleh komputer.
- Membutuhkan pemahaman dasar pemrograman bagi pembaca.

**Contoh Pseudocode:**

```
Algoritma CariBilanganTerbesar
Deklarasi:
    a, b, c: integer
    max: integer

Deskripsi:
    Input a, b, c
    max ← a
    Jika b > max maka
        max ← b
    Jika c > max maka
        max ← c
    Tampilkan "Bilangan terbesar adalah", max
```

## Diagram Alir (Flowchart)

Flowchart adalah representasi algoritma secara visual yang menggunakan simbol-simbol standar seperti persegi panjang (proses), belah ketupat (keputusan), jajar genjang (input/output), dan panah (alur). Flowchart banyak digunakan untuk menggambarkan alur logika dan keputusan.

**Kelebihan:**

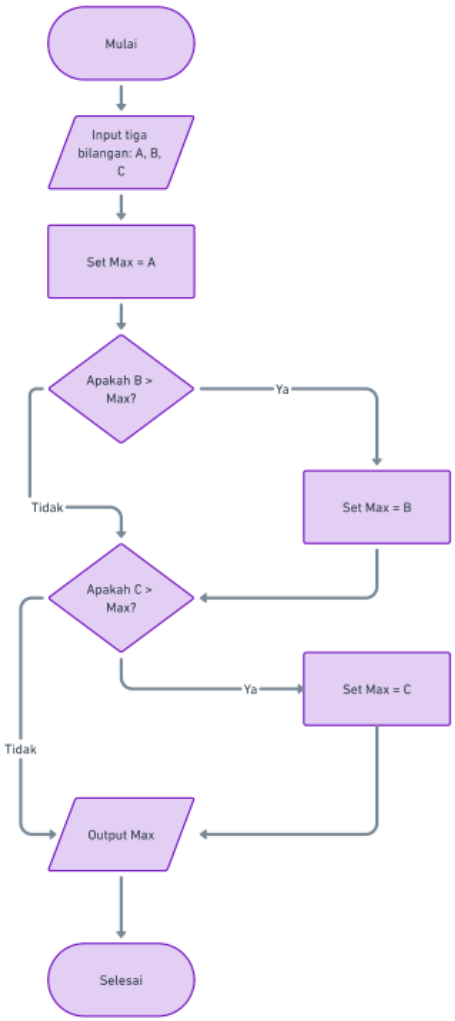
- Memberikan gambaran visual yang mudah diikuti.
- Cocok untuk menjelaskan proses kompleks dan percabangan.
- Memudahkan dalam mendeteksi kesalahan logika.

**Kekurangan:**

- Membutuhkan waktu lebih lama untuk menggambar.
- Tidak fleksibel untuk perubahan cepat seperti pada pseudocode.

### Contoh Flowchart:

Berikut ini adalah deskripsi visual flowchart untuk mencari bilangan terbesar dari tiga bilangan:




Gambar 1.2: Contoh Flowchart

Tabel 1.1: Perbandingan Representasi Algoritma

| Representasi | Cocok Untuk | Kelebihan | Kekurangan |
|---|---|---|---|
| Naratif | Penjelasan awal, non-teknis | Mudah dipahami umum, simpel | Tidak efisien untuk masalah kompleks |
| Pseudocode | Perancangan logika program | Mendekati kode, fleksibel | Tidak dapat dijalankan langsung |
| Flowchart | Visualisasi proses | Menarik secara visual, mudah diikuti | Sulit diubah, lebih lambat dibuat |

## 1.2 Struktur Data: Definisi dan Klasifikasi

Struktur data adalah cara menyusun, mengelola, dan menyimpan data secara efisien untuk digunakan dalam program komputer. Struktur data memungkinkan pengolahan data secara sistematis agar informasi dapat diakses dan dimanipulasi dengan mudah. Pemilihan struktur data yang tepat sangat penting karena berdampak langsung pada efisiensi algoritma yang menggunakannya. Struktur data diklasifikasikan ke dalam dua kategori utama: struktur data primitif dan struktur data non-primitif. Struktur data primitif meliputi tipe data dasar yang disediakan langsung oleh bahasa pemrograman, seperti *integer* dan *boolean*. Sementara itu, struktur data non-primitif mencakup tipe data yang lebih kompleks seperti *array, list, tree*, dan *graph* yang biasanya dibentuk dari kombinasi struktur data primitif dan memungkinkan representasi data yang lebih fleksibel dan efisien (Goodrich et al., 2014).

### 1.2.1 Tipe Data Primitif

Tipe data primitif adalah jenis data dasar yang disediakan langsung oleh sebagian besar bahasa pemrograman, termasuk Python. Tipe data ini merupakan fondasi dari representasi dan manipulasi informasi dalam

program. Dengan menggunakan tipe data primitif, programmer dapat melakukan operasi dasar seperti perhitungan aritmatika, manipulasi karakter, dan pengambilan keputusan logika. Beberapa contoh tipe data primitif meliputi (Lambert, 2019):

- `Integer`: digunakan untuk menyimpan bilangan bulat, seperti 1, -10, 250.
- `Float`: digunakan untuk menyimpan bilangan desimal atau pecahan, seperti 3.14, -0.001, 2.0.
- `Character`: menyimpan satu karakter, misalnya 'a', 'Z', atau simbol seperti '#'. Dalam Python, karakter direpresentasikan sebagai string dengan panjang satu.
- `Boolean`: merepresentasikan nilai logika, yaitu *True* dan *False*. Tipe data ini sering digunakan dalam ekspresi kondisi dan kontrol alur program seperti *if-else* atau perulangan.

### 1.2.2 Abstraksi Data dan ADT (Abstract Data Type)

Abstraksi data adalah konsep penting dalam pemrograman yang memungkinkan pengembang perangkat lunak untuk fokus pada operasi yang dapat dilakukan terhadap data, tanpa perlu mengetahui detail bagaimana data tersebut disimpan atau diimplementasikan. Pendekatan ini membantu dalam menciptakan sistem yang modular dan lebih mudah untuk dikembangkan dan dirawat. Dengan abstraksi data, kita menyembunyikan kompleksitas implementasi dan hanya mengekspos antarmuka atau operasi dasar yang relevan. *Abstract Data Type* (ADT) adalah model matematis dari struktur data yang mendefinisikan jenis data beserta operasi-operasi yang dapat dilakukan terhadapnya, tanpa mempermasalahkan bagaimana operasi tersebut diimplementasikan. Contoh-contoh ADT yang umum digunakan dalam pemrograman antara lain (Kurniawan, 2018):

- ***List* (Daftar)**: Kumpulan elemen yang diakses berdasarkan urutan.
- ***Stack* (Tumpukan)**: Struktur data dengan prinsip LIFO (*Last In First Out*).

- ***Queue* (Antrian)**: Struktur data dengan prinsip FIFO (*First In First Out*).
- ***Tree* (Pohon)**: Struktur hierarkis dengan simpul (*node*) dan cabang.
- ***Graph* (*Graf*)**: Kumpulan simpul yang saling terhubung melalui sisi (*edg*e).

ADT membantu dalam pengembangan algoritma yang efisien dan struktur program yang lebih bersih dan terorganisir.

## 1.3 Sejarah dan Evolusi Algoritma


Gambar 1.3: Tokoh Awal Alogoritma

Algoritma telah digunakan jauh sebelum kemunculan komputer modern. Istilah "algoritma" berasal dari nama ilmuwan Persia, Al-Khwarizmi, yang hidup pada abad ke-9. Ia menulis buku berjudul *Al-Kitab al-Mukhtasar fi Hisab al-Jabr wal-Muqabala*, yang menjadi dasar dari ilmu aljabar dan konsep algoritmik. Pada masa itu, algoritma digunakan terutama dalam konteks matematika untuk memecahkan permasalahan numerik seperti sistem persamaan, konversi bilangan, dan pencarian akar. Seiring perkembangan ilmu pengetahuan, algoritma mulai diterapkan secara lebih luas, termasuk dalam bidang

logika dan aritmetika oleh tokoh-tokoh seperti *Blaise Pascal* dan *Charles Babbage* (Knuth, 1997).

Perkembangan komputer digital pada abad ke-20 menjadi titik balik penting dalam evolusi algoritma. Algoritma tidak hanya digunakan untuk menyelesaikan persoalan matematis, tetapi juga diterapkan untuk mengendalikan logika program dalam perangkat lunak komputer. Di era modern, algoritma memainkan peran vital dalam pengembangan teknologi mutakhir, termasuk kecerdasan buatan, kriptografi, big data, dan komputasi awan. Algoritma tidak lagi hanya dihitung berdasarkan efisiensi waktu, tetapi juga dari aspek skalabilitas, adaptabilitas, dan kompleksitasnya dalam memecahkan masalah.

### 1.3.1 Algoritma Klasik

Algoritma klasik adalah metode-metode yang telah teruji waktu dan menjadi pilar dalam dunia algoritma dan pemrograman. Meskipun sebagian besar ditemukan berabad-abad lalu atau pada masa awal perkembangan komputer, algoritma-algoritma ini tetap digunakan karena kesederhanaannya, keandalan, dan kemampuannya menyelesaikan berbagai masalah fundamental. Mereka bukan hanya berfungsi sebagai alat praktis, tetapi juga sebagai sarana edukatif dalam memperkenalkan konsep logika, efisiensi, dan struktur dalam algoritma.

Berikut adalah tiga algoritma klasik paling penting beserta penjelasan dan contoh penggunaannya:

#### Euclidean Algorithm

Euclidean Algorithm adalah salah satu algoritma tertua dalam sejarah matematika yang ditemukan oleh Euclid. Tujuannya adalah untuk menghitung Faktor Persekutuan Terbesar (FPB) dari dua bilangan bulat. Prinsip dasarnya adalah bahwa FPB dari dua bilangan tidak berubah jika bilangan yang lebih besar dikurangi (atau dibagi) dengan bilangan yang lebih kecil secara berulang.

**Langkah-Langkah:**

```
1. Misalkan dua bilangan: a dan b, dengan a > b.
2. Bagi a dengan b, dan simpan sisanya r.
3. Gantikan a dengan b, dan b dengan r.
4. Ulangi langkah 2 dan 3 sampai r = 0.
5. Ketika r = 0, nilai b saat itu adalah FPB.
```

**Contoh**:

Mencari FPB dari 252 dan 105:

- 252 ÷ 105 = 2 sisa 42
- 105 ÷ 42 = 2 sisa 21
- 42 ÷ 21 = 2 sisa 0
- FPB = 21

**Kelebihan:**

- Sangat efisien bahkan untuk bilangan besar.
- Diterapkan dalam teori bilangan, kriptografi, dan sistem bilangan.

### Bubble Sort

Bubble Sort adalah algoritma pengurutan sederhana yang bekerja dengan cara membandingkan dua elemen berdekatan dan menukarnya jika urutannya salah. Proses ini diulang sampai seluruh elemen terurut. Meskipun lambat dibanding metode modern seperti Quick Sort atau Merge Sort, algoritma ini tetap digunakan untuk tujuan pendidikan karena konsepnya yang mudah dimengerti.

**Langkah-Langkah:**

```
1. Bandingkan elemen ke-1 dan ke-2, tukar jika perlu.
2. Bandingkan elemen ke-2 dan ke-3, tukar jika perlu.
3. Lanjutkan hingga akhir array – satu iterasi selesai.
4. Ulangi proses sampai tidak ada lagi pertukaran.
```

**Contoh (Data: [5, 1, 4, 2, 8]):**

- Iterasi 1: [1, 4, 2, 5, 8]

- Iterasi 2: [1, 2, 4, 5, 8]
- Iterasi 3: [1, 2, 4, 5, 8] (tidak ada pertukaran, selesai)

**Kelebihan:**

- Sederhana dan intuitif.
- Cocok untuk dataset kecil.

**Kekurangan:**

Kompleksitas waktu $O(n^2)$ — sangat lambat untuk data besar.

### Binary Search

Binary Search adalah algoritma pencarian cepat yang bekerja hanya pada daftar yang telah diurutkan. Algoritma ini membagi ruang pencarian menjadi dua bagian setiap iterasi, sehingga mempercepat proses pencarian secara signifikan.

**Langkah-Langkah:**

```
1. Tentukan indeks awal (low) dan akhir (high) dari array.
2. Hitung indeks tengah: mid = (low + high) / 2.
3. Bandingkan nilai di mid dengan target:
    • Jika sama: selesai.
    • Jika lebih kecil: geser pencarian ke kanan (low = mid
      + 1).
    • Jika lebih besar: geser pencarian ke kiri (high = mid
      - 1).
4. Ulangi hingga ditemukan atau low > high.
```

**Contoh (Data: [3, 5, 8, 10, 14], cari 10):**

- mid = 2 → 8 → kurang → cari di kanan
- mid = 3 → 10 → cocok → selesai

**Kelebihan:**

- Kompleksitas waktu O(log n) — sangat efisien.
- Ideal untuk pencarian dalam data besar yang terurut.

**Kekurangan:**

Tidak dapat digunakan pada data yang belum diurutkan.

Ketiga algoritma ini: Euclidean Algorithm, Bubble Sort, dan Binary Search adalah contoh klasik yang tidak hanya memiliki nilai historis, tetapi juga pedagogis. Mereka membantu memperkenalkan:

- Logika algoritmik dasar
- Efisiensi waktu dan kompleksitas
- Struktur kontrol (perulangan, kondisi, rekursi)

Sebagai fondasi dari berbagai algoritma lanjutan, memahami algoritma klasik merupakan langkah awal yang sangat penting bagi setiap pembelajar dalam bidang ilmu komputer, matematika komputasi, dan teknik pemrograman.

### 1.3.2 Algoritma Modern

Algoritma modern adalah kumpulan metode komputasi yang dirancang untuk menangani masalah yang kompleks, besar, dan dinamis dengan efisiensi yang tinggi. Tidak hanya berfokus pada kecepatan dan optimalisasi, algoritma modern juga sering dikombinasikan dengan pendekatan statistik, pembelajaran mesin, serta teknologi paralel dan terdistribusi (Dasgupta et al., 2006).

Algoritma modern menjadi tulang punggung berbagai aplikasi kontemporer, mulai dari pencarian di mesin Google, pengenalan wajah, rekomendasi belanja daring, hingga pengambilan keputusan otonom dalam kendaraan tanpa pengemudi.

**Divide and Conquer**

Divide and Conquer adalah strategi algoritmik yang memecah suatu masalah menjadi sub-masalah yang lebih kecil, menyelesaikannya (sering kali secara rekursif), dan kemudian menggabungkan hasilnya untuk membentuk solusi akhir.

**Langkah Umum:**

- **Divide**: Bagi masalah menjadi dua atau lebih sub-masalah.
- **Conquer**: Selesaikan sub-masalah tersebut secara rekursif.
- **Combine**: Gabungkan hasil sub-masalah menjadi solusi akhir.

**Contoh:**

- **Merge Sort**: Mengurutkan daftar dengan membagi menjadi dua bagian, mengurutkan masing-masing, lalu menggabungkannya.
- **Quick Sort**: Memilih pivot, mempartisi data menjadi dua berdasarkan pivot, lalu mengurutkan masing-masing bagian secara rekursif.
- **Fast Fourier Transform (FFT)**: Digunakan dalam pemrosesan sinyal dan citra.

**Kelebihan:**

- Efisien untuk dataset besar.
- Cocok untuk pemrosesan paralel dan rekursif.
- Kompleksitas waktu umumnya O(n log n) untuk pengurutan.

### Greedy Algorithm

Algoritma greedy mengambil keputusan terbaik secara lokal di setiap langkah dengan harapan bahwa keputusan tersebut menghasilkan solusi optimal secara global. Meski tidak selalu optimal, pendekatan ini sering kali efisien dan cukup baik untuk banyak kasus praktis.

**Contoh Aplikasi:**

- **Dijkstra's Algorithm**: Menemukan jalur terpendek dari satu titik ke titik lainnya dalam graf berbobot non-negatif.
- **Huffman Coding**: Algoritma pengkodean untuk kompresi data lossless.
- **Activity Selection Problem**: Memilih sebanyak mungkin aktivitas yang tidak tumpang tindih.

**Kelebihan:**

- Eksekusi cepat dan efisien.
- Sederhana diimplementasikan.
- Cocok untuk optimasi di bawah batasan waktu.

**Kekurangan:**

Tidak selalu menghasilkan solusi optimal untuk semua jenis masalah (misalnya pada Knapsack Problem).

### Dynamic Programming (DP)

Dynamic Programming menyelesaikan masalah kompleks dengan memecahnya menjadi sub-masalah yang tumpang tindih dan menyimpan hasil perhitungan sebelumnya untuk digunakan kembali (caching atau memoization). Ini menghindari penghitungan ulang yang tidak perlu, meningkatkan efisiensi secara signifikan.

**Teknik Umum:**

- **Top-down (Memoization)**: Menggunakan rekursi + penyimpanan hasil.
- **Bottom-up (Tabulation)**: Mengisi tabel dari kasus terkecil.

**Contoh Masalah:**

- **Fibonacci Sequence**: Menghitung elemen ke-n dalam deret Fibonacci.
- **Knapsack Problem**: Memilih barang dengan nilai maksimum tanpa melebihi kapasitas.
- **Edit Distance / Levenshtein Distance**: Mengukur perbedaan antara dua string.

**Kelebihan:**

- Efisien untuk masalah rekursif dengan sub-masalah yang tumpang tindih.
- Banyak digunakan dalam bioinformatika, NLP, dan optimasi keuangan.

## Algoritma dalam Machine Learning

Machine Learning (ML) menggunakan algoritma yang memungkinkan sistem belajar dari data dan meningkatkan performa tanpa diprogram ulang secara eksplisit. Algoritma ML dapat dikategorikan menjadi:

- Supervised Learning
- Unsupervised Learning
- Reinforcement Learning

**Contoh Algoritma ML:**

1. **Decision Tree**
   - Struktur pohon untuk pengambilan keputusan.
   - Mudah divisualisasikan dan diinterpretasikan.
   - Contoh: CART, ID3, C4.5
2. **K-Nearest Neighbors (KNN)**
   - Algoritma berbasis kedekatan (similarity).
   - Klasifikasi berdasarkan mayoritas tetangga terdekat dalam ruang fitur.
3. **Artificial Neural Networks (ANN)**
   - Terinspirasi dari jaringan saraf biologis.
   - Digunakan dalam deep learning untuk tugas seperti pengenalan wajah, NLP, dan visi komputer.

**Kelebihan Algoritma ML:**

- Kemampuan generalisasi tinggi.
- Mampu menangani data kompleks dan tidak terstruktur.
- Digunakan dalam personalisasi, prediksi, deteksi anomali, dan klasifikasi.

**Kekurangan:**

- Membutuhkan data besar.
- Rentan terhadap overfitting jika tidak ditangani dengan baik.

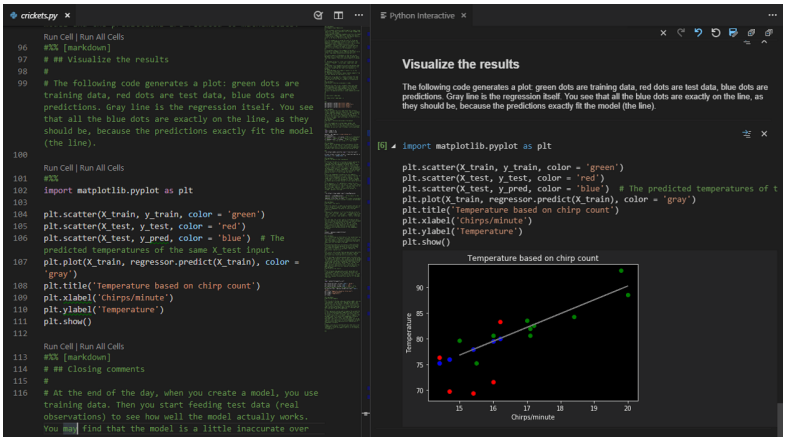
- Interpretabilitas rendah untuk model kompleks seperti deep learning.

## 1.4 Lingkungan Pengembangan Python

Python merupakan salah satu bahasa pemrograman yang sangat populer dan banyak digunakan dalam berbagai bidang, termasuk dalam pembelajaran algoritma dan struktur data. Keunggulan Python terletak pada sintaksisnya yang sederhana dan mudah dipahami, sehingga sangat cocok untuk pemula. Selain itu, Python memiliki ekosistem pustaka yang kaya serta komunitas global yang besar dan aktif, membuatnya menjadi bahasa yang fleksibel dan serbaguna. Lingkungan pengembangan Python terdiri dari berbagai alat bantu seperti *interpreter, editor teks*, dan IDE (*Integrated Development Environment*) yang memudahkan pengguna dalam menulis, menguji, dan *men-debug* program. Selain itu, Python juga mendukung pengelolaan pustaka eksternal melalui manajer paket serta pembuatan lingkungan virtual untuk memisahkan dependensi antar proyek. Semua elemen ini menjadikan Python sebagai lingkungan yang ideal bagi pengembangan perangkat lunak berbasis algoritma dan struktur data.

### 1.4.1 Interpreter, IDE, dan Editor Teks

Dalam pengembangan perangkat lunak menggunakan Python, terdapat berbagai alat bantu yang memudahkan penulisan dan pengujian program. Alat-alat ini mencakup *interpreter*, *editor teks* dan IDE (*Integrated Development Environment*) yang memiliki fitur masing-masing sesuai kebutuhan pengguna. *Interpreter* adalah komponen utama yang digunakan untuk mengeksekusi kode Python secara langsung, baris demi baris. *Interpreter* sangat berguna saat melakukan eksperimen singkat atau pengujian cepat terhadap potongan kode. Python menyediakan interpreter bawaan yang dapat dijalankan melalui terminal atau *command prompt* dengan mengetik python atau python3, tergantung pada sistem operasinya (Sweigart, 2019).




Gambar 1.4: Tampilan Visual Studio Code

IDE (*Integrated Development Environment*) merupakan lingkungan terintegrasi yang menyediakan fitur lengkap seperti penyorotan *sintaks*, *auto-completion*, *debugging* dan integrasi sistem kontrol versi. Beberapa IDE populer untuk Python adalah PyCharm, Visual Studio Code (VSCode), dan Spyder. Penggunaan IDE sangat disarankan bagi pengembang yang bekerja pada proyek besar atau yang memerlukan alat bantu canggih dalam pengembangan. Editor Teks adalah aplikasi yang digunakan untuk menulis kode tanpa fitur tambahan seperti pada IDE. Meskipun ringan dan sederhana, beberapa *editor teks modern* seperti Sublime Text, Atom, dan Notepad++ dapat dikonfigurasi dengan plugin untuk mendukung penulisan kode Python secara lebih produktif. *Editor teks* sangat cocok bagi pengguna yang menginginkan kontrol lebih besar atas lingkungan kerja mereka atau memiliki keterbatasan sumber daya perangkat.

# Bab 2
# Dasar Pemrograman Python

*"Belajarlah dari bahasa yang mudah, karena kemudahan itu adalah jembatan menuju pemahaman mendalam."* — **Guido van Rossum**

## 2.1 Instalasi dan Setup Python

Python adalah bahasa pemrograman tingkat tinggi yang bersifat interpreted dan mendukung berbagai paradigma pemrograman. Dikenal karena sintaksnya yang ringkas dan menyerupai bahasa manusia, Python dirancang untuk meningkatkan produktivitas pengembang serta memudahkan pembelajaran, terutama dalam memahami konsep dasar pemrograman, struktur data, dan algoritma (Zelle, 2004). Bahasa ini pertama kali diperkenalkan oleh Guido van Rossum pada tahun 1991 dan terus berkembang melalui pembaruan yang konsisten hingga kini.

Sebelum mulai menulis kode, langkah awal yang penting adalah melakukan instalasi dan menyiapkan lingkungan pengembangan Python. Proses ini memastikan bahwa sistem telah siap digunakan dan kompatibel dengan pustaka-pustaka yang dibutuhkan. Python mendukung berbagai sistem operasi seperti Windows, macOS, dan Linux, sehingga pengguna memiliki fleksibilitas dalam memilih platform. Dokumentasi resmi yang disediakan oleh (Python Software Foundation, 2024) juga sangat membantu, terutama bagi pengguna pemula yang ingin memulai dengan langkah yang tepat.

Gambar 2.1: Logo Resmi Python

### 2.1.1 Versi Python dan Cara Instalasi

Python merupakan bahasa pemrograman yang terus berkembang, dengan pembaruan versi mayor yang kerap menghadirkan peningkatan signifikan baik dalam sintaks, performa, maupun fitur. Versi Python 3.x, khususnya Python 3.11 ke atas, saat ini menjadi standar utama dalam dunia pendidikan dan industri karena kestabilan, kecepatan eksekusi, serta kelengkapan pustaka standarnya (Python Software Foundation, 2024).

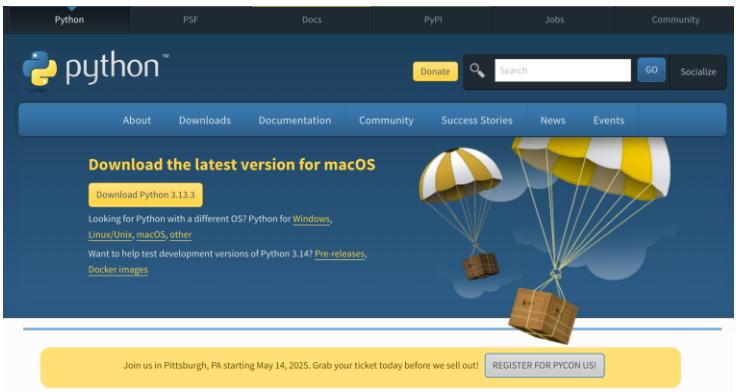
**Langkah Instalasi Python**

1. **Buka situs resmi Python**

   Langkah pertama dalam memasang Python adalah dengan mengunjungi situs resmi yang dikelola oleh Python Software Foundation, yaitu: https://www.python.org/downloads.

2. **Unduh installer untuk Windows atau macOS**

   Pengguna dapat mengunduh installer Python dari situs resmi python.org, python.org secara otomatis mendeteksi sistem operasi yang digunakan (Windows atau macOS) dan menampilkan tombol unduh yang sesuai, misalnya "**Download Python 3.13.3 for Windows**" atau "**Download Python 3.13.3 for macOS**".




Gambar 2.2: Tampilan Halaman Unduh Python

(Sumber: python.org)

Klik tombol unduhan besar berwarna kuning yang muncul di halaman tersebut. Misalnya, di Windows, file instalasi seperti `python-3.13.1.exe`, sementara bagi pengguna macOS, installer berupa file `.pkg` (Zelle, 2004).

3. **Jalankan Installer dan Tambahkan PATH (untuk Windows)**

   Untuk pengguna Windows, klik dua kali file installer `.exe`. Pada jendela awal instalasi, centang kotak **"Add Python to PATH"** lalu klik **"Install Now".** Langkah ini penting agar Python dapat dijalankan dari Command Prompt.

4. **Jalankan Installer (untuk macOS)**

   Untuk pengguna macOS, buka file `.pkg` yang telah diunduh, dan ikuti instruksi instalasi seperti pada aplikasi umumnya. Python akan terinstal di direktori default sistem.

5. **Verifikasi Instalasi**

   Buka Terminal (macOS) atau *Command Prompt* (Windows), lalu jalankan perintah.

### 2.1.2 Virtual Environment (venv, conda)

Dalam pengembangan perangkat lunak modern, pengelolaan versi pustaka dan dependensi merupakan hal yang esensial guna menjaga stabilitas dan reusabilitas kode. Python, sebagai bahasa pemrograman yang fleksibel, menyediakan solusi melalui sistem virtual environment yang memungkinkan setiap proyek berjalan dalam lingkungan yang terisolasi.

Di lingkungan pendidikan, penerapan *virtual environment* sangat dianjurkan karena memberi ruang bagi mahasiswa untuk bereksperimen tanpa risiko merusak konfigurasi sistem utama. Dengan lingkungan yang terisolasi, mahasiswa dapat dengan mudah mengelola paket, mencoba pustaka baru, dan memahami ekosistem Python secara lebih menyeluruh (Severance, 2016).

Terdapat dua pendekatan umum dalam membangun lingkungan virtual Python: menggunakan modul bawaan `venv`, dan menggunakan manajer

lingkungan conda dari distribusi Anaconda. Modul `venv` memungkinkan pembuatan lingkungan kerja lokal yang berisi interpreter Python beserta pustaka-pustaka yang dibutuhkan proyek. Pendekatan ini membantu mencegah konflik versi antar proyek dan tidak memengaruhi konfigurasi global sistem (Python Software Foundation, 2024).

Sementara itu, `conda` menawarkan fitur yang lebih luas, termasuk kemampuan untuk mengelola pustaka *non*-Python seperti R dan Julia, sehingga sangat sesuai digunakan dalam proyek multidisiplin dan analisis data (Grus, 2022).

**1. Lingkungan Virtual dengan `venv`**

Modul `venv` diperkenalkan sejak Python 3.3 sebagai solusi internal untuk menciptakan lingkungan kerja yang terisolasi. Modul ini memungkinkan pengguna untuk membuat direktori lingkungan yang berisi interpreter Python dan pustaka yang terpisah dari instalasi sistem. Hal ini berguna untuk menjaga kestabilan sistem dan menghindari ketergantungan global yang berlebihan (Zelle, 2004).

**Langkah-langkah menggunakan `venv`:**

1. **Buat *environment* baru:**

```
python -m venv nama_env
```

2. **Aktifkan environment:**

   - **Widows:**

```
nama_env\\Scripts\\activate
```

   - **macOS/Linux:**

```
source nama_env/bin/activate
```

3. **Menonaktifkan:**

```
deactivate
```

## 2. Lingkungan Virtual dengan `conda`

Alternatif dari `venv` adalah `conda`, yang disediakan oleh Anaconda dan Miniconda. `conda` tidak hanya dapat mengelola pustaka Python, tetapi juga pustaka *non*-Python seperti R dan Julia. Hal ini menjadikan conda sangat populer dalam komunitas data science dan penelitian ilmiah multidisiplin (Grus, 2022).

**Langkah-langkah dasar menggunakan `conda`:**

1. **Membuat environment:**

```
conda create --name nama_env python=3.12
```

2. **Aktifkan environment:**

```
conda activate nama_env
```

3. **Menonaktifkan:**

```
conda deactivate
```

4. **Menghapus *environment*:**

```
conda remove --name nama_env --all
```

Tabel 2.1: Perbandingan `venv` dan `conda`

| Aspek | `venv` | `conda` |
|---|---|---|
| **Manajer Paket** | `pip` | conda, juga bisa `pip` |
| **Bahasa** | Hanya Python | Python, R, Julia, dll. |
| **Ukuran** | Ringan | Lebih besar (Anaconda) |

| | | |
|---|---|---|
| **Dukungan Paket *Non*-Python** | Tidak tersedia | Tersedia |
| **Kemudahan Setup** | Bawaan Python | Perlu instalasi Anaconda/Miniconda |

## 2.2 Sintaks Dasar Python

Bahasa pemrograman Python dikenal dengan sintaksnya yang bersih, sederhana, dan menyerupai bahasa manusia, sehingga sangat cocok digunakan dalam konteks pembelajaran. Salah satu filosofi utama Python adalah *readability counts*, yang berarti keterbacaan kode menjadi pertimbangan utama dalam desain bahasa ini (Lutz & Ascher, 2009). Sintaks Python yang konsisten dan minimalis memungkinkan mahasiswa pemula memahami konsep-konsep dasar pemrograman tanpa dibebani oleh aturan penulisan yang rumit seperti pada beberapa bahasa lain.

Sintaks Python juga tidak menggunakan kurung kurawal (`{}`) atau titik koma (`;`) seperti pada C/C++ atau Java. Sebagai gantinya, Python menggunakan **indentasi** (spasi atau tab) untuk menandai blok kode. Hal ini mendorong praktik penulisan kode yang lebih rapi dan terstruktur sejak awal. Pemahaman awal terhadap variabel, tipe data, dan operator merupakan dasar untuk memahami konstruksi program lebih kompleks, termasuk struktur data dan algoritma yang akan dibahas pada bab-bab selanjutnya.

### 2.2.1 Variabel dan Tipe Data

Variabel dalam Python berfungsi sebagai penampung data yang dapat berubah-ubah. Python menggunakan sistem *dynamic typing*, di mana jenis data ditentukan secara otomatis saat nilai diberikan pada variabel. Tidak seperti bahasa pemrograman lain yang mengharuskan deklarasi tipe data secara eksplisit, Python cukup dengan memberikan nilai secara langsung (Downey, 2015).

**Contoh:**

```
nama = "Python"
umur = 25
tinggi = 1.72
aktif = True
```

Tabel 2.2: Tipe Data Dasar di Python

| Tipe Data | Contoh Nilai | Keterangan |
|---|---|---|
| `int` | 25 | Bilangan bulat |
| `float` | 1.72 | Bilangan desimal |
| `str` | "Python" | Rangkaian karakter (teks) |
| `bool` | `True, False` | Nilai logika |

Dalam konteks pendidikan, pendekatan ini mengurangi hambatan awal bagi mahasiswa baru dan mendorong mereka untuk fokus pada pemahaman logika program (Zelle, 2004).

### 2.2.2 Operator Aritmatika, Perbandingan, dan Logika

Python menyediakan operator dasar seperti `+, -, *, /` untuk aritmatika, `==, !=, <, >` untuk perbandingan, serta `and`, `or`, `not` untuk logika. Operator ini sangat esensial dalam proses pengambilan keputusan dan manipulasi data (Sweigart, 2020).

#### 2.2.2.1 Operator Aritmatika

Digunakan untuk operasi matematika dasar.

Tabel 2.3: Operator Aritmatika

| Operator | Fungsi | Contoh |
|---|---|---|
| `+` | Penjumlahan | `x + y` |

| Operator | Fungsi | Contoh |
|---|---|---|
| - | Pengurangan | x - y |
| * | Perkalian | x * y |
| / | Pembagian float | x / y |
| // | Pembagian bulat | x // y |
| % | Sisa bagi | x % y |
| ** | Pangkat | x ** y |

**Contoh:**

```
a = 10
b = 3

print(a + b)   # Penjumlahan → 13
print(a - b)   # Pengurangan → 7
print(a * b)   # Perkalian → 30
print(a / b)   # Pembagian float → 3.333...
print(a % b)   # Sisa bagi → 1
```

**Output:**

```
Output: 13
Output: 7
Output: 30
Output: 3.333…
Output: 1
```

**Pada pemanggilan di atas:**

Variabel a dan b menyimpan bilangan bulat. Operator +, –, *, /, dan % digunakan untuk menjumlahkan, mengurangkan, mengalikan, membagikan, dan menghasilkan sisa bagi nilai tersebut.

### 2.2.2.2 Operator Perbandingan

Digunakan untuk membandingkan dua nilai.

Tabel 2.4: Operator Perbandingan

| Operator | Makna | Contoh |
|---|---|---|
| == | Sama dengan | x == y |
| != | Tidak sama dengan | x != y |
| > | Lebih besar | x > y |
| < | Lebih kecil | x < y |
| >= | Lebih besar sama dengan | x >= y |
| <= | Lebih kecil sama dengan | x <= y |

**Contoh:**

```
x = 7
y = 5

print(x == y)   # Sama dengan → False
print(x > y)    # Lebih besar → True
```

**Output:**

```
Output: False
Output: True
```

**Pada pemanggilan di atas:**

Operator perbandingan membandingkan dua nilai dan menghasilkan `True` atau `False`. Contohnya, `x == y` berarti "apakah x sama dengan y?", sedangkan `x > y` memeriksa "apakah x lebih besar dari y".

### 2.2.2.3 Operator Logika

Digunakan dalam ekspresi logika Boolean.

Tabel 2.5: Operator Logika

| Operator | Makna | Contoh |
|---|---|---|
| `and` | Dan | `True and False → False` |
| `or` | Atau | `True or False → True` |
| `not` | Negasi | `not True → False` |

**Contoh:**

```
p = True
q = False

print(p and q)  # Output: False
print(p or q)   # Output: True
print(not p)    # Output: False
```

**Output:**

```
Output: False
Output: True
```

**Pada pemanggilan di atas:**

Operator logika digunakan untuk ekspresi Boolean.

- `and` hanya menghasilkan `True` jika kedua operand bernilai `True`.
- `or` menghasilkan `True` jika salah satu operand bernilai `True`.
- `not` membalik nilai Boolean operand-nya.

## 2.3 Struktur Kontrol

Penguasaan terhadap struktur kontrol merupakan prasyarat untuk dapat merancang algoritma yang efisien dan dapat diandalkan. Struktur

kontrol dalam Python memungkinkan aliran program berjalan secara kondisional atau berulang, serta memberikan fleksibilitas dalam menyusun logika komputasi. Penggunaan struktur seperti `if`, `for`, dan `while` memperkaya ekspresi algoritmik yang akan banyak digunakan dalam pembahasan struktur data lanjutan (Lutz & Ascher, 2009).

### 2.3.1 Percabangan (if, else, elif)

Struktur kontrol percabangan memungkinkan program menjalankan perintah tertentu berdasarkan kondisi. Python menggunakan kata kunci `if`, `elif`, dan `else`. Pendekatan menggunakan percabangan membuat logika program mudah dibaca dan dipelihara (Lutz & Ascher, 2009).

**Contoh:**

```
nilai = 82

if nilai >= 90:
    print("Nilai: A")
elif nilai >= 75:
    print("Nilai: B")
elif nilai >= 60:
    print("Nilai: C")
else:
    print("Nilai: D")
```

**Output:**

```
Output: Nilai: A
```

**Pada pemanggilan di atas:**

Variabel `nilai` memiliki nilai 82. Python akan mengevaluasi kondisi dari atas ke bawah:

- `nilai >= 90` → salah,
- `nilai >= 75` → benar → maka print ("Nilai: B") dijalankan.

Kondisi setelahnya diabaikan karena satu kondisi sudah terpenuhi.

### 2.3.2 Perulangan (for, while, comprehension)

Python mendukung perulangan dengan `for` dan `while`. Untuk manipulasi koleksi data, list comprehension menawarkan sintaks yang ringkas dan efisien. Menurut Severance (2016), penggunaan comprehension meningkatkan efisiensi penulisan kode untuk transformasi data.

**Contoh:**

```
for i in range(1, 6):
    print(i)
```

**Output:**

```
Output:
1
2
3
4
5
```

**Pada pemanggilan di atas:**

Fungsi `range(1, 6)` menghasilkan angka dari 1 hingga 5 (batas atas 6 tidak termasuk). Variabel i akan berisi setiap angka dari range tersebut secara berurutan dan ditampilkan dengan `print()`.

## 2.4 Modul dan Pustaka Bawaan

Dalam pengembangan program yang kompleks, efisiensi kode dan keterbacaan menjadi hal yang sangat penting. Python mendukung konsep modularitas melalui penggunaan modul, yang memungkinkan program dibagi menjadi bagian-bagian kecil yang dapat digunakan kembali. Python juga menyediakan banyak pustaka bawaan (*standard libraries*) yang mencakup berbagai fungsi, seperti operasi matematika, manipulasi waktu, dan pengacakan data. Kemampuan ini menjadikan

Python sangat kuat tanpa perlu menulis semuanya dari awal (Martelli, 2017).

### 2.4.1 Import, Alias, dan Paket Standar

Untuk menggunakan modul, Python menyediakan perintah import. Dengan perintah ini, kita dapat mengakses fungsi atau kelas yang ada di modul lain. Kita juga bisa menggunakan alias untuk menyederhanakan nama modul saat digunakan dalam program.

**Contoh:**

```
import math
print(math.sqrt(16))
```

**Output:**

```
Output: 4.0
```

**Pada pemanggilan di atas:**

Modul `math` diimpor, dan fungsi `sqrt()` digunakan untuk menghitung akar kuadrat dari 16.

**Menggunakan alias:**

```
import math as m
print(m.pow(2, 3))
```

**Output:**

```
Output: 8.0
```

**Pada pemanggilan di atas:**

Modul `math` diimpor dengan nama alias `m`. Fungsi `pow()` menghitung pangkat dua dipangkatkan tiga.

**Menggunakan alias:**

```
from math import pi, sin
print(pi)
```

```
print(sin(0))
```

**Output:**

```
Output:
3.141592653589793
0.0
```

**Pada pemanggilan di atas:**

Kita hanya mengimpor `pi` dan `sin` dari modul `math`, bukan seluruh isi modul. Ini menghemat memori dan memperjelas kode.

### 2.4.2 Contoh Modul: math, random, datetime

Python dilengkapi dengan pustaka standar yang luas dan mencakup beragam kebutuhan pengolahan data, matematika, dan manajemen waktu. Modul-modul seperti `math`, `random`, dan `datetime` termasuk yang paling sering digunakan karena menyediakan fungsi-fungsi penting dalam analisis numerik, pemodelan probabilistik, dan penjadwalan sistem. Ketiga modul ini merupakan bagian dari Python Standard Library yang terdistribusi secara resmi bersama instalasi Python (Python Software Foundation, 2024).

Modul `math` menyediakan fungsi matematis tingkat lanjut yang dirancang untuk perhitungan presisi seperti logaritma, trigonometri, dan fungsi eksponensial. Modul `random` digunakan untuk menghasilkan nilai acak yang penting dalam simulasi, pengacakan data, dan algoritma berbasis probabilitas. Sedangkan modul `datetime` menangani objek waktu dan tanggal, sangat berguna dalam analisis data temporal dan pengolahan waktu sistem.

Penggunaan modul-modul ini tidak memerlukan instalasi tambahan karena termasuk dalam pustaka bawaan Python dan secara luas didokumentasikan serta didukung oleh komunitas dan literatur akademik (Martelli, 2017; Sweigart, 2020) .

Python menyediakan banyak modul bawaan yang siap digunakan. Berikut adalah tiga contoh yang sering dipakai dalam pembelajaran dan pengembangan aplikasi.

### Modul `math`

Digunakan untuk melakukan operasi matematika.

**Contoh:**

```
import math
print(math.factorial(5))
```

**Output:**

```
Output: 120
```

**Pada pemanggilan di atas:**

Fungsi `factorial()` menghitung faktorial dari angka 5.

Tabel 2.6: Fungsi Umum dalam Modul `math`

| Fungsi | Keterangan |
|---|---|
| `sqrt(x)` | Akar kuadrat dari x |
| `pow(x, y)` | x pangkat y |
| `factorial(n)` | Faktorial dari n |
| `sin(x)` | Sinus dari x (radian) |
| `pi` | Konstanta $\pi \approx 3.1416$ |

### Modul `random`

Digunakan untuk menghasilkan angka acak.

**Contoh:**

```
import random
angka = random.randint(1, 10)
print("Angka acak:", angka)
```

**Output:**

```
Output: Angka acak: 7
```

**Pada pemanggilan di atas:**

Fungsi `randint(1,10)` menghasilkan bilangan bulat acak antara 1 hingga 10.

Tabel 2.7: Fungsi Umum dalam Modul `random`

| Fungsi | Keterangan |
|---|---|
| `randint(a, b)` | Bilangan bulat acak antara a dan b |
| `random()` | Bilangan float acak antara 0.0 dan 1.0 |
| `choice(list)` | Memilih satu elemen secara acak dari list |

### Modul datetime

Digunakan untuk manipulasi tanggal dan waktu.

**Contoh:**

```
import datetime
sekarang = datetime.datetime.now()
print("Waktu saat ini:", sekarang)
```

**Output (akan bervariasi):**

```
Output: Waktu saat ini: 2025-05-23 14:30:15.123456
```

**Pada pemanggilan di atas:**

Fungsi `datetime.datetime.now()` mengembalikan waktu sistem saat ini.

Tabel 2.8: Fungsi Umum dalam Modul `datetime`

| Fungsi/Kelas | Keterangan |
|---|---|
| `datetime.now()` | Mengambil waktu dan tanggal saat ini |
| `date.today()` | Mengambil tanggal hari ini |
| `timedelta(days=7)` | Representasi selisih waktu |

### 2.4.3 Manajemen Paket dan Virtual Environment

Dalam pengembangan aplikasi Python, penggunaan pustaka eksternal sangat umum dilakukan untuk mempercepat proses pengembangan dan menambah fungsionalitas. Untuk mengelola pustaka-pustaka tersebut, Python menyediakan manajer paket yang disebut **pip**. Dengan `pip`, pengguna dapat menginstal, memperbarui, dan menghapus pustaka eksternal dari repositori Python Package Index (PyPI) menggunakan perintah sederhana di terminal, seperti `pip install nama-paket`. Selain itu, Python juga menyediakan cara untuk membuat lingkungan pengembangan yang terisolasi melalui **virtual environment**. Dengan menggunakan modul seperti `venv` atau alat pihak ketiga seperti `virtualenv`, pengguna dapat membuat lingkungan khusus untuk setiap proyek sehingga pustaka dan dependensi yang digunakan tidak saling mengganggu antar proyek. Hal ini sangat berguna dalam pengembangan profesional, terutama ketika bekerja dengan proyek yang memiliki versi pustaka yang berbeda-beda. Penggunaan virtual environment membantu menjaga konsistensi lingkungan kerja dan mempermudah proses deployment aplikasi (Sweigart, 2019).

# Bab 3
# Fungsi dan Modularisasi

*Pisahkan kompleksitas menjadi fungsi-fungsi kecil, dan masalah besar akan menjadi sederhana."* — **Martin Fowler**

## 3.1 Dasar-dasar Fungsi

Fungsi (*function*) adalah blok kode tersusun yang dirancang untuk melakukan tugas tertentu dan dapat dipanggil berulang kali di dalam program (Downey, 2012). Dengan memanfaatkan fungsi, pengembang dapat memecah program besar menjadi bagian-bagian modular yang mudah dipahami, diuji, dan dipelihara. Penggunaan fungsi juga meningkatkan keterbacaan kode (*readability*) dan mengurangi pengulangan (*redundancy*), sehingga mendukung prinsip *Don't Repeat Yourself* (DRY).

Di Python, fungsi didefinisikan menggunakan kata kunci `def`, diikuti oleh nama fungsi, daftar parameter dalam tanda kurung, dan diakhiri titik dua, sebelum badan fungsi dituliskan dengan indentasi (Van Rossum & Drake, 2009). Secara konseptual, fungsi terdiri atas tiga bagian utama:

Tabel 3.1: Bagian Utama Fungsi

| Komponen | Deskripsi |
|---|---|
| Header | Mendefinisikan nama dan parameter input fungsi. |
| Docstring *(opsional)* | Menjelaskan tujuan, parameter, dan nilai balik fungsi dalam format string multiline. |
| Badan Fungsi | Blok kode yang melakukan operasi; dapat berisi pernyataan `return` untuk mengembalikan nilai ke pemanggil (Lutz & Ascher, 2009). |

**Contoh struktur dasar fungsi di Python:**

```
def nama_fungsi(param1, param2):
    """Docstring: Deskripsi singkat fungsi."""
    # Blok kode
    hasil = param1 + param2
    return hasil
```

Dengan struktur ini, fungsi menjadi unit independen: kode luar tidak perlu mengetahui detail implementasi di dalamnya, cukup memahami antarmuka (*interface*) berupa nama fungsi dan parameter input/output.

### 3.1.1 Definisi dan Pemanggilan Fungsi

Selain meningkatkan modularitas dan keterbacaan, penggunaan fungsi yang baik membantu pengembang untuk menerapkan prinsip **abstraction**—yaitu menyembunyikan detail implementasi yang kompleks di balik antarmuka yang sederhana. Dengan begitu, tim pengembang dapat bekerja secara paralel: satu anggota fokus memperbaiki logika internal fungsi, sementara yang lain menggunakan fungsi tersebut tanpa perlu memahami implementasi rinci. Pendekatan ini juga memudahkan **unit testing**, karena setiap fungsi dapat diuji secara terpisah untuk memastikan keluaran yang diharapkan sesuai dengan input tertentu (Beizer, 1995).

#### 3.1.1.1 Definisi Fungsi

Definisi fungsi di Python dimulai dengan kata kunci `def`, diikuti nama fungsi yang sesuai dengan konvensi penamaan (huruf kecil, dipisah garis bawah untuk readability), serta parameter (jika ada) di dalam tanda kurung. Setelah titik dua, baris berikutnya diindentasi untuk menandai badan fungsi (Zelle, 2004).

**Contoh:**

```
def hitung_luas_persegi(sisi):
    """
    Menghitung luas persegi.
    :param sisi: panjang sisi persegi (float/int)
```

```
    :return: luas persegi (float)
    """
    return sisi * sisi
```

**Penjelasan:**

- **Nama Fungsi**: `hitung_luas_persegi`
- **Parameter**: `sisi`
- **Docstring**: Mendeskripsikan kegunaan fungsi, tipe parameter, dan tipe nilai kembali.
- **Nilai Kembali**: `return sisi * sisi`

Penggunaan docstring memungkinkan dokumentasi otomatis melalui `help()` atau alat seperti Sphinx, sehingga mempermudah kolaborasi tim dan pemeliharaan kode (Lutz & Ascher, 2009).

### 3.1.1.2 Pemanggilan Fungsi

Untuk menggunakan fungsi, cukup menulis nama fungsi diikuti argumen sesuai urutan parameter:

```
luas = hitung_luas_persegi(5)
print(f"Luas persegi: {luas}")
```

**Output:**

```
Output: Luas persegi: 25
```

**Penjelasan:**

- Argumen `5` dipetakan ke parameter `sisi`.
- Nilai kembali dari fungsi (25) disimpan di variabel `luas`.

Fungsi juga dapat dipanggil di dalam ekspresi lain, misalnya:

```
total = hitung_luas_persegi(3) + hitung_luas_persegi(4)
```

**Output:**

```
Output: Luas persegi: 25
```

Dengan cara ini, fungsi berperan sebagai *black box* yang menyembunyikan detail implementasi, sehingga memudahkan pengujian unit (*unit testing*) dan penggunaan kembali (*reusability*) (Downey, 2012).

### 3.1.2 Parameter Positional vs Keyword

Selain itu, penggunaan parameter keyword meningkatkan **self-documenting code**, di mana setiap argumen yang diberikan mencerminkan makna parameter itu sendiri. Hal ini sangat berguna ketika fungsi memiliki banyak parameter boolean atau numeric yang tidak intuitif, misalnya `connect(host, port, use_ssl=False)` lebih jelas bila dipanggil sebagai `connect("db.example.com", use_ssl=True)` daripada hanya `connect("db.example.com", 3306, True)`. Dengan praktik ini, risiko kesalahan pemanggilan berkurang dan memudahkan pengecekan kode otomatis dalam proses **code review** (Fowler, 2010).

#### 3.1.2.1 Parameter Positional

Parameter *positional* adalah parameter yang pengirimannya berdasarkan urutan saat pemanggilan fungsi. Pengguna wajib memberikan argumen sesuai urutan definisi:

```
def cetak_biodata(nama, usia, kota):
    print(f"Nama: {nama}, Usia: {usia}, Kota: {kota}")

cetak_biodata("Andi", 30, "Jakarta")
```

Di atas, `"Andi"` → `nama`, `30` → `usia`, `"Jakarta"` → `kota`. Jika urutan argumen keliru, akan terjadi kesalahan logika:

```
cetak_biodata(25, "Budi", "Bandung")
# Output salah: Nama: 25, Usia: Budi, Kota: Bandung
```

### 3.1.2.2 Parameter Keyword

Parameter *keyword* memungkinkan pemanggilan fungsi dengan menyebut nama parameter secara eksplisit, terlepas dari urutan:

```
cetak_biodata(usia=25, kota="Bandung", nama="Budi")
```

Hasilnya sama dengan pemanggilan urut, namun lebih eksplisit dan mudah dibaca, terutama jika fungsi memiliki banyak parameter atau beberapa parameter bersifat opsional (Sweigart, 2015).

### 3.1.2.3 Kombinasi Positional dan Keyword

Python mengizinkan kombinasi ringkas antara dua gaya ini: argumen posisional ditulis terlebih dahulu, diikuti argumen keyword. Contoh:

```
def connect(host, port=3306, use_ssl=False):
    # Kode koneksi
    pass

connect("db.example.com", use_ssl=True)
```

Dalam pemanggilan di atas, `"db.example.com"` dipetakan ke `host` (positional), `use_ssl=True` di-set eksplisit, sedangkan `port` menggunakan nilai default `3306` (Van Rossum & Drake, 2009).

### 3.1.2.4 Best Practices

Sebagai pedoman umum, susun parameter fungsi dengan nama yang deskriptif dan urutan yang logis agar pengguna dapat memahami maksud setiap argumen tanpa harus melihat dokumentasi eksternal. Praktik ini secara langsung meningkatkan *readability*, meminimalisir kesalahan pemanggilan, dan memperlancar proses *code review*.


Gambar 3.1: Best Practices Fungsi dan Modularisasi

1. **Berikan Nilai Default**

   Untuk parameter yang sering memiliki nilai umum, tentukan nilai default agar pemanggilan fungsi lebih ringkas (e.g., `port=3306`).

2. **Gunakan Keyword untuk Parameter Opsional**

   Jika suatu parameter sifatnya opsional, pengguna wajib menyebutkan nama parameter agar kode lebih jelas.

3. **Urutan Parameter**

   Daftar parameter idealnya mengikuti urutan:

   - **Positional-only** (jika diperlukan, menggunakan sintaks `/`)
   - **Positional or keyword**
   - **Keyword-only** (setelah `*`)

4. **Variabel panjang (*args, **kwargs)**

   Penataan ini meningkatkan konsistensi antarmuka fungsi (Lutz & Ascher, 2009).

5. **Hindari Banyak Parameter**

   Jika fungsi memerlukan lebih dari lima parameter, pertimbangkan untuk memecahnya menjadi beberapa fungsi atau mengelompokkan ke dalam objek/struktur data (Downey, 2012).

Dengan memahami perbedaan dan penggunaan *positional* serta *keyword* parameter, pengembang dapat merancang antarmuka fungsi yang intuitif, fleksibel, dan mudah dipelihara.

## 3.2 Fungsi Lanjutan

Pada bagian ini, kita akan mempelajari fitur-fitur lanjutan dalam fungsi Python yang memungkinkan kode menjadi **lebih fleksibel, dinamis, dan ekspresif**. Topik yang dibahas meliputi penggunaan **parameter default**, **parameter panjang variabel (`*args`, `**kwargs`)**, serta fungsi anonim **(lambda)** dan fungsional **(map, filter, reduce)**.

## 3.2.1 Argumen Default dan Variable-length

### 3.2.1.1 Argumen Default

Argumen default memungkinkan kita mendefinisikan nilai awal untuk parameter. Jika saat pemanggilan fungsi parameter tersebut tidak diberi nilai, maka nilai default akan digunakan.

**Contoh:**

```
def salam(nama="Pengunjung"):
    print(f"Halo, {nama}!")
```

**Pemanggilan Fungsi:**

```
salam()
salam("Andi")
```

**Penjelasan:**

- `salam()` menggunakan nilai default `"Pengunjung"`.
- `salam("Andi")` menimpa nilai default dengan `"Andi"`.

**Output:**

```
Halo, Pengunjung!
Halo, Andi!
```

### 3.2.1.2 Parameter Panjang Variabel: `*args`

Digunakan untuk menerima **jumlah argumen tak terbatas secara positional**. Argumen disimpan sebagai tuple.

**Contoh:**

```
def jumlahkan(*angka):
    total = sum(angka)
    print(f"Jumlah total: {total}")
```

**Pemanggilan Fungsi:**

```
jumlahkan(1, 2, 3)
jumlahkan(5, 10, 15, 20)
```

**Output:**

```
Jumlah total: 6
Jumlah total: 50
```

#### 3.2.1.3 Parameter Panjang Variabel: `**kwargs`

Digunakan untuk menerima argumen **keyword yang tidak diketahui jumlahnya**. Argumen disimpan dalam bentuk dictionary (`dict`).

**Contoh:**

```
def cetak_info(**data):
    for k, v in data.items():
        print(f"{k}: {v}")
```

**Pemanggilan Fungsi:**

```
cetak_info(nama="Dina", usia=25, kota="Surabaya")
```

**Output:**

```
nama: Dina
usia: 25
kota: Surabaya
```

**Kelebihan:**

- Fleksibel terhadap struktur data.
- Cocok digunakan untuk fungsi-fungsi konfigurasi atau API.

### 3.2.2 Lambda, Map, Filter, dan Reduce

Python mendukung paradigma **pemrograman fungsional**, di mana fungsi dapat diperlakukan sebagai objek — artinya dapat disimpan

dalam variabel, dikirim sebagai argumen, dan dikembalikan dari fungsi lain.

### 3.2.2.1 Fungsi Lambda

**Lambda** adalah fungsi anonim satu baris, digunakan untuk ekspresi sederhana.

**Sintaks:**

```
lambda arg1, arg2: ekspresi
```

**Contoh:**

```
kuadrat = lambda x: x ** 2
print(kuadrat(5))
```

**Output:**

```
25
```

**Catatan:**

- Lambda cocok untuk operasi singkat dan inline.
- Tidak disarankan untuk logika kompleks — gunakan `def`.

### 3.2.2.2 Fungsi `map()`

Digunakan untuk **menerapkan fungsi ke setiap elemen iterable**.

**Contoh:**

```
angka = [1, 2, 3, 4]
hasil = list(map(lambda x: x * 2, angka))
print(hasil)
```

**Output:**

```
[2, 4, 6, 8]
```

**Penjelasan:**

- Setiap elemen angka dikalikan 2.
- Hasilnya adalah `list` baru dengan hasil transformasi.

### 3.2.2.3 Fungsi `filter()`

Digunakan untuk **menyaring elemen berdasarkan kondisi boolean**.

**Contoh:**

```
angka = [1, 2, 3, 4, 5]
ganjil = list(filter(lambda x: x % 2 != 0, angka))
print(ganjil)
```

**Output:**

```
[1, 3, 5]
```

**Penjelasan:**

`filter()` hanya menyertakan elemen yang bernilai `True` saat diuji fungsi lambda.

### 3.2.2.4 Fungsi `reduce()`

`reduce()` digunakan untuk **mengakumulasi elemen menjadi satu nilai** berdasarkan operasi berulang. Fungsi ini berada di modul `functools`.

**Contoh:**

```
from functools import reduce

hasil = reduce(lambda x, y: x * y, [1, 2, 3, 4])
print(hasil)
```

**Output:**

```
24
```

**Penjelasan:**

- Proses: `(((1×2)×3)×4) = 24`
- Cocok untuk perkalian, penjumlahan, penggabungan string, dll.

Tabel 3.2: Ringkasan Fungsi Fungsional

| Fungsi | Tujuan | Keluaran |
|---|---|---|
| `lambda` | Membuat fungsi anonim | Fungsi satu ekspresi |
| `map()` | Menerapkan fungsi ke setiap elemen iterable | List/Map Object |
| `filter()` | Menyaring elemen iterable berdasarkan kondisi | List/Filter Object |
| `reduce()` | Menggabungkan elemen menjadi satu nilai | Nilai Tunggal |

Dengan memahami konsep ini, pengembang dapat menulis kode Python yang **modular, deklaratif, dan efisien** serta lebih mudah diuji dan digunakan kembali di berbagai konteks pemrograman.

## 3.3 Modularisasi Kode

Dalam pengembangan perangkat lunak berskala besar, menjaga struktur kode yang rapi dan mudah dipelihara merupakan kebutuhan mendasar. Salah satu pendekatan yang efektif untuk mencapai tujuan tersebut adalah melalui modularisasi kode**,** yaitu teknik membagi program menjadi bagian-bagian kecil yang disebut *modul*. Setiap modul bertanggung jawab atas satu aspek spesifik dari sistem, sehingga kode menjadi lebih terorganisir, dapat digunakan kembali (*reusable*), dan lebih mudah diuji serta dikembangkan secara kolaboratif.

Python mendukung modularisasi secara eksplisit melalui penggunaan modul dan paket. Modul memungkinkan kita memisahkan fungsi dan kelas ke dalam file terpisah, sementara paket menyediakan cara untuk mengelompokkan beberapa modul ke dalam struktur yang lebih besar

dan logis. Subbab ini membahas dua aspek utama modularisasi dalam Python: pembuatan dan penggunaan modul serta pengemasan ke dalam paket.

### 3.3.1 Membuat dan Menggunakan Modul

Modul adalah berkas Python berekstensi `.py` yang berisi kumpulan fungsi, kelas, atau variabel yang saling terkait. Modul merupakan unit dasar dari modularisasi kode dalam Python dan sangat berguna untuk memisahkan logika program ke dalam bagian-bagian yang mudah dikelola. Dengan membuat modul, kita dapat mendefinisikan fungsi sekali dan menggunakannya di banyak tempat tanpa menyalin ulang kode.

Sebagai contoh, berikut adalah modul sederhana bernama `matematika.py`:

```
# File: matematika.py

def tambah(a, b):
    return a + b

def kali(a, b):
    return a * b
```

Modul ini dapat digunakan di file lain menggunakan perintah `import`. Misalnya, pada file `main.py`:

```
# File: main.py

import matematika

hasil1 = matematika.tambah(4, 6)
hasil2 = matematika.kali(3, 5)

print(f"Hasil tambah: {hasil1}")
print(f"Hasil kali: {hasil2}")
```

**Output:**

```
Hasil tambah: 10
Hasil kali: 15
```

Python juga memungkinkan impor fungsi tertentu dari sebuah modul menggunakan sintaks `from ... import ...`:

```
from matematika import tambah

print(tambah(2, 7))
```

Dengan memanfaatkan modul, kode menjadi lebih ringkas, dapat digunakan kembali, dan memudahkan proses pengujian unit serta dokumentasi.

### 3.3.2 Pengemasan dalam Paket (Package)

Ketika jumlah modul dalam sebuah proyek bertambah, pengelompokan modul-modul tersebut menjadi suatu struktur direktori yang lebih terorganisir menjadi penting. Untuk itu, Python menyediakan konsep **paket**. Paket adalah sebuah direktori yang berisi satu atau lebih modul serta berkas `__init__.py` yang menunjukkan bahwa direktori tersebut merupakan bagian dari namespace Python.

Paket digunakan untuk membagi proyek menjadi bagian-bagian logis dan fungsional. Misalnya, kita dapat memiliki paket `kalkulasi` yang berisi beberapa modul seperti `tambah.py, kali.py`, dan lain-lain.

**Struktur Direktori Paket:**

```
proyek/
│
├── kalkulasi/
│   ├── __init__.py
│   ├── tambah.py
│   └── kali.py
│
└── main.py
```

**Isi `kalkulasi/tambah.py`:**

```
def tambah(a, b):
    return a + b
```

**Isi `kalkulasi/kali.py`:**

```
def kali(a, b):
    return a * b
```

**Isi `kalkulasi/__init__.py`:**

```
from .tambah import tambah
from .kali import kali
```

**File `main.py`:**

```
from kalkulasi import tambah, kali

print(tambah(5, 3))
print(kali(2, 4))
```

**Output:**

```
8
8
```

Dengan struktur paket yang baik, kita tidak hanya mempermudah pengelolaan kode, tetapi juga membuka peluang untuk mendistribusikan dan mengelola pustaka Python secara profesional melalui sistem manajemen paket seperti `pip`.

### Rangkuman Praktik Terbaik Modularisasi

Beberapa prinsip penting yang perlu diperhatikan dalam modularisasi kode Python:

- **Single Responsibility**: Setiap modul atau fungsi harus memiliki satu tanggung jawab utama.

- **Penamaan Deskriptif**: Gunakan nama file dan fungsi yang jelas dan bermakna.
- **Gunakan `__init__.py`**: Untuk mendeklarasikan direktori sebagai bagian dari paket Python.
- **Pemisahan Logis**: Kelompokkan modul berdasarkan domain atau fungsi program.
- **Gunakan Modul Standar**: Hindari penulisan ulang fungsi yang telah tersedia dalam modul standar Python.

Dengan memahami konsep modularisasi serta penerapannya melalui modul dan paket, pengembang dapat membangun sistem yang lebih terstruktur, efisien, dan profesional.

## 3.4 Penanganan Eksepsi

Dalam pengembangan perangkat lunak, kesalahan (*error*) adalah hal yang tidak dapat dihindari, baik yang disebabkan oleh kesalahan logika, data masukan yang tidak valid, maupun gangguan sistem eksternal. Untuk mengatasi hal tersebut secara elegan dan terkontrol, Python menyediakan mekanisme penanganan eksepsi (exception handling). Dengan eksepsi, alur program tidak langsung terhenti ketika terjadi kesalahan, melainkan dapat ditangani secara eksplisit untuk menghindari crash dan memberikan informasi yang jelas kepada pengguna atau sistem.

Penanganan eksepsi merupakan bagian penting dalam penerapan prinsip robustness dan fault-tolerance dalam perangkat lunak. Pada subbab ini, akan dibahas struktur dasar penanganan eksepsi menggunakan `try`, `except`, dan `finally`, serta cara mendefinisikan eksepsi kustom sesuai kebutuhan aplikasi.

### 3.4.1 Try, Except, Finally

Struktur dasar penanganan eksepsi di Python menggunakan blok `try-except`. Blok `try` berisi kode yang berpotensi menghasilkan kesalahan, sementara blok `except` menangani jenis eksepsi tertentu yang mungkin

terjadi. Python juga menyediakan blok `finally`, yang digunakan untuk mengeksekusi kode apa pun yang terjadi**,** baik terjadi eksepsi maupun tidak.

**Struktur Umum:**

```
try:
    # Blok kode yang dicoba dijalankan
except JenisEksepsi:
    # Blok penanganan eksepsi
finally:
    # Blok yang selalu dieksekusi
```

**Contoh Penerapan:**

```
try:
    angka = int(input("Masukkan angka bulat: "))
    hasil = 10 / angka
    print(f"Hasil pembagian: {hasil}")
except ZeroDivisionError:
    print("Kesalahan: Tidak dapat membagi dengan nol.")
except ValueError:
    print("Kesalahan: Masukan bukan angka bulat.")
finally:
    print("Proses selesai.")
```

**Output 1 (input: `2`):**

```
Masukkan angka bulat: 2
Hasil pembagian: 5.0
Proses selesai.
```

**Output 2 (input: `0`):**

```
Masukkan angka bulat: 0
Kesalahan: Tidak dapat membagi dengan nol.
Proses selesai.
```

**Output 3 (input: `abc`):**

```
Masukkan angka bulat: abc
Kesalahan: Masukan bukan angka bulat.
```

```
Proses selesai.
```

**Penjelasan**

- **`try`**: Menampung kode utama yang berpotensi gagal saat dijalankan.
- **`except`**: Menangkap dan menangani jenis eksepsi tertentu seperti `ZeroDivisionError` dan `ValueError`.
- **`finally`**: Selalu dijalankan, cocok untuk pembersihan sumber daya, seperti menutup file atau koneksi database.

Blok `except` juga dapat menggunakan variabel untuk menangkap objek eksepsi:

```
except ZeroDivisionError as e:
    print(f"Kesalahan spesifik: {e}")
```

### 3.4.2 Membuat Eksepsi Kustom

Dalam beberapa kasus, kita perlu mendefinisikan **eksepsi buatan sendiri (custom exception)** untuk menangani kondisi khusus yang tidak tercakup oleh eksepsi bawaan Python. Eksepsi kustom dibuat dengan cara mendefinisikan kelas baru yang diturunkan dari kelas `Exception`.

**Struktur Umum:**

```
class NamaEksepsi(Exception):
    pass
```

**Contoh Kasus: Validasi Umur**

```
class UmurNegatifError(Exception):
    """Eksepsi untuk umur yang bernilai negatif."""
    def __init__(self, umur):
        super().__init__(f"Umur tidak boleh negatif: {umur}")
```

**Penggunaan dalam program:**

```
def cek_umur(umur):
    if umur < 0:
        raise UmurNegatifError(umur)
    print(f"Umur Anda: {umur}")

try:
    cek_umur(-5)
except UmurNegatifError as e:
    print(f"Terjadi kesalahan: {e}")
```

**Output:**

```
Terjadi kesalahan: Umur tidak boleh negatif: -5
```

**Penjelasan:**

- Kelas `UmurNegatifError` mewarisi dari `Exception`.
- Metode `__init__` digunakan untuk menyisipkan pesan eksepsi secara kustom.
- `raise` digunakan untuk melempar eksepsi secara eksplisit.

Eksepsi kustom ini sangat berguna untuk memisahkan *logic error handling* yang bersifat domain-spesifik dari eksepsi umum Python, seperti `TypeError` atau `IndexError`.

### Praktik Terbaik Penanganan Eksepsi

- **Tangkap Eksepsi Spesifik**

  Hindari `except:` tanpa jenis eksepsi karena dapat menyembunyikan bug yang tidak terduga.

- **Gunakan `finally` untuk Clean-up**

  Misalnya, menutup file atau koneksi jaringan, meskipun terjadi kesalahan.

- **Buat Eksepsi Kustom untuk Validasi Domain**

  Hal ini membantu membuat kode lebih mudah dipahami dan terorganisir.

- **Jangan Gunakan Eksepsi sebagai Alur Logika**

  Eksepsi sebaiknya digunakan untuk penanganan situasi luar biasa, bukan sebagai kontrol alur normal.

# Bab 4
# Kompleksitas dan Efisiensi Algoritma

*"Bukan hanya tentang menyelesaikan masalah, tapi seberapa efisien kita melakukannya."* — **Donald Knuth**

## 4.1 Analisis Kompleksitas Waktu

Kompleksitas algoritma merupakan aspek fundamental dalam ilmu komputer yang digunakan untuk mengevaluasi performa algoritma secara teoritis. Evaluasi ini mencakup dua dimensi utama, yaitu kompleksitas waktu **dan** kompleksitas ruang, yang masing-masing berperan dalam mengukur seberapa cepat dan hemat memori sebuah algoritma dijalankan. Pemahaman terhadap kompleksitas sangat penting dalam perancangan algoritma efisien dan optimal untuk skala sistem yang besar dan dinamis (Cormen et al., 2009).

Kompleksitas waktu mengukur jumlah langkah atau operasi elementer yang dilakukan algoritma relatif terhadap ukuran input (`n`). Analisis ini tidak bergantung pada implementasi spesifik atau mesin yang digunakan, melainkan fokus pada estimasi *growth rate* performa saat ukuran input membesar. Tujuannya adalah untuk membandingkan efisiensi algoritma, khususnya dalam skenario terburuk, rata-rata, dan terbaik (Weiss, 2012).

Kompleksitas waktu sering kali dinyatakan menggunakan notasi matematis seperti Big O**,** Theta**,** dan Omega, yang masing-masing mewakili batas atas, eksak, dan batas bawah dari waktu eksekusi algoritma. Evaluasi seperti ini penting dalam pengambilan keputusan desain sistem, terutama pada aplikasi real-time, basis data besar, atau perangkat dengan sumber daya terbatas (Levitin, 2011).

### 4.1.1 Notasi Big O

Notasi **Big O** (`O`) menggambarkan **batas atas asimtotik** dari waktu eksekusi algoritma, khususnya dalam skenario *worst-case*. Big O menunjukkan seberapa cepat waktu eksekusi meningkat seiring dengan pertambahan ukuran input. Misalnya, jika sebuah algoritma memiliki kompleksitas `O(n²)`, maka waktu eksekusinya akan meningkat kuadratik terhadap ukuran input `n`.

Beberapa contoh umum kompleksitas Big O meliputi:

- **`O(1)`** – **Konstan**: tidak bergantung pada input (misalnya akses array langsung).
- **`O(log n)`** – **Logaritmik**: seperti pada binary search.
- **`O(n)`** – **Linear**: misalnya pada linear search.
- **`O(n log n)`** – **Linear-logaritmik**: seperti merge sort dan quicksort.
- **`O(n²)`** – **Kuadratik**: misalnya pada bubble sort atau insertion sort.
- **`O(2ⁿ)`** – **Eksponensial**: pada brute-force solusi masalah kombinatorial.

Big O sangat berguna dalam menentukan efisiensi algoritma pada skala besar dan menjadi acuan utama dalam literatur algoritma modern (Cormen et al., 2009; Goodrich & Tamassia, 2014).

### 4.1.2 Notasi Θ (Theta) dan Ω (Omega)

Selain Big O, terdapat dua notasi penting lainnya yang memberikan gambaran yang lebih lengkap tentang performa algoritma, yaitu **Theta (Θ)** dan **Omega (Ω)**.

- **Notasi Theta (Θ)** merepresentasikan **batas eksak** dari kompleksitas waktu. Ini berarti bahwa waktu eksekusi akan tumbuh secara simetris terhadap fungsi tertentu baik dari bawah maupun atas. Misalnya, algoritma dengan `Θ(n log n)` memiliki performa waktu yang konsisten di semua skenario input.

- **Notasi Omega (Ω)** menunjukkan **batas bawah** dari waktu eksekusi, biasanya digunakan untuk kasus terbaik (*best-case scenario*). Sebagai contoh, pencarian linear memiliki $\Omega(1)$ jika elemen yang dicari berada di awal daftar.

Kombinasi ketiga notasi ini ($O$, $\Theta$, $\Omega$) memberikan kerangka komprehensif dalam mengevaluasi dan membandingkan berbagai algoritma secara matematis dan praktis (Levitin, 2011; Shaffer, 2013).

## 4.2 Analisis Kompleksitas Waktu

Selain kompleksitas waktu, aspek penting lain dalam evaluasi algoritma adalah kompleksitas ruang (space complexity). Kompleksitas ruang menggambarkan jumlah memori tambahan yang diperlukan oleh algoritma untuk menjalankan prosesnya, relatif terhadap ukuran input. Tidak hanya mencakup variabel input, tetapi juga mencakup penggunaan variabel lokal, struktur data tambahan, dan tumpukan pemanggilan fungsi rekursif (Cormen et al., 2009).

Pemahaman kompleksitas ruang sangat penting terutama pada sistem dengan keterbatasan memori seperti perangkat embedded, mobile, dan sistem real-time. Dalam banyak kasus, terdapat trade-off antara waktu dan ruang, yaitu peningkatan efisiensi waktu sering kali membutuhkan konsumsi memori yang lebih besar, dan sebaliknya (Levitin, 2011).

### 4.2.1 Menghitung Penggunaan Memori

Untuk menghitung kompleksitas ruang, kita meninjau total ruang memori yang digunakan selama eksekusi program, termasuk:

- **Ruang untuk input**: biasanya sebesar $O(n)$ jika data diakses tanpa duplikasi.
- **Ruang untuk variabel sementara**: misalnya, variabel indeks, penampung hasil sementara, dll.
- **Ruang untuk struktur data tambahan**: seperti stack, queue, atau array tambahan.

- **Ruang untuk pemanggilan rekursif**: setiap pemanggilan fungsi membutuhkan ruang stack tambahan.

Contoh, algoritma rekursif seperti *Fibonacci* versi rekursif membutuhkan `O(n)` ruang stack karena setiap pemanggilan menyimpan konteks fungsi yang belum selesai. Sebaliknya, versi iteratif hanya membutuhkan `O(1)` ruang untuk variabel loop (Goodrich & Tamassia, 2014).

Dalam konteks sorting, algoritma seperti merge sort menggunakan `O(n)` ruang tambahan untuk menyimpan hasil sementara, sedangkan quicksort dapat dilakukan secara in-place dengan ruang tambahan `O(log n)` untuk stack pemanggilan rekursif (Shaffer, 2013).

### 4.2.2 Trade-off Waktu vs Ruang

Banyak algoritma menghadirkan dilema antara efisiensi waktu dan efisiensi ruang. Dalam beberapa kasus, menggunakan lebih banyak ruang dapat

**Contohnya adalah:**

- Memoisasi pada pemrograman dinamis, yang menyimpan hasil perhitungan dalam tabel agar tidak dihitung ulang, menghasilkan efisiensi waktu yang jauh lebih baik, tetapi dengan biaya memori tambahan (`O(n)`).
- Penggunaan struktur data seperti *hash table* memungkinkan pencarian data dalam `O(1)` waktu rata-rata, tetapi membutuhkan ruang tambahan untuk menyimpan pasangan kunci-nilai (`O(n)`), dibandingkan dengan pencarian linear yang hanya membutuhkan `O(1)` ruang tetapi `O(n)` waktu.

Penting bagi pengembang untuk mengevaluasi kebutuhan sistem dan batasan sumber daya sebelum memilih pendekatan algoritmik. Dalam aplikasi dunia nyata, keputusan untuk menggunakan ruang atau waktu lebih banyak sering bergantung pada konteks aplikasi dan arsitektur perangkat lunak (Weiss, 2012).

## 4.3 Pengukuran dan Profiling

Dalam praktik pemrograman, pengukuran performa algoritma tidak cukup hanya dengan analisis kompleksitas waktu atau ruang secara teoretis. Diperlukan juga pengukuran empiris, yaitu observasi waktu eksekusi nyata dan konsumsi sumber daya saat program berjalan. Proses ini dikenal sebagai profiling. Profiling membantu pengembang untuk:

- Mengetahui bagian program yang paling memakan waktu,
- Mengidentifikasi fungsi atau modul yang boros memori,
- Menentukan bagian mana yang perlu dioptimalkan (Beazley & Jones, 2013).

Bahasa Python menyediakan modul-modul standar seperti `timeit` dan `cProfile` untuk mengukur dan menganalisis performa kode program.

### 4.3.1 Menggunakan `timeit`

Modul `timeit` berguna untuk melakukan *micro-benchmarking*, yaitu mengukur waktu eksekusi dari potongan kode kecil dengan akurasi tinggi. `timeit` menjalankan kode berulang kali dan menghitung rata-rata waktu yang dibutuhkan, sehingga hasilnya lebih stabil dibanding hanya memakai `time()` biasa.

**Contoh:**

```
import timeit

kode = '''
def hitung_faktorial(n):
    return 1 if n == 0 else n * hitung_faktorial(n - 1)
hitung_faktorial(10)
'''

print(timeit.timeit(stmt=kode, number=1000))
```

Kode di atas menjalankan fungsi `hitung_faktorial` sebanyak 1000 kali, lalu mencetak waktu total yang dibutuhkan. Modul ini sangat

bermanfaat untuk membandingkan dua algoritma berbeda berdasarkan performa waktu mereka dalam eksekusi berulang.

**Tips penggunaan**

- Gunakan `number` dan `repeat` untuk hasil yang konsisten.
- Hindari pengaruh `print()` atau I/O saat benchmarking.
- Menurut Hetland (2017), `timeit` adalah alat utama dalam pengujian efisiensi potongan kode Python karena presisinya dalam mengukur waktu hingga milidetik.

### 4.3.2 Profiling dengan `cProfile`

Untuk analisis performa yang lebih menyeluruh, Python menyediakan modul `cProfile`. Modul ini mencatat statistik runtime dari setiap fungsi dalam program, seperti:

- Jumlah pemanggilan fungsi (calls),
- Waktu total yang dihabiskan dalam fungsi (tottime),
- Waktu total inklusif fungsi pemanggil (cumtime),
- Rata-rata waktu per pemanggilan.

**Contoh:**

```
import cProfile

def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

cProfile.run("fibonacci(10)")
```

**Contoh Output `cProfile`:**

```
         177 function calls (11 primitive calls) in 0.000 seconds

   Ordered by: standard name
```

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       89    0.000    0.000    0.000    0.000 <stdin>:2(fibonacci)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method builtins.print}
       85    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
```

Tabel 4.1: Penjelasan Kolom

| Kolom | Makna |
|---|---|
| **ncalls** | Jumlah pemanggilan fungsi (rekursif dihitung berulang kali) |
| **tottime** | Waktu total yang dihabiskan di fungsi itu sendiri |
| **percall** | `tottime / ncalls` |
| **cumtime** | Total waktu kumulatif termasuk fungsi lain yang dipanggil di dalamnya |
| **percall** | `cumtime / ncalls` |
| **filename:lineno(function)** | Lokasi dan nama fungsi |

### Penjelasan:

- Fungsi `fibonacci` dipanggil **89 kali** (karena rekursif),
- Total waktu (sangat kecil) karena `n=10` masih ringan,
- Fungsi lain seperti `print` atau `isinstance` juga ikut terekam jika digunakan secara tidak langsung.

Jika Anda mencoba dengan `fibonacci(30)`, jumlah pemanggilan dan waktu akan jauh lebih besar dan relevan untuk melihat bottleneck secara signifikan.

### Kelebihan cProfile:

- Cocok untuk aplikasi kompleks dan berlapis,
- Dapat digabung dengan visualisasi seperti SnakeViz atau gProfiler untuk grafik interaktif.

- Menurut Beazley & Jones (2013), penggunaan `cProfile` sangat penting dalam proses refactoring dan optimasi performa, terutama untuk mendeteksi *bottleneck* dalam proyek skala menengah hingga besar.

## 4.4 Optimasi Dasar

Optimasi algoritma adalah langkah penting untuk meningkatkan efisiensi eksekusi dan penggunaan sumber daya. Tanpa optimasi, algoritma dapat berjalan lambat atau memakan memori secara berlebihan, terutama ketika berhadapan dengan skala data besar. Dua pendekatan dasar yang umum digunakan adalah teknik memoisasi dan pemilihan struktur data yang tepat.

### 4.4.1 Teknik Memoisasi

Memoisasi adalah teknik untuk menyimpan hasil perhitungan fungsi agar tidak perlu dihitung ulang ketika diperlukan kembali. Teknik ini sangat berguna dalam pemrograman rekursif yang mengandung submasalah berulang, seperti pada kasus perhitungan deret Fibonacci, penghitungan kombinasi, atau dalam algoritma dynamic programming.

**Contoh penerapan memoization pada fungsi Fibonacci:**

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(50))
```

Dengan pendekatan ini, fungsi tidak akan menghitung kembali nilai `fibonacci(n)` yang sudah pernah diproses, sehingga efisiensinya meningkat signifikan. Tanpa memoisasi, kompleksitas waktu fungsi

Fibonacci bisa mencapai $O(2^n)$, namun dengan memoisasi dapat ditekan menjadi $O(n)$.

### 4.4.2 Pemilihan Struktur Data yang Tepat

Struktur data memainkan peran penting dalam kinerja algoritma. Pemilihan struktur data yang sesuai dapat mempercepat proses pencarian, penyisipan, dan penghapusan elemen. Sebaliknya, kesalahan dalam memilih struktur data dapat menyebabkan program menjadi lambat dan tidak efisien.

Tabel berikut menggambarkan struktur data yang umum dan kegunaannya:

Tabel 4.2: Struktur Data yang Umum dan Kegunaannya

| Tujuan Penggunaan | Struktur Data yang Direkomendasikan |
|---|---|
| Akses elemen berdasarkan indeks | List / Array |
| Pencarian berdasarkan kunci | Dictionary (Hash Map) |
| Penyimpanan data unik | Set |
| Antrian dua arah | `collections.deque` |
| Prioritas elemen | `heapq` (Priority Queue) |

**Contoh perbandingan efisiensi:**

```
data_list = [1, 2, 3, 4, 5]
print(3 in data_list)  # Kompleksitas waktu: O(n)

data_set = {1, 2, 3, 4, 5}
print(3 in data_set)   # Kompleksitas waktu: O(1)
```

Dari contoh tersebut terlihat bahwa penggunaan `set` lebih efisien daripada `list` dalam operasi pencarian jika tidak membutuhkan urutan elemen.

# Bab 5
# Struktur Data Fundamental

*"Pemilihan struktur data yang tepat seringkali lebih penting dari algoritma itu sendiri."* — **Robert Lafore**

## 5.1 Array

Struktur data array merupakan salah satu konsep dasar yang sangat penting dalam pemrograman komputer, terutama dalam konteks algoritma dan struktur data (Åkerblom et al., 2020; Moffat & Mackenzie, 2022). Meskipun dalam bahasa pemrograman tertentu, array merupakan tipe data yang terpisah, di Python konsep array lebih sering diterjemahkan ke dalam struktur data list. Python tidak memiliki array secara eksplisit seperti bahasa pemrograman lain (misalnya C++ atau Java), namun list di Python memiliki kemampuan yang serupa dengan array, bahkan lebih fleksibel dalam banyak kasus.

### 5.1.1 Konsep Array dalam Struktur Data

Pada dasarnya, array adalah struktur data yang menyimpan elemen-elemen data dalam urutan tertentu, di mana setiap elemen dapat diakses menggunakan indeks numerik (Åkerblom & Castegren, 2023). Array sangat berguna ketika kita membutuhkan struktur data yang memungkinkan kita untuk menyimpan elemen-elemen yang memiliki tipe data yang sama dan membutuhkan akses cepat berdasarkan indeks

**Kelebihan array**

- Penyimpanan elemen secara berurutan di memori (Åkerblom & Castegren, 2023).
- Akses elemen berdasarkan indeks yang sangat cepat (O(1)) (Makor et al., 2022).

- Memudahkan dalam pengelolaan elemen-elemen dengan urutan tertentu (Soroush et al., 2011).

Namun, array juga memiliki keterbatasan:

- Ukuran array bersifat tetap, dan harus ditentukan pada saat pembuatan (Tuersley, 2004).
- Tidak fleksibel dalam hal penambahan atau penghapusan elemen setelah array dibuat (Kumakiri et al., 2006).

### 5.1.2 List dalam Python: Implementasi Array

Di Python, list merupakan struktur data yang sangat mirip dengan array, tetapi dengan fleksibilitas yang lebih besar. List Python dapat menyimpan elemen-elemen dari berbagai tipe data dalam satu struktur, termasuk angka, string, dan objek lainnya. List Python juga memiliki kemampuan untuk menambah, menghapus, dan mengubah elemen, sehingga lebih fleksibel dibandingkan array tradisional yang memiliki ukuran tetap.

Berikut adalah contoh implementasi dasar array menggunakan list di Python:

```
# Membuat list
array = [10, 20, 30, 40, 50]

# Akses elemen menggunakan indeks
print(array[0])  # Output: 10
print(array[3])  # Output: 40

# Mengubah elemen pada indeks tertentu
array[1] = 25
print(array)  # Output: [10, 25, 30, 40, 50]
```

**Output:**

```
10
40
[10, 25, 30, 40, 50]
```

Pada contoh di atas, kita membuat list dengan lima elemen yang berisi angka. List tersebut dapat diakses menggunakan indeks, dan kita dapat mengubah elemen pada indeks tertentu. Python tidak membatasi ukuran list, yang berarti kita dapat menambah elemen baru pada list kapan saja.

### 5.1.3 Operasi Dasar pada List: Indexing dan Slicing

Di dalam list Python, kita dapat melakukan operasi dasar seperti indexing dan slicing. Operasi indexing memungkinkan kita untuk mengakses elemen tertentu dalam list dengan memberikan indeksnya, sementara slicing memungkinkan kita untuk mengambil sublist dari list asli berdasarkan rentang indeks yang ditentukan.

#### Indexing

Indexing adalah cara untuk mengakses elemen berdasarkan indeks. Di Python, indeks dimulai dari angka 0, artinya elemen pertama memiliki indeks 0, elemen kedua memiliki indeks 1, dan seterusnya. Berikut adalah contoh penggunaan indexing:

```
array = [10, 20, 30, 40, 50]

# Akses elemen pertama
print(array[0])  # Output: 10

# Akses elemen terakhir (indeks -1 untuk elemen terakhir)
print(array[-1])  # Output: 50
```

**Output:**

```
10
50
```

#### Slicing

Slicing adalah operasi yang digunakan untuk mengambil bagian dari list. Dengan slicing, kita dapat menentukan rentang indeks yang ingin kita ambil. Sintaks dasar slicing adalah: list[start:end], di mana start adalah indeks awal dan end adalah indeks akhir (indeks ini tidak termasuk dalam hasil). Berikut adalah contoh penggunaan slicing:

```
array = [10, 20, 30, 40, 50]

# Mengambil elemen dari indeks 1 sampai 3 (indeks 3 tidak
termasuk)
print(array[1:3])  # Output: [20, 30]

# Mengambil elemen dari indeks 2 sampai akhir
print(array[2:])  # Output: [30, 40, 50]

# Mengambil elemen dari awal hingga indeks 3
print(array[:3])  # Output: [10, 20, 30]
```

**Output:**

```
[20, 30]
[30, 40, 50]
[10, 20, 30]
```

Dengan slicing, kita bisa mengambil sublist dengan berbagai cara yang fleksibel, yang sangat berguna dalam berbagai algoritma pengolahan data.

### Fleksibilitas List di Python

Python menyediakan berbagai operasi tambahan pada list yang memungkinkan kita untuk menambahkan, menghapus, dan mengubah elemen dalam list dengan sangat mudah. Beberapa operasi dasar lainnya meliputi:

- `append()`: Menambahkan elemen ke akhir list.
- `insert()`: Menyisipkan elemen di indeks tertentu.
- `remove()`: Menghapus elemen berdasarkan nilai.
- `pop()`: Menghapus dan mengembalikan elemen pada indeks tertentu.

```
array = [10, 20, 30]

# Menambahkan elemen ke akhir list
array.append(40)
```

```
print(array)  # Output: [10, 20, 30, 40]

# Menyisipkan elemen di indeks 1
array.insert(1, 15)
print(array)  # Output: [10, 15, 20, 30, 40]

# Menghapus elemen dengan nilai tertentu
array.remove(20)
print(array)  # Output: [10, 15, 30, 40]

# Menghapus dan mengembalikan elemen terakhir
popped_element = array.pop()
print(popped_element)  # Output: 40
print(array)  # Output: [10, 15, 30]
```

**Output:**

```
[10, 20, 30, 40]
[10, 15, 20, 30, 40]
[10, 15, 30, 40]
40
[10, 15, 30]
```

### 5.1.4 Kelebihan dan Kekurangan List di Python

Seperti halnya struktur data lainnya, list di Python memiliki kelebihan dan kekurangan. Berikut adalah beberapa kelebihan dan kekurangan utama dari list Python:

**Kelebihan:**

- Fleksibilitas: List di Python dapat menyimpan berbagai tipe data dalam satu list, menjadikannya sangat fleksibel.
- Dinamis: Ukuran list di Python tidak tetap, sehingga kita dapat menambah atau mengurangi elemen dengan mudah.
- Akses cepat: Akses elemen dengan indeks di list sangat cepat (O(1)).

**Kekurangan:**

- Kinerja: Operasi penghapusan atau penyisipan elemen di tengah list dapat mempengaruhi kinerja, karena list harus menggeser elemen-elemen yang ada (O(n)).
- Keterbatasan Tipe Data: Meskipun fleksibel, list tidak memiliki jaminan tipe data yang sama seperti array dalam bahasa lain, yang dapat menjadi masalah dalam konteks aplikasi tertentu.

List di Python adalah implementasi yang sangat fleksibel dari konsep array dalam struktur data. Dengan kemampuan untuk menyimpan berbagai tipe data dan operasi dasar seperti indexing, slicing, serta berbagai metode lainnya, list memberikan kemudahan dalam pengelolaan data. Meskipun terdapat beberapa keterbatasan dalam hal efisiensi pada operasi tertentu, list tetap menjadi salah satu struktur data yang paling sering digunakan dalam pemrograman Python.

## 5.2 List

### 5.2.1 Metode List dalam Python

Python menyediakan berbagai metode untuk memanipulasi dan mengelola elemen dalam list. Metode-metode ini memungkinkan kita untuk menambah, menghapus, mencari, dan memodifikasi elemen-elemen dalam list dengan cara yang efisien. Beberapa metode yang paling sering digunakan adalah `append()`, `insert()`, `remove()`, dan `pop()`, yang telah kita bahas sekilas pada sub bab sebelumnya, di sini kita akan mengulasnya lebih mendalam beserta beberapa metode tambahan yang berguna.

#### 5.2.1.1 append()

Metode `append()` digunakan untuk menambahkan elemen ke akhir list. Metode ini tidak mengembalikan nilai apapun dan menambahkan elemen dalam posisi terakhir.

**Contoh:**

```
list_data = [1, 2, 3]
list_data.append(4)
print(list_data)  # Output: [1, 2, 3, 4]
```

**Output:**

```
[1, 2, 3, 4]
```

### 5.2.1.2 insert()

Metode `insert()` memungkinkan kita untuk menyisipkan elemen pada posisi tertentu dalam list. Kita perlu menentukan dua parameter: indeks di mana elemen akan disisipkan dan elemen yang akan disisipkan.

**Contoh:**

```
list_data = [1, 2, 3]
list_data.insert(1, 1.5)
print(list_data)  # Output: [1, 1.5, 2, 3]
```

**Output:**

```
[1, 1.5, 2, 3]
```

Di sini, kita menyisipkan nilai 1.5 di indeks ke-1, sehingga elemen yang sebelumnya ada di indeks ke-1 akan bergeser ke kanan.

### 5.2.1.3 remove()

Metode `remove()` digunakan untuk menghapus elemen pertama yang ditemukan dengan nilai tertentu. Jika elemen tersebut tidak ditemukan, maka akan terjadi `ValueError`.

Contoh:

```
list_data = [1, 2, 3, 2, 4]
list_data.remove(2)
print(list_data)  # Output: [1, 3, 2, 4]
```

Output:

```
[1, 3, 2, 4]
```

Dalam contoh di atas, kita menghapus elemen 2 pertama yang ditemukan dalam list.

### 5.2.1.4 pop()

Metode `pop()` digunakan untuk menghapus elemen dari list dan mengembalikannya. Jika tidak ada indeks yang diberikan, maka elemen terakhir yang ada dalam list akan dihapus dan dikembalikan. Jika indeks diberikan, elemen pada posisi tersebut akan dihapus.

**Contoh:**

```
list_data = [1, 2, 3]
popped_value = list_data.pop()
print(popped_value)  # Output: 3
print(list_data)  # Output: [1, 2]

# Menghapus elemen di indeks 0
popped_value = list_data.pop(0)
print(popped_value)  # Output: 1
print(list_data)  # Output: [2]
```

**Output:**

```
3
[1, 2]
1
[2]
```

### 5.2.1.5 index()

Metode `index()` digunakan untuk mencari posisi indeks dari elemen pertama yang ditemukan dalam list. Jika elemen tidak ditemukan, akan muncul `ValueError`.

**Contoh:**

```
list_data = [1, 2, 3, 4]
index_of_three = list_data.index(3)
print(index_of_three)  # Output: 2
```

**Output:**

```
2
```

### 5.2.1.6 count()

Metode `count()` digunakan untuk menghitung berapa kali suatu elemen muncul dalam list.

**Contoh:**

```
list_data = [1, 2, 3, 2, 4, 2]
count_of_two = list_data.count(2)
print(count_of_two)  # Output: 3
```

**Output:**

```
3
```

### 5.2.1.7 extend()

Metode `extend()` digunakan untuk menambahkan elemen-elemen dari iterable (seperti list lain, tuple, atau set) ke dalam list yang ada. Metode ini mirip dengan `append()`, tetapi `append()` menambahkan satu elemen, sementara `extend()` menambahkan elemen-elemen dari iterable.

**Contoh:**

```
list_data = [1, 2, 3]
list_data.extend([4, 5])
print(list_data)  # Output: [1, 2, 3, 4, 5]
```

**Output:**

```
[1, 2, 3, 4, 5]
```

### 5.2.2 List Comprehension

List comprehension adalah salah satu fitur yang sangat populer dan efisien di Python untuk membuat list baru berdasarkan list yang sudah ada atau iterable lainnya. Dengan menggunakan list comprehension, kita dapat menulis kode yang lebih ringkas dan lebih mudah dibaca. List comprehension memungkinkan kita untuk mengaplikasikan suatu operasi atau kondisi tertentu pada setiap elemen dalam iterable dan menghasilkan list baru.

Sintaks dasar list comprehension adalah:

```
[expression for item in iterable if condition]
```

- `expression`: operasi atau ekspresi yang akan diterapkan pada setiap elemen dalam iterable.
- `item`: elemen yang sedang diproses dari iterable.
- `iterable`: koleksi data (seperti list, range, dll) yang akan diiterasi.
    - `condition`: (opsional) kondisi yang harus dipenuhi oleh elemen untuk diproses.

**Contoh Penggunaan List Comprehension**

1. Membuat list kuadrat dari angka-angka dalam rentang tertentu:

    ```
    numbers = [1, 2, 3, 4, 5]
    squares = [x**2 for x in numbers]
    print(squares)  # Output: [1, 4, 9, 16, 25]
    ```

    **Output:**

    ```
    [1, 4, 9, 16, 25]
    ```

2. Mengambil angka genap dari sebuah list:

```
numbers = [1, 2, 3, 4, 5, 6]
evens = [x for x in numbers if x % 2 == 0]
print(evens)  # Output: [2, 4, 6]
```

**Output:**

```
[2, 4, 6]
```

3. Membuat list dari string yang diubah menjadi huruf kapital:

```
words = ['apple', 'banana', 'cherry']
capitalized_words = [word.upper() for word in words]
print(capitalized_words)  # Output: ['APPLE', 'BANANA',
'CHERRY']
```

**Output:**

```
['APPLE', 'BANANA', 'CHERRY']
```

### Keuntungan Menggunakan List Comprehension:

- **Lebih efisien**: List comprehension sering kali lebih cepat daripada menggunakan metode lain seperti loop for untuk membuat list baru.
- **Kode lebih ringkas**: List comprehension memungkinkan kita untuk menulis kode yang lebih singkat, sehingga lebih mudah dipahami dan lebih mudah dipelihara.
- **Dapat mencakup kondisi**: List comprehension dapat mencakup kondisi yang hanya akan memasukkan elemen yang memenuhi syarat tertentu, memungkinkan kita untuk menghindari pengolahan elemen yang tidak perlu.

Walaupun list comprehension sangat efisien dan ringkas, kita harus berhati-hati untuk tidak menggunakannya dalam situasi yang terlalu rumit, karena hal ini dapat membuat kode menjadi lebih sulit untuk dibaca. Ketika operasi yang akan dilakukan terlalu kompleks atau melibatkan banyak kondisi, menggunakan loop for biasa mungkin lebih mudah dipahami.

## 5.3 Tuple dan String

### 5.3.1 Tuple

Tuple adalah struktur data yang mirip dengan list, tetapi dengan perbedaan utama bahwa tuple bersifat tidak dapat diubah (immutable) setelah dibuat. Artinya, setelah tuple diciptakan, elemen-elemen di dalamnya tidak dapat diubah, ditambah, atau dihapus. Hal ini menjadikan tuple sangat berguna ketika kita memerlukan koleksi data yang tetap dan tidak ingin elemen-elemen dalam koleksi tersebut dimodifikasi secara tidak sengaja.

**Keunggulan menggunakan tuple antara lain:**

- Keamanan Data: Karena sifatnya yang immutable, data dalam tuple tidak bisa diubah, yang menjadikannya lebih aman digunakan dalam situasi di mana data tidak boleh dimodifikasi.
- Kinerja Lebih Cepat: Tuple biasanya memiliki kinerja yang lebih cepat dibandingkan dengan list, terutama dalam hal pencarian dan iterasi. Hal ini disebabkan oleh sifat tuple yang tidak perlu mengelola perubahan data seperti halnya list.
- Penggunaan Sebagai Kunci dalam Dictionary: Karena tuple bersifat immutable, tuple dapat digunakan sebagai kunci dalam dictionary, sementara list tidak bisa.

**Membuat dan Menggunakan Tuple**

Tuple dibuat dengan cara yang sangat mirip dengan list, namun dengan menggunakan tanda kurung biasa () (bukan tanda kurung siku []). Tuple dapat menyimpan berbagai tipe data yang berbeda, seperti angka, string, atau bahkan objek lainnya.

```
# Membuat tuple
tuple_data = (1, 2, 3, 4, 5)

# Mengakses elemen dalam tuple menggunakan indexing
print(tuple_data[2])  # Output: 3
```

```
# Menggunakan tuple sebagai kunci dalam dictionary
dict_data = {tuple_data: "nilai tuple"}
print(dict_data)  # Output: {(1, 2, 3, 4, 5): 'nilai tuple'}
```

**Output:**

```
3
{(1, 2, 3, 4, 5): 'nilai tuple'}
```

## Operasi pada Tuple

Meskipun tuple bersifat immutable, kita masih bisa melakukan operasi-operasi dasar seperti indexing dan slicing, yang memungkinkan kita untuk mengakses elemen-elemen dalam tuple.

**Contoh:**

- **Indexing**: Sama seperti pada list, kita dapat mengakses elemen dalam tuple dengan menggunakan indeks.
- **Slicing**: Mengambil bagian tertentu dari tuple dengan cara yang mirip dengan list.
- **Penggabungan Tuple**: Kita dapat menggabungkan dua atau lebih tuple dengan menggunakan operator +.

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)

# Menggabungkan tuple
tuple_combined = tuple1 + tuple2
print(tuple_combined)  # Output: (1, 2, 3, 4, 5, 6)

# Slicing
print(tuple_combined[1:4])  # Output: (2, 3, 4)
```

**Output:**

```
(1, 2, 3, 4, 5, 6)
(2, 3, 4)
```

Namun, karena tuple tidak dapat diubah, kita tidak bisa menambah atau menghapus elemen dalam tuple setelah tuple dibuat.

### Kapan Menggunakan Tuple?

Tuple sering digunakan ketika kita ingin memastikan bahwa data yang ada tidak akan berubah selama program berjalan. Beberapa contoh penggunaannya adalah:

- **Data yang tidak boleh diubah**: Misalnya, koordinat titik (x, y) atau konfigurasi tetap yang tidak boleh diubah setelah ditetapkan.
- **Sebagai kunci dictionary**: Karena tuple bersifat immutable, tuple dapat digunakan sebagai kunci dalam dictionary, sementara list tidak dapat.

## 5.3.2 String

String adalah urutan karakter yang digunakan untuk menyimpan dan memanipulasi teks. Di Python, string adalah tipe data yang sangat penting dan sering digunakan dalam banyak aplikasi. Seperti tuple, string juga bersifat immutable, yang berarti kita tidak dapat mengubah isi string setelah string tersebut dibuat.

### Karakteristik String

- **Immutable**: String di Python tidak dapat diubah setelah dibuat. Jika kita ingin mengubah string, kita harus membuat string baru berdasarkan modifikasi yang kita inginkan.
- **Mudah Diproses**: Python menyediakan berbagai metode yang memungkinkan kita untuk melakukan berbagai operasi pada string, seperti pencarian, penggabungan, pemotongan, dan pengubahan format.
- **Dukungan Unicode**: String di Python mendukung karakter Unicode, yang memungkinkan kita untuk bekerja dengan teks dalam berbagai bahasa dan karakter khusus.

### Membuat dan Menggunakan String

Untuk membuat string di Python, kita cukup menulis karakter-karakter dalam tanda kutip ganda (" ") atau tunggal (' ').

```
# Membuat string
string_data = "Halo, Dunia!"
# Mengakses karakter dalam string menggunakan indexing
print(string_data[0])  # Output: H

# Menggunakan slicing untuk mendapatkan substring
print(string_data[5:10])  # Output: Dunia
```

**Output:**

```
H
 Duni
```

### Metode pada String

Python menyediakan berbagai metode yang sangat berguna untuk bekerja dengan string. Beberapa metode paling umum digunakan adalah:

- `upper()`: Mengubah semua karakter dalam string menjadi huruf kapital.
- `lower()`: Mengubah semua karakter dalam string menjadi huruf kecil.
- `replace()`: Mengganti bagian dari string dengan substring yang baru.
- `split()`: Memecah string menjadi list berdasarkan pemisah tertentu.
- `strip()`: Menghapus karakter tertentu (biasanya spasi) dari awal dan akhir string.

**Contoh penggunaan metode-metode string:**

```
# Mengubah string menjadi huruf kapital
string_data = "halo"
```

```
print(string_data.upper())  # Output: HALO

# Mengganti kata dalam string
new_string = "Halo, Dunia!".replace("Dunia", "Python")
print(new_string)  # Output: Halo, Python!

# Memecah string menjadi list
words = "saya suka python".split()
print(words)  # Output: ['saya', 'suka', 'python']

# Menghapus spasi di awal dan akhir string
whitespace_string = "   halo   "
print(whitespace_string.strip())  # Output: halo
```

**Output:**

```
HALO
Halo, Python!
['saya', 'suka', 'python']
Halo
```

## String dan Pengolahan Teks

String sangat sering digunakan dalam pengolahan teks. Beberapa operasi dasar yang biasa dilakukan pada string adalah:

- **Pencarian**: Menggunakan metode seperti `find()` atau `in` untuk memeriksa apakah suatu substring ada dalam string.
- **Penggabungan**: Menggunakan operator `+` atau metode `join()` untuk menggabungkan beberapa string.
- **Format String**: Python mendukung format string yang sangat kuat, memungkinkan kita untuk menyisipkan variabel atau ekspresi dalam string.

**Contoh format string:**

```
name = "John"
age = 25
message = f"Nama saya {name} dan saya berumur {age} tahun."
print(message)  # Output: Nama saya John dan saya berumur 25
tahun.
```

**Output:**

```
Nama saya John dan saya berumur 25 tahun.
```

Tuple dan string adalah dua struktur data fundamental di Python yang memiliki karakteristik yang sangat berguna dalam pengolahan data. Keduanya bersifat immutable, yang memberikan keunggulan dalam hal keamanan dan efisiensi kinerja, namun juga membatasi fleksibilitas dalam modifikasi data.

- Tuple sangat berguna ketika kita ingin menyimpan data yang tidak boleh diubah dan sering digunakan sebagai kunci dalam dictionary atau untuk mewakili data yang berpasangan (seperti koordinat).
- String adalah tipe data yang sangat penting dalam pemrograman dan pengolahan teks. Metode-metode yang ada pada string memungkinkan kita untuk melakukan berbagai operasi pengolahan teks yang sangat efisien.

## 5.4 Koleksi Bawaan: set dan dict

### 5.4.1 Set

Set di Python adalah kumpulan elemen yang tidak terurut dan tidak ada duplikat. Konsep ini sangat mirip dengan set dalam matematika, di mana kita hanya menyimpan elemen-elemen unik, dan tidak peduli dengan urutannya. Set di Python sangat berguna untuk operasi seperti pencarian elemen yang cepat, serta untuk melakukan operasi matematika seperti irisan (*intersection*), gabungan (*union*), dan selisih (*difference*).

**Karakteristik Set**

- **Tidak terurut**: Elemen-elemen dalam set tidak memiliki urutan tertentu, sehingga kita tidak dapat mengaksesnya menggunakan indeks seperti list atau tuple.

- **Unik**: Set hanya dapat berisi elemen-elemen yang unik. Jika kita mencoba untuk menambahkan elemen yang sudah ada, maka elemen tersebut tidak akan ditambahkan lagi.
- **Mutable**: Set di Python adalah tipe data yang dapat dimodifikasi (mutable), meskipun elemen-elemennya harus bersifat hashable (dapat dihitung nilai hash-nya).

### Membuat dan Menggunakan Set

Set di Python dapat dibuat menggunakan kurung kurawal `{}` atau menggunakan fungsi `set()`.

```
# Membuat set menggunakan kurung kurawal
set_data = {1, 2, 3, 4, 5}

# Membuat set dari list (duplikat otomatis dihapus)
set_from_list = set([1, 2, 2, 3, 4])
print(set_from_list)  # Output: {1, 2, 3, 4}
```

**Output:**

```
Output: {1, 2, 3, 4}
```

### Operasi Dasar pada Set

- **Menambah elemen**: Untuk menambahkan elemen ke dalam set, kita menggunakan metode `add()`.

  ```
  set_data = {1, 2, 3}
  set_data.add(4)
  print(set_data)  # Output: {1, 2, 3, 4}
  ```

- **Menghapus elemen**: Untuk menghapus elemen dari set, kita dapat menggunakan metode `remove()` atau `discard()`. Perbedaan utamanya adalah `remove()` akan memunculkan error jika elemen tidak ditemukan, sementara `discard()` tidak.

  ```
  set_data.remove(2)
  print(set_data)  # Output: {1, 3, 4}
  ```

```
set_data.discard(5)  # Tidak akan error meskipun 5 tidak
ada di dalam set
```

- **Operasi matematika**: Set menyediakan berbagai operasi matematika yang sangat berguna untuk manipulasi data, seperti gabungan (*union*), irisan (*intersection*), dan selisih (*difference*).
- **Gabungan (*union*)**: Menggabungkan dua set untuk menghasilkan elemen-elemen dari keduanya.

```
set_a = {1, 2, 3}
set_b = {3, 4, 5}
union_set = set_a | set_b
print(union_set)  # Output: {1, 2, 3, 4, 5}
```

- **Irisan (*intersection*)**: Mengambil elemen yang ada di kedua set.

```
intersection_set = set_a & set_b
print(intersection_set)  # Output: {3}
```

- **Selisih (*difference*)**: Mengambil elemen yang ada di set pertama tetapi tidak ada di set kedua.

```
difference_set = set_a - set_b
print(difference_set)  # Output: {1, 2}
```

### Aplikasi Set

Set sangat berguna dalam berbagai kasus, terutama untuk operasi yang melibatkan keunikan elemen atau pencarian yang cepat. Beberapa contoh penggunaan set adalah:

- **Menghapus elemen duplikat**: Set secara otomatis menghapus elemen yang duplikat ketika kita mengonversi list menjadi set.
- **Mencari elemen yang sama antara dua koleksi**: Dengan operasi irisan, kita bisa mencari elemen yang sama antara dua set.
- **Penggunaan untuk memeriksa keanggotaan**: Karena set menggunakan struktur data hash, pemeriksaan apakah suatu elemen ada dalam set sangat cepat.

### 5.4.2 Dictionary: Key-Value Storage

Dictionary di Python adalah struktur data yang menyimpan data dalam bentuk pasangan key-value. Setiap elemen dalam dictionary terdiri dari kunci (key) yang unik, dan nilai (value) yang terkait dengan kunci tersebut. Dictionary sangat berguna ketika kita membutuhkan pemetaan data, di mana kita bisa mengakses data berdasarkan kunci alih-alih mengaksesnya berdasarkan indeks seperti pada list atau tuple.

Beberapa karakteristik dictionary antara lain:

- **Unik Kunci (Key)**: Setiap kunci dalam dictionary harus unik. Namun, nilai yang terkait dengan kunci bisa berulang.
- **Mutable**: Dictionary bersifat mutable, yang berarti kita dapat menambah, menghapus, atau mengubah pasangan kunci-nilai setelah dictionary dibuat.
- **Tidak terurut (Pada versi sebelumnya)**: Sejak Python 3.7, dictionary mempertahankan urutan penyisipan elemen, meskipun masih lebih sering dianggap tidak terurut karena urutan tidak dijamin dalam semua versi Python sebelumnya.
- **Kunci harus bersifat hashable**: Kunci dalam dictionary harus bersifat hashable, yang berarti kunci tersebut harus memiliki nilai hash yang tetap (seperti string, angka, atau tuple).

#### 5.4.2.1 Membuat dan Menggunakan Dictionary

Dictionary dibuat menggunakan kurung kurawal {} dengan pasangan kunci-nilai yang dipisahkan oleh titik dua (:).

```
# Membuat dictionary
dict_data = {'nama': 'John', 'usia': 30, 'pekerjaan':
'programmer'}

# Mengakses nilai dengan kunci
print(dict_data['nama'])  # Output: John

# Menambahkan elemen baru
dict_data['alamat'] = 'Jakarta'
```

```
print(dict_data)   # Output: {'nama': 'John', 'usia': 30,
'pekerjaan': 'programmer', 'alamat': 'Jakarta'}
```

Output:

```
John
{'nama': 'John', 'usia': 30, 'pekerjaan': 'programmer',
'alamat': 'Jakarta'}
```

### 5.4.2.2 Operasi Dasar pada Dictionary

**1. Mengakses nilai dengan kunci**

Nilai dalam dictionary dapat diakses dengan menggunakan kunci.

```
print(dict_data['usia'])  # Output: 30
```

**Output:**

```
30
```

**2. Menghapus elemen**

Elemen dalam dictionary dapat dihapus menggunakan metode del atau metode pop(). Metode pop() menghapus pasangan kunci-nilai dan mengembalikan nilai yang terkait dengan kunci tersebut.

```
# Menghapus elemen dengan kunci
del dict_data['alamat']
print(dict_data)  # Output: {'nama': 'John', 'usia': 30,
'pekerjaan': 'programmer'}

# Menggunakan pop untuk menghapus dan mengembalikan nilai
usia = dict_data.pop('usia')
print(usia)  # Output: 30
print(dict_data)  # Output: {'nama': 'John', 'pekerjaan':
'programmer'}
```

**Output:**

```
{'nama': 'John', 'usia': 30, 'pekerjaan': 'programmer'}
30
```

```
{'nama': 'John', 'pekerjaan': 'programmer'}
```

**3. Menambah atau mengubah nilai**

Nilai yang terkait dengan kunci dapat diubah dengan cara mengakses kunci tersebut dan memberi nilai baru.

```
dict_data['pekerjaan'] = 'data scientist'
print(dict_data)  # Output: {'nama': 'John', 'pekerjaan':
'data scientist'}
```

**Output:**

```
{'nama': 'John', 'pekerjaan': 'data scientist'}
```

### 5.4.2.3 Aplikasi Dictionary

Dictionary adalah struktur data yang sangat berguna dalam pengolahan data yang melibatkan pasangan kunci-nilai. Beberapa contoh penggunaannya adalah:

- **Pemetaan data**: Dictionary digunakan untuk memetakan data seperti nama pengguna ke ID pengguna atau alamat email ke nama pengguna.
- **Pengelolaan pengaturan**: Dictionary sangat berguna untuk menyimpan konfigurasi pengaturan yang berupa pasangan kunci-nilai.
- **Pengolahan data JSON**: Dictionary sering digunakan untuk menangani data dalam format JSON, yang terdiri dari pasangan kunci-nilai.

Set dan dictionary adalah dua koleksi bawaan yang sangat berguna di Python, dengan karakteristik yang membuatnya sangat efisien dalam pengolahan data yang melibatkan elemen unik atau pemetaan data. Set berguna untuk operasi yang melibatkan keanggotaan cepat dan operasi matematika seperti gabungan, irisan, dan selisih. Sementara itu, dictionary sangat efisien dalam menyimpan dan mengakses data yang terkait dengan pasangan kunci-nilai.

# Bab 6
# Array dan List dalam Python

*"Dalam dunia Python, list adalah pisau serbaguna: sederhana namun sangat kuat."* — **Anonim**

## 6.1 Pendahuluan

Dalam pemrograman Python, struktur data memegang peranan penting dalam pengelolaan dan manipulasi data. Dua jenis struktur data yang sering digunakan adalah `list` dan `array`. `List` merupakan struktur data bawaan Python yang bersifat dinamis, fleksibel, dan mampu menyimpan berbagai jenis data dalam satu wadah, seperti angka, string, hingga objek lainnya. Elemen dalam *list* bersifat terurut dan dapat diakses menggunakan indeks. Keunggulan utama dari *list* adalah kemudahannya dalam penggunaan serta fleksibilitas dalam menyimpan tipe data yang berbeda (Lutz & Ascher, 2009).

Sementara itu, *array* dalam Python umumnya merujuk pada struktur data dari modul eksternal seperti array atau `numpy.array`. Modul array dari pustaka standar hanya mendukung elemen dengan tipe data homogen dan menawarkan efisiensi ruang penyimpanan lebih baik dibanding *list*. Di sisi lain, `numpy.array` dari pustaka NumPy menjadi pilihan utama untuk operasi komputasi numerik karena mendukung operasi vektor dan matriks yang efisien serta cepat(Oliphant, 2020). Python memiliki jenis tipe data yang dapat direpresentasikan mirip seperti array, tetapi memiliki karakteristik dan keunikannya tersendiri. Tipe data tersebut, antara lain list, tuple, dictionary, dan set.

Dengan demikian, pemilihan antara `list` dan `array` bergantung pada kebutuhan spesifik pengguna, apakah lebih mengutamakan fleksibilitas atau efisiensi performa.

## 6.2 List

Tipe data pertama yang akan dibahas adalah `list`. Dalam Python, `list` merupakan salah satu tipe data yang paling sering digunakan karena fleksibilitasnya. `List` adalah struktur data yang menyimpan kumpulan nilai atau elemen yang terurut. Setiap elemen dalam `list` dapat berupa berbagai tipe data, seperti angka, string, atau bahkan list lainnya. Elemen-elemen tersebut disebut juga sebagai item. Urutan elemen dalam `list` sangat penting karena dapat diakses menggunakan indeks tertentu. Karena sifatnya yang dinamis dan serbaguna, `list` sangat berguna dalam berbagai konteks pemrograman Python (Lubanovic, 2019).

### 6.2.1 Membuat Sebuah List

Terdapat beberapa cara untuk membuat tipe data `list` dalam Python. Salah satunya adalah dengan menggunakan tanda kurung siku `[]`, di mana elemen-elemen dalam `list` dipisahkan oleh tanda koma, misalnya `[value1, value2, ...]`. Selain itu, `list` juga dapat dibuat menggunakan fungsi bawaan `list()`, yang memungkinkan pembuatan *list* dari berbagai jenis iterable. `List` dapat berisi nol atau lebih elemen, tergantung pada kebutuhan pengguna (Winardi, Pemrograman Python Untuk Pemula, 2023). Fleksibilitas ini menjadikan *list* sebagai salah satu struktur data yang paling umum digunakan dalam pemrograman Python.

**Pertama**, untuk membuat list dapat menggunakan kurung `[]`**.** Untuk membuatnya juga harus disiapkan variabel yang akan digunakan sebagai penampung `list` tersebut.

**Contoh:**

```
# membuat list menggunakan kurung siku
list_kosong = []
tahun_lahir = [2009, 2010, 2011, 2021]
nama_teman  = ["aArum", "Rizal", "Dyan", "Ajie"]
nilai_uts   = [90.5, 75, 8, 77, 9]
lulus       = [True, False]
```

```
biodata     = ["Alif", 12, True]

print(tahun_lahir)
print(nama_teman)
print(nilai_uts)
print(lulus)
print(biodata)
```

**Output:**

```
[2009, 2010, 2011, 2021]
['aArum', 'Rizal', 'Dyan', 'Ajie']
[90.5, 75, 8, 77, 9]
[True, False]
['Alif', 12, True]
```

List juga bisa berisi berbagai macam jenis tipe data yaitu dari integer, float, string hingga boolean seperti pada contoh program di atas.

**Kedua**, selain dengan menggunakan kurung siku, untuk membuat list dapat juga menggunakan fungsi internal/bawaan di Python, yaitu `list()`.

**Contoh:**

```
# membuat list menggunakan fungsi list
list_kosong = []
tahun_lahir = list([2009, 2010, 2011, 2021])
nama_teman  = list(["Arum", "Rizal", "Dyan", "Ajie"])
nilai_uts   = list([90.5, 75, 8, 77, 9])
lulus       = list([True, False])
biodata     = list(["Alif", 12, True])

print(tahun_lahir)
print(nama_teman)
print(nilai_uts)
print(lulus)
print(biodata)
```

**Output:**

```
[2009, 2010, 2011, 2021]
['Arum', 'Rizal', 'Dyan', 'Ajie']
[90.5, 75, 8, 77, 9]
[True, False]
['Alif', 12, True]
```

Apabila dilakukan ekseskusi (*running*) hasilnya outputnya akan sama saja kareena sama-sama menjadi type data list, ketika menggunakan kurung siku [] ataupun menggunakan fungsi bawaan list (Kadir, 2019a). Namun untuk meyakinkan apakah type data tersebut benar-benar termasuk list atau bukan, maka bisa digunakan sebuah fungsi yaitu `type()`. Perhatikan contoh berikut:

```
lulus = [True, False]
# memeriksa jenis type data
print(type(list_kosong))
print(type(lulus))

list_kosong = list()
biodata     = list(["Alif", 12, True])

print(type(list_kosong))
print(type(biodata))
```

**Output:**

```
<class 'list'>
<class 'list'>

<class 'list'>
<class 'list'>
```

Selain list yang biasa, bisa dibuat juga list yang ada dalam list yang populer disebut sebagai nested list. Contohnya sebagai berikut:

```
# membuat contoh nested list
kategori = ["ikan", ["biawak", "komodo"], "carnivora"]

print(kategori)
print(type(kategori))
```

**Output:**

```
['ikan', ['biawak', 'komodo'], 'carnivora']
<class 'list'>
```

Selain beberapa fungsi di atas, list juga mempunyai kelebihan yaitu duplikasi item, artinya dalam list dapat dimasukkan beberapa item yang sama dan tidak dianggap salah dalam pemrograman Python. Adapun contohnya sebagai berikut:

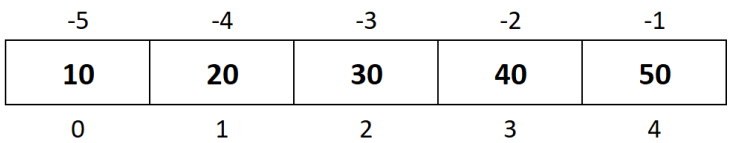
```
tahun_masehi = [2002, 2002, 2003, 2003]
lulus = [True, False, False, True]
```

### 6.2.2 List Indexing dan Slicing dalam Python

Pada Python, `list` merupakan struktur data yang mendukung akses elemen berdasarkan indeksnya. Dua teknik utama yang digunakan untuk mengakses elemen dalam `list` adalah **indexing** dan **slicing**. Dengan *indexing*, kita bisa mengakses satu elemen, sementara *slicing* memungkinkan kita mengambil bagian tertentu dari `list` dengan fleksibilitas tinggi (Kadir, 2018).

#### 6.2.2.1 List Indexing

*Indexing* digunakan untuk mengakses elemen tertentu dalam *list* berdasarkan posisi indeksnya. Indeks dalam Python dimulai dari angka `0` untuk elemen pertama, `1` untuk elemen kedua, dan seterusnya. Python juga mendukung **indeks negatif**, di mana `-1` mengacu pada elemen terakhir, `-2` untuk elemen sebelum terakhir, dan seterusnya (Kadir, 2019a). Sebagai ilustrasi lihat gambar berikut:




Gambar 6.1: Ilustrasi List Indexing

Indeks positif (dari kiri) dimulai dari `0`, sedangkan indeks negatif (dari kanan) dimulai dari `-1`. Untuk mengakses item dalamlist dapat menggunakan angka indeks atau langsung nama item spesifik terkait.

**Contoh:**

```
my_list = [10, 20, 30, 40, 50]
print(my_list[0])
print(my_list[-1])
# Output 10
# Output 50
```

Perlu diperhatikan disini, ketika akan menentukan offset yang lebih panjang dari panjang list, itu akan mengembalikan `error`.

**Contoh:**

```
my_list = [10, 20, 30, 40, 50]
print(my_list[0])
print(my_list[-1])
print(my_list[6])
```

**Output:**

```
10
50
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-1-0a68961c11ec> in <cell line: 1>()
      3 print(my_list[-1])
      4
----> 5 print(my_list[6])

IndexError: list index out of range
```

Terjadinya error karena offset melebihi dari panjang `list offset 6` sementara panjang list hanya ada `5`.

### 6.2.2.2 List Slicing

*Slicing* memungkinkan kita mengambil sebagian elemen dari `list` dengan menggunakan notasi `list[start:stop:step]` (Kadir, 2019b). Parameter:

- `Start` : Indeks awal (termasuk dalam hasil)
- `stop` : Indeks akhir (tidak termasuk dalam hasil)
- `step` : Jarak antar elemen (opsional)

**Contoh Program:**

```
nomor = [1, 2, 3, 4, 5, 6, 7, 8]
print(nomor[2:6])   # slice dari indeks 2 sampai sebelum 6
print(nomor[:4])    # slice dari awal sampai sebelum indeks 4
print(nomor[3:])    # slice dari indeks 3 sampai akhir
print(nomor[::2])   # slice seluruh list dengan step 2
```

**Output:**

```
[3, 4, 5, 6]
[1, 2, 3, 4]
[4, 5, 6, 7, 8]
[1, 3, 5, 7]
```

Sama seperti saat melakukan pengindeksan, untuk melakukan pengirisan (`slicing`) juga harus menggunakan kurung siku. Saat kalian mengiris sebuah item pada list, dalam tanda kurung siku, kalian dapat menentukan offset awal, offset akhir, dan jumlah langkah di antara keduanya (opsional).

**Contoh Program:**

```
# membuat list
bulan = ['Januari', 'Februari', 'Maret', 'April', 'Mei',
'Juni']

#slicing
print(bulan[:]) # untuk mendapatkan seluruh item
```

```
print(bulan[0]) # untuk mendapatkan item pertama
print(bulan[1:]) # untuk mendapatkan item kedua saampai terakhir
print(bulan[:2]) # untuk mendapatkan item pertama sampai ketiga
print(bulan[1:3]) # untuk mendapatkan item kedua sampai keempat
print(bulan[0:3:2]) # untuk mendapatkan item keempat dengan melewatkan 1 item
print(bulan[::-1]) # untuk membalikkan uruta string
```

**Output:**

```
['Januari', 'Februari', 'Maret', 'April', 'Mei', 'Juni']
Januari
['Februari', 'Maret', 'April', 'Mei', 'Juni']
['Januari', 'Februari']
['Februari', 'Maret']
['Januari', 'Maret']
['Juni', 'Mei', 'April', 'Maret', 'Februari', 'Januari']
```

Perlu diperhatikan bahwa nilai untuk pengindeksan dan pengirisan harus berupa bilangan bulat (`integer`). Jika kalian menggunakan `float`, itu akan mengembalikan `error`.

**Contoh:**

```
# membuat list
bulan = ['Januari', 'Februari', 'Maret', 'April', 'Mei', 'Juni']

bulan[1.5]
```

**Output:**

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-748bb345ff49> in <cell line: 0>()
      2 bulan = ['Januari', 'Februari', 'Maret', 'April', 'Mei', 'Juni']
      3
----> 4 bulan[1.5]

TypeError: list indices must be integers or slices, not float
```

Type data `list` ini bersifat *mutable* atau dapat diubah secara mudah, oleh sebab itu untuk mengubah item pada list dapat menggunakan metode *indexing dan slicing* (Khomsah, 2022).

Cara untuk mengubah item pada list dapat menggunakan cara penindeksan, untuk itu harus menentukan ofsetnya dahulu baru kemudian dilanjutkan menggunakan *assignmet operator* (=).

```
# membuat list
bulan = ['Januari', 'Februari', 'Maret', 'April', 'Mei',
'Juni']

# mengganti item pada list
bulan[0] = "Desember"

print(bulan)
```

**Output:**

```
['Desember', 'Februari', 'Maret', 'April', 'Mei', 'Juni']
```

Sedangkan jika akan mengubah dengan menggunakan metode slicing (pengirisan) adalah sebgai berikut:

```
# membuat list
bulan = ['Januari', 'Februari', 'Maret', 'April', 'Mei',
'Juni']

# mengganti item pada list dengan slicing
bulan[0:2] = ["Desember", "Januari"]

print(bulan)
```

**Output:**

```
['Desember', 'Januari', 'Maret', 'April', 'Mei', 'Juni']
```

### 6.2.3 Operator Pada List

Pada Pemrograman Python menyediakan berbagai operasi yang bisa dilakukan pada struktur data `list`, yang memungkinkan manipulasi data secara fleksibel dan efisien. Berikut adalah beberapa operasi umum pada `list`:

#### 6.2.3.1 Operator Penjumlahan (*Concatenation*)

Operator penjumlahan digunakan untuk menggabungkan dua atau lebih list. Operator yang digunakan berupa tanda (+).

**Contoh 1:**

```
# Penggabungan List
x = [1, 2, 3, 4, 5]
y = [6, 7, 8, 9]
z = x + y
print(z)
```

**Output:**

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

**Contoh 2:**

```
# Penggabungan List
Nama1 = ['Fakultas ']
Nama2 = ['Sains ']
Nama3 = ['dan Teknologi']
Gabung = Nama1 + Nama2 + Nama3

print(Gabung)
```

**Output:**

```
['Fakultas ', 'Sains ', 'dan Teknologi']
```

### Operator Pengulangan *(Repetition)*

Operator pengulangan digunakan untuk mengulangi elemen yang dimasukkan ke dalam `list`. Operator yang digunakan berupa tanda asterisk (*).

**Contoh 1:**

```
# Mengulang List

angka1 = [0, 1]
ulang   = angka1 * 3
print(ulang)
```

**Output:**

```
[0, 1, 0, 1, 0, 1]
```

**Contoh 2:**

```
# Mengulang List

angka1 = ['Belajar Python ']
ulang   = angka1 * 3
print(ulang)
```

**Output:**

```
['Belajar Python ', 'Belajar Python ', 'Belajar Python ']
```

Operator yang dapat digunakan hanya penambahan dan perkalian. Operator lain tidak dapat digunakan pada List. Apabila tetap memaksakan untuk menggunakan operator lainnya maka python akan mengembalikan *error*.

**Contoh:**

```
# Operator Python

Nilai = [30, 60, 90]
Bagi = Nilai/3
```

```
print{bagi}
```

**Output:**

```
  File "<ipython-input-14-d94b3fe8bcbc>", line 5
    print{bagi}
          ^
SyntaxError: Missing parentheses in call to 'print'. Did you
mean print(...)?
```

## Operator Perbandingan List

Python juga menyediakan operator perbandingan untuk sebuah List. Operator perbandingan yang digunakan diantaranya seperti ==, <, !=, dan lain sebagainya. Operator perbandingan akan membandingkan item-item dengan ofset yag sama pada kedua list atau lebih (Kadir, 2019a).

**Contoh:**

```
# Menggunakan Operator Perbandingan
a = [1, 2, 3]
b = [1, 2]

# membandingkan list
print(f"a == b --> {a == b}")
print(f"a >= b --> {a >= b}")
print(f"a <= b --> {a <= b}")
print(f"a > b  --> {a > b}")
print(f"a < b  --> {a < b}")
print(f"a != b --> {a != b}")
```

Output:

```
a == b --> False
a >= b --> True
a <= b --> False
a > b  --> True
a < b  --> False
```

```
a != b --> True
```

Pada contoh program tersebut, list a lebih panjang daripada list b. Oleh karena itu, jika kita menerapkan operasi ==, akan mengembalikan `False` walaupun terdapat nilai yang sama.

### 6.2.4 Metode pada List *(List Method)*

Dalam Python, `list` memiliki berbagai metode bawaan (`list` methods) yang memungkinkan manipulasi data secara efisien. Beberapa metode umum antara lain `append()` untuk menambahkan elemen di akhir list, `insert()` untuk menambahkan elemen pada indeks tertentu, `remove()` untuk menghapus elemen berdasarkan nilai, dan `pop()` untuk menghapus elemen berdasarkan indeks. Selain itu, ada juga `sort()` untuk mengurutkan elemen, reverse() untuk membalik urutan, dan `clear()` untuk mengosongkan `list`. Metode-metode ini sangat berguna dalam pengolahan data karena memberikan fleksibilitas dalam menambah, mengubah, maupun menghapus elemen dalam `list`. Berikut daftar lengkap method pada list.

Tabel 6.1: Daftar Method

| Metode | Deskripsi |
|---|---|
| `append()` | Menambahkan elemen di akhir daftar |
| `clear()` | Menghapus semua elemen dari daftar |
| `copy()` | Mengembalikan salinan dari daftar |
| `count()` | Mengembalikan jumlah elemen dengan nilai yang ditentukan |
| `extend()` | Menambahkan elemen dari daftar lain (atau iterable lain) ke akhir daftar saat ini |
| `index()` | Mengembalikan indeks dari elemen pertama dengan nilai yang ditentukan |
| `insert()` | Menambahkan elemen pada posisi tertentu |
| `pop()` | Menghapus elemen pada posisi tertentu |
| `remove()` | Menghapus item dengan nilai yang ditentukan |

| Metode | Deskripsi |
|---|---|
| `reverse()` | Membalik urutan elemen dalam daftar |
| `sort()` | Mengurutkan elemen dalam daftar |

Setiap metode pada `list` memiliki fungsi dan kegunaan yang berbeda-beda. Berikut ini merupakan beberapa metode yang umum digunakan dalam pengolahan `list`. Untuk menambahkan item ke dalam `list`, dapat menggunakan `metode .append()` untuk menambahkan satu elemen di akhir, atau `.extend()` untuk menambahkan beberapa elemen sekaligus dari iterable lain.

**Contoh:**

```
# penggunaan append()
Angka = [10, 20, 30, 40, 50]
Angka.append(100)
print(Angka)

# penggunaan extend()
Nama = ["Dito", "Surya", "Alif", "Rino"]
Nama.extend(Angka)
print(Nama)
```

**Output:**

```
[10, 20, 30, 40, 50, 100]
['Dito', 'Surya', 'Alif', 'Rino', 10, 20, 30, 40, 50, 100]
```

Method `.append()` menambahkan elemen baru ke akhir pada list, sedangkan method `.extend()` akan mengambil list (atau iterable) sebagai argumen dan menambahkan semua elemennya pada list [1].

Kemudian, untuk menghapus item spesifik pada list, kalian dapat menggunakan method `.pop()` dan `.remove()`.

```
# penggunaan pop
Buah = ["Jeruk", "Apel", "Melon", "Semangka"]
Buah.pop(0)
print(Buah)
# penggunaan remove()
```

```
Sayur = ["Kol", "Sawi", "Mentimun", "Bayam"]
Sayur.remove("Mentimun")
print(Sayur)
```

**Output:**

```
['Apel', 'Melon', 'Semangka']
['Kol', 'Sawi', 'Bayam']
```

Pada contoh di atas kedua method digunakan untuk menghapus item, namun method `.pop()` membutuhkan angka indeks (posisi) sebagai argumen list yang akan dihapus, sedangkan methods .remove() membutuhkan nilai spesifik sebagai argumen list yang akan dihapus (Winardi, 2023).

Selanjutnya `method .sort()` memiliki parameter reverse yang dapat bernilai *True* atau *False* (secara default False). Ketika kita ingin mengurutkan item pada list dimulai dari yang terbesar hingga terkecil, maka kita dapat mengatur parameter `reverse=True` dan kebalikannya.

**Contoh:**

```
# menggunakan method sort()
Urut = [99, 77, 87, 75, 100, 85, 65]
# mengurutkan secara descending (dari besar ke kecil)
Urut.sort(reverse=True)
print(Urut)
# mengurutkan secara ascending (dari kecil ke besar)
Urut.sort(reverse=False)
print(Urut)
```

**Output:**

```
[100, 99, 87, 85, 77, 75, 65]
[65, 75, 77, 85, 87, 99, 100]
```

## 6.3 Tuple

Dalam Python, tuple merupakan salah satu struktur data yang mirip dengan list, namun memiliki sifat yang membedakannya secara mendasar. Perbedaan utama antara tuple dan list terletak pada sifat immutability atau ketidakberubahannya. Jika elemen dalam list dapat dimodifikasi, ditambahkan, atau dihapus setelah list dibuat, maka tuple bersifat immutable, yang berarti bahwa data atau elemen di dalamnya tidak dapat diubah setelah tuple tersebut dibuat (Winardi, 2023).

Dengan kata lain, ketika sebuah tuple telah didefinisikan, kita tidak dapat menambahkan elemen baru, menghapus elemen yang ada, atau mengubah nilainya. Hal ini menjadikan tuple lebih aman digunakan untuk menyimpan data tetap atau data yang tidak boleh dimodifikasi secara tidak sengaja selama program berjalan. Secara sintaksis, tuple didefinisikan menggunakan tanda kurung biasa `()` atau bahkan tanpa tanda kurung dalam beberapa kasus, sedangkan list menggunakan tanda kurung siku `[]`.

Contoh: `my_tuple = (1, 2, 3).`

Meskipun terbatas dalam manipulasi, tuple tetap mendukung operasi dasar seperti pengindeksan, iterasi, dan penggabungan. Sifat immutable-nya juga membuat tuple lebih efisien dalam penggunaan memori dan lebih cepat dalam proses eksekusi dibandingkan list untuk data yang tidak perlu diubah.

### 6.3.1 Membuat tuple

Ada beberapa cara untuk membuat tuple pada Python, yaitu menggunakan tanda kurung lengkung `((value1, value2, ...))` dan menggunakan fungsi `tuple()`. Sama seperti list, setiap item yang ada di tuple dipisahkan dengan tanda koma.

**Contoh Menggunakan kurung `()`:**

```
# Contoh membuat Tuple
huruf = ('a', 'b', 'c', 'd', 'e')
print(huruf)
```

```
print(type(huruf))
```

**Output:**

```
('a', 'b', 'c', 'd', 'e')
<class 'tuple'>
```

Pada contoh di atas untuk membuat tuple dibuat dengan menggunakan kurung lengkung `()`. Namun sebenarnya tuple bisa dibuat juga tanpa menggunakan kurung lengkung tetapi tetap menggunakan tanda koma.

**Contoh tanpa menggunakan kurung `()`:**

```
# Contoh Membuat Tuple tanpa kurung
huruf = "a", "b", "c", "d", "e"
nama1 = "Sains",
nama2 = ("teknologi")

print(f"variabel huruf: {huruf} bertipe data{type(huruf)}")
print(f"variabel nama1: {nama1} bertipe data{type(nama1)}")
print(f"variabel nama2: {nama2} bertipe data{type(nama2)}")
```

**Output:**

```
variabel huruf: ('a', 'b', 'c', 'd', 'e') bertipe data<class
'tuple'>
variabel nama1: ('Sains',) bertipe data<class 'tuple'>
variabel nama2: teknologi bertipe data<class 'str'>
```

**Contoh tuple menggunakan kurung `()` dengan koma:**

```
# Contoh Tuple dengan Koma
Angka     = 100,
Kota      = ('Yogyakarta',)
# Ini bukan Tuple
Kabupaten = ("Sleman Yogyakarta")

print(f"Variabel Angka Bertipe {type(Angka)}")
print(f"Variabel Kota Bertipe {type(Kota)}")
print(f"Variabel Kabupaten Bertipe {type(Kabupaten)}")
```

**Output:**

```
Variabel Angka Bertipe <class 'tuple'>
Variabel Kota Bertipe <class 'tuple'>
Variabel Kabupaten Bertipe <class 'str'>
```

Data nilai yang ada di dalam tanda kurung lengkung `()` belum tentu semua bertipe data tuple, namun nilai-nilai yang di pisahkan dengan koma serta berada dalam satu variabel merupakan tipe data tuple. Dengan demikian dapat disimpulkan bahwa tanda koma itulah yang paling menentukan dalam pembuatan tuple yang menggunakan tanda kurung lengkung maupun tidak. Sama halnya seperti list, tuple juga dapat berisi berbagai macam tipe data.

### 6.3.2 Tuple Slicing dan Indexing

Seperti list, *tuple* juga mendukung teknik *slicing*, yaitu cara untuk mengambil sebagian elemen dari tuple berdasarkan indeks. Karena tuple bersifat immutable (tidak dapat diubah), slicing tidak mengubah isi tuple asli, melainkan menghasilkan tuple baru berisi elemen yang dislice (Beazley, 2023). Slicing sangat berguna untuk mengambil bagian data tanpa mengubah tuple aslinya. Karena tuple tidak dapat diubah, slicing menjadi salah satu cara penting untuk mengekstrak atau memproses datanya. Sintaks umum slicing adalah:

`tuple[start:stop:step]`**.** Parameter sintaks slicing tuple adalah:

- `start`: indeks awal (opsional, default = 0)
- `stop`: indeks akhir (tidak termasuk, opsional)
- `step`: langkah lompatan (opsional)

**Contoh:**

```
my_tuple = (10, 20, 30, 40, 50, 60)
print(my_tuple[1:4])   # Output: (20, 30, 40)
print(my_tuple[:3])    # Output: (10, 20, 30)
print(my_tuple[::2])   # Output: (10, 30, 50)
```

Untuk Indexing, tidak ada perbedaan yang signifikan dan sama seperti pada List. Tiap-tiap item di dalam tuple juga memiliki indeks yang dimulai dari 0 sampai sejumlah item yang dimaksud. Seperti pada list untuk dapat mengakses dapat dibuat tuplenya terlebih dahulu, selanjutnya untuk menentukan ofsetnya adalah dengan menggunakan kurung siku [ ] setelah nama variabel tuple.

**Contoh:**

```
# membuat tuple
bulan = ('Januari', 'Februari', 'Maret', 'April')

# slicing
print(bulan[:])      # cetak semua item
print(bulan[0])      # cetak item pertama
print(bulan[1:])     # cetak item kedua sampai terakhir
print(bulan[:2])     # cetak item pertama sampai kedua
print(bulan[1:3])    # cetak item kedua sampai ketiga
print(bulan[0:3:2])  # cetak dari indeks 0 sampai sebelum 3
dengan step 2
print(bulan[::-1])   # membalik urutan
```

**Output:**

```
('Januari', 'Februari', 'Maret', 'April')
Januari
('Februari', 'Maret', 'April')
('Januari', 'Februari')
('Februari', 'Maret')
('Januari', 'Maret')
('April', 'Maret', 'Februari', 'Januari')
```

Tuple mempunyai sifat elemennya *immutable* (tidak dapat diubah) dengan menggunakan *Indexing* dan *Slicing* seperti halnya pada List. Jika mecoba melakukan penggantian elemen pada *tuple* menggunakan metode *indexing* ataupun *Slicing* seperti pada List, maka akan mendapat pengembalian error.

**Contoh:**

```
# membuat tuple
bulan = ('Januari', 'Februari', 'Maret', 'April')

# mengganti tuple dengan indexing
bulan[0] = "Desember"
print(bulan)
```

**Output:**

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-24-41e669ab5b41> in <cell line: 0>()
      4 # mengganti tuple dengan indexing
----> 5 bulan[0] = "Desember"
      6 print(bulan)

TypeError: 'tuple' object does not support item assignment
```

### 6.3.3 Perbandingan Tuple (Tuple Comparison)

Dalam Python, tuple dapat dibandingkan menggunakan operator perbandingan seperti `==`, `!=`, `<`, `>`, `<=`, dan `>=`. Perbandingan ini dilakukan secara leksikografis, yaitu membandingkan elemen-elemen dari kedua tuple satu per satu, dimulai dari elemen pertama. Proses ini berlanjut hingga ditemukan pasangan elemen yang berbeda atau hingga semua elemen dibandingkan (Foundation., 2024). Perhatikan contoh program berikut:

```
tuple1 = (1, 2, 3)
tuple2 = (1, 2, 4)
print(tuple1 < tuple2)  # Output: True
```

Dalam contoh di atas, elemen pertama dan kedua dari `tuple1` dan `tuple2` sama. Namun, elemen ketiga berbeda (3 vs 4), sehingga Python menentukan bahwa `tuple1` lebih kecil dari `tuple2`.

Jika semua elemen yang dibandingkan sama, maka tuple yang lebih pendek dianggap lebih kecil.

```
tuple3 = (1, 2)
```

```
tuple4 = (1, 2, 0)
print(tuple3 < tuple4)  # Output: True
```

Perbandingan tuple ini berguna dalam berbagai konteks, seperti pengurutan data dan evaluasi kondisi logika. Karena tuple bersifat immutable, mereka sering digunakan sebagai kunci dalam struktur data seperti dictionary.

### 6.3.4 Operasi Tuple (Tuple Operations)

Dalam Python, tuple mendukung berbagai operator yang memungkinkan manipulasi dan evaluasi data secara efisien. Operator-operator yang digunakan dalam operasi ini sebagian besar mirip dengan yang digunakan pada list, namun perlu diingat bahwa karena tuple bersifat immutable, hasil operasi tidak akan mengubah tuple asli, melainkan menghasilkan tuple baru jika diperlukan (Beazley, 2023).

Sama seperti list, operasi artimatika yang bisa kita terapkan pada tuple adalah operasi penambahan (+) dan perkalian (*). Operasi penambahan digunakan untuk menggabungkan sebuah tuple. Sedangkan, operasi perkalian digunakan untuk mengulangi tuple beberapa kali.

```
# membuat tuple
a = (10, 20, 30)
b = (100, 200, 300)
# operasi penambahan dan perkalian
print(f"a+b = {a + b}")
print(f"a*3 = {a * 3}")
```

**Output:**

```
a+b = (10, 20, 30, 100, 200, 300)
a*3 = (10, 20, 30, 10, 20, 30, 10, 20, 30)
```

Selain operasi artimatika penambahan dan perkalian, akan mengembalikan error jika kita terapkan pada tuple.

```
# membuat tuple
```

```
a = (10, 20, 30)

# operasi penambahan dan perkalian
print((nilai - 10))
```

**Output:**

```
NameError                                 Traceback (most recent call
last)
<ipython-input-29-0202836741f4> in <cell line: 0>()
      4 # operasi penambahan dan perkalian
----> 5 print((nilai - 10))

NameError: name 'nilai' is not defined
```

### 6.3.5 Tuple Method

Sama halnya seperti `list`, `tuple` juga merupakan sebuah objek. Dengan demikian berarti `tuple` juga mempunyai beberapa method yang dapat dioperasikan. Namun demikian, hanya ada 2 method yang bisa dioperasikan pada tuple.

Tabel 6.2: Tuple Method

| Method | Deskripsi |
|---|---|
| `count()` | Mengembalikan jumlah kemunculan suatu nilai tertentu di dalam tuple |
| `index()` | Mencari nilai tertentu di dalam tuple dan mengembalikan posisi (indeks) tempat nilai tersebut ditemukan |

`Method .count()` mempunyai fungsi untuk menghitung jumlah item spesifik yang ada pada tuple. Pada Method ini hanya memiliki satu parameter yaitu value. Value berupa item yang akan dicari. Sedangkan method `.index()` digunakan untuk mencari nomor indeks suatu item pada tuple. Sama halnya seperti `.count()`, method ini juga memerlukan parameter berupa value. Yang perlu diperhatika adalah ketika terdapat 2 item atau lebih pada suatu tuple, maka nilai indeks dari

item pada urutan pertama lah yang akan digunakan. Akan tetapi, jika tidak ditemukan nilai indeksnya, akan mengembalikan error.

```
# membuat tuple
nilai = (90, 80, 70, 50, 100, 75, 70, 88, 70)

# menggunakan method count()
nilai_70 = nilai.count(70)
print(f"Total nilai yang mendapat nilai 70 : {nilai_70}")
```

**Output:**

```
Total nilai yang mendapat nilai 70 : 3
```

**Contoh yang error:**

```
# membuat tuple
nilai = (90, 80, 70, 50, 100, 75, 70, 88, 70)

# menggunakan method count()
nilai.append(50)
```

**Output:**

```
AttributeError: 'tuple' object has no attribute 'append'
```

## 6.4 Dictionary

*Dictionary* adalah salah satu fitur paling kuat dalam Python karena menjadi dasar bagi banyak algoritma yang efisien dan elegan (Beazley, 2023). Dalam bahasa pemrograman lain, struktur data ini dikenal dengan istilah *associative array*, *hashes*, atau *hashmaps* (Python Software Foundation, 2024).

Mirip dengan `list`, `dictionary` adalah tipe data yang digunakan untuk menyimpan kumpulan data. Namun, berbeda dari `list` yang menggunakan indeks numerik, *dictionary* menyimpan data dalam bentuk pasangan *key-value* (kunci dan nilai). Setiap `key` terhubung dengan satu nilai tertentu, membentuk satu entri yang disebut `item`.

Struktur ini bersifat tidak berurutan (*unordered*), dapat diubah (*mutable*), dan dapat diindeks menggunakan *key*. Satu hal penting: setiap *key* dalam `dictionary` harus bersifat unik dan tidak boleh ada duplikasi.

### 6.4.1 Membuat Dictionary

Untuk membuat *dictionary* di Python, terdapat dua cara utama, yaitu menggunakan kurung kurawal `{}` dengan format `{key1: value1, key2: value2, ...}` atau menggunakan fungsi bawaan `dict()`. Setiap pasangan `key-value` dipisahkan oleh tanda koma. `Key` bisa dianalogikan sebagai indeks dalam `list`, sedangkan `value` adalah data yang disimpan.

Menggunakan tanda kurung kurawal:

```
my_dict = {'nama': 'Ari', 'usia': 25}
```

Selain dengan menggunakan tanda kurung kurawal `{}`, dictionary juga bisa dibuat dengan menggunakan fungsi `dict()`.

```
# membuat dictionary
nilai = {
    1: 'sepuluh',
    2: 'duapuluh',
    3: 'tigapuluh',
    4: 'empat puluh',
    5: 'lima puluh'
}
print(nilai, type(nilai))
```

**Output:**

```
{1: 'sepuluh', 2: 'duapuluh', 3: 'tigapuluh', 4: 'empat puluh', 5:
'lima puluh'} <class 'dict'>
```

Selain dengan cara tersebut di atas, Dictionary juga dapat menggunakan cara berikut:

```
# membuat dictionary
nilai = {1: 'sepuluh', 2: 'duapuluh', 3: 'tigapuluh', 4:
'empat puluh', 5: 'lima puluh'}
print(nilai, type(nilai))
```

**Output:**

```
{1: 'sepuluh', 2: 'duapuluh', 3: 'tigapuluh', 4: 'empat
puluh', 5: 'lima puluh'} <class 'dict'>
```

Dictionary mempunyai hal yang berbeda dengan list dan tuple, key dalam dictionary tidak diijinkan adanya duplikasi. Namun demikian, untuk value dalam dictionary masih diijinkan adanya duplikasi. Jika dalam key terdapat adanya duplikasi, maka hanya key terakhirlah yang akan digunakan.

```
# membuat dictionary
nilai = {1: 'sepuluh', 2: 'duapuluh', 3: 'tigapuluh', 4:
'empat puluh', 5: 'lima puluh'}
print(nilai, type(nilai))
```

**Output:**

```
{1: 'sepuluh', 2: 'duapuluh', 3: 'tigapuluh', 4: 'empat
puluh', 5: 'lima puluh'} <class 'dict'>
```

### 6.4.2 Menambahkan Item Dictionary

Berbeda dengan `tuple` yang bersifat immutable, struktur data *dictionary* dalam Python bersifat mutable, artinya nilai-nilai atau *value* yang tersimpan di dalamnya masih dapat diubah setelah `dictionary` didefinisikan. Hal ini memungkinkan pengguna untuk melakukan pembaruan terhadap isi *dictionary* tanpa harus membuat struktur baru. Namun, ada hal penting yang perlu diperhatikan, yaitu bahwa `key` dalam `dictionary` bersifat tetap dan tidak dapat diubah setelah ditetapkan. `Key` harus unik dan bersifat hashable, sehingga tidak bisa

berupa tipe data yang bisa berubah seperti list atau dictionary lainnya. Meskipun `key` tidak dapat dimodifikasi, kita tetap dapat menambahkan pasangan `key-value` baru ke dalam `dictionary` hanya dengan menetapkan *key* baru dan mengisinya dengan *value* tertentu. Jika `key` tersebut sudah ada sebelumnya, maka nilainya akan diperbarui. Jika *key* belum ada, maka Python akan menambahkannya sebagai item baru. Jika akan mengubah sebuah item pada `dictionary`, maka harus merujuk pada *key* nya, setelah itu tinggal menetapkan *value* yang baru. Selanjutnya, ada beberapa cara untuk melakukan akses pada sebuah item dalam *dictionary*. Untuk itu dapat digunakan cara merujuk pada `key` nya juga atau bisa dengan cara menggunakan method `.get`. Dengan menggunakan cara-cara tersebut dapat memberikan fleksibilitas tinggi dalam manajemen data secara dinamis.

```
# membuat dictionary
hewan = {
    "darat": "kijang",
    "air": "arowana",
    "udara": "rajawali",
    "darat": "gajah"
}

print(hewan)
```

**Output:**

```
{'darat': 'gajah', 'air': 'arowana', 'udara': 'rajawali'}
```

Selain cara-cara di atas, dapat diakses juga key nya saja atau *value* nya saja, bahkan dapat juga diakses *key* nya dan *value* nya saja. Hal ini bisa dilakukan dengan menggunakan *method* yang dapat dioperasikan pada *Dictionary*.

```
# membuat dictionary
biodata = {
    "nama": "zahra",
    "id": 123456,
    "umur": 25,
```

```
    "jenis": "perempuan"
}

# mengakses item
print(biodata.keys())
print(biodata.values())
print(biodata.items())
```

**Output:**

```
dict_keys(['nama', 'id', 'umur', 'jenis'])
dict_values(['zahra', 123456, 25, 'perempuan'])
dict_items([('nama', 'zahra'), ('id', 123456), ('umur', 25),
('jenis', 'perempuan')])
```

Pada saat sedang mengakses *key* saja, dapat menggunakan `method .keys()`. Namun demikian untuk untuk mengkases *value*-nya saja dapat menggunakan `method .values()`. Kemudian, untuk mengakses *key-value* kita bisa menggunakan `method .items()`.

### 6.4.3 Perbandingan Dictionary *(Dictionary Comparison)*

Sama halnya dengan list dan tuple, dictionary juga mempunyai fasilitas operator perbandingan. Namun demikian hanya operator `==` dan `!=` yang dapat dioperasikan pada perbandingan `dictionary`. Operator lain jika dipaksakan bekerja pada `dictionary`, maka akibatnya Python akan memberikan atau mengembalikan pesan `error`.

```
# membuat dictionary
x1 = {10:20, 30:40, 50:60, 70:80}
x2 = {10:20, 30:40, 50:60}
# membandingkan dictionary
print(f"x1 == x2 --> {x1 == x2}")
print(f"x1 != x2 --> {x1 == x2}")
```

**Output:**

```
x1 == x2 --> False
x1 != x2 --> False
```

Selain dari operator di atas, operator perbandingan yang lainnya tidak bisa bekerja jika dioperasikan pada dictionary. Jika tetap memaksakan untuk dipakai di dictionary, maka Python akan memberikan pengembalian error.

```
# membuat dictionary
x1 = {10:20, 30:40, 50:60, 70:80}
x2 = {10:20, 30:40, 50:60}

# membandingkan dictionary
print(f"x1 > x2 --> {x1 > x2}")
print(f"x1 != x2 --> {x1 == x2}")
```

**Output:**

```
TypeError: '>' not supported between instances of 'dict' and
'dict'
```

### 6.4.4 Methode pada Dictionary (Dictionary Method).

Sama hal nya dengan tipe data yang lain yang mempunyai method. `Dictionary` juga memiliki method yang dapat dioperasikan pada `dictionary`. Tabel berikut berisi method yang dapat dioperasikan pada `dictionary`.

Tabel 6.3: `Dictionary` Method

| Method | Deskripsi |
|---|---|
| `clear()` | Menghapus semua elemen dari dictionary |
| `copy()` | Mengembalikan salinan dari dictionary |
| `fromkeys()` | Mengembalikan dictionary baru dengan kunci yang ditentukan dan nilai yang sama untuk semua kunci |
| `get()` | Mengembalikan nilai dari kunci yang ditentukan |

| Method | Deskripsi |
|---|---|
| `items()` | Mengembalikan daftar yang berisi tuple dari setiap pasangan kunci-nilai |
| `keys()` | Mengembalikan daftar berisi semua kunci dalam dictionary |
| `pop()` | Menghapus elemen dengan kunci yang ditentukan |
| `popitem()` | Menghapus pasangan kunci-nilai terakhir yang dimasukkan |
| `setdefault()` | Mengembalikan nilai dari kunci yang ditentukan. Jika kunci tidak ada: tambahkan kunci tersebut dengan nilai baru |
| `update()` | Memperbarui dictionary dengan pasangan kunci-nilai yang ditentukan |
| `values()` | Mengembalikan daftar berisi semua nilai dalam dictionary |

Semua method-method yang ada pada table di atas memiliki fungsi dan kegunaannya masing-masing. Beberapa method di atas sudah dibahas yaitu diantaranya antara lain `.get()`, `.keys()`, `.values()`, dan `.items()`. Namun demikin masih ada lagi method yang sering digunakan yaitu method `.update()` dan `.pop()`.

Method `.update()` digunakan untuk menggabungkan atau menambahkan sebuah dictionary baru, sedangkan method `.pop()` digunakan untuk menghapus sebuah item pada dictionary.

```
# membuat dictionary
a = {'A': 100,
     'B': 85,
     'C': 70}
b = {'D': 60}

# menggunakan method .update
a.update(b)
print(a)
```

**Output:**

```
{'A': 100, 'B': 85, 'C': 70, 'D': 60}
```

**Contoh lain:**

```
# membuat dictionary
a = {'A': 100,
     'B': 85,
     'C': 70,
     'D': 60}

# menggunakan method .pop
a.pop("D")
print(a)
```

**Output:**

```
{'A': 100, 'B': 85, 'C': 70}
```

## 6.5 Set

Dalam dunia pemrograman, pengelolaan dan manipulasi kumpulan data merupakan aspek penting dalam pengembangan perangkat lunak yang efisien. Salah satu struktur data yang sangat berguna untuk keperluan tersebut adalah `set`. Python menyediakan set sebagai tipe data bawaan (built-in type) yang dirancang untuk menyimpan sekumpulan elemen yang tidak berurutan, tidak memiliki indeks, dan yang paling penting, tidak mengizinkan duplikasi elemen.

Secara konsep, set dalam Python mirip dengan himpunan dalam matematika. Artinya, sebuah set hanya dapat menyimpan elemen-elemen yang unik dan mendukung berbagai operasi himpunan seperti union (gabungan), intersection (irisan), difference (selisih), dan symmetric difference (selisih simetris). Hal ini menjadikan set sangat efisien untuk berbagai algoritma yang melibatkan pengecekan keanggotaan atau penghapusan duplikasi.

Untuk membuat set, Python menyediakan dua cara utama: menggunakan tanda kurung kurawal `{}` atau fungsi bawaan `set()`. Penting untuk dicatat bahwa elemen dalam set harus bersifat hashable, sehingga tipe data seperti list atau dictionary tidak dapat dimasukkan sebagai elemen.

Penggunaan `set` memberikan berbagai keunggulan dibandingkan list ketika performa dalam pencarian dan penghilangan duplikasi data menjadi prioritas utama. Oleh karena itu, pemahaman terhadap tipe data `set` menjadi penting dalam penguasaan struktur data dasar Python yang efisien dan elegan.

### 6.5.1 Membuat Set

`Set` dalam Python adalah koleksi elemen **unik** yang tidak memiliki urutan dan tidak dapat diindeks. Python menyediakan dua cara utama untuk membuat *set*, yaitu:

yaitu menggunakan kurung kurawal (`{key1, key2, ...}`) dan menggunakan fungsi `set()`. Sama halnya seperti tipe data yang lainnya, setiap item dalam set juga dipisahkan dengan tanda koma. Meskipun sama-sama menggunakan kurung kurawal seperti halnya pada `dictionary`, namun bedanya adalah setiap item pada `set` tidak memiliki `key-value pair`.

```
# membuat set
bil_genap = {2, 4, 2, 6, 2, 2, 8, 10, 4}  # angka duplikasi
akan dihapus
sekolah = {'IKU'}

print(bil_genap, type(bil_genap))
print(sekolah, type(sekolah))
```

**Output:**

```
{2, 4, 6, 8, 10} <class 'set'>
{'IKU'} <class 'set'>
```

Ada hal yang perlu diperhatikan disini, adalah bahwa tidak bisa membuat set kosong menggunakan kurung kurawal, namun ketika mencoba untuk membuatnya, Python akan membaca hal tersebut sebagai sebuah `dictionary`. Untuk menanggulangi hal tersebut, bisa dibuat dengan menggunakan fungsi `set()`. Selain itu juga, fungsi `set()` juga dapat digunakan untuk mengkonversi jenis tipe data lain menjadi untuk dijadikan data *set*.

Ketika akan mengubah string menjadi set, Python akan melakukan pemecahan string tersebut menjadi substring dan tidak memperbolehkan ada duplikasi. Seprti pada contoh pada string **"brawijaya"** menjadi {'j', 'a', 'i', 'w', 'b', 'y', 'r'} ketika diubah menjadi set. Kemudian untuk list dan tuple tidak ada perubahan yang signifikan saat dikonversi menjadi `set` (hanya saja tetap tidak ada duplikasi data). Namun ketika `dictionary` diubah menjadi `set`, maka akan menyisakan *key*-nya saja. Perhatikan contohnya pada program di atas seperti dictionary {"A": 100, "B": 85, "C": 70, "D": 60} yang awalnya terdiri dari `key-value pair`, tersisa {'A', 'D', 'B', 'C'} saat diubah menjadi `set`.

```
# type data lainnya
sekolah = 'puntadewa'
buah = ['apel', 'mangga', 'melon']  # type list
angka = (100, 60, 70, 66, 75, 90)   # type tuple
peringkat = {'A': 100, 'B': 86, 'c': 70, 'D': 65}  # type
dictionary

# mengkonversi menjadi set
sekolah = set(sekolah)
buah = set(buah)
angka = set(angka)
peringkat = set(peringkat)

print(f"sekolah: {sekolah} bertipe {type(sekolah)}")
print(f"buah: {buah} bertipe {type(buah)}")
print(f"angka: {angka} bertipe {type(angka)}")
print(f"peringkat: {peringkat} bertipe {type(peringkat)}")
```

Output:

```
sekolah: {'a', 'p', 't', 'u', 'd', 'w', 'e', 'n'} bertipe <class
'set'>
buah: {'apel', 'melon', 'mangga'} bertipe <class 'set'>
angka: {66, 100, 70, 75, 90, 60} bertipe <class 'set'>
peringkat: {'B', 'D', 'A', 'c'} bertipe <class 'set'>
```

### 6.5.2 Set operations

Operator aritmatika tidak dapat dioperasikan pada set. Ketika mecoba di operasikan, maka akan mengembalikan *error*.

```
# membuat set
a = {1, 2, 3}
b = {5, 6, 7}

# operator aritmatika
print(a / 7)
```

**Output:**

```
TypeError: unsupported operand type(s) for /: 'set' and 'int'
```

Akan tetapi, terdapat operasi yang bisa dilakukan pada set. Operasi tersebut seperti *union* (gabungan), *intersection* (irisan), *difference*, dan *symmetric difference*. Cara kerja dari masing-masing operasi ini sama seperti yang ada pada operator matematika berikut :

Tabel 6.4: *Operator Set*

| Operator | Deskripsi | Contoh |
|---|---|---|
| \| | Union | {1, 2, 3} \| {3, 4} → {1,2,3,4} |
| & | Intersection | {1, 2, 3} & {3, 4} → {3} |
| - | Difference | {1, 2, 3} - {3, 4} → {1,2} |

| Operator | Deskripsi | Contoh |
|---|---|---|
| ^ | Symetric Difference | `{1, 2, 3} ^ {3, 4}` → `{1,2,4}` |
| in | Is an element off | `3 in {1,2,3}` → `True` |

### 6.5.3 Set Methods

`Set` juga merupakan sebuah objek. Dengan demikian terdapat juga beberapa method yang dapat dioperasikan pada `set`. Berikut table yang berisi method pada `set`.

Tabel 6.5: *Set Method*

| 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 perbedaan antara dua atau lebih set |
| `difference_update()` | `-=` | Menghapus elemen dari set ini yang juga ada di set lain |
| `discard()` | | Menghapus elemen tertentu dari set (tidak error jika elemen tidak ditemukan) |
| `intersection()` | `&` | Mengembalikan set hasil irisan antara dua set |
| `intersection_update()` | `&=` | Memperbarui set dengan menyisakan elemen yang juga ada di set lain |

| Method | Shortcut | Deskripsi |
|---|---|---|
| `isdisjoint()` | | Mengembalikan `True` jika dua set tidak memiliki elemen yang sama |
| `issubset()` | <= | Mengembalikan `True` jika set ini adalah subset dari set lain |
| `(issubset alternatif)` | < | Mengembalikan `True` jika semua elemen dalam set ini ada di set lain, dan ukurannya lebih kecil |
| `issuperset()` | >= | Mengembalikan `True` jika set ini adalah superset dari set lain |
| `(issuperset alternatif)` | > | Mengembalikan `True` jika semua elemen set lain ada di set ini, dan ukurannya lebih besar |
| `pop()` | | Menghapus dan mengembalikan elemen acak dari set |
| `remove()` | | Menghapus elemen tertentu dari set (error jika elemen tidak ditemukan) |
| `symmetric_difference()` | ^ | Mengembalikan set dengan elemen yang tidak sama di kedua set |
| `symmetric_difference_update()` | ^= | Memperbarui set dengan elemen yang tidak sama di kedua set |
| `union()` | ` | ` |
| `update()` | ` | =` |

Selanjutnya akan dibahas beberapa method set yang sering digunakan di dalam pemrograman.

### `.add()`

Method ini digunakan untuk menambahkan satu elemen ke dalam set.

```
# membuat set
angka = {90, 100, 80, 50, 75, 35, 45}

# menggunakan method .add()
angka.add(150)
print(angka)
```

**Output:**

```
{80, 50, 35, 100, 150, 90, 75, 45}
```

### `.union()`

Menggabungkan dua `set` dan mengembalikan `set` baru tanpa duplikasi.

```
# membuat set
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

# menggunakan .union
z = x.union(y)

print(z)
```

Output:

```
# membuat set
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

# menggunakan .union
z = x.union(y)

print(z)
```

### `.update()`

Method ini digunakan untuk menambahkan beberapa elemen sekaligus dari iterable lain (misalnya list atau `set`).

```
# membuat set
x = {"apple", "pisang", "cherry"}
y = {"google", "microsoft", "apple"}

# menggunakan .update
x.update(y)

print(x)
```

**Output:**

```
{'cherry', 'apple', 'google', 'pisang', 'microsoft'}
```

### `.remove()`

Menghapus elemen dari `set`. Jika elemen tidak ditemukan, akan menghasilkan `error`.

```
# membuat set
buah = {"apel", "pisang", "cherry"}

# menggunakan .remove
buah.remove("pisang")

print(buah)
```

**Output:**

```
{'cherry', 'apel'}
```

### `.discard()`

Mirip seperti `.remove()`, tapi **tidak menghasilkan error** jika elemen tidak ditemukan.

```
# membuat set
thisset = {"apple", "banana", "cherry"}

# menggunakan .discard
thisset.discard("banana")

print(thisset)
```

Output:

```
{'cherry', 'apple'}
```

### `.pop()`

Menghapus dan mengembalikan satu elemen acak dari `set`.

```
# membuat set
nama = {"Surya", "Alif", "Momo"}

# menggunakan method .pop
nama.pop()

print(nama)
```

**Output:**

```
{'Momo', 'Alif'}
```

### .clear()

Menghapus semua elemen dalam set.

Berikut adalah contoh penggunaan method `.clear()` pada `set` di Python:

```
# membuat set
hobi = {"membaca", "menulis", "bersepeda"}

# menampilkan isi set sebelum dihapus
print("Sebelum clear:", hobi)
```

```
# menghapus semua elemen
hobi.clear()

# menampilkan isi set setelah dihapus
print("Setelah clear:", hobi)
```

**Output:**

```
Sebelum clear: {'menulis', 'bersepeda', 'membaca'}
Setelah clear: set()
```

Metode `.clear()` akan menghapus **semua elemen** dalam set, dan hasil akhirnya adalah set kosong `set()`.

### `.intersection()`

Mengembalikan elemen yang **sama** dari dua set.

```
# membuat set
buah = {"apel", "pisang", "rambutan", "kuda"}
hewan = {"monyet", "harimau", "kuda", "pisang"}

# menggunakan method .intersection
hasil = buah.intersection(hewan)

print(hasil)
```

**Output:**

```
{'pisang', 'kuda'}
```

### `.difference()`

Mengembalikan elemen yang hanya ada di satu set dan tidak di set lainnya.

```
# membuat set
buah = {"apel", "pisang", "ceri", "mangga"}
soft = {"google", "mangga", "microsoft", "aple"}

# menggunakan method .difference
```

```
hasil = buah.difference(soft)

print(hasil)
```

Output:

```
{'ceri', 'apel', 'pisang'}
```

# Bab 7
# Stack dan Queue: Konsep dan Implementasi

*"Tumpukan dan antrian mengajarkan kita urutan dan prioritas dalam logika."* — Alan Perlis

## 7.1 Konsep Dasar Stack dan Queue

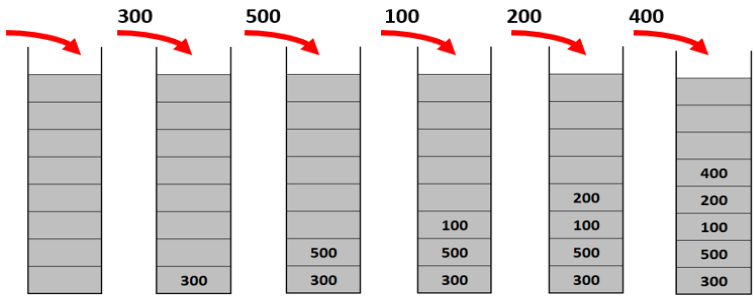
Stack dan Queue merupakan struktur data linear yang sangat penting dan banyak digunakan dalam berbagai aplikasi. Mulai dari aplikasi tingkat dasar seperti pengelolaan memori, hingga tingkat lanjutan seperti algoritma penelusuran graf dan sistem antrean. Stack dan Queue memiliki sifat yang unik dalam mengelola urutan data. Dalam dunia nyata bentuk struktur data ini memiliki penerapan yang luas. Stack dan Queue adalah metode atau teknik dalam menyimpan atau mengambil data ke dan dari memori komputer.

Stack dapat diibaratkan sebuah tumpukan barang dalam sebuah tempat yang hanya memiliki satu pintu di atasnya. Jadi untuk memasukkan dan mengambil barang hanya dapat dilakukan melalui pintu tersebut. Ukuran barang tersebut pas dengan pintunya, sehingga barang yang akan dikeluarkan pertama kali adalah barang yang terakhir kali dimasukkan. Dengan demikian kaidah Stack adalah *First In Last Out* atau *Last In First Out*. Artinya, elemen yang terakhir dimasukkan ke dalam stack akan menjadi elemen pertama yang keluar atau elemen yang pertama di masukkan ke dalam stack menjadi elemen terakhir di keluarkan dari dalam stack.


Queue diibaratkan seperti jejeran orang yang sedang menunggu antrean di sebuah loket. Jadi orang pertama yang datang adalah yang pertama kali dilayani. Contoh dalam dalam dunia nyata adalah antrean pada sebuah *service center*, praktek dokter di sebuah rumah sakit, pelayanan pengurusan data kependudukan di kelurahan dan sebagainya. Jadi pelayanan dilakukan berdasarkan urutan kedatangan atau urutan pendaftaran. Dengan demikian kaidah Queue adalah *First In, First Out*.

## 7.2 Stack

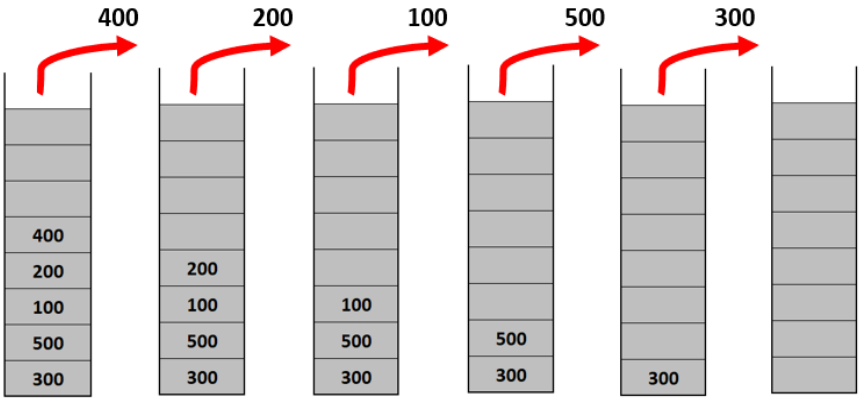
Stack adalah struktur data linear yang menerapkan prinsip LIFO (*Last In, First Out* atau *First In, Last Out*). Pengaksesan data (memasukan dan mengeluarkan) dilakukan hanya dari satu sisi yaitu top. Top adalah posisi data terakhir dimasukkan, dengan kata lain data tersebut berada dibagian atas Stack. Misalkan terdapat data 300, 500, 100, 200, 400. Ilustrasi atau gambaran pemasukan data ke dalam Stack ditunjukkan pada gambar 7.1.




Gambar 7.1: Ilustrasi pemasukan data pada Stack

Sedangkan untuk mengeluarkan data dari dalam sebuah Stack dilakukan hanya dari satu sisi yaitu pada bagian bagian atas. Data yang dikeluarkan berdasarkan urutan masuknya ke dalam Stack. Dimana data terakhir dimasukan akan menjadi data pertama yang dikeluarkan.

Gambar 7.2 merupakan gambaran cara kerja struktur data Stack dalam mengeluarkan data.



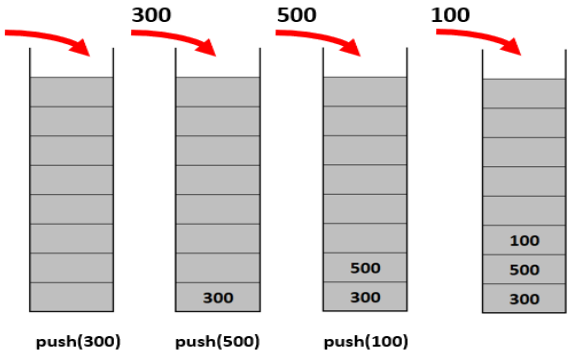

Gambar 7.2: Ilustrasi mengeluarkan data dari Stack

### 7.2.1 Operasi Stack

Untuk mengakses data dari dan ke dalam struktur data Stack menggunakan operasi atau fungsi-fungsi seperti berikut:

1. **Fungsi `push(elemen)`**

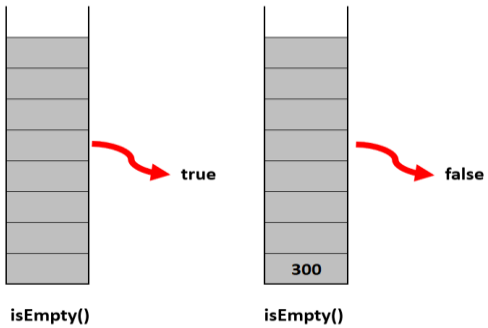
   Fungsi `push(elemen)` digunakan untuk memasukkan elemen atau data ke dalam Stack.




Gambar 7.3: Fungsi `push(elemen)`

**2. Fungsi `isEmpty()`**

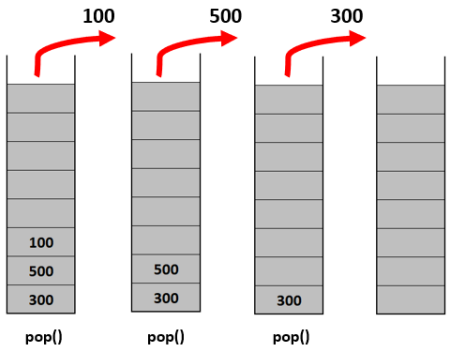
Fungsi `isEmpty()` digunakan untuk memeriksa apakah Stack dalam kondisi kosong atau tidak. Fungsi ini biasanya di panggil sebelum mengeluarkan data dari dalam Stack. Fungsi `isEmpty()` akan mengembalikan hasil yaitu ***true*** jika Stack dalam kondisi kosong dan ***false*** jika Stack tidak kosong. Penggunaan fungsi ini penting untuk menghindari kesalahan (*error*) pada saat mengeluarkan data ketika Stack dalam kondisi kosong.




Gambar 7.4: Fungsi `isEmpty()`

**3. Fungsi `pop()`**

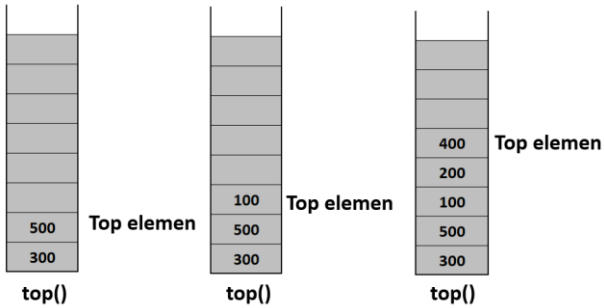
Fungsi `pop()` digunakan untuk mengeluarkan data dari Stack. Data yang dikeluarkan dari Stack berada pada posisi atas (top).




Gambar 7.5: Fungsi `pop()`

**4. Fungsi `top()`**

Fungsi `top()` digunakan untuk membaca elemen paling atas dari sebuah Stack.




Gambar 7.6: Fungsi `top()`

**5. Fungsi `clear()`**

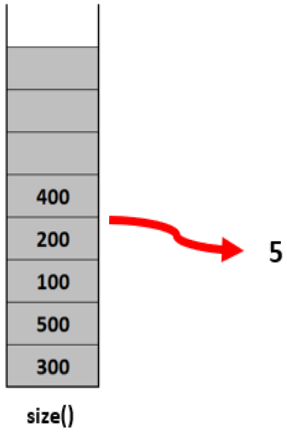
Fungsi clear() digunakan untuk menghapus semua elemen dari Stack.


Gambar 7.7: Fungsi clear()

Untuk membuktikan bahwa Stack dalam kondisi kosong, harus memanggil fungsi `isEmpty()`. Jika Stack dalam kondisi kosong maka nilai yang dikembalikan fungsi `isEmpty()` adalah ***true***.

6. **Fungsi `size()`**

   Fungsi size() digunakan untuk mendapatkan ukuran stack. Pemanggilan fungsi ini akan mengembalikan seberapa banyak data atau jumlah elemen dalam Stack saat ini.




Gambar 7.8: Fungsi `size()`

## 7.2.2 Implementasi Stack

Struktur data Stack dapat di implementasikan dengan dua cara yaitu *array based stack* dan *linked-list based stack*.

### Array Based Stack

Array-Based Stack adalah implementasi struktur stack menggunakan array statis atau dinamis sebagai wadah penyimpanan datanya. Pada Python, dapat mengimplementasikan stack berbasis array dengan menggunakan list bawaan Python. Meskipun list di Python bersifat dinamis, dari sudut pandang struktur data, dapat diperlakukan sebagai array. Berikut adalah implementasi Array Based Stack yang ditulis menggunakan bahasa pemrograman Python.

```
1.  class Stack:
2.      def __init__(self):
3.          self.items = []
4.
```

```
5.      def push(self, data):
6.          self.items.append(data)
7.          print(f"Data: {data}")
8.
9.      def pop(self):
10.         if not self.isEmpty():
11.             data = self.items.pop()
12.             print(f"Keluarkan data (pop) : {data}")
13.             return data
14.         else:
15.             print("Tidak bisa keluarkan data, karena Stack
    kosong!")
16.             return None
17.
18.     def size(self):
19.         return len(self.items)
20.
21.     def isEmpty(self):
22.         return len(self.items) == 0
23.
24.     def top(self):
25.         if not self.isEmpty():
26.             return self.items[-1]
27.         else:
28.             print("Stack  kosong!  Tidak  ada  elemen  di
    top.")
29.             return None
30.
31.     def kosongkan(self):
32.         self.items.clear()
33.         print("Tidak ada data di dalam Stack.")
34.
35.     def tampilkan(self):
36.         print("Tampilkan  isi  Stack  (bawah  ->  atas):",
    self.items)
37. # Membuat objek Stack
38. objekStack = Stack()
39.
40. print("Input Data ke dalam Stack:")
41. objekStack.push(300)
42. objekStack.push(500)
43. objekStack.push(100)
44. objekStack.push(200)
45. objekStack.push(400)
46.
47. objekStack.tampilkan()
```

```
48.
49. print("Posisi data terakhir (top) \t: ", objekStack.top())
50. print("Ukuran atau jumlah elemen \t: ", objekStack.size())
51. print("Apakah Stack kosong? \t\t: ", objekStack.isEmpty())
52.
53. objekStack.pop()
54. objekStack.tampilkan()
55.
56. objekStack.kosongkan()
57. objekStack.tampilkan()
58. print("Apakah Stack kosong? : ", objekStack.isEmpty())
59. objekStack.top()
```

Keluaran atau *output* dari program tersebut ketika dijalankan adalah seperti berikut ini:

```
Input Data ke dalam Stack:
Data: 300
Data: 500
Data: 100
Data: 200
Data: 400

Tampilkan isi Stack (bawah -> atas): [300, 500, 100, 200, 400]
Posisi data terakhir (top)   : 400
Ukuran atau jumlah elemen    : 5
Apakah Stack kosong?         : False

Keluarkan data (pop) : 400
Tampilkan isi Stack (bawah -> atas): [300, 500, 100, 200]

Tidak ada data di dalam Stack.
Tampilkan isi Stack (bawah -> atas): []
Apakah Stack kosong? : True
Stack kosong! Tidak ada elemen di top.
```

### Penjelasan Kode Program

Perhatikan potongan kode pada baris 1-3 berikut ini:

```
class Stack:
    def __init__(self):
        self.items = []
```

Baris 1 merupakan definisi sebuah kelas yang diberi nama **Stack**. Baris 2-3 adalah merupakan konstruktor default yaitu **__init__(self):** yang akan dijalankan secara otomatis saat objek dibuat (lihat baris 38 pada listing program di atas). Pada kosntruktor ini membuat list kosong yang akan digunakan untuk menyimpan elemen-elemen Stack yaitu **self.items = []**.

Parameter ***self*** mengacu pada objek saat ini.

Berikut adalah kode pembuatan objek Stack (baris 38) yaitu:

```
objekStack = Stack()
```

Perhatikan potongan kode pada baris 5-7 berikut ini:

```
def push(self, data):
    self.items.append(data)
    print(f"Data: {data}")
```

Potongan kode tersebut digunakan untuk membuat fungsi yang memasukkan elemen data ke dalam Stack dengan nama `push(self, data):`, di dalam fungsi ini memanggil fungsi `append()` yang berguna untuk memasukan elemen data ke dalam list dengan nama **items** yang sudah didefinisikan pada baris 3, data akan ditambahkan ke elemen akhir (top). Fungsi `append()` sendiri merupakan fungsi bawaan dari Python yang di gunakan untuk menambah data ke dalam list. Perintah atau kode selanjutnya adalah memanggil fungsi `print()` yang digunakan untuk menampilkan hasil atau *output* ke layar.

Pemanggilan fungsi ini terdapat pada baris 41-45 seperti berikut:

```
objekStack.push(300)
objekStack.push(500)
objekStack.push(100)
objekStack.push(200)
objekStack.push(400)
```

Selanjutnya perhatikan potongan kode pada baris 9-16 berikut ini:

```
def pop(self):
    if not self.isEmpty():
        data = self.items.pop()
        print(f"Keluarkan data (pop) : {data}")
        return data
    else:
        print("Tidak bisa keluarkan data, karena Stack
kosong!")
        return None
```

Potongan kode tersebut digunakan untuk membuat fungsi yang mengeluarkan data pada elemen terakhir (*top*) dari dalam Stack dengan nama **pop(self):**, di dalam fungsi ini mengerjakan hal-hal sebagai berikut:

1. Memeriksa apakah Stack dalam kondisi kosong atau tidak.
2. Jika perintah `if not self.isEmpty()` mengembalikan nilai **true**, yang berarti Stack tidak kosong maka keluarkan data pada posisi terakhir atau *top* yang disimpan pada variabel items. Fungsi `pop()` adalah fungsi bawaan dari Python untuk mengeluarkan data dari sebuah list.
3. Elemen data yang dikeluarkan menggunakan perintah `self.items.pop()` akan disimpan ke variabel dengan nama data.
4. Selanjutnya menampilkan elemen data yang dikembalikan saat pemanggilan fungsi.
5. Jika perintah `if not self.isEmpty()` mengembalikan nilai **false**, yang berarti Stack dalam kondisi kosong maka akan menampilkan pesan yang ditulis di dalam fungsi `print()`.
6. Selanjutnya kembalikan None.

Pemanggilan fungsi ini terdapat pada baris 53 seperti berikut:

```
objekStack.pop()
```

Perhatikan potongan kode pada baris 18-19 berikut ini:

```
def size(self):
```

```
    return len(self.items)
```

Potongan kode tersebut digunakan untuk mengetahui ukuran atau jumlah elemen data dalam Stack dengan nama `size(self):`, di dalam fungsi ini memanggil fungsi `len()`. Fungsi `len()` adalah fungsi bawaan dari Python untuk mendapatkan jumlah elemen dalam sebuah list. Ketika fungsi tersebut di panggil oleh maka akan mengembalikan jumlah elemen dalam Stack. Pemanggilan fungsi ini terdapat pada baris 50 seperti berikut:

```
objekStack.size()
```

Selanjutnya perhatikan potongan kode pada baris 21-22 berikut ini:

```
def isEmpty(self):
    return len(self.items) == 0
```

Potongan kode tersebut digunakan untuk memeriksa apakah Stack dalam kondisi kosong atau tidak dengan nama **`isEmpty(self):`**, di dalam fungsi ini memanggil fungsi `len().` Pernyataan `len(self.items) == 0` digunakan untuk membandikan apakah variable items sama dengan nol. Jika ya akan mengembalikan ***true***, jika tidak maka akan mengembalikan ***false***. Fungsi ini biasanya dipanggil untuk memeriksa data sebelum mengeluarkan data dari dalam Stack dan untuk memeriksa data pada posisi terakhir atau *top* pada sebuah Stack. Contoh pemanggilan fungsi ini juga terdapat pada baris 51 dan 58 seperti berikut:

```
objekStack.isEmpty()
```

Selanjutnya perhatikan potongan kode pada baris 24-29 berikut ini:

```
def top(self):
    if not self.isEmpty():
        return self.items[-1]
    else:
        print("Stack kosong! Tidak ada elemen di top.")
        return None
```

Potongan kode tersebut digunakan untuk membuat fungsi yang memeriksa data pada elemen terakhir (*top*) dari Stack dengan nama **top(self):**, di dalam fungsi ini mengerjakan hal-hal sebagai berikut:

1. Memeriksa apakah Stack dalam kondisi kosong atau tidak.
2. Jika perintah **if not self.isEmpty()** mengembalikan nilai **true**, yang berarti Stack tidak kosong, baca elemen terakhir dengan perintah **return self.items[-1]**.
3. Pada Python **items[-1]** berarti elemen terakhir dari list. Elemen ini tidak di hapus.
4. Jika perintah **if not self.isEmpty()** mengembalikan nilai **false**, yang berarti Stack dalam kondisi kosong maka akan menampilkan pesan yang ditulis di dalam fungsi print().
5. Selanjutnya kembalikan None.

Selanjutnya perhatikan potongan kode pada baris 31-33 berikut ini:

```
def kosongkan(self):
    self.items.clear()
    print("Tidak ada data di dalam Stack.")
```

Potongan kode tersebut digunakan untuk membuat fungsi yang mengosongkan semua elemen dari Stack. Fungsi clear() adalah fungsi bawaan Python yang digunakan untuk menghapus semua elemen di dalam list yaitu variabel items, sehingga stack menjadi kosong.

Selanjutnya potongan kode pada baris 35-36 berikut ini:

```
def tampilkan(self):
    print("Tampilkan isi Stack (bawah -> atas):", self.items)
```

Potongan kode tersebut digunakan untuk membuat fungsi yang menampilkan data dari dalam Stack.

### Linked-List-Based Stack

Linked-List-Based Stack adalah struktur data stack yang diimplementasikan menggunakan linked list. Artinya, elemen-elemen stack disimpan dalam node-node yang saling terhubung. Berikut adalah implementasi Linked-List-Based Stack yang ditulis menggunakan bahasa pemrograman Python.

```
1.  # Node untuk menyimpan data dan referensi ke node
    berikutnya
2.  class Node:
3.      def __init__(self, data):
4.          self.data = data
5.          self.next = None
6.
7.  # Stack menggunakan struktur Linked List
8.  class LinkedListStack:
9.      def __init__(self):
10.         self.top = None  # stack kosong
11.
12.     def isEmpty(self):
13.         return self.top is None
14.
15.     def push(self, data):
16.         # Buat node baru
17.         new_node = Node(data)
18.         # Hubungkan node baru ke node top lama
19.         new_node.next = self.top
20.         # Tetapkan node baru sebagai top
21.         self.top = new_node
22.         print(f"Push: {data}")
23.
24.     def pop(self):
25.         if self.isEmpty():
26.             print("Stack kosong! Tidak bisa pop.")
27.             return None
```

```
28.         popped = self.top.data
29.         # Pindahkan top ke node berikutnya
30.         self.top = self.top.next
31.         print(f"Pop: {popped}")
32.         return popped
33.
34.     def peek(self):
35.         if self.isEmpty():
36.             print("Stack kosong! Tidak ada elemen pada
    top.")
37.             return None
38.         return self.top.data
39.
40.     def kosongkan(self):
41.         self.top = None
42.         print("Stack dikosongkan.")
43.
44.     def tampilkan(self):
45.         if self.isEmpty():
46.             print("Stack kosong.")
47.         else:
48.             print("Isi Stack:")
49.             current = self.top
50.             while current:
51.                 print(f" {current.data}", end=" ->")
52.                 current = current.next
53.
54. # Contoh penggunaan
55. if __name__ == "__main__":
56.     objekStack = LinkedListStack()
57.     print("Input Data ke dalam Stack:")
58.     objekStack.push(300)
59.     objekStack.push(500)
60.     objekStack.push(100)
61.     objekStack.push(200)
62.     objekStack.push(400)
63.     objekStack.tampilkan()
64.     print("Peek:", objekStack.peek())
65.     objekStack.pop()
66.     objekStack.tampilkan()
67.     objekStack.kosongkan()
68.     objekStack.tampilkan()
```

Keluaran atau *output* dari program tersebut ketika dijalankan adalah seperti berikut ini:

```
Input Data ke dalam Stack:
Push: 300
Push: 500
Push: 100
Push: 200
Push: 400

Isi Stack:
  400 -> 200 -> 100 -> 500 -> 300 ->

Peek: 400
Pop: 400

Isi Stack:
  200 -> 100 -> 500 -> 300 ->

Stack dikosongkan.
Stack kosong.
```

**Penjelasan Kode:**

Program yang disajikan merupakan implementasi struktur data Stack menggunakan pendekatan Linked List dalam bahasa Python. Stack adalah struktur data linear yang mengikuti prinsip Last In First Out (LIFO), di mana elemen terakhir yang ditambahkan akan menjadi elemen pertama yang dikeluarkan. Dalam implementasi ini, kelas `Node` digunakan untuk merepresentasikan setiap elemen dalam Stack, yang menyimpan data dan referensi ke node berikutnya. Sementara itu, kelas `LinkedListStack` digunakan untuk membangun dan mengelola operasi-operasi Stack dengan memanfaatkan struktur node tersebut.

Metode `__init__` pada kelas `LinkedListStack` menginisialisasi Stack dalam keadaan kosong dengan menetapkan atribut `top` sebagai `None`. Metode `isEmpty()` digunakan untuk memeriksa apakah Stack dalam keadaan kosong, dan mengembalikan nilai boolean berdasarkan kondisi `self.top`. Untuk menambahkan data ke dalam Stack, digunakan metode `push(data)` yang membuat objek `Node` baru dan

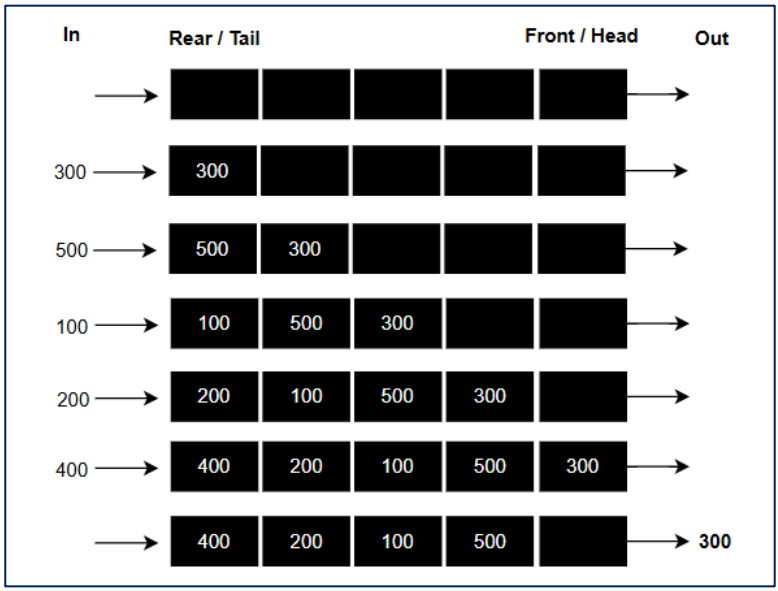
menempatkannya di atas Stack, dengan menghubungkan node baru ke node sebelumnya melalui referensi `next`.

Proses penghapusan elemen paling atas dilakukan dengan metode `pop()`, yang akan mengembalikan dan menghapus elemen paling atas jika Stack tidak kosong. Bila Stack kosong, maka akan ditampilkan pesan bahwa operasi `pop` tidak dapat dilakukan. Untuk melihat data pada posisi teratas tanpa menghapusnya, digunakan metode `peek()` yang mengembalikan nilai data dari node paling atas jika tersedia. Selain itu, metode `kosongkan()` digunakan untuk menghapus seluruh elemen Stack dengan mengatur ulang pointer `top` ke `None`, sedangkan metode `tampilkan()` digunakan untuk menelusuri dan mencetak semua elemen dalam Stack dari atas ke bawah.

Bagian akhir dari kode merupakan blok eksekusi utama (`main`) yang mendemonstrasikan penggunaan Stack. Lima elemen (300, 500, 100, 200, dan 400) dimasukkan ke dalam Stack, kemudian isi Stack ditampilkan. Selanjutnya, dilakukan operasi `peek()` dan `pop()` untuk melihat serta menghapus elemen teratas. Isi Stack kembali ditampilkan setelah penghapusan elemen, dan akhirnya dilakukan pengosongan Stack diikuti dengan proses verifikasi bahwa Stack telah kosong. Keseluruhan program ini memberikan ilustrasi yang sistematis dan efisien mengenai pengelolaan Stack berbasis Linked List, serta dapat dijadikan sebagai referensi implementatif dalam pengembangan aplikasi yang memerlukan mekanisme tumpukan data.

## 7.3 Queue

Queue atau antrean adalah struktur data untuk menyimpan berbagai elemen data. Queue menyusun elemen-elemen data secara linier. Prinsip dasar dari struktur data ini adalah "*First In, First Out*" (FIFO). Jadi elemen data yang pertama dimasukkan ke dalam antrean akan menjadi elemen pertama juga yang dikeluarkan.




### 7.3.1 Operasi Queue

Untuk mengakses data dari dan ke dalam struktur data Stack menggunakan operasi atau fungsi-fungsi seperti berikut:

1. `enqueue(item)`

   Fungsi ini digunakan untuk memasukkan elemen **item** pada akhir queue yaitu pada posisi rear atau tail.

2. `dequeue()`

   Fungsi ini digunakan untuk mengambil elemen pertama dari queue yaitu pada posisi front atau head.

3. `firstEl()`

   Fungsi ini digunakan untuk membaca elemen pertama dari queue tanpa menghapus datanya.

4. `isEmpty()`

   Fungsi ini digunakan untuk memeriksa apakah Queue dalam kondisi kosong atau tidak. Fungsi `isEmpty()` akan

mengembalikan hasil yaitu `true` jika Queue dalam kondisi kosong dan `false` jika Queue tidak kosong.

5. `clear()`

   Fungsi ini digunakan untuk menghapus atau membersihkan queue

### 7.3.2 Jenis-jenis Queue

Jenis struktur data Queue ini diklasifikasikan berdasarkan cara implementasi dan penggunaannya. Berdasarkan implementasinya dibagi menjadi dua yaitu:

1. **Linear atau Simple Queue**

   Elemen-elemen data disusun dalam barisan linear dan penambahan serta penghapusan elemen hanya terjadi pada dua ujung barisan tersebut. Contoh queue yang menggunakan array statis, ketika elemen di-dequeue dari awal, tempatnya akan tetap kosong tetapi tidak bisa digunakan lagi. Jika ada tempat kosong di depan tetapi rear sudah di akhir array, akan tetap dianggap penuh meskipun ada ruang kosong. Misalnya sebuah queue array statis dengan `maxSize = 5`

   ```
   Front : [300, 500, 100, 200, 400] Rear (penuh)
   Dequeue() : 300
   Setelah melakukan operasi dequeue () yaitu data 300
   dikeluarkan maka hasilnya adalah seperti berikut:
   Front : [-, 500, 100, 200, 400]  Rear (tetap
   diakhir)
   ```

   Jika ingin menambahkan elemen baru, akan dianggap penuh meskipun masih ada ruang kosong di depan.

2. **Circular Queue**

   Mirip dengan jenis linear, tetapi ujung-ujung barisan terhubung satu sama lain, menciptakan struktur antrean yang berputar. Circular Queue mengatasi masalah pada simple queue dengan cara menggunakan indeks secara melingkar. Jika rear berada di indeks terakhir (`maxSize-1`), maka elemen baru akan dimasukkan ke indeks 0 jika kosong. Circular queue dapat mengoptimalkan

penggunaan memori, karena tidakada ruang kosong yang tidak bisa digunakan. Representasi Circular Queue dengan Array, misalkan ukuran Circular Queue dengan maxSize = 5. Berikut adalah langkah-langkahnya:

**Langkah 1 – Inisialisasi**

```
Index:  [0] [1] [2] [3] [4]
Queue:  [-] [-] [-] [-] [-]
Front = -1, Rear = -1  (Queue Kosong)
```

**Langkah 2 - Enqueue 300, 500, 100**

```
Enqueue(300), Enqueue(500), Enqueue(100)
Index:  [0]  [1]  [2]  [3]  [4]
Queue:  [300] [500] [100] [-]  [-]
Front = 0, Rear = 2
```

**Langkah 3 - Dequeue () - 300 keluar dari Quueue**

```
Dequeue()
Index:  [0]  [1]  [2]  [3]  [4]
Queue:  [-]  [500] [100] [-]  [-]
Front = 1, Rear = 2
```

**Langkah 4 - Enqueue 200, 400, 700 (menggunakan ruang kosong)**

```
Enqueue(200), Enqueue(400), Enqueue(700)
Index:  [0]  [1]  [2]  [3]  [4]
Queue:  [700] [500] [100] [200] [400]
Front = 1, Rear = 0 (Circular)
```

Jenis queue berdasarkan penggunaan juga dibagi menjadi dua yaitu:

1. **Priority Queue**

   Setiap elemen memiliki tingkat prioritas tertentu. Elemen dengan prioritas tertinggi akan diproses lebih dulu, terlepas dari urutan masuknya. Jadi elemen tidak hanya memiliki data, tetapi juga memiliki nilai prioritas. Contoh penggunaan adalah:

   - Sistem antrian di rumah sakit (pasien gawat darurat didahulukan).
   - Penjadwalan proses pada sistem operasi (proses penting mendapatkan layanan CPU terlebih dadulu).
   - Pengaturan bandwidth pada jaringan (paket prioritas tinggi dikirim lebih dulu).

2. **Double-ended Queue (Dequeue)**

   Elemen memungkinkan untuk ditambahkan atau dihapus dari kedua ujung antrean. Jadi dapat melakukan operasi `enqueue()` dan `dequeue()` dari dua arah yaitu Front dan Rear. Contoh penggunaan adalah implementasi undo atau redo pada aplikasi,

### 7.3.3 Implementasi Queue

Struktur data Queue dapat di implementasikan dengan dua cara yaitu *array based queue* dan *linked-list based queue*.

#### Array-Based Queue

Berikut adalah implementasi Array-Based Queue yang ditulis menggunakan bahasa pemrograman Python.

```
1.  class Queue:
2.      def __init__(self):
3.          self.items = []
4.
5.      def isEmpty(self):
6.          return len(self.items) == 0
7.
8.      def enqueue(self, item):
9.          #Menambah data di akhir
10.         self.items.append(item)
```

```
11.
12.     def dequeue(self):
13.         if not self.isEmpty():
14.             # Mengeluarkan data dari depan
15.             return self.items.pop(0)
16.         else:
17.             print("Queue kosong!")
18.             return None
19.
20.     def peek(self):
21.         if not self.isEmpty():
22.             return self.items[0]
23.         else:
24.             return None
25.
26.     def size(self):
27.         return len(self.items)
28.
29.     def tampilkan(self):
30.         print("Isi Queue:", self.items)
31.
32. # Pembuatan objek Queue
33. objekQueue = Queue()
34. objekQueue.enqueue(300)
35. objekQueue.enqueue(500)
36. objekQueue.enqueue(100)
37. objekQueue.enqueue(200)
38. objekQueue.enqueue(400)
39. objekQueue.tampilkan()
40. print("Dequeue:", objekQueue.dequeue())
41. objekQueue.tampilkan()
```

**Output:**

```
Isi Queue: [300, 500, 100, 200, 400]
Dequeue: 300
Isi Queue: [500, 100, 200, 400]
```

**Penjelasan Kode:**

Program Python di atas merupakan implementasi dari struktur data **Queue (antrian)** menggunakan list sebagai media penyimpanan data. Kelas `Queue` dimulai pada baris 1 dan diinisialisasi melalui konstruktor `__init__` pada baris 2–3 dengan `self.items = []`, yang berarti

antrian dimulai dalam keadaan kosong. Untuk memeriksa apakah antrian kosong, digunakan metode `isEmpty` (baris 5–6), yang akan mengembalikan `True` jika panjang list `self.items` sama dengan nol.

Penambahan data ke dalam antrian dilakukan dengan metode `enqueue` pada baris 8–10. Pada bagian ini, `self.items.append(item)` digunakan untuk menambahkan elemen ke akhir antrian, sesuai dengan prinsip **FIFO (First In First Out)**. Sebaliknya, penghapusan elemen dari antrian dilakukan dengan metode `dequeue` (baris 12–18), yang akan mengeluarkan elemen pertama dari antrian menggunakan `self.items.pop(0)`. Namun, jika antrian kosong, maka akan ditampilkan pesan "Queue kosong!" dan metode akan mengembalikan `None`.

Untuk melihat elemen pertama dalam antrian tanpa menghapusnya, metode `peek` digunakan (baris 20–24), dengan mengembalikan `self.items[0]` jika antrian tidak kosong. Metode `size` (baris 26–27) mengembalikan jumlah elemen dalam antrian dengan `len(self.items)`. Sementara itu, metode `tampilkan` pada baris 29–30 menampilkan seluruh isi antrian dalam urutan yang sesuai.

Di bagian akhir program, yaitu pada baris 32–41, dilakukan demonstrasi penggunaan kelas `Queue`. Sebuah objek `objekQueue` dibuat, kemudian lima data (300, 500, 100, 200, 400) dimasukkan ke dalam antrian menggunakan `enqueue`. Setelah itu, antrian ditampilkan dengan `tampilkan`, lalu satu elemen dikeluarkan dari depan dengan `dequeue`, dan antrian ditampilkan kembali untuk menunjukkan perubahan. Implementasi ini mencerminkan prinsip dasar queue dan dapat digunakan dalam berbagai aplikasi pemrograman seperti penjadwalan tugas atau simulasi antrian.

### Linked List Based Queue

Linked-List Based Queue adalah struktur data queue yang diimplementasikan menggunakan linked list. Artinya, elemen-elemen queue disimpan dalam node-node yang saling terhubung. Berikut

adalah implementasi Linked-List-Based Queue yang ditulis menggunakan bahasa pemrograman Python.

```
1.  class Node:
2.      def __init__(self, data):
3.          self.data = data
4.          self.next = None
5.
6.  class LinkedQueue:
7.      def __init__(self):
8.          self.front = None
9.          self.rear = None
10.
11.     def isEmpty(self):
12.         return self.front is None
13.
14.     def enqueue(self, item):
15.         nodeBaru = Node(item)
16.         if self.isEmpty():
17.             self.front = self.rear = nodeBaru
18.         else:
19.             self.rear.next = nodeBaru
20.             self.rear = nodeBaru
21.
22.     def dequeue(self):
23.         if self.isEmpty():
24.             print("Queue kosong!")
25.             return None
26.         data = self.front.data
27.         self.front = self.front.next
28.         # Jika  dequeue kosong
29.         if self.front is None:
30.             self.rear = None
31.         return data
32.
33.     def peek(self):
34.         if self.isEmpty():
35.             return None
36.         return self.front.data
37.
38.     def tampilkan(self):
39.         current = self.front
40.         print("Isi Queue:", end=" ")
41.         while current:
42.             print(current.data, end=" -> ")
43.             current = current.next
```

```
44.         print("None")
45.
46.
47. # Pembuatan Objek Linked List
48. objekLQ = LinkedQueue()
49. #input data ke Queue
50. objekLQ.enqueue(300)
51. objekLQ.enqueue(500)
52. objekLQ.enqueue(100)
53. objekLQ.enqueue(200)
54. objekLQ.enqueue(400)
55. objekLQ.tampilkan()
56. print("Dequeue:", objekLQ.dequeue())
57. objekLQ.tampilkan()
```

**Output:**

```
Isi Queue: 300 -> 500 -> 100 -> 200 -> 400 -> None
Dequeue: 300
Isi Queue: 500 -> 100 -> 200 -> 400 -> None
```

**Penjelasan Kode:**

Program ini merupakan implementasi struktur data Queue (antrian) menggunakan Linked List dalam bahasa Python. Struktur dasar node didefinisikan pada baris 1–4 melalui kelas `Node`, yang memiliki atribut `data` untuk menyimpan nilai dan `next` untuk menunjuk ke node berikutnya dalam antrian.

Kelas `LinkedQueue` dimulai pada baris 6 dan bertanggung jawab untuk mengelola operasi-operasi pada queue. Konstruktor `__init__` (baris 7–9) menginisialisasi dua pointer: `front` dan `rear`, yang masing-masing menunjuk ke elemen pertama dan terakhir dalam antrian. Metode `isEmpty` pada baris 11–12 digunakan untuk memeriksa apakah antrian kosong dengan mengembalikan `True` jika `front` adalah `None`.

Operasi untuk menambahkan elemen baru ke antrian didefinisikan dalam metode `enqueue` (baris 14–20). Sebuah objek `Node` baru dibuat pada baris 15 dan dimasukkan ke antrian. Jika antrian kosong (baris 16), maka `front` dan `rear` keduanya akan menunjuk ke node baru tersebut. Namun, jika tidak kosong, node baru akan ditambahkan di belakang

antrian dengan menghubungkannya melalui `rear.next`, lalu `rear` diperbarui (baris 19–20).

Proses menghapus elemen dilakukan melalui metode `dequeue` (baris 22–31). Jika antrian kosong (baris 23–25), akan dicetak pesan "Queue kosong!" dan mengembalikan `None`. Jika tidak kosong, data pada node `front` diambil (baris 26), lalu pointer `front` dipindahkan ke node berikutnya (baris 27). Jika setelah penghapusan tidak ada lagi node (antrian kosong), maka `rear` juga diset menjadi `None` (baris 29–30).

Metode `peek` (baris 33–36) memungkinkan pengguna untuk melihat data di elemen paling depan tanpa menghapusnya. Jika antrian kosong, maka akan mengembalikan `None`. Sedangkan untuk melihat seluruh isi antrian, digunakan metode `tampilkan` pada baris 38–44, yang akan mencetak data dari depan ke belakang hingga node terakhir (ditutup dengan `None`).

Bagian implementasi atau demonstrasi dari penggunaan antrian dimulai pada baris 47. Objek `LinkedQueue` dibuat dengan nama `objekLQ` (baris 48). Kemudian, lima elemen (300, 500, 100, 200, 400) dimasukkan ke dalam antrian secara berurutan menggunakan `enqueue` (baris 50–54). Setelah itu, isi antrian ditampilkan (baris 55), sebuah elemen dihapus dari depan antrian menggunakan `dequeue` (baris 56), dan isi antrian kembali ditampilkan untuk memperlihatkan hasil perubahan (baris 57).

Secara keseluruhan, kode ini memberikan gambaran utuh mengenai operasi dasar queue berbasis linked list, yaitu enqueue, dequeue, peek, dan traversal, dengan efisiensi waktu yang optimal untuk setiap operasi karena tidak memerlukan pemindahan elemen seperti pada list biasa.

# Bab 8
# Linked List dan Operasinya

*"Terhubung satu sama lain, node membentuk aliran data yang fleksibel dan efisien."* — **Niklaus Wirth**

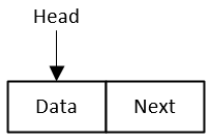
## 8.1 Single Linked List

Single Linked List (SLL) merupakan bentuk dasar dari struktur data linked list, di mana setiap elemen saling terhubung secara sekuensial melalui pointer tunggal. Berbeda dengan array yang memiliki panjang tetap dan memori kontigu, SLL bersifat dinamis dan efisien dalam penggunaan memori. Struktur ini banyak digunakan dalam pengembangan sistem yang memerlukan alokasi dan dealokasi data secara fleksibel, seperti dalam manajemen memori, penjadwalan proses, atau implementasi tumpukan (*stack*) dan antrian (*queue*) berbasis pointer (Cormen et al., 2009).

### 8.1.1 Struktur Node dan Pointer

Dalam implementasinya, setiap node pada single linked list memiliki dua komponen utama, yaitu data dan pointer ke node berikutnya. Node pertama disebut sebagai **head**, dan pointer terakhir pada list akan menunjuk ke `null` atau `None`, yang menandakan akhir dari list.

Struktur umum node dapat direpresentasikan dalam bentuk berikut:




Gambar 8.1: Representasi Node

Tabel 8.1: Representasi Node

| Field | Deskripsi |
|---|---|
| `data` | Menyimpan nilai atau informasi |
| `next` | Menunjuk ke node berikutnya |

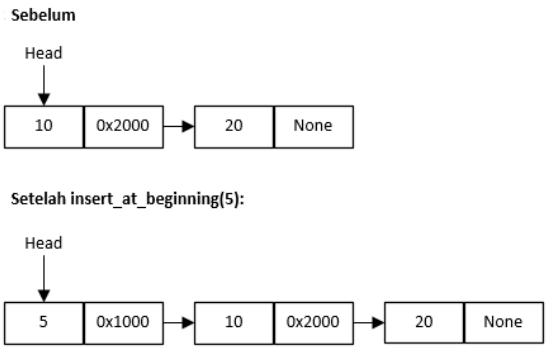
Struktur ini memungkinkan traversal satu arah dari depan ke belakang. Ketiadaan pointer ke node sebelumnya membuat traversal hanya dapat dilakukan secara maju. Konsep ini sederhana namun sangat efisien dalam aplikasi yang tidak memerlukan navigasi dua arah (Knuth, 1997).

### 8.1.2 Operasi Dasar: Insert, Delete, Traverse

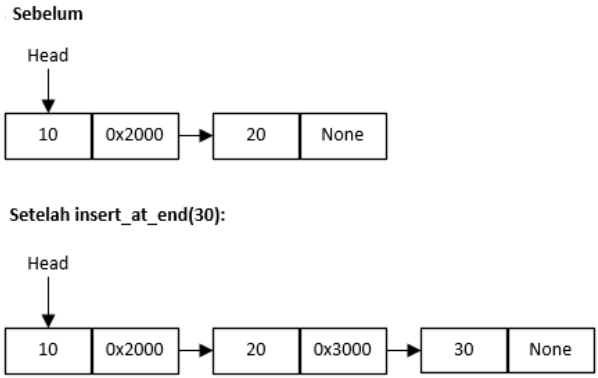
Operasi-operasi dasar dalam single linked list meliputi penyisipan (*insert*), penghapusan (*delete*), dan penelusuran (*traverse*). Ketiga operasi ini merupakan dasar dari manipulasi data dalam struktur linked list.

#### Insert (Penyisipan Node)

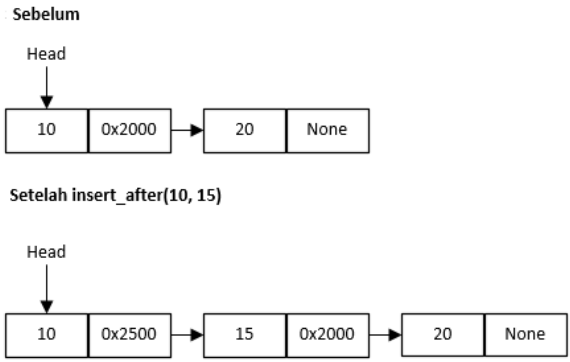
Penyisipan node baru dapat dilakukan di tiga posisi: di awal list (insert at beginning), di akhir list (insert at end), atau setelah node tertentu (insert after node).




Gambar 8.2: Insert di Awal (I*nsert at Beginning*)




Gambar 8.3: Insert di Akhir (*Insert at End*)



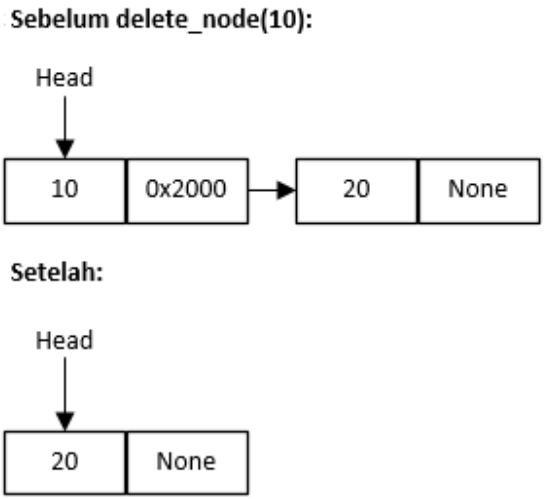

Gambar 8.4: Insert Setelah Node Tertentu (*Insert After Node*)

Proses penyisipan dilakukan dengan membuat node baru, kemudian mengatur pointer agar node tersebut terhubung secara logis ke dalam list yang sudah ada. Penyisipan ini bersifat efisien karena tidak memerlukan pergeseran elemen seperti pada array (Weiss, 2013).

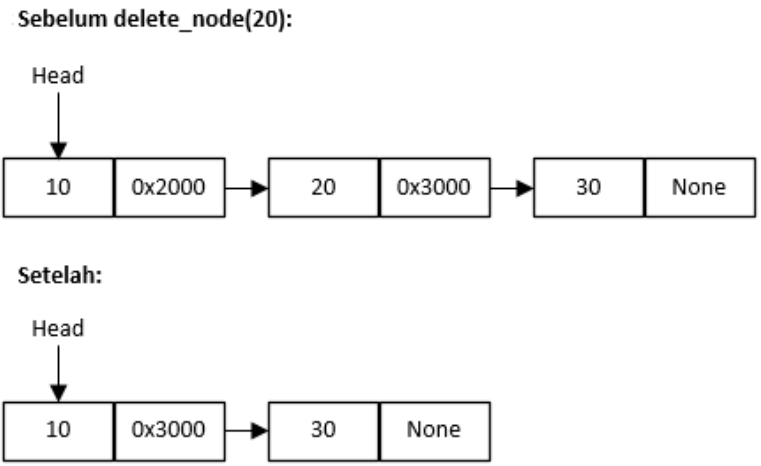
## Delete (Penghapusan Node)

Penghapusan node dilakukan berdasarkan nilai yang ingin dihapus. Jika nilai ditemukan di `head`, maka `head` akan dialihkan ke node berikutnya. Jika berada di tengah atau akhir, pointer node sebelumnya

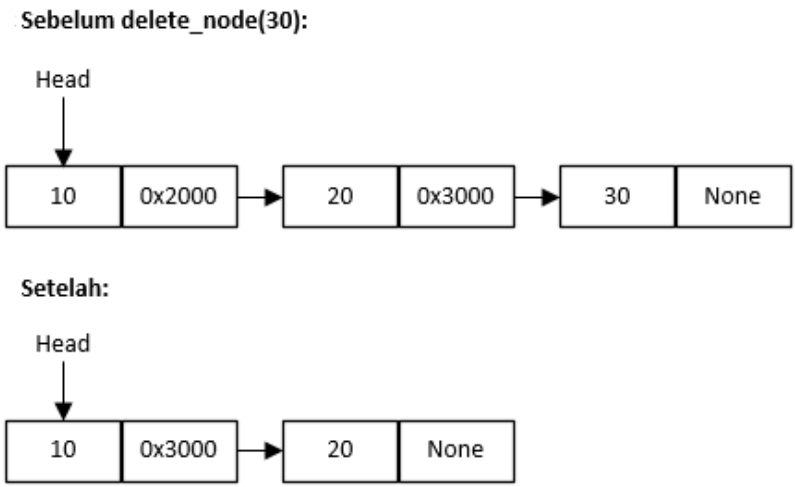
diatur untuk melewati node yang dihapus. Penting untuk menjaga integritas referensi agar tidak terjadi memory leak.




Gambar 8.5: Delete Node di Awal (Head)



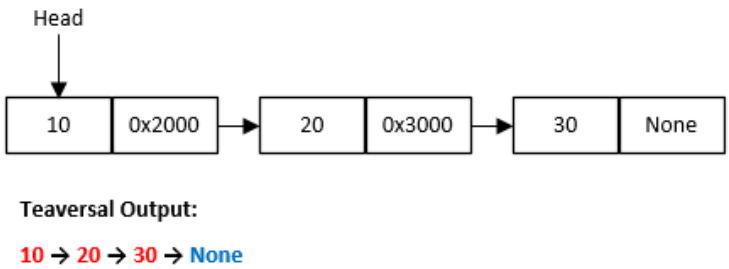

Gambar 8.6: Delete Node di Tengah




Gambar 8.7: Delete Node di Akhir

## Traverse (Penelusuran List)

Traversal merupakan proses menelusuri linked list dari `head` hingga `null`, umumnya untuk menampilkan semua elemen atau mencari nilai tertentu. Traversal dilakukan secara linear dengan kompleksitas waktu O(n), di mana n adalah jumlah node dalam list. Traversal sangat penting dalam verifikasi dan manipulasi data (Shaffer, 2011).




Gambar 8.8: Traversal

### Implementasi dalam Python:

Berikut ini adalah implementasi single linked list menggunakan bahasa pemrograman Python, yang mencerminkan prinsip dasar struktur data dinamis:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class SingleLinkedList:
    def __init__(self):
        self.head = None

    def insert_at_beginning(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    def insert_at_end(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node

    def delete_node(self, key):
        current = self.head
        if current and current.data == key:
            self.head = current.next
            return
        prev = None
        while current and current.data != key:
            prev = current
            current = current.next
        if current is None:
            return
```

```
        prev.next = current.next

    def traverse(self):
        current = self.head
        while current:
            print(current.data, end=" → ")
            current = current.next
        print("None")
```

### Penjelsan Kode:

Kode di atas mengimplementasikan struktur data Single Linked List menggunakan bahasa Python. Struktur ini terdiri dari dua kelas utama, yaitu `Node` dan `SingleLinkedList`. Kelas `Node` merepresentasikan elemen dasar dari linked list, yang memiliki dua atribut: `data` untuk menyimpan nilai dan `next` untuk menunjuk ke node berikutnya dalam daftar. Setiap node hanya mengetahui siapa penerusnya, tidak memiliki referensi ke pendahulunya.

Kelas `SingleLinkedList` berfungsi sebagai pengelola daftar. Atribut `self.head` menyimpan referensi ke node pertama dalam list. Metode `insert_at_beginning(data)` digunakan untuk menyisipkan elemen di awal list. Metode ini membuat node baru dan menunjukannya ke node yang sebelumnya berada di posisi head, kemudian memperbarui head ke node baru tersebut. Metode `insert_at_end(data)` digunakan untuk menambahkan elemen di akhir list. Jika list kosong, maka node baru menjadi head. Jika tidak, pointer `current` bergerak dari head hingga mencapai node terakhir (yang `next`-nya bernilai `None`), lalu `next` dari node terakhir tersebut diarahkan ke node baru.

Metode `delete_node(key)` berfungsi untuk menghapus node berdasarkan nilai data. Jika node yang ingin dihapus adalah head, maka head langsung diarahkan ke node berikutnya. Jika node yang ingin dihapus berada di tengah atau akhir list, maka pointer `prev` digunakan untuk menyimpan referensi ke node sebelumnya, dan penghapusan dilakukan dengan memutus sambungan node yang bersangkutan dari list. Sementara itu, metode `traverse()` digunakan untuk menelusuri

list dari head hingga akhir, mencetak setiap nilai data yang ditemukan, dan memvisualisasikan hubungan antar node menggunakan panah →. Traversal berakhir saat pointer `current` mencapai `None`, menandai ujung dari linked list.

Secara keseluruhan, kode ini menunjukkan operasi dasar pada single linked list, yaitu penyisipan di awal dan akhir, penghapusan node berdasarkan nilai, dan penelusuran seluruh elemen dalam list, yang merupakan fondasi penting dalam pemahaman struktur data dinamis satu arah.

**Contoh Penggunaan:**

```
sll = SingleLinkedList()
sll.insert_at_end(10)
sll.insert_at_end(20)
sll.insert_at_beginning(5)
sll.insert_at_end(30)

print("Isi linked list:")
sll.traverse()

sll.delete_node(20)
print("Setelah menghapus 20:")
sll.traverse()
```

**Output:**

```
Isi linked list:
5 → 10 → 20 → 30 → None
Setelah menghapus 20:
5 → 10 → 30 → None
```

Single Linked List merupakan struktur data fundamental yang banyak digunakan dalam berbagai aplikasi informatika. Keunggulan utamanya terletak pada efisiensi penyisipan dan penghapusan elemen tanpa perlu pergeseran data. Meskipun traversal hanya satu arah dan akses elemen acak tidak efisien, fleksibilitas dan kesederhanaan implementasinya

menjadikan SLL sangat relevan dalam berbagai konteks pemrograman dan sistem (Goodrich et al., 2021).

## 8.2 Double Linked List

Double Linked List (DLL) atau *doubly linked list* merupakan bentuk lanjutan dari single linked list yang memiliki dua pointer di setiap nodenya. Jika pada single linked list hanya terdapat pointer ke node berikutnya, maka pada double linked list terdapat dua pointer: satu menunjuk ke node sebelumnya (`prev`) dan satu lagi ke node berikutnya (`next`). Struktur ini sangat berguna dalam aplikasi yang membutuhkan akses dua arah terhadap elemen-elemen list, seperti pada fitur undo-redo dalam aplikasi pengolah teks atau sistem navigasi berbasis data linier (Goodrich et al., 2021).

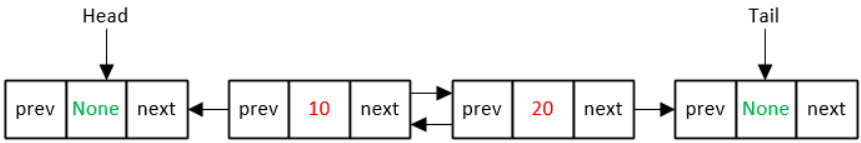
### 8.2.1 Struktur Ganda (prev, next)

Node dalam double linked list memiliki tiga bagian utama:

Tabel 8.2: Bagian Utama Node Linked List

| Atribut | Deskripsi |
|---|---|
| `data` | Menyimpan nilai dari node |
| `prev` | Menunjuk ke node sebelumnya |
| `next` | Menunjuk ke node berikutnya |

Struktur ini membuat traversal tidak hanya terbatas dari `head` ke `tail`, melainkan juga memungkinkan dari `tail` ke `head`. Node pertama memiliki `prev = None`, dan node terakhir memiliki `next = None`.




Gambar 8.9: Struktur Double Linked List

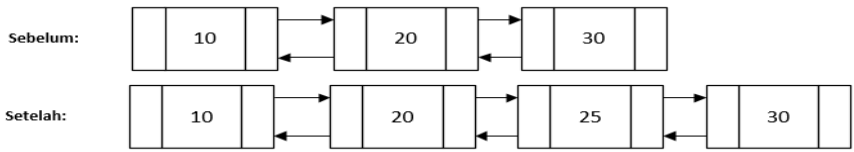
Struktur dua arah ini memungkinkan efisiensi tinggi dalam penyisipan dan penghapusan node di tengah-tengah list, karena tidak perlu melakukan traversal ulang dari awal list. Operasi seperti reverse traversal pun menjadi lebih efisien dibandingkan dengan single linked list (Shaffer, 2011).

### 8.2.2 Operasi Insert/Delete Dua Arah

Double linked list mendukung operasi dasar sebagai berikut:

**Insert**

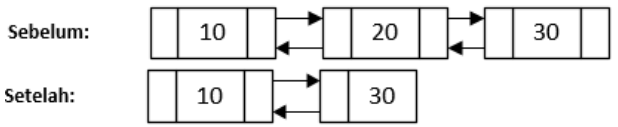
- **Di awal**: pointer `prev` node pertama diatur menunjuk ke node baru.
- **Di akhir**: pointer `next` node terakhir diatur menunjuk ke node baru, dan `prev` node baru menunjuk ke node sebelumnya.
- **Setelah node tertentu**: node baru dihubungkan di antara dua node eksisting.




Gambar 8.10: Insert setelah node 20

### Delete

- Node dihapus dengan cara mengatur pointer `next` node sebelumnya ke node setelahnya, dan pointer `prev` node setelahnya ke node sebelumnya.
- Jika node yang dihapus adalah `head`, maka pointer `head` perlu diperbarui.




Gambar 8.11: Delete node 20

Keunggulan double linked list terlihat jelas dalam operasi delete dan insert karena operasi dapat dilakukan dari dua arah, dan tidak perlu traversal dari awal jika posisi diketahui atau ditelusuri dari `tail` (Weiss, 2013).

### Implementasi Double Linked List dalam Python:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None

class DoubleLinkedList:
    def __init__(self):
        self.head = None

    def insert_at_beginning(self, data):
        new_node = Node(data)
        new_node.next = self.head
        if self.head:
            self.head.prev = new_node
        self.head = new_node
```

```
    def insert_at_end(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = new_node
        new_node.prev = temp

    def delete_node(self, key):
        current = self.head
        while current:
            if current.data == key:
                if current.prev:
                    current.prev.next = current.next
                else:
                    self.head = current.next
                if current.next:
                    current.next.prev = current.prev
                return
            current = current.next

    def traverse_forward(self):
        current = self.head
        while current:
            print(current.data, end=" ⇄ ")
            current = current.next
        print("None")

    def traverse_backward(self):
        current = self.head
        if not current:
            return
        # Arahkan ke node terakhir
        while current.next:
            current = current.next
        # Traversal mundur
        while current:
            print(current.data, end=" ⇄ ")
```

```
            current = current.prev
        print("None")
```

**Penjelasan Kode:**

Kode di atas merupakan implementasi dari struktur data Double Linked List menggunakan bahasa Python. Struktur ini memungkinkan traversal dua arah karena setiap node menyimpan dua referensi, yaitu `prev` (ke node sebelumnya) dan `next` (ke node berikutnya). Kelas `Node` mendefinisikan elemen dasar dalam double linked list dengan tiga atribut: `data` untuk menyimpan nilai, `prev` untuk menunjuk ke node sebelumnya, dan `next` untuk menunjuk ke node berikutnya. Dengan dua pointer tersebut, double linked list dapat dilewati dari kedua arah, berbeda dengan single linked list yang hanya dapat dilalui secara satu arah.

Kelas `DoubleLinkedList` bertugas mengelola struktur list. Atribut `self.head` menunjuk ke node pertama dalam list. Metode `insert_at_beginning(data)` digunakan untuk menyisipkan elemen di awal list. Metode ini membuat node baru, mengarahkan `next` dari node baru ke head saat ini, dan jika head tidak kosong, maka `prev` dari node head akan diarahkan ke node baru. Kemudian, head diperbarui ke node baru tersebut. Metode `insert_at_end(data)` digunakan untuk menyisipkan elemen di akhir list. Jika list kosong, maka node baru akan menjadi head. Jika tidak, pointer `temp` digunakan untuk menelusuri hingga node terakhir, lalu node baru disisipkan di akhir dan pointer `prev` milik node baru diatur agar menunjuk ke node sebelumnya.

Untuk menghapus elemen, digunakan metode `delete_node(key)` yang akan menelusuri node satu per satu hingga menemukan node dengan nilai `data` yang sesuai dengan `key`. Jika node ditemukan, maka relasi antar node sebelumnya dan sesudahnya akan diperbarui: jika node memiliki `prev`, maka `next` dari node sebelumnya akan diarahkan ke `next` dari node saat ini; jika node adalah head, maka head diperbarui ke node berikutnya. Jika node tersebut memiliki `next`, maka `prev` dari

node setelahnya akan diarahkan ke node sebelumnya. Dengan demikian, node yang dihapus sepenuhnya terputus dari struktur list.

Metode `traverse_forward()` digunakan untuk menampilkan seluruh isi list dari head hingga node terakhir. Nilai-nilai node dicetak menggunakan simbol panah dua arah ⇄ untuk merepresentasikan keterhubungan dua arah antar node. Sebaliknya, metode `traverse_backward()` digunakan untuk menampilkan isi list dari belakang ke depan. Untuk melakukannya, traversal dimulai dari head hingga node terakhir, kemudian dicetak mundur menggunakan pointer `prev`.

Secara keseluruhan, implementasi ini menunjukkan keunggulan double linked list dalam fleksibilitas traversal dan kemudahan penghapusan atau penyisipan node baik di awal, tengah, maupun akhir list, karena pointer dapat diatur dari kedua arah. Struktur ini sangat bermanfaat dalam pengembangan sistem yang memerlukan navigasi maju dan mundur, seperti penelusuran riwayat (*undo/redo*), navigasi halaman, atau sistem cache.

**Contoh Penggunaan:**

```
dll = DoubleLinkedList()
dll.insert_at_end(10)
dll.insert_at_end(20)
dll.insert_at_end(30)

print("Traversal Maju:")
dll.traverse_forward()

print("Traversal Mundur:")
dll.traverse_backward()

dll.delete_node(20)

print("Setelah Hapus 20:")
dll.traverse_forward()
```

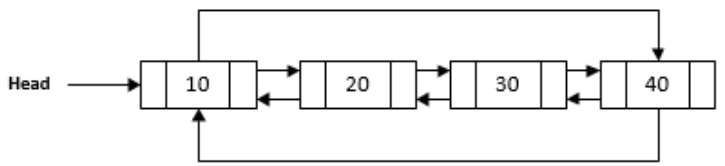
**Output**

```
Traversal Maju:
10 ⇄ 20 ⇄ 30 ⇄ None
Traversal Mundur:
30 ⇄ 20 ⇄ 10 ⇄ None
Setelah Hapus 20:
10 ⇄ 30 ⇄ None
```

Double Linked List memberikan fleksibilitas yang jauh lebih besar dibandingkan dengan single linked list. Dengan dua pointer `prev` dan `next`, operasi insert dan delete dapat dilakukan lebih efisien, khususnya dalam struktur data dinamis. Meskipun memori yang dibutuhkan lebih besar karena dua pointer per node, keuntungannya sebanding dengan kemampuannya untuk traversal dua arah dan pengelolaan list yang kompleks (Cormen et al., 2009; Shaffer, 2011).

## 8.3 Circular Linked List

Circular Linked List adalah salah satu bentuk lanjutan dari struktur linked list yang bersifat melingkar. Dalam struktur ini, node terakhir tidak menunjuk ke `None` seperti pada single linked list, tetapi kembali menunjuk ke node pertama (head). Sifat sirkular ini memungkinkan traversal berulang tanpa batas, menjadikannya cocok untuk berbagai aplikasi yang memerlukan siklus data terus-menerus.




Gambar 8.12: Representasi Circular Linked List dengan 4 Node

Circular linked list banyak digunakan dalam konteks seperti penjadwalan sistem operasi (*round-robin scheduling*), buffer sirkular,

dan navigasi menu yang berulang. Karena tidak ada titik akhir eksplisit, pengelolaan dan pemrosesan data dalam circular linked list memerlukan perhatian khusus terhadap kondisi batas.

### 8.3.1 Struktur Node dan Circular Linked List

Struktur dasar circular linked list terdiri dari kumpulan node yang masing-masing menyimpan nilai data dan referensi ke node berikutnya. Perbedaan utamanya dengan single linked list adalah pada node terakhir, di mana pointer `next` tidak menunjuk ke `None`, melainkan ke `head`. Hal ini menciptakan struktur sirkular yang memungkinkan traversal tak berujung.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None

    def tambah_di_akhir(self, data):
        node_baru = Node(data)
        if self.head is None:
            self.head = node_baru
            node_baru.next = self.head
        else:
            temp = self.head
            while temp.next != self.head:
                temp = temp.next
            temp.next = node_baru
            node_baru.next = self.head

    def tampilkan(self):
        if self.head is None:
            print("List kosong.")
            return
        temp = self.head
        while True:
```

```
            print(temp.data, end=' -> ')
            temp = temp.next
            if temp == self.head:
                break
        print("(kembali ke head)")
```

### Penjelasan Kode

Kode di atas mengimplementasikan struktur data Circular Linked List dalam bahasa Python. Circular linked list adalah jenis linked list di mana node terakhir tidak menunjuk ke `None`, melainkan kembali ke node pertama (head), membentuk struktur sirkular. Hal ini memungkinkan traversal yang tidak terbatas karena tidak ada ujung eksplisit pada list.

Kelas `Node` digunakan untuk merepresentasikan elemen dasar dalam list, dengan dua atribut: `data` untuk menyimpan nilai dan `next` untuk menunjuk ke node berikutnya. Kelas `CircularLinkedList` bertindak sebagai pengelola list dan memiliki atribut `head` sebagai penunjuk ke node pertama. Metode `tambah_di_akhir(data)` digunakan untuk menambahkan elemen di akhir list. Jika list kosong, node baru menjadi `head` dan `next`-nya menunjuk ke dirinya sendiri. Jika list sudah memiliki elemen, pointer `temp` digunakan untuk menelusuri hingga node terakhir (yaitu node yang `next`-nya menunjuk kembali ke `head`). Setelah ditemukan, node baru disisipkan setelah node terakhir, dan `next` dari node baru diarahkan kembali ke `head` untuk menjaga sifat sirkular.

Metode `tampilkan()` digunakan untuk menampilkan semua elemen dalam circular linked list. Traversal dimulai dari `head` dan mencetak setiap nilai `data` yang ditemukan, diikuti panah `->`, untuk menunjukkan arah sambungan antar node. Proses berakhir ketika traversal kembali ke node `head`, yang menandakan bahwa seluruh elemen telah dilalui satu putaran penuh. Pesan "(kembali ke head)" dicetak untuk menunjukkan bahwa traversal telah menyentuh kembali node awal dan tidak berlanjut tanpa batas.

Struktur circular linked list seperti ini sangat bermanfaat dalam aplikasi yang memerlukan pemrosesan berulang atau rotasi tanpa akhir, seperti

dalam sistem penjadwalan CPU berbasis *round-robin*, buffer sirkular, atau menu navigasi yang dapat dilalui secara terus-menerus. Meskipun sederhana, implementasi circular linked list memerlukan pengelolaan pointer yang hati-hati agar tidak terjadi infinite loop atau kesalahan saat menyisipkan dan menelusuri data.

### Contoh Penggunaan Circular Linked List:

Untuk memahami implementasi circular linked list secara praktis, berikut ini adalah contoh penggunaan di mana beberapa elemen dimasukkan ke dalam list, kemudian ditampilkan. Traversal dimulai dari node `head` dan berakhir ketika traversal kembali ke node `head`.

```
cll = CircularLinkedList()
cll.tambah_di_akhir(10)
cll.tambah_di_akhir(20)
cll.tambah_di_akhir(30)
cll.tampilkan()
```

### Output:

```
10 -> 20 -> 30 -> (kembali ke head)
```

Contoh ini menunjukkan bahwa pointer dari node terakhir mengarah ke node pertama. Dengan demikian, list tidak memiliki ujung dan traversal akan berulang secara terus-menerus jika tidak dihentikan secara eksplisit.

## 8.3.2 Keunggulan Circular Linked List

Circular linked list menawarkan berbagai kelebihan dibandingkan bentuk linked list lainnya. Salah satunya adalah kemampuan untuk melakukan traversal dari node mana pun dan tetap mengunjungi seluruh elemen. Hal ini sangat berguna untuk implementasi sistem berulang seperti navigasi siklik, atau sistem penjadwalan bergiliran.

- Traversal dapat dimulai dari node mana pun dan kembali ke awal.
- Cocok untuk aplikasi yang bersifat siklik atau berulang.

- Efisien dalam sistem antrian berbasis rotasi (round-robin).

### 8.3.3 Kekurangan Circular Linked List

Meskipun memiliki keunggulan dalam efisiensi dan fleksibilitas, circular linked list juga memiliki sejumlah kekurangan yang harus diperhatikan. Kompleksitas dalam mendeteksi akhir traversal serta risiko loop tak terbatas jika pointer tidak dikelola dengan baik menjadi tantangan utama.

- Deteksi akhir traversal lebih kompleks karena tidak ada `None`.
- Penanganan penambahan dan penghapusan node lebih rumit.
- Berisiko menyebabkan infinite loop jika pointer rusak.

### 8.3.4 Perbandingan Circular dan Single Linked List

Untuk memahami perbedaan fungsional dan struktural antara circular linked list dan single linked list, berikut ini disajikan tabel perbandingan yang menguraikan perbedaan utama berdasarkan karakteristik struktural dan aplikatif.

Tabel 8.3: Perbandingan Circular dan Single Linked List

| Aspek | Single Linked List | Circular Linked List |
|---|---|---|
| Node terakhir | Menunjuk ke `None` | Menunjuk ke head |
| Awal traversal | Hanya dari head | Bisa dari node mana pun |
| Arah navigasi | Linear | Sirkular |
| Deteksi akhir | Mudah (`None`) | Perlu pengecekan terhadap head |
| Aplikasi umum | Stack, queue | Round-robin, buffer, navigasi |

Circular linked list merupakan alternatif struktur data yang menawarkan fleksibilitas tinggi untuk aplikasi-aplikasi yang menuntut pemrosesan berulang tanpa titik akhir eksplisit. Walaupun lebih kompleks dalam implementasi dibandingkan Single linked list, circular linked list sangat

ideal untuk skenario real-time dan interaktif. Pemahaman yang baik terhadap pengelolaan pointer dan batas traversal menjadi kunci sukses dalam memanfaatkan struktur ini secara optimal.

# Bab 9
# Teknik Pencarian Data

*"Mencari bukan soal keberuntungan, tapi strategi dan efisiensi."*

— **Jon Bentley**

## 9.1 Konsep Pencarian (*Searching*)

Pencarian, dalam konteks ilmu komputer, adalah proses fundamental untuk menemukan elemen atau informasi tertentu dalam suatu kumpulan data berdasarkan nilai atau kriteria yang telah ditentukan (Ramadhan, 2022). Tujuan utamanya adalah untuk mengidentifikasi keberadaan suatu entitas data yang memenuhi kondisi spesifik, atau untuk mengambilnya jika ditemukan (Nurhasanah et al., 2015).

Pencarian dapat diterapkan dalam berbagai konteks, mulai dari skenario yang paling sederhana hingga yang paling kompleks. Sebagai contoh, pencarian dapat dilakukan dalam daftar atau array untuk menemukan elemen tertentu dalam urutan data. Dalam sistem basis data, pencarian menjadi krusial untuk menemukan catatan atau entri spesifik berdasarkan kriteria yang diberikan Bahkan dalam struktur data yang lebih kompleks seperti pohon pencarian biner atau tabel hash, pencarian memainkan peran sentral. Di ranah yang lebih luas, seperti pada grafik atau jaringan, pencarian digunakan untuk menemukan jalur atau relasi antar node (Nurhasanah et al., 2015).

Efisiensi pencarian sangat bergantung pada metode yang digunakan dan karakteristik struktur data tempat data disimpan (Nurhasanah et al., 2015). Struktur data sendiri merupakan cara penyimpanan, pengorganisasian, dan pengelolaan data dalam komputer agar dapat diakses dan dimanfaatkan secara efisien (Nugroho, 2019). Pemilihan struktur data yang tepat dapat mempermudah akses data, mengelola data secara efektif, serta memudahkan penambahan dan penghapusan

data (Hafiz et al., 2024). Misalnya, array memungkinkan akses cepat berdasarkan indeks, sementara linked list unggul dalam penambahan dan penghapusan elemen (Sofianti et al., 2025)(Sofianti et al., 2025). Tanpa struktur data yang terorganisir, pencarian data akan menjadi proses yang sangat lambat dan tidak efisien, terutama untuk kumpulan data yang besar (Hafiz et al., 2024). Oleh karena itu, hubungan antara struktur data dan algoritma pencarian bersifat simbiotik; struktur data yang baik menjadi prasyarat bagi algoritma pencarian yang efisien, dan algoritma pencarian yang tepat memaksimalkan potensi dari struktur data yang ada (Nurhasanah et al., 2015).

## 9.2 Algoritma Pencarian

Dalam disiplin ilmu komputer, terdapat beragam algoritma pencarian yang dirancang untuk skenario dan karakteristik data yang berbeda. Secara garis besar, metode pencarian dapat diklasifikasikan menjadi dua kategori utama: metode pencarian data tanpa penempatan data khusus dan metode pencarian data dengan penempatan data khusus (Nurhasanah et al., 2015).

### 9.2.1 Metode Pencarian Data Tanpa Penempatan Data

Metode ini tidak memerlukan pengorganisasian atau pengaturan data tertentu sebelum proses pencarian dimulai. Data dapat dicari langsung dalam urutan aslinya (Nurhasanah et al., 2015).

#### 9.2.1.1 Pencarian Linear (Linear Search / Sequential Search)

Pencarian linear adalah algoritma pencarian paling sederhana yang bekerja dengan memeriksa setiap elemen dalam daftar secara berurutan, satu per satu, hingga nilai target ditemukan atau seluruh daftar telah diperiksa (Ray et al., 2021). Jika elemen target ditemukan, algoritma mengembalikan indeks atau posisi elemen tersebut. Jika tidak, ia menandakan bahwa elemen tidak ada dalam kumpulan data (Singh & Singh, 2014).

Kelebihan utama dari pencarian linear adalah kesederhanaannya dalam implementasi dan kemampuannya untuk bekerja pada array yang tidak terurut (Singh & Singh, 2014). Ini menjadikannya pilihan yang cocok untuk kumpulan data yang kecil atau skenario di mana data tidak dapat diurutkan (. Namun, kelemahannya terletak pada efisiensinya. Dalam kasus terburuk, di mana elemen target berada di akhir daftar atau tidak ada sama sekali, algoritma harus memeriksa semua n elemen, menghasilkan kompleksitas waktu O(n). Ini membuatnya sangat tidak efisien untuk kumpulan data yang besar (Ray et al., 2021).

### 9.2.1.2 Pencarian Biner (*Binary Search*)

Pencarian biner adalah algoritma yang jauh lebih efisien dibandingkan pencarian linear, namun memiliki prasyarat penting: data harus sudah terurut, baik secara menaik (*ascending*) maupun menurun (*descending*) (Sari & Rosadi, 2017). Algoritma ini bekerja dengan membagi interval pencarian menjadi dua secara berulang kali. Proses dimulai dengan membandingkan nilai target dengan elemen tengah dari daftar. Jika elemen tengah adalah target, pencarian berhasil. Jika target lebih kecil, pencarian dilanjutkan di paruh kiri daftar; jika lebih besar, pencarian dilanjutkan di paruh kanan. Proses ini berlanjut hingga target ditemukan atau interval pencarian menjadi kosong (Nurhasanah et al., 2015).

Keunggulan utama pencarian biner adalah kecepatannya yang signifikan untuk kumpulan data besar. Dengan setiap perbandingan, ruang pencarian berkurang setengahnya, menghasilkan kompleksitas waktu O(log n) dalam kasus terburuk maupun rata-rata (Craig, 2019). Ini menjadikannya pilihan yang sangat efisien untuk mencari elemen dalam basis data besar atau array yang telah diurutkan. Namun, ketergantungannya pada data yang terurut adalah kekurangannya; jika data tidak terurut, overhead pengurutan awal mungkin diperlukan (Rumapea, 2017).

### 9.2.1.3 Pencarian Interpolasi (Interpolation Search)

Pencarian interpolasi dapat dianggap sebagai versi yang ditingkatkan dari pencarian biner, terutama efektif untuk array besar yang terurut dan

terdistribusi secara seragam. Berbeda dengan pencarian biner yang selalu memeriksa elemen tengah, pencarian interpolasi "menebak" posisi yang mungkin dari nilai target berdasarkan nilai kunci dan rentang ruang pencarian. Ini dilakukan menggunakan rumus interpolasi yang memperhitungkan nilai target relatif terhadap elemen batas bawah dan batas atas dari interval pencarian (Bitner & Reingold, 1975).

Jika nilai target lebih dekat ke batas atas, algoritma akan cenderung melompat ke posisi yang lebih dekat ke sana, dan sebaliknya (Nurhasanah et al., 2015). Efisiensi ini menghasilkan kompleksitas waktu rata-rata yang sangat baik, O(log log n), yang bahkan lebih cepat daripada pencarian biner untuk data yang terdistribusi seragam (Bitner & Reingold, 1975). Namun, jika data tidak terdistribusi secara seragam, kinerjanya dapat menurun drastis hingga O(n) dalam kasus terburuk. Seperti pencarian biner, pencarian interpolasi juga memerlukan data yang terurut (Nurhasanah et al., 2015).

#### 9.2.1.4 Pencarian Lompat (Jump Search)

Pencarian lompat, atau dikenal juga sebagai pencarian blok, adalah algoritma pencarian lain yang cocok untuk array yang sudah diurutkan. Alih-alih memeriksa setiap elemen secara berurutan seperti pencarian linear, atau membagi dua secara ketat seperti pencarian biner, pencarian lompat melompat maju dengan sejumlah langkah tetap. Setelah algoritma menemukan blok di mana elemen target mungkin berada (yaitu, elemen terakhir blok lebih besar dari target), ia kemudian melakukan pencarian linear tradisional dalam blok yang lebih kecil tersebut (Harabor & Koenig, 2021).

Algoritma ini menawarkan kompromi antara pencarian linear dan biner. Meskipun tidak secepat pencarian biner, ia jauh lebih baik daripada pencarian linear untuk dataset besar yang terurut. Kompleksitas waktu pencarian lompat adalah O($\sqrt{n}$). Kelebihannya meliputi implementasi yang relatif sederhana dan pengurangan jumlah perbandingan dibandingkan pencarian linear. Namun, seperti pencarian biner dan interpolasi, ia memerlukan data yang terurut (Harabor & Koenig, 2021)

### 9.2.2 Metode Pencarian Data dengan Penempatan Data

Metode ini melibatkan pengorganisasian data ke dalam struktur khusus sebelum pencarian dilakukan, dengan tujuan utama untuk meningkatkan efisiensi pencarian secara drastis (Nurhasanah et al., 2015).

#### Pencarian Berbasis Hash (*Hash-based Search*)

Pencarian berbasis hash menggunakan fungsi hash untuk memetakan kunci (nilai yang dicari) ke indeks atau alamat tertentu dalam struktur data yang disebut tabel hash. Konsep ini sangat kuat karena dalam kasus ideal, pencarian dapat dilakukan dalam waktu konstan, yaitu O(1). Python dict (*dictionary*) adalah contoh implementasi tabel hash bawaan yang sangat efisien (Wang, 2025).

Fungsi hash yang baik akan menghasilkan indeks unik untuk setiap kunci. Namun, tantangan utama dalam pencarian berbasis hash adalah kolisi (collision), di mana dua kunci berbeda menghasilkan indeks hash yang sama (Nurhasanah et al., 2015; Wang, 2025). Untuk mengatasi ini, berbagai strategi penanganan kolisi digunakan, seperti separate chaining (menyimpan beberapa elemen di indeks yang sama dalam bentuk daftar atau linked list) atau linear probing (mencari slot kosong berikutnya). Meskipun rata-rata waktu pencarian adalah O(1), dalam kasus terburuk (misalnya, semua kunci berkolisi ke indeks yang sama), kompleksitasnya bisa menurun menjadi O(n). Meskipun demikian, untuk sebagian besar aplikasi, pencarian berbasis hash menawarkan performa yang luar biasa cepat, menjadikannya pilihan yang sangat populer (Nurhasanah et al., 2015; Wang, 2025)

### 9.2.3 Perbandingan Algoritma Pencarian Utama

Tabel berikut merangkum karakteristik, kompleksitas waktu dan ruang, serta kelebihan dan kekurangan dari algoritma pencarian yang telah dibahas.

Tabel 9.1: Perbandingan Sederhana Algoritma Pencarian

| Algoritma | Syarat Data | Kecepatan (Rata-rata) | Kelebihan | Kekurangan |
|---|---|---|---|---|
| **Linear Search** | Tidak perlu urut | Lambat (O(n)) | Mudah digunakan, cocok untuk data acak | Tidak efisien untuk data besar |
| **Binary Search** | Harus terurut | Cepat (O(log n)) | Sangat cepat, efisien untuk data besar | Hanya untuk data terurut |
| **Interpolation** | Terurut & distribusi seragam | Sangat cepat (O(log log n)) | Lebih cepat dari binary jika data merata | Lambat jika data tidak merata |
| **Jump Search** | Terurut | Sedang (O(√n)) | Lebih cepat dari linear, mudah diimplementasi | Masih kalah cepat dibanding binary search |
| **Hash Search** | Kunci unik (idealnya) | Sangat cepat (O(1)) | Akses langsung, cocok untuk pencarian berdasarkan kunci | Bisa lambat jika banyak kolisi (benturan data) |

Penjelasan:

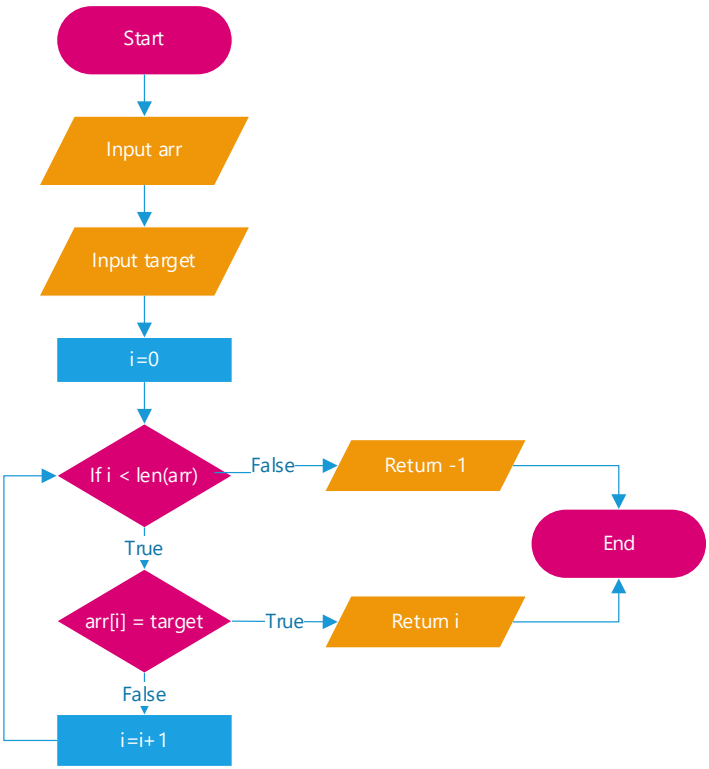
- **Linear Search** cocok digunakan saat data tidak disortir dan hanya sesekali digunakan.
- **Binary Search** merupakan pilihan terbaik untuk data besar yang terurut.
- **Interpolation Search** mengasumsikan distribusi data merata, cocok untuk data numerik yang tersebar rapi.
- **Jump Search** merupakan kompromi antara Linear dan Binary.
- **Hash-based Search** sangat efisien, tapi bergantung pada desain *hash function* yang baik.

## 9.3 Flowchart (Diagram Alir)

Flowchart adalah representasi visual langkah-langkah dalam suatu algoritma, menggunakan simbol-simbol standar untuk menunjukkan urutan operasi dan keputusan (Khesya, 2021). Penggunaan flowchart sangat membantu dalam memahami logika dan alur kerja suatu algoritma sebelum diimplementasikan ke dalam kode.

### 9.3.1 Flowchart Pencarian Linear

Flowchart untuk pencarian linear menggambarkan proses pemeriksaan setiap elemen secara berurutan.

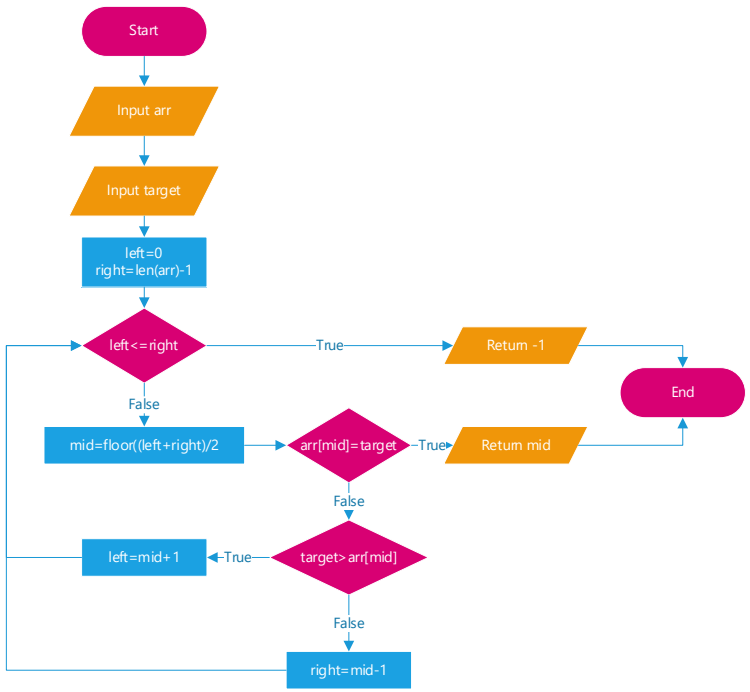



Gambar 9.1: Flowchart Pencarian Linear

Proses dimulai dengan inisialisasi indeks i ke 0. Kemudian, dalam sebuah loop, algoritma memeriksa apakah i masih dalam batas panjang daftar dan apakah elemen pada `arr[i]` sama dengan nilai target. Jika elemen ditemukan (`arr[i] == target`), pencarian berhasil dan indeks i dikembalikan. Jika tidak ada kecocokan, i ditingkatkan sebesar 1 untuk melanjutkan ke elemen berikutnya. Proses ini berulang hingga elemen ditemukan atau seluruh daftar telah diperiksa. Jika loop selesai tanpa menemukan elemen (yaitu, i mencapai akhir daftar), pencarian dinyatakan gagal dan mengembalikan -1 (Armawati, 2023).

### 9.3.2 Flowchart Pencarian Biner

Flowchart untuk pencarian biner, yang dirancang untuk array terurut, menunjukkan pendekatan divide-and-conquer.




Gambar 9.2: Flowchart Pencarian Biner

Proses dimulai dengan menginisialisasi batas bawah (left) ke indeks pertama (0) dan batas atas (right) ke indeks terakhir dari array (n-1). Dalam sebuah loop, algoritma menghitung indeks tengah (mid) sebagai `(left + right) // 2`. Kemudian, nilai target dibandingkan dengan elemen di arr[mid]. Jika sama, elemen ditemukan dan mid dikembalikan. Jika target lebih besar dari `arr[mid]`, batas bawah (left) digeser ke `mid + 1` (mencari di paruh kanan). Jika target lebih kecil dari `arr[mid]`, batas atas (right) digeser ke `mid - 1` (mencari di paruh kiri). Loop ini berlanjut selama low kurang dari atau sama dengan high. Jika loop berakhir dan target tidak ditemukan, algoritma mengembalikan indikasi bahwa elemen tidak ada (`-1`) (Buana & Setiawan, 2021).

## 9.4 Implementasi Algoritma Pencarian Menggunakan Python

Python dikenal sebagai bahasa pemrograman yang intuitif dan mudah dibaca, menjadikannya pilihan ideal untuk mengimplementasikan dan memahami algoritma struktur data. Bagian ini akan menampilkan implementasi Python untuk beberapa algoritma pencarian utama, disertai penjelasan rinci.

### 9.4.1 Implementasi Pencarian Linear (*Linear Search*)

**Pencarian linear (Linear Search)** adalah salah satu algoritma pencarian paling sederhana. Algoritma ini bekerja dengan memeriksa setiap elemen dalam daftar secara berurutan mulai dari indeks pertama hingga terakhir, sampai elemen yang dicari ditemukan atau daftar habis diperiksa. Keunggulan utama dari metode ini adalah kesederhanaan implementasinya, serta tidak adanya persyaratan bahwa data harus terurut.

**Implementasi Linear Search dalam Python:**

```
def cari_linear(daftar, target):
    """
    Mencari nilai target dalam daftar menggunakan metode
linear search.
```

```
    Parameter:
    - daftar : list
        List angka yang akan diperiksa.
    - target : int
        Nilai yang ingin ditemukan.

    Return:
    - int : indeks target jika ditemukan, atau -1 jika tidak
ditemukan.
    """
    for i in range(len(daftar)):
        if daftar[i] == target:
            return i  # Nilai ditemukan di indeks i
    return -1  # Nilai tidak ditemukan
```

**Contoh Penggunaan:**

```
# Daftar data
data = [23, 45, 12, 9, 78, 70, 33]
nilai_cari = 70

# Panggil fungsi cari_linear
hasil = cari_linear(data, nilai_cari)

# Tampilkan hasil pencarian
if hasil != -1:
    print(f"Nilai {nilai_cari} ditemukan di indeks
{hasil}.")
else:
    print(f"Nilai {nilai_cari} tidak ada di
daftar.")
```

**Output:**

```
Nilai 70 ditemukan di indeks 5.
```

**Penjelasan Algoritma**

1. **Inisialisasi**

   Fungsi `cari_linear` menerima dua argumen: `daftar` (list angka) dan `target` (nilai yang akan dicari).

2. **Iterasi Elemen**

   - Algoritma memeriksa setiap elemen di dalam `daftar` menggunakan loop `for`.
   - Jika `daftar[i] == target`, indeks `i` segera dikembalikan.
   - Jika pencarian mencapai akhir daftar tanpa menemukan target, fungsi mengembalikan `-1`.

**Contoh Kasus:**

Dengan daftar `[23, 45, 12, 9, 78, 70, 33]`, pencarian `70` menghasilkan indeks `5` (karena indeks dimulai dari `0`).

Tabel 9.2: Ilustrasi Proses Linear Search (*nilai 70*)

| Langkah | Indeks | Nilai pada Indeks | Cocok dengan Target? |
|---|---|---|---|
| 1 | 0 | 23 | Tidak |
| 2 | 1 | 45 | Tidak |
| 3 | 2 | 12 | Tidak |
| 4 | 3 | 9 | Tidak |
| 5 | 4 | 78 | Tidak |
| 6 | 5 | 70 | Ya |

Tabel 9.3: Kelebihan dan Kekurangan Linear Search

| Kelebihan | Kekurangan |
|---|---|
| Implementasi sederhana dan mudah dipahami | Waktu pencarian lambat untuk daftar yang panjang |
| Tidak memerlukan data terurut | Kompleksitas waktu O(n) pada kasus terburuk |

| Kelebihan | Kekurangan |
|---|---|
| Dapat digunakan pada semua jenis data | Kurang efisien dibanding algoritma pencarian lain |

Tabel 9.4: Kompleksitas Linear Search

| Jenis Kompleksitas | Notasi Big-O |
|---|---|
| Waktu Terburuk | O(n) |
| Waktu Rata-rata | O(n) |
| Waktu Terbaik | O(1) |
| Kompleksitas Ruang | O(1) |

### 9.4.2 Implementasi Pencarian Biner (*Binary Search*)

**Binary Search** adalah algoritma pencarian efisien yang hanya dapat diterapkan pada data yang telah **terurut secara menaik atau menurun**. Alih-alih memeriksa elemen satu per satu seperti pada *linear search*, binary search membagi ruang pencarian menjadi dua bagian secara berulang. Dengan demikian, jumlah elemen yang harus diperiksa menurun drastis setiap langkah, menghasilkan kompleksitas waktu **O(log n)** dalam kasus terbaik dan rata-rata.

**Implementasi Rekursif Binary Search dalam Python:**

```
def cari_biner_rekursif(daftar, target, kiri, kanan):
    """
    Mencari nilai target dalam daftar terurut menggunakan
metode binary search rekursif.

    Parameter:
    - daftar : list[int]
        Daftar angka yang sudah terurut.
    - target : int
        Nilai yang ingin dicari.
    - kiri : int
        Indeks awal pencarian.
    - kanan : int
```

```
        Indeks akhir pencarian.

    Return:
    - int : Indeks target jika ditemukan, atau -1 jika tidak
ditemukan.
    """
    if kiri <= kanan:
        tengah = (kiri + kanan) // 2
        if daftar[tengah] == target:
            return tengah
        elif daftar[tengah] < target:
            return cari_biner_rekursif(daftar, target, tengah
+ 1, kanan)
        else:
            return cari_biner_rekursif(daftar, target, kiri,
tengah - 1)
    return -1
```

**Contoh Penggunaan:**

```
# Daftar data yang sudah terurut
data = [9, 12, 23, 33, 45, 70, 78]
nilai_cari = 23

# Panggil fungsi binary search
hasil = cari_biner_rekursif(data, nilai_cari, 0, len(data) -
1)

# Tampilkan hasil
if hasil != -1:
    print(f"Nilai {nilai_cari} ditemukan di indeks {hasil}.")
else:
    print(f"Nilai {nilai_cari} tidak ada di daftar.")
```

**Output:**

```
Nilai 23 ditemukan di indeks 2.
```

### Penjelasan Algoritma

Binary Search bekerja melalui proses rekursif sebagai berikut:

1. **Cek Kondisi Awal**

   Jika indeks `kiri` lebih besar dari `kanan`, maka ruang pencarian telah habis dan nilai tidak ditemukan.

2. **Hitung Titik Tengah**

   Dihitung menggunakan `(kiri + kanan) // 2`. Elemen di indeks `tengah` dibandingkan dengan nilai `target`.

3. **Bandingkan Nilai Tengah dengan Target**

   - Jika cocok, maka indeks `tengah` dikembalikan.
   - Jika `target < daftar[tengah]`, pencarian dilakukan pada separuh **kiri** daftar.
   - Jika `target > daftar[tengah]`, pencarian dilanjutkan pada separuh **kanan** daftar.

4. **Pencarian Berlanjut Secara Rekursif**

   Fungsi `cari_biner_rekursif` dipanggil kembali hingga nilai ditemukan atau ruang pencarian habis.

Tabel 9.5: Ilustrasi Proses Binary Search untuk Mencari 23

| Langkah | Indeks Kiri | Indeks Kanan | Indeks Tengah | Nilai Tengah | Arah Pencarian |
|---|---|---|---|---|---|
| 1 | 0 | 6 | 3 | 33 | Kiri |
| 2 | 0 | 2 | 1 | 12 | Kanan |
| 3 | 2 | 2 | 2 | 23 | Ditemukan |

Tabel 9.6: Kelebihan dan Kekurangan Binary Search

| Kelebihan | Kekurangan |
|---|---|
| Kompleksitas waktu yang sangat efisien (O(log n)) | Data harus **terurut** sebelum dilakukan pencarian |
| Cocok untuk dataset besar | Implementasi rekursif dapat menimbulkan *stack overflow* |

| Kelebihan | Kekurangan |
|---|---|
| Lebih cepat dari linear search dalam mayoritas kasus | Kurang fleksibel dibanding pencarian berbasis hash table |

Tabel 9.7: Kompleksitas Algoritma Binary Search

| Jenis Kompleksitas | Notasi Big-O |
|---|---|
| Waktu Terburuk | O(log n) |
| Waktu Rata-rata | O(log n) |
| Waktu Terbaik | O(1) |
| Kompleksitas Ruang | O(log n) (rekursif), O(1) (iteratif) |

### 9.4.3 Implementasi Pencarian Interpolasi (*Interpolation Search*)

**Interpolation Search** adalah algoritma pencarian yang merupakan pengembangan dari binary search. Algoritma ini **mengasumsikan data terdistribusi secara seragam**, sehingga dapat memperkirakan posisi elemen target lebih akurat dibandingkan binary search yang selalu memeriksa elemen tengah. Pendekatan ini sangat efisien dalam dataset besar yang memiliki *uniform distribution.*

**Implementasi Interpolation Search:**

```
def cari_interpolasi(daftar, target):
    """
    Mencari nilai target dalam daftar terurut menggunakan
metode interpolation search.

    Parameter:
    - daftar : list[int]
        Daftar angka yang sudah terurut dan terdistribusi
merata.
    - target : int
        Nilai yang ingin dicari.

    Return:
```

```
    - int : Indeks target jika ditemukan, atau -1 jika tidak
ditemukan.
    """
    kiri = 0
    kanan = len(daftar) - 1

    while kiri <= kanan and target >= daftar[kiri] and target
<= daftar[kanan]:
        # Tangani kasus pembagian nol
        if daftar[kanan] == daftar[kiri]:
            if daftar[kiri] == target:
                return kiri
            return -1

        # Perkiraan posisi target menggunakan rumus
interpolasi
        pos = kiri + ((kanan - kiri) * (target - daftar[kiri])
//
                      (daftar[kanan] - daftar[kiri]))

        if daftar[pos] == target:
            return pos
        elif daftar[pos] < target:
            kiri = pos + 1
        else:
            kanan = pos - 1
    return -1
```

**Contoh Penggunaan:**

```
data = [10, 20, 30, 40, 50, 60, 70, 80, 90]
nilai_cari = 70

hasil = cari_interpolasi(data, nilai_cari)

if hasil != -1:
    print(f"Nilai {nilai_cari} ditemukan di indeks {hasil}.")
else:
    print(f"Nilai {nilai_cari} tidak ada di daftar.")
```

**Contoh Output 1 – Ketika nilai ditemukan:**

```
Nilai 70 ditemukan di indeks 6.
```

**Contoh Output 2 – Ketika nilai tidak ditemukan:**

```
Nilai 100 tidak ada di daftar.
```

**Penjelasan:**

1. **Kondisi Awal**

   Fungsi menerima daftar terurut dan target pencarian. Indeks kiri dimulai dari 0 dan kanan dari `len(daftar) - 1`.

2. **Validasi Rentang**

   Pencarian dilanjutkan hanya jika `target` berada dalam rentang `[daftar[kiri], daftar[kanan]]`.

3. **Rumus Interpolasi**

   Posisi perkiraan `pos` dihitung menggunakan formula:

$$pos = kiri + \frac{(target - daftar[kiri]) \times (kanan - kiri)}{daftar[kanan] - daftar[kiri]}$$

4. **Bandingkan dengan Target**
   - Jika `daftar[pos] == target`, pencarian berhasil.
   - Jika lebih kecil, pencarian dilanjutkan ke sebelah kanan (`kiri = pos + 1`).
   - Jika lebih besar, pencarian pindah ke kiri (`kanan = pos - 1`).

5. **Kasus Khusus**

   Jika `daftar[kiri] == daftar[kanan]`, pencarian berhenti untuk menghindari pembagian dengan nol.

Tabel 9.8: Ilustrasi Proses Interpolation Search untuk Target = 70

| Langkah | Indeks Kiri | Indeks Kanan | Estimasi Posisi | Nilai di Posisi | Arah Pencarian |
|---|---|---|---|---|---|
| 1 | 0 | 8 | 6 | 70 | Ditemukan |

Tabel 9.9: Kelebihan dan Kekurangan Interpolation Search

| Kelebihan | Kekurangan |
|---|---|
| Sangat cepat jika data terdistribusi merata | Tidak efisien jika data tidak merata |
| Kompleksitas mendekati O(log log n) dalam kasus ideal | Pembagian dengan nol harus ditangani |
| Cocok untuk pencarian dalam big data numerik | Hanya dapat digunakan untuk data numerik yang terurut |

Tabel 9.10: Kompleksitas Interpolation Search

| Jenis Kompleksitas | Notasi Big-O |
|---|---|
| Waktu Terburuk | O(n) |
| Waktu Rata-rata | O(log log n) |
| Waktu Terbaik | O(1) |
| Kompleksitas Ruang | O(1) |

### 9.4.4 Implementasi Pencarian Lompat (*Jump Search*)

Jump Search adalah salah satu algoritma pencarian pada struktur data terurut yang menggabungkan kelebihan binary search dan linear search. Algoritma ini bekerja dengan melakukan lompatan sejumlah langkah tertentu (biasanya sebesar akar kuadrat dari panjang daftar), dan kemudian melakukan pencarian linear dalam blok yang kemungkinan mengandung nilai target. Pendekatan ini mempercepat pencarian dibandingkan linear search biasa, namun tetap lebih sederhana dari binary search.

**Implementasi Algoritma Jump Search:**

```
import math

def cari_lompat(daftar, target):
    """
    Mencari nilai target dalam daftar terurut menggunakan
metode Jump Search.

    Parameter:
    - daftar: list of int → daftar angka yang sudah terurut.
    - target: int → nilai yang ingin dicari.

    Return:
    - int → indeks dari target jika ditemukan, atau -1 jika
tidak ditemukan.
    """
    n = len(daftar)
    langkah = int(math.sqrt(n))  # Menentukan ukuran langkah
optimal
    sebelumnya = 0

    # Melompat hingga menemukan blok yang memungkinkan
mengandung target
    while sebelumnya < n and daftar[min(langkah, n) - 1] <
target:
        sebelumnya = langkah
        langkah += int(math.sqrt(n))
        if sebelumnya >= n:
            return -1   # Target berada di luar jangkauan
daftar

    # Pencarian linear di dalam blok
    while sebelumnya < min(langkah, n) and daftar[sebelumnya]
< target:
        sebelumnya += 1

    # Memverifikasi apakah nilai ditemukan
    if sebelumnya < n and daftar[sebelumnya] == target:
        return sebelumnya
    return -1
```

**Contoh Penggunaan:**

```
data = [10, 20, 30, 40, 50, 60, 70, 80, 90]
nilai_cari = 80
hasil = cari_lompat(data, nilai_cari)

if hasil != -1:
    print(f"Nilai {nilai_cari} ditemukan di indeks {hasil}.")
else:
    print(f"Nilai {nilai_cari} tidak ada di daftar.")
```

**Output:**

```
Nilai 80 ditemukan di indeks 7.
```

**Penjelasan Algoritma**

Algoritma Jump Search bekerja dengan dua fase utama:

1. **Fase Lompatan**

   Program melompat sejumlah $\sqrt{n}$ elemen dalam daftar untuk mempercepat pencarian blok. Jika elemen pada ujung blok lebih kecil dari target, maka pencarian akan dilanjutkan ke blok berikutnya.

2. **Fase Pencarian Linear**

   Setelah menemukan blok yang mungkin mengandung target, pencarian dilakukan satu per satu di dalam blok tersebut untuk memastikan apakah target ada di sana.

Tabel 9.11: Ilustrasi Proses Jump Search

| Langkah | Indeks | Nilai pada Indeks | Keterangan |
|---|---|---|---|
| 1 | 2 | 30 | Lompat ke indeks 2 |
| 2 | 5 | 60 | Lompat ke indeks 5 |
| 3 | 8 | 90 | Lompat ke indeks 8 → lewat |
| 4 | 6-7 | 70 → 80 | Linear search, ditemukan di 7 |

Tabel 9.12: Kelebihan dan Kekurangan

| Kelebihan | Kekurangan |
|---|---|
| Lebih cepat dari linear search pada daftar besar | Hanya bisa digunakan pada daftar terurut |
| Implementasi sederhana | Tidak seefisien binary search dalam banyak kasus |
| Mengurangi jumlah perbandingan | Membutuhkan perhitungan `sqrt(n)` |

Tabel 9.13: Kompleksitas Waktu dan Ruang

| Jenis Kompleksitas | Notasi Big-O |
|---|---|
| Waktu Terburuk | O(√n) |
| Waktu Rata-rata | O(√n) |
| Waktu Terbaik | O(1) |
| Kompleksitas Ruang | O(1) |

### 9.4.5 Implementasi Pencarian Berbasis Hash (Contoh Sederhana)

**Hash table** adalah struktur data yang sangat efisien untuk menyimpan dan mengambil pasangan *key-value*. Dengan menggunakan fungsi hash, data dapat diakses dalam waktu hampir konstan, yaitu *O(1)* dalam kasus terbaik. Berikut ini disajikan implementasi sederhana tabel hash di Python dengan *chaining* untuk menangani kolisi.

**Implementasi Kelas Tabel Hash:**

```
class TabelHash:
    def __init__(self, ukuran):
        """
        Konstruktor untuk inisialisasi tabel hash dengan
jumlah bucket tertentu.
        Setiap bucket direpresentasikan sebagai list kosong
(menggunakan chaining).
        """
        self.ukuran = ukuran
```

```
        self.tabel = [[] for _ in range(ukuran)]

    def _hitung_hash(self, kunci):
        """
        Fungsi privat untuk menghitung indeks hash dari kunci.
        Menggunakan built-in hash() dan modulo terhadap
ukuran tabel.
        """
        return hash(kunci) % self.ukuran

    def tambah(self, kunci, nilai):
        """
        Menambahkan pasangan kunci-nilai ke dalam tabel hash.
        Jika kunci sudah ada, maka nilainya diperbarui.
        """
        indeks = self._hitung_hash(kunci)
        bucket = self.tabel[indeks]
        for i, (kunci_ada, _) in enumerate(bucket):
            if kunci_ada == kunci:
                bucket[i] = (kunci, nilai)
                print(f"[UPDATE] Kunci '{kunci}' diperbarui
di bucket {indeks}.")
                return
        bucket.append((kunci, nilai))
        print(f"[TAMBAH] Kunci '{kunci}' ditambahkan di
bucket {indeks}.")

    def ambil(self, kunci):
        """
        Mengambil nilai berdasarkan kunci. Mengembalikan
'Tidak ada' jika tidak ditemukan.
        """
        indeks = self._hitung_hash(kunci)
        bucket = self.tabel[indeks]
        for kunci_ada, nilai in bucket:
            if kunci_ada == kunci:
                return nilai
        return "Tidak ada"

    def hapus(self, kunci):
        """
```

```
        Menghapus entri berdasarkan kunci dari tabel hash.
        """
        indeks = self._hitung_hash(kunci)
        bucket = self.tabel[indeks]
        for i, (kunci_ada, _) in enumerate(bucket):
            if kunci_ada == kunci:
                bucket.pop(i)
                print(f"[HAPUS] Kunci '{kunci}' dihapus dari
bucket {indeks}.")
                return
        print(f"[GAGAL] Kunci '{kunci}' tidak ditemukan.")

    def __str__(self):
        """
        Mengembalikan representasi string dari isi tabel
hash.
        Menampilkan hanya bucket yang memiliki isi.
        """
        isi = []
        for i, bucket in enumerate(self.tabel):
            if bucket:
                isi.append(f"Bucket {i}: {bucket}")
        return "\n".join(isi) if isi else "Tabel hash kosong."
```

**Contoh Penggunaan:**

```
# Membuat tabel hash dengan 10 bucket
tabel_hash = TabelHash(ukuran=10)

# Menambahkan data
tabel_hash.tambah("apel", 10)
tabel_hash.tambah("pisang", 20)
tabel_hash.tambah("ceri", 30)
tabel_hash.tambah("anggur", 40)
tabel_hash.tambah("aprikot", 50)

# Menampilkan isi tabel hash
print("\nIsi tabel hash setelah penambahan:")
print(tabel_hash)

# Mengambil data
```

```
print(f"\nAmbil nilai 'pisang': {tabel_hash.ambil('pisang')}")
print(f"Ambil nilai 'ceri': {tabel_hash.ambil('ceri')}")
print(f"Ambil nilai 'mangga': {tabel_hash.ambil('mangga')}")

# Menghapus data
tabel_hash.hapus("pisang")

# Menampilkan kembali isi tabel hash setelah penghapusan
print("\nIsi tabel hash setelah menghapus 'pisang':")
print(tabel_hash)

# Mencoba mengambil kembali data yang sudah dihapus
print(f"\nAmbil nilai 'pisang' setelah dihapus: {tabel_hash.ambil('pisang')}")
```

**Penjelasan:**

Kode ini mengimplementasikan tabel hash, sebuah struktur data yang menyimpan pasangan kunci-nilai (misalnya, "apel" → 10) untuk memungkinkan penyimpanan dan pengambilan data yang cepat. Bayangkan tabel hash seperti rak buku dengan beberapa laci (bucket), di mana setiap laci bisa menyimpan beberapa buku (pasangan kunci-nilai) jika terjadi "tabrakan" (kolisi). Kelas TabelHash dibuat dengan ukuran tertentu (misalnya, 10 laci) saat diinisialisasi. Setiap laci adalah list kosong untuk menampung pasangan kunci-nilai, yang memungkinkan penanganan kolisi dengan metode chaining (menyimpan beberapa pasangan dalam satu laci).

Metode _hitung_hash menghitung nomor laci (indeks) untuk kunci tertentu, seperti "apel", dengan mengubah kunci menjadi angka menggunakan fungsi hash Python dan membaginya dengan jumlah laci untuk mendapatkan sisa (modulo). Metode tambah digunakan untuk menyimpan atau memperbarui pasangan kunci-nilai. Misalnya, saat menambahkan "apel" → 10, kode menghitung indeks laci untuk "apel", memeriksa apakah kunci sudah ada di laci tersebut, dan memperbarui nilainya jika ada, atau menambahkan pasangan baru jika tidak ada. Metode ambil mencari nilai berdasarkan kunci, seperti mencari nilai untuk "pisang", dan mengembalikan nilainya (misalnya, 20) atau

"Tidak ada" jika kunci tidak ditemukan. Metode hapus menghapus pasangan kunci-nilai dari laci yang sesuai, misalnya menghapus "pisang" dari lacinya. Metode __str__ menampilkan isi tabel hash dengan mencetak setiap laci yang tidak kosong, menunjukkan pasangan kunci-nilai di dalamnya.

Bagian contoh penggunaan membuat tabel hash dengan 10 laci, lalu menambahkan pasangan seperti "apel" → 10, "pisang" → 20, dan lainnya. Kode mencetak pesan setiap kali menambahkan atau memperbarui kunci, menampilkan isi tabel hash, mencari nilai untuk kunci seperti "pisang" dan "mangga", menghapus "pisang", dan menampilkan tabel hash lagi setelah penghapusan. Tabel hash ini efisien untuk mencari, menambah, atau menghapus data, terutama jika kolisi minim, dan kode ditulis dengan cara yang sederhana dan mudah dipahami.

**Output:**

```
[TAMBAH] Kunci 'apel' ditambahkan di bucket 6.
[TAMBAH] Kunci 'pisang' ditambahkan di bucket 4.
[TAMBAH] Kunci 'ceri' ditambahkan di bucket 8.
[TAMBAH] Kunci 'anggur' ditambahkan di bucket 2.
[TAMBAH] Kunci 'aprikot' ditambahkan di bucket 4.

Isi tabel hash setelah penambahan:
Bucket 2: [('anggur', 40)]
Bucket 4: [('pisang', 20), ('aprikot', 50)]
Bucket 6: [('apel', 10)]
Bucket 8: [('ceri', 30)]

Ambil nilai 'pisang': 20
Ambil nilai 'ceri': 30
Ambil nilai 'mangga': Tidak ada
[HAPUS] Kunci 'pisang' dihapus dari bucket 4.

Isi tabel hash setelah menghapus 'pisang':
Bucket 2: [('anggur', 40)]
Bucket 4: [('aprikot', 50)]
Bucket 6: [('apel', 10)]
Bucket 8: [('ceri', 30)]
```

```
Ambil nilai 'pisang' setelah dihapus: Tidak ada
```

Tabel 9.14: Operasi dan Kompleksitas Hash Table

| Operasi | Deskripsi | Kompleksitas Rata-rata | Kompleksitas Terburuk |
|---|---|---|---|
| `tambah()` | Menambah atau update kunci-nilai | O(1) | O(n) |
| `ambil()` | Mengambil nilai berdasarkan kunci | O(1) | O(n) |
| `hapus()` | Menghapus pasangan kunci-nilai | O(1) | O(n) |

*Catatan:* Kompleksitas terburuk terjadi bila semua elemen jatuh ke dalam satu bucket (*worst-case chaining*).

Tabel 9.15: Keunggulan dan Kelemahan Struktur Hash Table

| Keunggulan | Kelemahan |
|---|---|
| Waktu akses sangat cepat (O(1) dalam rata-rata) | Tidak efisien jika fungsi hash buruk |
| Efisien untuk dataset besar | Tidak cocok untuk data yang harus disimpan terurut |
| Mudah diimplementasikan | Perlu menangani kolisi (chaining atau probing) |

## 9.5 Studi Kasus: Aplikasi Pencarian dalam Dunia Nyata

Memahami algoritma secara teori dan implementasi adalah satu hal, tetapi mengetahui kapan dan di mana menggunakannya adalah keterampilan yang berbeda. Bagian ini akan membahas beberapa studi kasus yang menunjukkan bagaimana algoritma pencarian diterapkan dalam skenario dunia nyata.

### 9.5.1 Skenario 1: Mencari Produk di Toko Online

Deskripsi Kasus: Bayangkan Anda sedang membangun fitur pencarian untuk toko online yang memiliki jutaan produk. Pengguna mengetik nama produk atau kata kunci, dan sistem perlu menampilkan hasil yang relevan secepat mungkin.

**Pilihan Algoritma:**

1. **Pencarian Linear (Linear Search)**

   Algoritma ini jelas tidak cocok untuk skenario ini. Dengan jutaan produk, pencarian linear akan memiliki kompleksitas waktu O(n), yang berarti akan sangat lambat dan menyebabkan pengalaman pengguna yang buruk, bahkan bisa mengakibatkan timeout system (Ray et al., 2021).

2. **Pencarian Biner (Binary Search)**

   Algoritma ini sangat cocok jika produk disimpan dalam daftar yang terurut, misalnya berdasarkan ID Produk, nama alfabetis, atau harga. Jika data terurut, pencarian biner dengan kompleksitas O(log n) dapat menemukan produk dalam hitungan milidetik, bahkan untuk dataset yang sangat besar (Lin, 2019). Namun, penting untuk memiliki mekanisme pengurutan awal atau pemeliharaan urutan data setiap kali produk baru ditambahkan atau dihapus.

3. **Pencarian Berbasis Hash (*Hash-based Search*)**

   Ini adalah pilihan yang sangat baik, terutama jika pencarian sering dilakukan berdasarkan kunci unik seperti SKU (Stock Keeping Unit) produk. Dengan tabel hash, akses ke detail produk dapat hampir instan (rata-rata O(1)) (Wang, 2025). Inilah mengapa sistem basis data sering menggunakan indeks berbasis hash untuk pencarian cepat.

**Analisis Pemilihan Algoritma Terbaik:**

Untuk toko online dengan data produk yang besar, Pencarian Biner atau Pencarian Berbasis Hash adalah pilihan yang jauh lebih unggul daripada Pencarian Linear. Jika pencarian sering dilakukan berdasarkan

ID unik atau SKU, Pencarian Berbasis Hash adalah yang tercepat dan paling efisien. Jika pencarian berdasarkan nama produk yang diurutkan, Pencarian Biner adalah pilihan yang tepat. Meskipun ada overhead pengurutan awal atau pembangunan tabel hash, investasi awal ini akan terbayar dengan performa pencarian yang sangat cepat, yang krusial untuk pengalaman pengguna yang baik.

Studi kasus ini secara jelas menunjukkan bahwa dalam aplikasi skala besar, "pencarian" bukan hanya tentang memilih algoritma yang tepat, tetapi juga tentang bagaimana data diindeks dan distrukturkan. Basis data modern mengandalkan indeks (seringkali berbasis pohon B-Tree atau tabel hash) untuk memungkinkan pencarian cepat pada volume data yang masif. Ini adalah contoh nyata di mana investasi dalam desain struktur data dan pemilihan algoritma yang tepat secara langsung berkorelasi dengan pengalaman pengguna yang responsif dan kemampuan skalabilitas sistem secara keseluruhan.

### 9.5.2 Skenario 2: Mengelola Data Karyawan dalam Sistem Informasi

Deskripsi Kasus: Sebuah perusahaan memiliki sistem informasi karyawan yang menyimpan data ribuan karyawan. Setiap karyawan memiliki ID unik (Nomor Induk Karyawan/NIK). Manajer sering perlu mencari detail karyawan berdasarkan NIK mereka.

**Pilihan Algoritma**

1. **Pencarian Berbasis Hash (*Hash-based Search*)**

   Ini adalah pilihan yang ideal untuk skenario ini. Karena setiap karyawan memiliki NIK yang unik, NIK dapat berfungsi sebagai kunci dalam tabel hash. Pencarian berdasarkan NIK akan memberikan akses rata-rata O(1) ke data karyawan, menjadikannya sangat cepat dan efisien (Wang, 2025). Tipe data dict di Python adalah representasi sempurna untuk mengimplementasikan skenario ini.

**2. Pencarian Biner (Binary Search)**

Algoritma ini juga bisa digunakan jika data karyawan disimpan dalam array atau daftar yang terurut berdasarkan NIK. Namun, setiap kali karyawan baru ditambahkan atau dihapus, daftar harus diurutkan ulang atau dipertahankan urutannya, yang dapat menjadi overhead komputasi. Meskipun efisien (O(log n)), algoritma ini tidak secepat Pencarian Berbasis Hash untuk pencarian kunci unik.

## Analisis Pemilihan Algoritma Terbaik

Untuk skenario ini, Pencarian Berbasis Hash adalah pemenang yang jelas. Kemampuan untuk mencari berdasarkan kunci unik dengan kecepatan rata-rata O(1) sangat penting untuk sistem yang membutuhkan akses data yang sangat cepat dan sering. Ini meminimalkan waktu tunggu bagi pengguna dan memaksimalkan efisiensi operasional sistem informasi karyawan.

Studi kasus ini dengan tegas menyoroti bagaimana keberadaan "kunci unik" dalam kumpulan data dapat menjadi pendorong utama untuk mencapai efisiensi pencarian optimal. Ketika ada kunci yang dapat di-hash secara unik (seperti NIK), tabel hash menjadi pilihan yang tak tertandingi dalam hal kecepatan akses. Hal ini menunjukkan bahwa desain data yang cermat, misalnya dengan memastikan adanya pengidentifikasi unik untuk setiap entitas, dapat secara fundamental memengaruhi pilihan algoritma pencarian yang paling sesuai dan, pada akhirnya, performa keseluruhan sistem.

Tabel 9.16: Contoh Aplikasi Algoritma Pencarian dalam Dunia Nyata

| Skenario Aplikasi | Algoritma Pencarian yang Direkomendasikan | Alasan Rekomendasi |
|---|---|---|
| Mencari Produk di Toko Online (Jutaan produk, pencarian cepat) | Pencarian Berbasis Hash (berdasarkan SKU/ID Produk) atau Pencarian | Hash-based: O(1) rata-rata untuk kunci unik, akses instan (Wang, 2025). Binary: O(log n) untuk data terurut, sangat cepat untuk dataset besar (Lin, 2019). |

| Skenario Aplikasi | Algoritma Pencarian yang Direkomendasikan | Alasan Rekomendasi |
|---|---|---|
| | Biner (berdasarkan nama/harga terurut) | Linear Search tidak praktis karena O(n) (Ray et al., 2021). |
| Mengelola Data Karyawan dalam Sistem Informasi (Ribuan karyawan, pencarian berdasarkan NIK unik) | Pencarian Berbasis Hash | NIK sebagai kunci unik memungkinkan pencarian O(1) rata-rata, memberikan akses data yang sangat cepat dan efisien (Wang, 2025). |
| Mencari Kata dalam Kamus Digital (Kamus besar, pencarian kata) | Pencarian Biner | Kata-kata dalam kamus secara alami terurut alfabetis, menjadikan Binary Search (O(log n)) sangat efisien (Lin, 2019). |

Sistem Rekomendasi Film (Mencari film berdasarkan rating atau genre dalam daftar yang sangat besar) Pencarian Interpolasi (jika rating/genre terdistribusi seragam) atau Pencarian Biner

Interpolation Search (O(log log n)) lebih cepat jika data terdistribusi seragam (Bitner & Reingold, 1975). Binary Search (O(log n)) adalah pilihan umum yang kuat untuk data terurut (Lin, 2019).

Pencarian File di Sistem Operasi (Mencari file berdasarkan nama di direktori besar) Pencarian Lompat (jika filesystem memungkinkan akses blok) atau Pencarian Berbasis Hash (jika nama file diindeks)

Jump Search (O(√n)) dapat mengurangi lompatan acak pada disk (Harabor & Koenig, 2021). Hash-based Search memberikan akses cepat jika sistem file menggunakan indeks hash (Wang, 2025).

## Kesimpulan

Pencarian data merupakan operasi fundamental dalam komputasi, dan efisiensinya sangat bergantung pada pemilihan algoritma serta struktur data yang tepat. Algoritma Pencarian Linear, meskipun sederhana dan dapat diterapkan pada data tidak terurut, memiliki kinerja yang lambat (O(n)) untuk dataset besar. Sebaliknya, algoritma seperti Pencarian Biner, Pencarian Interpolasi, dan Pencarian Lompat menawarkan

efisiensi yang jauh lebih tinggi (masing-masing O(log n), O(log log n), dan O(√n)), namun dengan prasyarat bahwa data harus terurut. Pencarian Interpolasi menonjol karena efisiensinya yang luar biasa pada data yang terdistribusi secara seragam.

Pencarian Berbasis Hash, yang sering diimplementasikan melalui tabel hash, menawarkan kecepatan akses rata-rata O(1) untuk pencarian berdasarkan kunci unik, menjadikannya sangat kuat meskipun ada tantangan dalam penanganan kolisi. Python, dengan sintaksisnya yang intuitif, menyediakan lingkungan yang sangat baik untuk mengimplementasikan dan memahami algoritma-algoritma ini, bahkan menggunakannya secara internal untuk struktur data bawaan seperti dict.

Dalam memilih algoritma pencarian yang paling sesuai, beberapa faktor krusial perlu dipertimbangkan:

- **Karakteristik Data**: Apakah data yang akan dicari sudah terurut? Apakah distribusinya seragam? Apakah ada kunci unik yang dapat dimanfaatkan? Jawaban atas pertanyaan-pertanyaan ini akan sangat membatasi dan memandu pilihan algoritma.
- **Ukuran Dataset**: Untuk kumpulan data yang kecil, kesederhanaan Pencarian Linear mungkin lebih diutamakan. Namun, untuk dataset yang besar, penggunaan algoritma dengan kompleksitas logaritmik atau berbasis hash adalah suatu keharusan untuk memastikan kinerja yang dapat diterima.
- **Frekuensi Pencarian**: Jika data jarang dicari, overhead komputasi untuk pengurutan awal atau pembangunan tabel hash mungkin tidak sepadan. Sebaliknya, jika pencarian sering dilakukan, investasi awal dalam struktur data dan algoritma yang efisien akan memberikan keuntungan performa yang signifikan.
- **Batasan Sumber Daya**: Pertimbangkan kompleksitas ruang dan waktu, terutama dalam lingkungan dengan memori terbatas atau kebutuhan akan respons waktu nyata.
- **Trade-off**: Penting untuk diingat bahwa tidak ada satu algoritma "terbaik" yang cocok untuk semua skenario. Keputusan terbaik

seringkali melibatkan trade-off yang cermat antara kecepatan eksekusi, penggunaan memori, dan kompleksitas implementasi.

Dengan pemahaman mendalam tentang konsep-konsep ini, pengembang dan analis data dapat membuat keputusan yang lebih terinformasi untuk merancang sistem yang efisien dan tangguh dalam menghadapi tantangan pengelolaan data yang terus berkembang.

# Bab 10
# Teknik Pengurutan Data

*"Mengurutkan data adalah seni menyusun kekacauan menjadi keteraturan."* — **Charles Babbage**

## 10.1 Bubble Sort dan Variannya

Pengurutan (*sorting*) adalah kegiatan menyusun kembali sekumpulan objek yang tidak beraturan menjadi beraturan dengan aturan tertentu atau susunan tertentu. Mengurutkan data sangat bermanfaat oleh karena data yang terurut akan lebih mudah dicari dan diperbaiki jika terdapat kesalahan. Sejumlah metode pengurutan data dalam bentuk algoritma yang dapat untuk mengurutkan data diantaranya adalah *Bubble Sort, Merge Sort, Shell Sort, Radix Sort, Quick Sort*, dan sebagainya (Abdullah et al., 2023). Pengaturan urutan data bisa dilakukan baik dari yang terkecil ke yang terbesar (*ascending*) maupun sebaliknya (*descending*) (Marsellus Oton Kadang, 2021). Dalam pengurutan data selalu membandingkan data, menggunakan operator perbandingan > atau <.

### 10.1.1 Bubble Sort Dasar

Algoritma *bubble sort* adalah salah satu cara untuk mengatasi masalah pengurutan dalam pemrograman, menggunakan data yang ada sebagai masukan dan menghasilkan output yang merupakan hasil dari proses pengurutan yang dilakukan (Abdullah et al., 2023).

*Bubble Sort* merupakan salah satu teknik pengurutan yang tergolong mudah, yang bekerja dengan cara melintasi elemen yang perlu diurutkan berulang kali. Algoritma bubble sort bekerja dengan cara membandingkan dua item data yang dekat satu sama lain dan menukarnya jika keduanya tidak berada dalam urutan yang tepat.

Proses ini dilakukan berulang-ulang untuk setiap elemen sampai tidak ada lagi pertukaran yang diperlukan. Dengan demikian, seluruh data telah disusun dengan benar (Topperworld.in, n.d.).

*Pseudo-code* algoritma *buble-sort* standar mengurutkan data secara *ascending*.

```
void Bubble_Sort(int a[], int n)
{
    int i, j;

    // Loop utama untuk setiap pass
    for (i = 0; i < n; i++)
    {
        // Loop untuk membandingkan elemen yang berdekatan
        for (j = 0; j < n - i - 1; j++)
        {
            // Jika elemen saat ini lebih besar dari elemen
berikutnya, tukar
            if (a[j] > a[j + 1])
            {
                // Tukar posisi elemen
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
        }
    }
}
```

*Pseudo-code* algoritma *bubble sort* di atas bekerja dengan cara sebagai berikut:

1. Mulailah dengan membandingkan dua elemen pertama dalam vektor data.
2. Jika elemen pertama lebih besar dari yang kedua, ubah posisi keduanya.
3. Lanjutkan ke pasangan elemen berikutnya dan ulangi langkah nomor 2
4. Teruskan langkah ini hingga sampai ujung larik.

5. Pada titik ini, elemen terbesar akan "mengapung" ke ujung larik.

Misalkan diberikan data random dalam vektor data berikut:

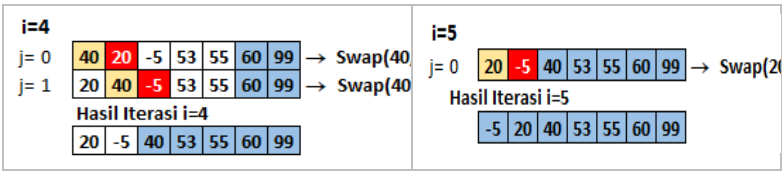
| 99 | 55 | 53 | 60 | 40 | 20 | -5 |
|---|---|---|---|---|---|---|

## Iterasi Pertama (Bandingkan dan Tukar)

1. Mulai dengan indeks pertama. Kemudian bandingkan elemen pertama dan kedua.
2. Tukar posisi keduanya jika elemen pertama lebih besar dari kedua.
3. Kemudian bandingkan elemen kedua dan ketiga. Jika tidak berurutan, ubah posisinya
4. Proses di atas berlanjut hingga elemen terakhir dicapai.

Tabel 10.1: Simulasi metode *bubble sort*

<table>
<tr>
<td>Iterasi Pertama<br>i=0<br>j= 0 | 99 | 55 | 53 | 60 | 40 | 20 | -5 | → Swap(99<br>j= 1 | 55 | 99 | 53 | 60 | 40 | 20 | -5 | → Swap(99<br>j= 2 | 55 | 53 | 99 | 60 | 40 | 20 | -5 | → Swap(99<br>j= 3 | 55 | 53 | 60 | 99 | 40 | 20 | -5 | → Swap(99<br>j= 4 | 55 | 53 | 60 | 40 | 99 | 20 | -5 | → Swap(99<br>j= 5 | 55 | 53 | 60 | 40 | 20 | 99 | -5 | → Swap(99<br>Hasil Iterasi i=0<br>55 | 53 | 60 | 40 | 20 | -5 | 99</td>
<td>Iterasi Kedua<br>i=1<br>j= 0 | 55 | 53 | 60 | 40 | 20 | -5 | 99 | → Swap(55,<br>j= 1 | 53 | 55 | 60 | 40 | 20 | -5 | 99<br>j= 2 | 53 | 55 | 60 | 40 | 20 | -5 | 99 | → Swap(60,<br>j= 3 | 53 | 55 | 40 | 60 | 20 | -5 | 99 | → Swap(60,<br>j= 4 | 53 | 55 | 40 | 20 | 60 | -5 | 99 | → Swap(60<br>Hasil Iterasi i=1<br>53 | 55 | 40 | 20 | -5 | 60 | 99</td>
</tr>
<tr>
<td>Iterasi Ketiga<br>i=2<br>j= 0 | 53 | 55 | 40 | 20 | -5 | 60 | 99 | → Swap(55<br>j= 1 | 53 | 55 | 40 | 20 | -5 | 60 | 99 | → Swap(55<br>j= 2 | 53 | 40 | 55 | 20 | -5 | 60 | 99 | → Swap(55<br>j= 3 | 53 | 40 | 20 | 55 | -5 | 60 | 99 | → Swap(55<br>Hasil Iterasi i=2<br>53 | 40 | 20 | -5 | 55 | 60 | 99</td>
<td>Iterasi Keempat<br>i=3<br>j= 0 | 53 | 40 | 20 | -5 | 55 | 60 | 99 | → Swap(53<br>j= 1 | 40 | 53 | 20 | -5 | 55 | 60 | 99 | → Swap(53<br>j= 2 | 40 | 20 | 53 | -5 | 55 | 60 | 99 | → Swap(53<br>Hasil Iterasi i=3<br>40 | 20 | -5 | 53 | 55 | 60 | 99</td>
</tr>
<tr>
<td>Iterasi Kelima</td>
<td>Iterasi Keenam</td>
</tr>
</table>




Implementasi algoritma di atas menggunakan bahasa *python* sebagai berikut:

```
# Program 10.1 - Implementasi Bubble Sort

def bubble_sort(a):
    n = len(a)
    # Loop untuk setiap pass
    for i in range(n):
        # Loop untuk membandingkan elemen berdekatan
        for j in range(0, n - i - 1):
            # Tukar jika urutan salah (elemen kiri > dari
kanan)
            if a[j] > a[j + 1]:
                a[j], a[j + 1] = a[j + 1], a[j]

# Data sebelum diurutkan
data = [99, 55, 53, 60, 40, 20, -5]
print("Sebelum diurutkan:", data)

# Pemanggilan fungsi bubble_sort
bubble_sort(data)

# Data setelah diurutkan
print("Setelah diurutkan:", data)
```

Output:

```
Sebelum diurutkan: [99, 55, 53, 60, 40, 20, -5]
Setelah diurutkan: [-5, 20, 40, 53, 55, 60, 99]
```

### 10.1.2 Optimasi Early-Exit

Ketika algoritma *sort bubble* biasa digunakan, data mungkin sudah terurut, tetapi algoritma terus bekerja sampai *looping* berakhir. Tentu saja, memproses data membutuhkan lebih banyak waktu. Untuk mencegah hal ini terjadi, dapat digunakan *Exit Early*, atau *Early-Stopping*, yang memaksa algoritma untuk berhenti ketika tidak ada lagi pertukaran data dalam satu iterasi. Berikut ini adalah bentuk algoritma sort buble yang dioptimasi dengan stop awal:

```
void Bubble_Sort(int a[], int n)
{
    int i, j;
    bool swapped;

    // Loop utama untuk setiap pass
    for (i = 0; i < n; i++)
    {
        swapped = false;

        // Loop untuk membandingkan elemen berdekatan
        for (j = 0; j < n - i - 1; j++)
        {
            // Tukar jika elemen tidak dalam urutan yang benar
            if (a[j] > a[j + 1])
            {
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
                swapped = true;
            }
        }

        // Early stopping: jika tidak ada pertukaran
        if (!swapped)
            break;
    }
}
```

Implementasi algoritma *buble_sort* teroptimasi dengan *early-stopping* di atas ke dalam bahasa pemrograman *python* sebagai berikut:

Implementasi algoritma di atas menggunakan bahasa *python* sebagai berikut:

```
# Program 10.2 - Bubble Sort dengan Early Stopping

def bubble_sort(a):
    n = len(a)
    for i in range(n):
        swapped = False  # Penanda apakah terjadi pertukaran

        # Loop untuk membandingkan & menukar elemen tidak
berurutan
        for j in range(0, n - i - 1):
            if a[j] > a[j + 1]:
                a[j], a[j + 1] = a[j + 1], a[j]
                swapped = True

        # Jika tidak ada pertukaran, array sudah terurut
        if not swapped:
            break

# Data sebelum diurutkan
data = [99, 55, 53, 60, 40, 20, -5]
print("Sebelum diurutkan:", data)

# Pemanggilan fungsi bubble_sort
bubble_sort(data)

# Data setelah diurutkan
print("Setelah diurutkan:", data)
```

**Output:**

```
Sebelum diurutkan: [99, 55, 53, 60, 40, 20, -5]
Setelah diurutkan: [-5, 20, 40, 53, 55, 60, 99]
```

## 10.2 Selection Sort dan Insertion Sort

*Insertion sort* bekerja dengan membangun vektor yang telah terurut satu per satu, sementara elemen yang belum terurut disisipkan ke posisi yang

sesuai dalam vektor yang telah terurut, *selection sort* mencari elemen terbesar atau terkecil dalam vektor data untuk ditempatkan pada posisi yang sebenarnya.

### 10.2.1 Selection Sort

Beberapa definisi algoritma *selection sort*:

1. *Selection sort* adalah metode pengurutan angka dengan memilih elemen ke-i sampai n-1 dan menukar elemen yang dipilih dengan yang lain(Sandria et al., 2022).
2. *Selection sort* merupakan salah satu algoritma pengurutan sederhana yang berbasis perbandingan di tempat, di mana vektor data dibagi menjadi dua bagian, bagian data terurut di ujung kiri dan bagian belum terurut di ujung kanan (Yanti & Eriana, 2024).
3. *Selection sort* tergolong algoritma sorting sederhana dengan cara membandingkan semua elemen dalam vektor data untuk mencari nilai terkecil/terbesar disetiap perulangan, yang selanjutnya ditempatkan diposisi yang sesuai sampai semua elemen vektor terurut (Putri et al., 2022).
4. *Selection Sort* merupakan metode pengurutan dengan cara mencari nilai elemen yang terbesar atau yang terkecil dari sekumpulan elemen nilai pada sebuah data, dan cukup gampang dalam penulisan *coding*-nya (Evi Lestari Pratiwi, 2020).
5. *Selection sort* merupakan pengurutan dengan memilih data terkecil dalam vektor untuk ditempatkan pada bagian awal, lalu dari posisi ke-2 hingga data terakhir dicari kembali yang terkecil untuk diletakkan di posisi ke-2, dan selanjutnya untuk posisi ke-3 hingga ke-(N-1) (Ginting et al., 2023).

Merujuk ke beberapa pendapat tentang definisi *selection sort* di atas, dapat dikatakan bahwa *selection sort* adalah algoritma pengurutan data yang sederhana berbasis perbandingan yang bekerja dengan memilih elemen terkecil atau terbesar dari bagian yang belum terurut dan kemudian menempatkannya di tempat yang tepat. Algoritma ini menggabungkan dua vektor data yaitu vektor data yang terurut di kiri dan yang belum terurut di kanan.

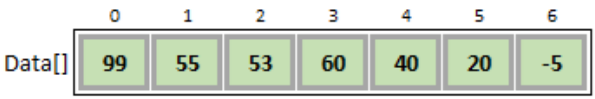
Karena kompleksitas waktu rata-rata dan terburuk algoritma ini sebesar $O(n^2)$, sehingga tidak efektif untuk data besar, walaupun algoritma *selection sort* mudah dipahami dan digunakan. Langkah-langkah metode *selection sort*:

1. Iterasi =0
2. Ambil elemen pertama sebagai maks/min yaitu maks/min= data[Iterasi]
3. Untuk j=iterasi+1 sampai dengan n-1:

   Bandingkan maks/min dengan data[j].

   - Jika tidak sesuai: jadikan maks/min=data[j],
   - Set nilai indeks=j
4. Jika iterasi <> indeks, Tukarkan data[iterasi] dengan data[Indeks]
5. Iterasi++
6. Ulangi dari langka 2 sampai data dalam vektor terurut

Untuk lebih jelasnya, perhatikan cara kerja *selection sort* untuk urutan *acsencing*. Dalam pengurutan *ascending*, dilakukan pencarian data minimum untuk ditempatkan di bagian terurut (bagian kiri vektor).




Berdasarkan data di atas, dapat diperoleh nilai n=7

Tabel 10.2: Simulasi metode *selection sort*


Implementasi *selection sort* dalam bahasa pemrograman *python*:

```
# Program 10.3 - Implementasi Selection Sort

def selection_sort(arr):
    n = len(arr)
```

```
    for i in range(n):
        min_index = i  # Asumsikan elemen ke-i adalah yang
terkecil

        # Cari elemen terkecil di sisa array
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:
                min_index = j

        # Tukar elemen minimum dengan elemen ke-i
        arr[i], arr[min_index] = arr[min_index], arr[i]

# Data sebelum diurutkan
data = [99, 55, 53, 60, 40, 20, -5]
print("Sebelum diurutkan:", data)

# Pemanggilan fungsi selection_sort
selection_sort(data)

# Data setelah diurutkan
print("Setelah diurutkan :", data)
```

**Output:**

```
Sebelum diurutkan: [99, 55, 53, 60, 40, 20, -5]
Setelah diurutkan : [-5, 20, 40, 53, 55, 60, 99]
```

### 10.2.2 Insertion Sort

*Insertion sort* biasa disebut dengan metode pengurutan yang paling dasar dan dikenal sebagai teknik peralihan yang memiliki kecepatan rata-rata berada di antara teknik pengurutan modern (*merge* dan *quick sort)* dan teknik pengurutan yang lebih *primitive* (*bubble sort* dan *selection sort*). Cara kerjanya bisa dibandingkan dengan permainan kartu sebagai berikut:

Misalkan di atas meja tersedia tumpukan sejumlah kartu permainan, di mana meja dapat dianalogikan sebagai vektor data dan masing-masing kartu dianalogikan elemen data yang ada dalam vektor tersebut.

Gambar 10.1: Analogi *Insert Section*

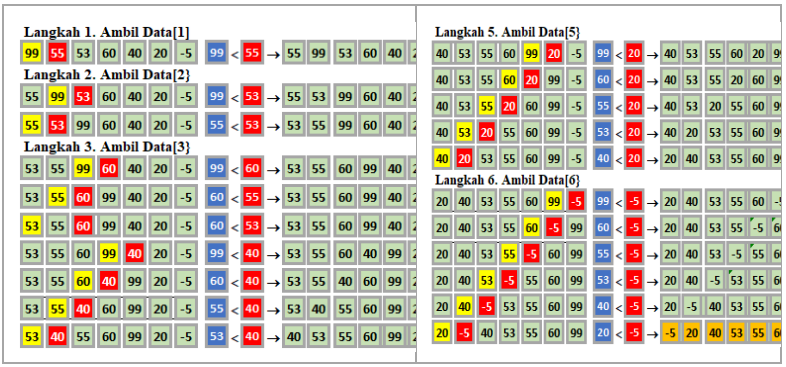
1. Misalkan di atas meja tersedia tumpukan sejumlah kartu permainan, di mana meja dapat dianalogikan sebagai vektor data dan masing-masing kartu dianalogikan elemen data yang ada dalam vektor tersebut.
2. Langkah pertama ambil satu kartu dari antara tumpukan tersebut kemudian letakkan di tangan kiri.
3. Selanjutnya ambilah kartu baru dari kartu tambahan yang ada di atas meja kemudian bandingkan dengan kumpulan kartu yang telah terurut di tangan kiri. Kemudian tempatkan kartu baru tersebut sesuai dengan posisinya.
4. Ulangi langkah 3 hingga semua kartu di meja habis dan kartu di tangan kiri terurut (Saptadi & Sari, 2012).

Misalkan diberikan isi vektor data berikut dalam keadaan random dan ingin diurutkan secara *ascending*.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Data[] | 99 | 55 | 53 | 60 | 40 | 20 | -5 |

Perhatikan cara kerja algoritma *insertion sort* berikut:

Tabel 10.3. Simulasi metode *insertion sort*




Implementasi metode *insertion sort* ke dalam bahasa pemrograman *phyton* sebagai berikut:

```
# Program 10.4 - Implementasi Insertion Sort
def insertion_sort(arr):
    n = len(arr)
    for i in range(1, n):
        key = arr[i]       # Elemen yang akan disisipkan
        j = i - 1

        # Geser elemen yang lebih besar dari key ke kanan
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

        # Tempatkan key pada posisi yang tepat
        arr[j + 1] = key
# Data sebelum diurutkan
data = [99, 55, 53, 60, 40, 20, -5]
print("Sebelum diurutkan:", data)

# Pemanggilan fungsi insertion_sort
insertion_sort(data)

# Data setelah diurutkan
print("Setelah diurutkan :", data)
```

**Output:**

```
Sebelum diurutkan: [99, 55, 53, 60, 40, 20, -5]
Setelah diurutkan: [-5, 20, 40, 53, 55, 60, 99]
```

## 10.3 Merge Sort dan Quick Sort

### Merger Sort

*Merger Sort* merupakan salah satu algoritma modern yang bekerja dengan cara memecah dan menggabungkan kembali elemen data sekaligus mengurutkan data. Metode *merge sort* dirancang untuk data yang besar.

Cara kerja metode ini:

1. Vektor data di bagi 2 menjadi vektor kiri dan dan kanan
2. Gabungkan (*merge*) kembali sambil mengurutkan baik secara *ascending* maupun *descending*.

Misalkan diberikan dua vektor A dan B berikut untuk diurutkan dengan metode *Merge Sort* secara *ascending*.


Penyelesaian:

Tabel 10.3: Simulasi metode *Merge Sort*

| Iterasi | A | B | Perbandingan | Hasil Merge Sort |
|---|---|---|---|---|
| 1 | 53 55 60 99 | -5 20 40 | 53 < -5 | → -5 |
| 2 | 53 55 60 99 | 20 40 | 53 < 20 | → -5 20 |
| 3 | 53 55 60 99 | 40 | 53 < 40 | → -5 20 40 |
| 4 | 53 55 60 99 | | 53 | → -5 20 40 53 |
| 5 | 55 60 99 | | 55 | → -5 20 40 53 55 |
| 6 | 60 99 | | 60 | → -5 20 40 53 55 60 |
| 7 | 99 | | 99 | → -5 20 40 53 55 60 99 |

Dari tabel 10.3 pada iterasi 5 s.d. 7 tidak ada lagi perbandingan elemen data A dengan B, oleh karena pada iterasi tersebut sudah tidak ada

elemen data dalam vektor B. Dengan demikian elemen data vektor A tinggal dipindahkan satu persatu ke vertor data hasil *merge sort*.

Implementasi metode *merge sort* di atas ke dalam bahasa pemrograman *python*:

```
# Program 10.5 - Penggabungan Dua Vektor Terurut (Merge)

def merge_vektors(A, B):
    i = j = 0
    result = []

    # Bandingkan elemen dari kedua vektor
    while i < len(A) and j < len(B):
        if A[i] < B[j]:
            result.append(A[i])
            i += 1
        else:
            result.append(B[j])
            j += 1

    # Tambahkan sisa elemen dari A jika masih ada
    while i < len(A):
        result.append(A[i])
        i += 1

    # Tambahkan sisa elemen dari B jika masih ada
    while j < len(B):
        result.append(B[j])
        j += 1

    return result

# Vektor yang sudah terurut
A = [53, 55, 60, 99]
B = [-5, 20, 40]

# Proses penggabungan
merged_result = merge_vektors(A, B)

# Tampilkan hasil
print("Hasil penggabungan A dan B:", merged_result)
```

**Output:**

```
Hasil penggabungan A dan B: [-5, 20, 40, 53, 55, 60, 99]
```

## Quick Sort

*Quick Sort* adalah algoritma yang dalam mengurutkan data memanfaatkan teknik memecah data menjadi sejumlah partisi data. Oleh karena itu algoritma ini biasa juga disebut dengan nama *partition exchange sort*. Pengurutan data dilakukan dengan pertama-tama memilih sebuah elemen, kemudian elemen-elemen data akan diurutkan sedemikian hingga didapatkan elemen data yang terurut (Evi Lestari Pratiwi, 2020).

Cara kerja metode ini *quick sort* sebagai berikut:

1. Pilih salah satu elemen vektor sebagai *pivot*, menggunakan strategi pemilihan elemen pertama, elemen terakhir, elemen tengah *Median-of-three*, acak (*random*).
2. Pisahkan vektor menjadi 2 yaitu bagian kiri dan kanan
3. Ulangi langkah 1-2 secara rekursif sehingga terbentuk sub vektor kiri dan sub vektor kanan sampai sub vektor kiri dan sub vektor kanan tidak bisa lagi dipartisi
4. Gabungkan semua *pivot* yang sudah didapatkan dalam bentuk terurut.

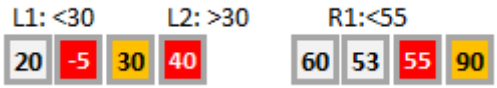
Misalkan akan diurutkan data random berikut menggunakan *Quick Sort* dengan strtategi pemilihan elemen akhir.


Data diurutkan secara *ascending*:

Langkah 1. Ambil 50 sebagai *pivot*

Langkah 2. Partisi vektor menjadi 2. Vektor kiri untuk nilai yang < 50, dan vektor kanan untuk nilai yang > 50

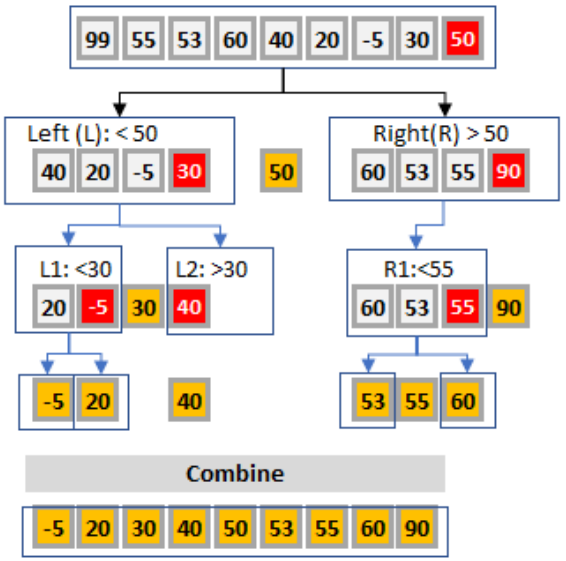
Langkah 3. Ambil 30 di vektor *Left* sebagai *pivot* dan 90 di vektor *Right* sebagai *pivot*. Partisi vektor Left menjadi sub vektor L1 untuk nilai yang < 30, sub vektor L2 untuk nilai>30. Partiri sub vektor Right menjadi 2: R1 untuk nilai<90, dan R2 untuk nilai>90.




Langkah 4. Ambil -5 di sub vektor L1 sebagai *pivot*. Partisi L1 menjadi 2. Ambil 55 sebagai *pivot* di R1, partisi sub vektor R1 menjadi 2.


Langkah 5. Gabungkan/combine semua *pivot* sehingga di dapat data yang terurut berikut:


Gambaran lengkap dari keseluruhan proses quick sort di atas sebagai berikut;




Gambar 10.2: Simulasi *Quick Sort Pivot* elemen terakhir

Implementasi metode *quick sort pivot* elemen terakhir dalam bahasa *python* sebagai berikut:

```
# Program 10.6 - Implementasi Quick Sort

def quick_sort(arr):
    # Basis rekursi: jika panjang array 0 atau 1, sudah
terurut
    if len(arr) <= 1:
        return arr

    pivot = arr[-1]  # Gunakan elemen terakhir sebagai pivot
    left = []        # Elemen lebih kecil dari pivot
    right = []        # Elemen lebih besar atau sama dengan
pivot

    # Bagi elemen ke dalam dua list berdasarkan pivot
    for i in range(len(arr) - 1):
        if arr[i] < pivot:
            left.append(arr[i])
        else:
```

```
            right.append(arr[i])

    # Gabungkan hasil pengurutan rekursif
    return quick_sort(left) + [pivot] + quick_sort(right)

# Data sebelum diurutkan
data = [99, 55, 53, 60, 40, 20, -5, 30, 50]
print("Data sebelum diurutkan:", data)

# Proses pengurutan dengan Quick Sort
sorted_data = quick_sort(data)

# Tampilkan hasil
print("Data setelah diurutkan:", sorted_data)
```

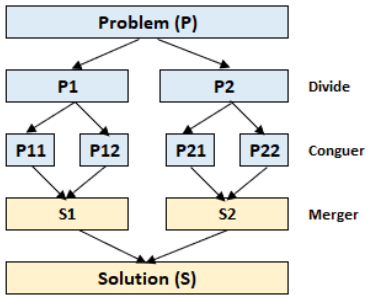
**Output:**

```
Data sebelum diurutkan: [99, 55, 53, 60, 40, 20, -5, 30, 50]
Data setelah diurutkan: [-5, 20, 30, 40, 50, 53, 55, 60, 99]
```

### 10.3.1 Merge Sort: Divide & Conquer

Algoritma *Divide and Conquer* merupakan metode pemecahan masalah untuk menyelesaikan suatu masalah dengan cara membagi masalah utama menjadi submasalah, menyelesaikannya secara individu, dan kemudian menggabungkannya untuk menghasilkan solusi penyelesaian masalah (EeksforGeeks, 2025).

Algoritma *Divide and Conquer* bekerja dengan tiga tahapan yaitu *Divide*, *Conquer* dan *Merge*.




Gambar 10.3: Gambaran Divide and Conquer

1. *Divide* yaitu proses untuk membagi masalah ke dalam beberapa submasalah yang lebih kecil.
2. *Conquer*: Setiap submasalah diselesaikan secara rekursif. Apabila masih ada submasalah masih besar, ulangi proses pembagian sampai masalah tidak bisa dibagi lagi.
3. *Merge*: Solusi dari submasalah yang telah diselesaikan digabungkan kembali untuk membentuk solusi keseluruhan.

Misalkan akan diurutkan isi vektor yang random berikut:

| 99 | 55 | 53 | 60 | 40 | 20 | -5 | 30 | 50 |
|---|---|---|---|---|---|---|---|---|

Penyelesaian:

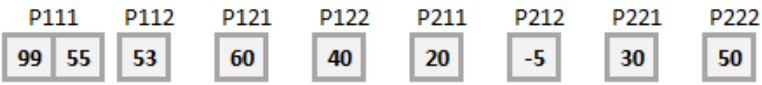
1. Bagi dua vektor data menjadi 2 bagian yaitu P1 dan P2

| P1 | | | | | P2 | | | |
|---|---|---|---|---|---|---|---|---|
| 99 | 55 | 53 | 60 | 40 | 20 | -5 | 30 | 50 |

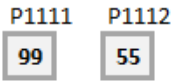
2. Partisi sub vektor P1 menjadi P11, P12 dan P2 menjadi P21, P22.

| P11 | | | P12 | | P21 | | P22 | |
|---|---|---|---|---|---|---|---|---|
| 99 | 55 | 53 | 60 | 40 | 20 | -5 | 30 | 50 |

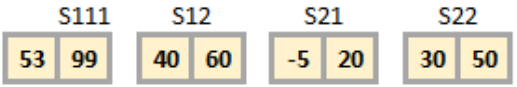
3. Partisi P11 menjadi P111, P112; Partisi P12 menjadi P121, P122; Partisi P21 menjadi P211, P212; Partisi P22 menjadi P221, P222.



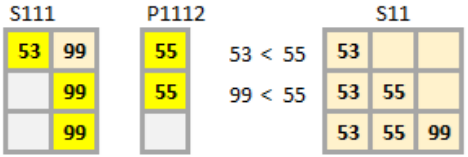

4. Partisi P111 menjadi P1111, P1112



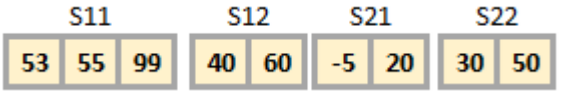

5. Sampai di sini semua sub vektor elemennya tinggal satu yang artinya tidak dapat lagi dipartisi. Langkah berikutnya adalah menggabungkan (*merge*) kembali sub vektor tersebut menggunakan *merge sort algorithm*.
6. Gabung dan urutkan P1111 dan P1112 ke S111, P121 dan P122 ke S12, P211 ke S21, P221 dan P222 ke S22.



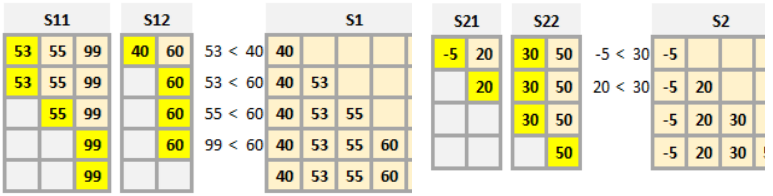

7. Gabungkan dan urutkan S111 dan P1112 ke S11



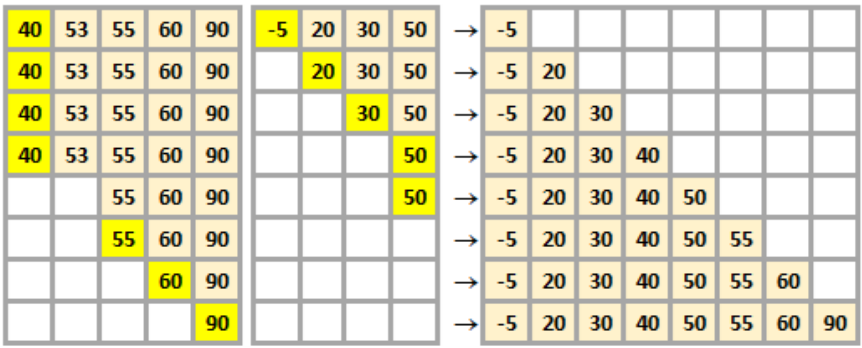

Sehingga di dapatkan empat sub vektor berikut:



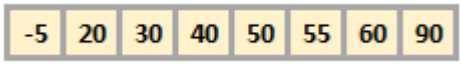

8. Combine S11 dan S12, S21 dan S22 dengan *merge sort*.



9. Combine S1 dan S2 membentuk solusi (S) menggunakan *merge sort*.




Dengan demikian, dapat diperoleh vektor data yang isinya terurut secara *descending*.




Implementasi metode *Merge Sort: Divide & Conquer* dalam bahasa pemrograman *python*:

```
# Program 10.7 - Implementasi Merge Sort (Ascending)

def merge_sort(arr):
    # Basis rekursi: jika panjang array 1 atau kurang, sudah
terurut
    if len(arr) <= 1:
        return arr

    # Pisahkan array menjadi dua bagian
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])   # Rekursif ke kiri
    right_half = merge_sort(arr[mid:])  # Rekursif ke kanan

    # Gabungkan dua bagian terurut
    return merge(left_half, right_half)

def merge(left, right):
    merged = []
```

```
    i = j = 0

    # Gabungkan dua array dengan perbandingan elemen
    while i < len(left) and j < len(right):
        if left[i] < right[j]:  # Untuk urutan ascending
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1

    # Tambahkan sisa elemen yang belum habis
    merged.extend(left[i:])
    merged.extend(right[j:])

    return merged

# Data sebelum diurutkan
data = [99, 55, 53, 60, 40, 20, -5, 30, 50]
print("Data sebelum diurutkan:", data)

# Pemanggilan merge_sort
sorted_data = merge_sort(data)

# Data setelah diurutkan
print("Data setelah diurutkan (ascending):", sorted_data)
```

**Output:**

```
Data sebelum diurutkan: [99, 55, 53, 60, 40, 20, -5, 30, 50]
Data setelah diurutkan (ascending): [-5, 20, 30, 40, 50, 53, 55,
60, 99]
```

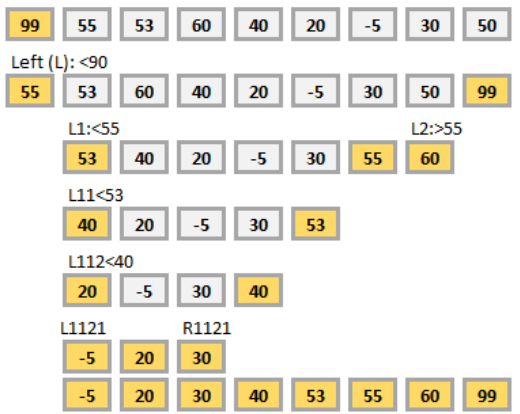
### 10.3.2 Quick Sort: *Pivot* Selection

*Pivot Selection* adalah proses untuk memilih *pivot* sebagai acuan untuk membagi dua vektor data menjadi dua bagian yaitu kiri untuk kumpulan data yang lebih kecil dari *pivot* dan kanan untuk kumpulan data yang lebih besar dari *pivot*. Ada empat cara yang dapat digunakan seseorang dalam memilih acuan(*pivot*) yaitu: Pilih elemen pertama, elemen

terakhir, elemen acak, dan elemen median. Untuk pemilihan elemen terakhir sudah dibahas pada pembahasan sebelumnya.

### *Pivot* Elemen Pertama

*Pivot* elemen pertama adalah cara penentuan *pivot*, dimana elemen yang dijadikan *pivot* adalah elemen pertama dalam vektor ataupun subvektor. Misalkan akan diurutkan data berikut secara *ascending*.




Gambar 10.4: Simulasi *Pivot* elemen pertama

Dalam gambar 10.4 diperlihatkan bahwa setiap tahapan, *pivot* selalu diambil di bagian pertama vektor data, sub vektor Left(L), Sub Vektor L1, L11, L112 sebagai patokan untuk menempatkan data yang lebih kecil di sebelah kiri *pivot* dan dan data yang lebih besar di sebelah kanan *pivot*.

Implementasi *pivot* elemen pertama dalam bahasa *python* sebagai berikut:

```
# Program 10.8 - Quick Sort Menggunakan List Comprehension

def quick_sort(arr):
    # Basis rekursi: jika panjang array 1 atau kurang, sudah
terurut
    if len(arr) <= 1:
        return arr
```

```
    pivot = arr[0]  # Gunakan elemen pertama sebagai pivot
    # Bagi array menjadi elemen <= pivot dan > pivot
    left = [x for x in arr[1:] if x <= pivot]
    right = [x for x in arr[1:] if x > pivot]

    # Gabungkan hasil rekursi
    return quick_sort(left) + [pivot] + quick_sort(right)

# Data sebelum diurutkan
data = [99, 55, 53, 60, 40, 20, -5, 30, 50]
print("Data sebelum diurutkan:", data)

# Proses pengurutan
sorted_data = quick_sort(data)

# Data setelah diurutkan
print("Data setelah diurutkan :", sorted_data)
```

**Output:**

```
Data sebelum diurutkan: [99, 55, 53, 60, 40, 20, -5, 30, 50]
Data setelah diurutkan: [-5, 20, 30, 40, 50, 53, 55, 60, 99]
```

### *Pivot* Random

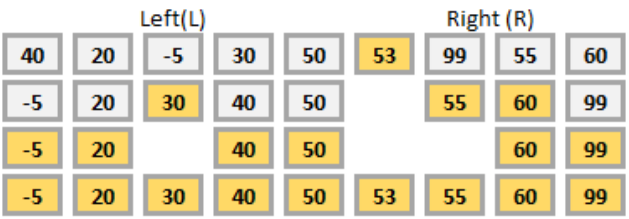
*Pivot* random adalah cara mengambil *pivot* secara acak di setiap tahapan pengurutan *quick sort* sebagai patokan pengelompokan data di sebelah kiri dan kanan *pivot*. Misalkan akan diurutkan data berikut secara *ascending*.


1. Pilih secara random salah satu elemen *data* dalam vektor yang akan menjadi *pivot*.
2. Misalnya 53 sehingga membentuk vektor kiri dan kanan berikut:


3. Ulangi dari no 1 untuk sub vektor kiri dan sub vektor kanan sampai

didapatkan elemen vektor yang terurut.

Secara lengkap tahapan untuk mengurutkan data dengan *pivot* random dapat dilihat pada gambar 10.4.




Gambar 10.4. Simulasi *Pivot Random*

Implementasi *pivot random* dalam bahasa pemrograman *python* sebagai berikut:

```
# Program 10.9 - Quick Sort dengan Pivot Acak

import random

def quick_sort_random_pivot(arr):
    # Basis rekursi
    if len(arr) <= 1:
        return arr

    # Pilih pivot secara acak
    pivot_index = random.randint(0, len(arr) - 1)
    pivot = arr[pivot_index]

    # Bagi elemen selain pivot menjadi dua bagian
    rest = arr[:pivot_index] + arr[pivot_index + 1:]
    left = [x for x in rest if x <= pivot]
    right = [x for x in rest if x > pivot]

    # Rekursi pada bagian kiri dan kanan
    return quick_sort_random_pivot(left) + [pivot] +
quick_sort_random_pivot(right)
# Data sebelum diurutkan
data = [99, 55, 53, 60, 40, 20, -5, 30, 50]
print("Data sebelum diurutkan:", data)
```

```
# Proses pengurutan
sorted_data = quick_sort_random_pivot(data)

# Data setelah diurutkan
print("Data setelah diurutkan :", sorted_data)
```

**Output:**

```
Data sebelum diurutkan: [99, 55, 53, 60, 40, 20, -5, 30, 50]
Data setelah diurutkan  : [-5, 20, 30, 40, 50, 53, 55, 60,
99]
```

### *Pivot* Median

Apa itu median? Median adalah nilai tengah dari sekumpulan data yang terurut. Misalnya:


, maka yang menjadi median adalah 20.

Berikut ini cara mencari *pivot* median dalam vektor data yang akan diurutkan:


Elemen pertama : 45

Elemen terakhir : 37

Elemen tengah : 41

Urutkan ketiga nilai ini menjadi: 


Mediannya adalah: 41

Bagaimana menentukan median jika elemen datanya genap?


Elemen pertama : 32

Elemen terakhir : -2

Elemen tengah : (n/2) = 4/2 = 2 artinya nilai tengah berada pada data ke-2.

Elemen tengah : 44

Urutkan ketiga nilai ini menjadi:

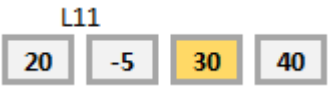

Mediannya adalah: 32

Misalkan akan diurutkan data berikut secara *ascending*.

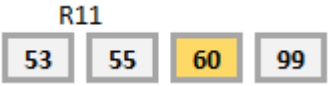

1. *Pivot* median: elemen pertama 99, elemen terakhir 50, dan tengah 40. Setelah diurutkan didapatkan 40, 50, 99. *Pivot*=50 sehingga bentuk subvektor kiri dan kanan menjadi:


2. *Pivot* median sub vektor kiri (L) dan sub vektor kanan (R):

   Sub vektor *Left*(L): elemen pertama 40, tengah 20, terakhir 30. Diurutkan menjadi 20, 30, 40. *Pivot* Sub vektor L adalah: 30.



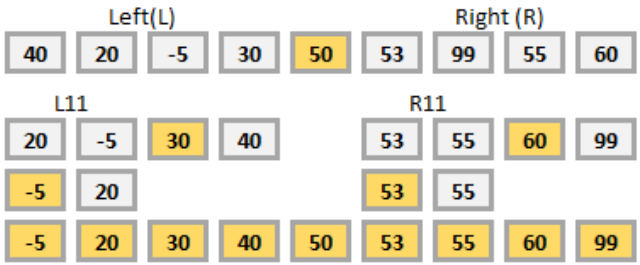

   Sub vektor *Right*(R): elemen pertama 53, tengah 99, terakhir 60. Diurutkan menjadi 53, 60, 99. *Pivot* Sub vektor L adalah: 60.




   Karena L11 dan R11 tinggal 2 elemen maka tidak bisa lagi dicari mediannya. *Pivot*-nya dapat diambil secara random di antara kedua elemen tersebut. L11 menjadi:


R11 menjadi:


Simulasi *pivot* median secara lengkap dapat dilihat pada gambar 10.5.




Gambar 10.5. Simulasi *Pivot Median*

Implementasi *pivot median* ke dalam bahasa pemrograman *python*:

```
# Program 10.10 - Quick Sort dengan Median-of-Three

def median_of_three(arr, low, high):
    mid = (low + high) // 2
    a, b, c = arr[low], arr[mid], arr[high]

    # Kembalikan indeks median dari tiga nilai
    if (a - b) * (c - a) >= 0:
        return low
    elif (b - a) * (c - b) >= 0:
        return mid
    else:
        return high

def quick_sort_median(arr, low, high):
    if low < high:
        # Pilih pivot dengan strategi median-of-three
        pivot_index = median_of_three(arr, low, high)
        # Pindah pivot ke depan
        arr[low],   arr[pivot_index]   =   arr[pivot_index],
arr[low]
```

```
        pivot = arr[low]

        # Partitioning
        i = low + 1
        for j in range(low + 1, high + 1):
            if arr[j] < pivot:
                arr[i], arr[j] = arr[j], arr[i]
                i += 1

        # Tempatkan pivot ke posisi yang tepat
        arr[low], arr[i - 1] = arr[i - 1], arr[low]

        # Rekursif ke kiri dan kanan
        quick_sort_median(arr, low, i - 2)
        quick_sort_median(arr, i, high)

# Data input
data = [99, 55, 53, 60, 40, 20, -5, 30, 50]
print("Sebelum diurutkan:", data)

# Proses pengurutan
quick_sort_median(data, 0, len(data) - 1)

# Tampilkan hasil
print("Setelah diurutkan:", data)
```

**Output:**

```
Sebelum diurutkan: [99, 55, 53, 60, 40, 20, -5, 30, 50]
Setelah diurutkan: [-5, 20, 30, 40, 50, 53, 55, 60, 99]
```

## 10.4 Fungsi Pengurutan Python

Di dalam bahasa pemrograman *python* telah tersedia fungsi bawaan untuk mengurutkan data. Fungsi bawaan *python* untuk mengurutkan data adalah `sorted()` dan fungsi `list.sort()`.

### 10.4.1 sorted() vs list.sort()

Fungsi `sorted()` dan `list_sort()` adalah fungsi yang dapat digunakan untuk mengurutkan isi sebuah `list`. Apa perbedaan antara `sorted()` dan `list_sorted()`.

Fungsi sorted() mengurutkan data dan menghasilkan *list* baru yang sudah terurut tanpa mengubah data asli, dalam artian bahwa hasil pengurutan metode ini akan tersimpan dalam `list` yang baru. Sedangkan hasil pengurutan metode `list.sort()` tidak menghasilkan *list* baru tetapi langsung diurutkan dalam *list* asli (*sort in-place*). Contoh penggunaan fungsi `sorted()`:

```
# Program 10.11 - Pengurutan dengan Fungsi Built-in sorted()

# Data awal
List_angka = [99, 55, 53, 60, 40, 20, -5, 30, 50]

# Pengurutan menggunakan fungsi sorted()
result = sorted(List_angka)

# Tampilkan hasil sebelum dan sesudah pengurutan
print("Data sebelum diurutkan :", List_angka)
print("Data setelah diurutkan :", result)
```

Dalam contoh program pengurutan data di atas pada baris ke-2 ada instruksi `result= sorted(List_angka)` yang artinya mengurutkan data dalam nama List_angka dan hasilnya dimasukkan ke dalam *list result*. Outputnya:

```
Data sebelum diurutkan : [99, 55, 53, 60, 40, 20, -5, 30, 50]
Data setelah diurutkan : [-5, 20, 30, 40, 50, 53, 55, 60, 99]
```

Contoh penggunaan fungsi `list_sort()`:

```
#Program 10.12
List_angka = [99, 55, 53, 60, 40, 20, -5, 30, 50]
print("Data sebelum diurutkan :", List_angka)
List_angka.sort()
print("Data setelah diurutkan :", List_angka)
```

### 10.4.2 Parameter Key dan Reverse

Dalam pengurutan data menggunakan fungsi *sort* bawaan *phyton*, dapat diatur model pengurutan data menggunakan kunci yang disebut dengan parameter. Ada dua macam parameter yang dapat digunakan dalam menurutkan data yaitu *key* dan *reverse*.

#### Key

Parameter *key* digunakan untuk mengatur bagaimana elemen-elemen *list* dapat diurutkan sesuai dengan kebutuhan pengguna. Parameter `key` dapat digunakan pada fungsi-fungsi bawaan *phyton* seperti `sorted()`, `min()`, `max()`, dan `sort()`.

**1. Fungsi `sorted()` dengan `key`**

```
# Program 10.13 - Pengurutan List String dengan dan tanpa
Parameter key

# Data awal
List_buah = ['Semangka', 'Apel', 'Salak', 'Duku', 'Mangga']

# Pengurutan alfabetis (default)
result1 = sorted(List_buah)

# Pengurutan berdasarkan panjang string
result2 = sorted(List_buah, key=len)

# Tampilkan hasil
print('Tanpa Parameter key       :', result1)
print('Dengan Parameter key=len  :', result2)
```

Dalam contoh 10.13 ada dua kali pengurutan yaitu pada baris 3 dan 4. Pengurutan pada baris 3 tidak menggunakan parameter *key*

sehingga isi *list* akan diurutkan berdasarkan abjad, sedangkan pada baris 4 menggunakan *key*=len yang artinya *list* diurutkan berdasarkan panjang setiap isi *list*. Output dari program di atas sebagai berikut:

```
Tanpa Parameter key: ['Apel', 'Duku', 'Mangga', 'Salak',
'Semangka']
Dengan Parameter key=len: ['Apel', 'Duku', 'Salak', 'Mangga',
'Semangka']
```

## 2. Fungsi `min()` dan `max()` dengan `key`

```
# Program 10.14 - Minimum dan Maksimum Berdasarkan Nilai
Absolut
# Data awal
Angka = [83, 15, 88, 62, 58, 20, 56, 59, -7, 100, -150,
0]

# Menentukan nilai minimum dan maksimum berdasarkan nilai
absolut
ResultMin = min(Angka, key=abs)
ResultMax = max(Angka, key=abs)

# Tampilkan hasil
print('Nilai     Minimum     (berdasarkan     abs)     :',
abs(ResultMin))
print('Nilai      Maksimum      (berdasarkan      abs):',
abs(ResultMax))
```

Dalam contoh 10.14 pada baris 3 untuk mencari bilangan minimum dengan *key=abs* artinya semua nilai dalam *list* angka diabsolutkan sehingga nilai-nilai tersebut semuanya bernilai positip, maka nilai terkecil adalah 0. Dan pada baris keempat untuk mencari bilangan terbesar/maksimum menggunakan parameter *key=abs*, sehingga nilai dalam *list* angka dianggap semuanya positip. Nilai yang dihasilkan adalah 150.

**Output:**

```
Nilai Minimum :   0
Nilai Maksimum: 150
```

### 3. Fungsi `sort()` dan key

```
# Program 10.15 - Pengurutan List dengan sort() dan key

# Data awal
List_buah = ['Semangka', 'Apel', 'Salak', 'Duku',
'Mangga']
# Pengurutan secara alfabetis
List_buah.sort()
print('Tanpa Parameter key      :', List_buah)
# Pengurutan berdasarkan panjang string
List_buah.sort(key=len)
print('Dengan Parameter key=len :', List_buah)
```

**Output:**

```
Tanpa Parameter key:    ['Apel', 'Duku', 'Mangga', 'Salak',
'Semangka']
Dengan Parameter key=len:  ['Apel', 'Duku', 'Salak', 'Mangga',
'Semangka']
```

### Reverse

*Reverse* adalah parameter yang digunakan untuk menentukan apakah isi *list* diurutkan secara *ascending* atau *descending*. Parameter *reverse* dapat bernilai *true* atau *false*, dengan nilai default *false* (*ascending*).

```
# Program 10.16 - Pengurutan Descending dengan
sort(reverse=True)

# Data awal
List_data = [99, 55, 53, 60, 40, 20, -5, 30, 50]

# Urutkan secara descending (menurun)
List_data.sort(reverse=True)

# Tampilkan hasil
print("Data setelah diurutkan secara descending:", List_data)
```

**Output:**

```
Data Terurut Descending: [99, 60, 55, 53, 50, 40, 30, 20, -5]
```

# Bab 11
# Studi Kasus dan Implementasi

*"Graph merepresentasikan dunia yang saling terhubung—dari jejaring sosial hingga rute terpendek."* — **Edsger W. Dijkstra**

## 11.1 Pendahuluan Graph dan Aplikasinya

Di era kecerdasan buatan dan pembelajaran mesin yang berkembang pesat, penguasaan struktur data dan algoritma bukan sekadar fondasi akademik, melainkan kebutuhan praktis yang mendesak. Python, sebagai bahasa pemrograman dominan dalam bidang data science dan deep learning, menawarkan sintaks yang elegan dan pustaka yang kaya untuk mengimplementasikan berbagai struktur data dan algoritma secara efisien. Bab ini bertujuan untuk menjembatani teori dengan praktik, khususnya dalam konteks pengembangan model Machine Learning (ML) dan Deep Learning (DL) yang andal.

Machine learning modern tidak hanya bergantung pada algoritma pembelajaran itu sendiri, tetapi juga pada bagaimana data disusun, disimpan, dan diakses. Misalnya, pemilihan struktur data seperti heap, tree, hash table, atau graph dapat secara signifikan memengaruhi efisiensi preprocessing data, pemodelan, dan evaluasi kinerja model. Dalam deep learning, struktur data digunakan secara ekstensif pada tahap pembuatan arsitektur jaringan saraf, penjadwalan pembelajaran, hingga manajemen cache GPU untuk batch processing.

Melalui bab ini, pembaca akan diajak mengeksplorasi implementasi praktis struktur data seperti list, dictionary, set, queue, stack, dan graph dalam Python. Setiap penjelasan didampingi contoh aplikatif: mulai dari pembentukan mini-batch dalam training neural network menggunakan deque, penerapan graph traversal untuk Graph Neural

Networks (GNN), hingga penggunaan priority queue dalam hyperparameter tuning otomatis.

Pendekatan pembelajaran berbasis proyek menjadi tulang punggung bab ini. Contoh-contoh seperti implementasi k-d tree untuk nearest neighbor search dalam klasifikasi gambar, atau penggunaan trie dalam sistem rekomendasi berbasis teks, diharapkan mampu memberikan pengalaman langsung yang memperkuat pemahaman teoretis pembaca. Dengan cara ini, struktur data dan algoritma tidak lagi dipandang sebagai materi abstrak, melainkan sebagai alat konkret dalam membangun sistem cerdas berbasis Python.

Akhirnya, bab ini juga menyinggung bagaimana pemahaman mendalam terhadap algoritma (seperti DFS, BFS, Dijkstra, A*) dapat membantu optimasi pipeline data dalam proyek AI berskala besar, termasuk dalam arsitektur distribusi model deep learning. Dengan menguasai keterampilan ini, pembaca diharapkan mampu tidak hanya menggunakan framework ML seperti TensorFlow atau PyTorch, tetapi juga memahami logika internalnya dan mengoptimalkannya sesuai kebutuhan spesifik proyek.

## 11.2 Dasar-Dasar Struktur Data Python

Python menyediakan struktur data bawaan (built-in) yang efisien dan fleksibel untuk berbagai kebutuhan pemrograman. Dalam konteks machine learning dan deep learning, pemahaman yang kuat terhadap struktur data ini sangat penting untuk manipulasi dataset, pembuatan model, serta proses training dan evaluasi (Yulianto et al., 2023). Empat struktur data dasar yang paling sering digunakan adalah: `List`, `Tuple`, `Dictionary`, dan `Set`.

### 11.2.1 List: Struktur Serbaguna dalam Preprocessing Data

`List` adalah struktur data sekuensial yang memungkinkan penyimpanan elemen dalam urutan tertentu. List mendukung operasi mutasi, yang artinya elemen dapat diubah setelah list dibuat. Dalam

proyek ML, list sering digunakan untuk menyimpan fitur, label, atau hasil prediksi.

```
# Contoh penggunaan list untuk menyimpan hasil prediksi
predictions = [0, 1, 1, 0, 1]
```

Dalam proses *mini-batch training*, list juga digunakan untuk membentuk batch data sebelum dikonversi menjadi tensor:

```
batch_data = [data[i] for i in batch_indices]
```

### 11.2.2 Tuple: Representasi Imutabel untuk Konfigurasi Model

Tuple mirip dengan list, namun bersifat imutabel (tidak bisa diubah setelah dibuat). ni ideal untuk menyimpan konfigurasi tetap seperti dimensi layer atau parameter input/output.

```
input_shape = (224, 224, 3)  # RGB image
```

Tuple banyak digunakan dalam model deep learning ketika mendefinisikan arsitektur layer convolutional (Karno et al., 2023), misalnya di Keras:

```
model.add(Conv2D(32,           kernel_size=(3,           3),
input_shape=input_shape))
```

### 11.2.3 Dictionary: Kunci-Value dalam Data Labeling dan Hyperparameter

Dictionary sangat berguna dalam penyimpanan pasangan kunci-nilai. Ini banyak digunakan dalam ML untuk menyimpan label kelas, konfigurasi hyperparameter, atau metrik evaluasi.

```
# Label klasifikasi
label_map = {'cat': 0, 'dog': 1}
# Hyperparameter
params = {'learning_rate': 0.001, 'batch_size': 32}
```

Dalam proyek NLP, dictionary juga digunakan sebagai word-to-index map dalam tokenisasi.

```
word2idx = {'the': 1, 'cat': 2, 'sat': 3}
```

### 11.2.4 Set: Pengolahan Data Unik dan Operasi Himpunan

`Set` menyimpan elemen yang unik dan tidak berurutan. Sangat berguna saat ingin menghapus duplikasi dari dataset, atau melakukan operasi himpunan seperti *intersection* dan *difference*.

```
labels = [1, 2, 2, 3, 1]
unique_labels = set(labels)  # Output: {1, 2, 3}
```

Dalam data deduplication atau validasi label, set sangat efisien:

```
if set(predictions) == set(ground_truth):
    print("Prediksi mencakup semua label")
```

### 11.2.5 Struktur Data Kompleks: List of Dict, Dict of List

Struktur data sering kali dikombinasikan untuk menciptakan struktur kompleks seperti:

```
# List of dicts (contoh data training)
dataset = [
    {'text': 'saya suka python', 'label': 1},
    {'text': 'belajar machine learning', 'label': 0}
]
# Dict of lists (pengelompokan hasil)
results = {
    'accuracy': [0.88, 0.91, 0.93],
    'loss': [0.45, 0.32, 0.29]
}
```

Penggunaan struktur ini mempermudah manajemen pipeline data dalam berbagai eksperimen ML/DL.

## 11.3 Algoritma Dasar untuk Analisis Data

Algoritma dasar merupakan blok bangunan utama dalam setiap pipeline machine learning. Meskipun banyak pustaka seperti `NumPy`, `pandas`, dan `scikit-learn` telah mengabstraksi banyak operasi, pemahaman terhadap cara kerja algoritma dasar tetap penting agar kita dapat menyesuaikan, mengoptimalkan, atau membangun kembali algoritma sesuai kebutuhan spesifik (Hastomo et al., 2021).

Subbab ini akan membahas beberapa algoritma penting untuk analisis data, yakni: sorting, searching, dan traversal, beserta contoh implementasinya dalam Python.

### 11.3.1 Sorting: Mengurutkan Data untuk Feature Selection atau Visualisasi

Mengurutkan data sangat penting dalam banyak kasus, seperti:

- Menemukan fitur terpenting berdasarkan korelasi atau skor informasi.
- Menampilkan hasil klasifikasi dengan confidence score tertinggi.
- Mengurutkan nilai loss atau akurasi dalam model ensemble

Contoh: Mengurutkan fitur berdasarkan korelasi terhadap target.

```
import pandas as pd

# Dataset dummy
df = pd.DataFrame({
    'feature1': [0.1, 0.4, 0.3],
    'feature2': [0.7, 0.2, 0.5],
    'target': [1, 0, 1]
})

# Hitung korelasi
correlations = df.corr()['target'].drop('target')
sorted_corr = correlations.sort_values(ascending=False)
print(sorted_corr)
Contoh lainnya: Mengurutkan hasil prediksi dari model.
```

```
import numpy as np

probabilities = np.array([0.45, 0.85, 0.65])
sorted_indices = np.argsort(probabilities)[::-1]   # Urutan
descending
print("Ranking prediksi:", sorted_indices)
```

### 11.3.2 Searching: Pencarian Efisien dalam Dataset atau Struktur Model

Algoritma pencarian digunakan dalam:

- Pencarian nilai maksimum/minimum.
- Pencarian threshold optimal dalam klasifikasi.
- Pencarian dalam dictionary word embedding atau tokenisasi.

Contoh: Pencarian biner pada data yang telah diurutkan (Binary Search).

```
def binary_search(arr, x):
    low, high = 0, len(arr)-1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
    return -1

data = [1, 3, 5, 7, 9]
print(binary_search(data, 7))  # Output: 3
Searching juga digunakan saat mengakses vocab index dalam NLP:
word2idx = {'hello': 1, 'world': 2}
idx = word2idx.get('hello', -1)
```

## 11.4 Traversal: Menjelajahi Dataset dan Struktur Model

Traversal sangat penting, terutama untuk:

- Mengecek isi dataset (iterasi).
- Traversal graph (dalam Graph Neural Network).
- Melakukan forward pass dalam jaringan neural.

Traversal umum menggunakan for-loop, tetapi traversal rekursif digunakan untuk tree atau graph.

**Contoh traversal data dan kalkulasi rata-rata label:**

```
dataset = [{'x': [1, 2], 'y': 1}, {'x': [2, 3], 'y': 0}, {'x': [3, 4], 'y': 1}]
avg_label = sum(d['y'] for d in dataset) / len(dataset)
print("Rata-rata label:", avg_label)
```

**Contoh traversal tree (misalnya pada Decision Tree):**

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def in_order_traversal(node):
    if node:
        in_order_traversal(node.left)
        print(node.value)
        in_order_traversal(node.right)

# Binary Tree
root = Node(10)
root.left = Node(5)
root.right = Node(15)
in_order_traversal(root)
```

### 11.4.1 Kombinasi Algoritma untuk Pengolahan Data

Dalam proyek nyata, kita sering menggabungkan sorting, searching, dan traversal.

Contoh: Pencarian dokumen dengan skor TF-IDF tertinggi (top-N retrieval).

```
from sklearn.feature_extraction.text import TfidfVectorizer

docs = ["machine learning is fun", "deep learning is
powerful", "python is great"]
vectorizer = TfidfVectorizer()
tfidf_matrix = vectorizer.fit_transform(docs)
query = tfidf_matrix[0].toarray()

# Hitung skor cosine (dot product sederhana)
scores = tfidf_matrix @ query.T
top_n = scores.toarray().flatten().argsort()[::-1][:2]
print("Dokumen dengan skor tertinggi:", top_n)
```

## 11.5 Struktur Data dalam Preprocessing Machine Learning

Tahap *preprocessing* adalah jembatan antara data mentah dan pembelajaran mesin. Kualitas preprocessing sering kali menentukan keberhasilan model, bahkan lebih dari kompleksitas algoritmanya. Python, dengan pustaka seperti `pandas`, `numpy`, dan `sklearn`, menawarkan berbagai struktur data yang sangat fleksibel dan efisien untuk menangani tugas ini.

Dalam subbab ini, kita akan mengeksplorasi struktur data penting yang umum digunakan dalam preprocessing machine learning: DataFrame, ndarray, serta struktur kompleks seperti dict of arrays, dan pipeline berbasis `sklearn`.

### 11.5.1 DataFrame: Struktur Tabular yang Dominan

`pandas.DataFrame` adalah struktur data dua dimensi mirip spreadsheet yang sangat kuat untuk memanipulasi data terstruktur. Ia mendukung indexing, filtering, grouping, dan operasi statistik secara langsung.

Contoh: Menangani dataset untuk klasifikasi.

```
import pandas as pd

data = pd.DataFrame({
    'usia': [23, 45, 31],
    'gaji': [5000000, 8000000, 6500000],
    'label': [0, 1, 0]
})

# Normalisasi sederhana
data['gaji_norm'] = data['gaji'] / data['gaji'].max()
```

Fitur `.iloc`, `.loc`, `.groupby()`, dan `.apply()` memungkinkan proses pembersihan dan transformasi data yang kompleks secara ringkas dan deklaratif.

### 11.5.2 NumPy ndarray: Struktur Dasar untuk Model ML/DL

`numpy.ndarray` adalah struktur array multidimensi yang menjadi fondasi bagi pustaka seperti `scikit-learn`, `TensorFlow`, dan `PyTorch`.

Contoh: Konversi DataFrame ke `ndarray` untuk model ML

```
import numpy as np

X = data[['usia', 'gaji_norm']].values  # Matrix fitur
y = data['label'].values  # Vektor target
```

Array dua dimensi ini bisa langsung digunakan sebagai input ke model dari `sklearn`.

### 1.1.1 Struktur Kompleks: Dict of Arrays dan List of Tuples

Ketika data menjadi lebih heterogen, struktur seperti dict of arrays menjadi berguna untuk menyimpan data fitur dan metadata secara terpisah.

```
dataset = {
    'features': np.array([[23, 0.6], [45, 1.0], [31, 0.8]]),
    'labels': np.array([0, 1, 0]),
    'names': ['Andi', 'Budi', 'Citra']
}
```

Dalam proyek deep learning, list of tuples atau list of dict sering digunakan untuk membentuk dataset kustom:

```
train_data = [
    ({'image': img1, 'text': 'kucing'}, 0),
    ({'image': img2, 'text': 'anjing'}, 1),
]
```

Struktur semacam ini dapat digunakan dalam `Dataset` PyTorch atau `tf.data.Dataset`.

### 1.1.2 Pipeline Scikit-learn: Struktur Modular untuk Preprocessing

`Pipeline` dari `scikit-learn` adalah struktur tinggi (high-level) untuk merangkai preprocessing dan model dalam satu alur yang rapi dan dapat direproduksi.

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

clf = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', LogisticRegression())
])
clf.fit(X, y)
```

Pipeline ini memungkinkan integrasi preprocessing (scaling, encoding, imputasi) dan modeling tanpa perlu menulis ulang langkah preprocessing di setiap eksperimen.

### 1.1.3 Struktur Data dalam Proyek DL Skala Besar

Dalam deep learning, struktur data perlu memenuhi syarat tambahan: efisiensi memory, kompatibilitas tensor, dan batch processing. `TensorDataset` dan `DataLoader` dalam `PyTorch`, atau `tf.data.Dataset` dalam `TensorFlow`, adalah contoh struktur data khusus untuk deep learning.

```
import torch
from torch.utils.data import TensorDataset, DataLoader

tensor_X = torch.tensor(X, dtype=torch.float32)
tensor_y = torch.tensor(y, dtype=torch.long)

dataset = TensorDataset(tensor_X, tensor_y)
loader = DataLoader(dataset, batch_size=2, shuffle=True)
```

## 11.6 Algoritma untuk Training dan Evaluasi Model

Tahap training dan evaluasi adalah inti dari proses machine learning dan deep learning. Meskipun pustaka modern menyediakan fungsi siap pakai, memahami bagaimana algoritma-algoritma ini bekerja akan memberi kita fleksibilitas dan kendali dalam menyesuaikan pipeline model, terutama saat menghadapi masalah yang tidak standar atau skenario real-world yang kompleks.

Subbab ini membahas beberapa algoritma penting dan implementasinya dalam Python, yakni: training loop, cross-validation, grid search, early stopping, dan metrik evaluasi berbasis distribusi data.

### 11.6.1 Training Loop: Algoritma Dasar dalam Deep Learning

Training loop adalah algoritma yang mengatur bagaimana model belajar dari data secara iteratif. Meskipun framework seperti PyTorch dan TensorFlow menyediakan abstraksi, loop eksplisit tetap digunakan untuk fleksibilitas.

Contoh: Training loop manual dalam PyTorch.

```
import torch
import torch.nn as nn
import torch.optim as optim

model = nn.Linear(10, 1)
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

for epoch in range(10):
    for X_batch, y_batch in loader:
        optimizer.zero_grad()
        output = model(X_batch)
        loss = criterion(output, y_batch)
        loss.backward()
        optimizer.step()
    print(f"Epoch {epoch+1}, Loss: {loss.item():.4f}")
```

Dengan loop ini, kita dapat menyisipkan custom logic seperti regulasi, adversarial training, atau dynamic sampling.

### 1.1.4 Cross-Validation: Evaluasi Generalisasi Model

Cross-validation adalah teknik untuk mengevaluasi performa model secara adil, terutama saat data terbatas. Algoritma dasar dari *k-fold cross-validation* melibatkan:

- Membagi data menjadi k subset.
- Melatih model pada k–1 subset, menguji pada satu sisanya.
- Mengulangi proses dan menghitung rata-rata metrik.

Contoh dengan `scikit-learn`:

```
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier
scores = cross_val_score(RandomForestClassifier(), X, y,
cv=5, scoring='accuracy')
print(f"Rata-rata akurasi: {scores.mean():.3f}")
```

### 1.1.5 Grid Search: Algoritma Pencarian Hyperparameter Optimal

Grid Search mencari kombinasi hyperparameter terbaik berdasarkan metrik evaluasi tertentu.

**Contoh:**

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC

param_grid = {'C': [0.1, 1, 10], 'kernel': ['linear', 'rbf']}
grid = GridSearchCV(SVC(), param_grid, cv=3)
grid.fit(X, y)

print("Best parameters:", grid.best_params_)
```

Untuk proyek skala besar, kita dapat mengganti Grid Search dengan Random Search atau Bayesian Optimization (mis. `optuna`, `ray.tune`).

### 1.1.6 Early Stopping: Algoritma Pencegah Overfitting

Early stopping menghentikan training saat model mulai overfit, biasanya berdasarkan validasi loss yang stagnan atau naik.

Contoh pseudocode (umum dalam DL):

```
best_val_loss = float('inf')
patience, wait = 5, 0

for epoch in range(max_epochs):
    train()
```

```
    val_loss = validate()
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        wait = 0
    else:
        wait += 1
        if wait >= patience:
            print("Early stopping...")
            break
```

Contoh dalam Keras:

```
from tensorflow.keras.callbacks import EarlyStopping

callback = EarlyStopping(monitor='val_loss', patience=3)
model.fit(X_train,       y_train,       validation_split=0.2,
callbacks=[callback])
```

### 1.1.7 Metrik Evaluasi Berbasis Distribusi Data

Evaluasi model tidak selalu cukup dengan *accuracy* saja. Pada data imbalanced, kita perlu mempertimbangkan metrik lain: *precision*, *recall*, *F1-score*, *ROC-AUC*, dan *confusion matrix*.

```
from    sklearn.metrics    import    classification_report,
confusion_matrix
y_pred = model.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

Untuk regresi:

```
from sklearn.metrics import mean_absolute_error, r2_score

print("MAE:", mean_absolute_error(y_test, y_pred))
print("R^2:", r2_score(y_test, y_pred))
```

Dengan memahami dan mengimplementasikan algoritma training dan evaluasi ini, pembaca dapat:

- Merancang strategi eksperimen yang sistematis.

- Menghindari overfitting dan underfitting.
- Memastikan model memiliki performa yang stabil dan terukur saat digunakan pada data nyata

## 11.7 Optimasi dan Kompleksitas Waktu pada Proyek AI

Dalam pengembangan sistem AI yang berskala besar—baik pada tahap pelatihan, inferensi, maupun deployment optimasi menjadi faktor krusial. Optimasi tidak hanya menyangkut pemilihan model, tetapi juga mencakup efisiensi algoritma, struktur data, penggunaan memori, dan waktu komputasi. Konsep seperti kompleksitas waktu dan ruang, serta strategi profiling dan paralelisasi, menjadi alat penting untuk meningkatkan performa dan skalabilitas.

### 11.7.1 Kompleksitas Algoritma: Teori dan Dampaknya di Dunia Nyata

Kompleksitas waktu (`time complexity`) dan kompleksitas ruang (`space complexity`) digunakan untuk mengukur efisiensi algoritma. Dalam proyek AI, pemahaman terhadap kompleksitas membantu memilih pendekatan yang tepat untuk preprocessing data besar, pelatihan model, atau pencarian hyperparameter.

Contoh:

| **Operasi** | **Kompleksitas Waktu** |
|---|---|
| Linear Search | O(n) |
| Binary Search | O(log n) |
| Matrix Multiplication (Naïve) | $O(n^3)$ |
| Forward Pass Neural Network | ~O(n × m) |

Contoh dampak nyata:

- Dataset 1 juta record → pemrosesan $O(n^2)$ akan sangat lambat dibanding $O(n \log n)$.
- Model NLP dengan 300 juta parameter → membutuhkan teknik efisiensi memory dan GPU.

### 11.7.2 Profiling: Mengukur dan Mengoptimalkan Kode Python

Python menyediakan alat profiling seperti `time`, `cProfile`, dan pustaka `line_profiler` untuk menganalisis bagian kode yang lambat atau boros memori.

Contoh profiling sederhana:

```
import time

start = time.time()
# ... kode berat di sini ...
end = time.time()
print(f"Waktu eksekusi: {end - start:.4f} detik")
```

Contoh profiling fungsi:

```
import cProfile

def train_model():
    # fungsi training
    pass

cProfile.run('train_model()')
```

### 11.7.3 Optimasi Struktur Data dan Algoritma

Pemilihan struktur data yang tepat dapat menghemat waktu dan memori:

Tabel 11.1: Pemilihan Sturuktur Data yang Efisien

| Tujuan | Struktur Efisien |
|---|---|
| Lookup cepat | `dict, set (O(1))` |
| Antrian batch | `collections.deque` |
| Akses elemen berurutan | `list, np.ndarray` |
| Penyimpanan besar efisien | `np.memmap, HDF5` |

Contoh penggantian list → `deque` untuk efisiensi:

```
from collections import deque
buffer = deque(maxlen=100)  # lebih efisien untuk append/pop
```

Contoh menghindari loop eksplisit → gunakan NumPy:

```
# Loop lambat
squared = [x**2 for x in arr]

# NumPy cepat
squared = np.square(arr)
```

### 11.7.4 Paralelisme dan GPU Acceleration

Untuk proyek AI berskala besar, paralelisme adalah kunci. Python mendukung multithreading (`threading`), multiprocessing (`multiprocessing`), dan GPU acceleration via CUDA (PyTorch/TensorFlow).

Contoh multiprocessing:

```
from multiprocessing import Pool

def square(x):
    return x * x

with Pool(4) as p:
    result = p.map(square, range(10))

Contoh pemanfaatan GPU di PyTorch:
import torch
```

```
device = torch.device('cuda' if torch.cuda.is_available()
else 'cpu')
model = model.to(device)
inputs = inputs.to(device)
```

GPU sangat bermanfaat untuk:

- Model NLP besar (BERT, GPT) (Yang et al., 2023; Zheng et al., 2021)
- CNN pada image classification (Hastomo, 2021; Hastomo et al., 2022; Hastomo & Bayangkari, 2021; Satyo et al., 2021)
- Deep reinforcement learning (Aini et al., 2022; Sujatna et al., 2023)

### 1.1.8 Strategi Optimasi Proyek AI End-to-End

Berikut adalah pendekatan sistematis untuk mengoptimalkan pipeline AI:

1. **Preprocessing**
   - Gunakan vectorized operation (NumPy, pandas).
   - Simpan data dalam format efisien (Parquet, HDF5).
2. **Training**
   - Gunakan batching dan DataLoader.
   - Manfaatkan GPU dan mixed precision training.
3. **Evaluasi**
   - Gunakan subset data bila perlu.
   - Parallelize evaluasi model dengan joblib.
4. **Deployment**
   - Gunakan ONNX atau TorchScript untuk model ringan.
   - Caching hasil inference untuk query berulang.

Memahami struktur data dan algoritma tidak cukup hanya dari sisi teori pengaruhnya terhadap performa, kecepatan, dan efektivitas sistem AI

sangat nyata. Bab ini menunjukkan bagaimana keputusan kecil (misalnya pemilihan dict alih-alih list) bisa berdampak besar pada efisiensi dan skalabilitas sistem.

Dengan landasan yang kuat ini, pembaca siap untuk membangun pipeline machine learning dan deep learning yang efisien, dapat direproduksi, dan siap produksi.

# Daftar Pustaka


Abdullah, M. F., Hafiza, I., Wahyuni, R., & Syahputra, A. (2023). *Penggunaan Algoritma Bubble Sort dalam Pengurutan Nomor Induk Mahasiswa*. 0–5.

Aho, A. V, Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Pearson Education.

Åkerblom, B., & Castegren, E. (2023). Arrays in practice: An Empirical Study of Array Access Patterns on the JVM. *Art, Science, and Engineering of Programming*, *8*(3), 14:1-14:31. https://doi.org/10.22152/programming-journal.org/2024/8/14

Åkerblom, B., Castegren, E., & Wrigstad, T. (2020). Reference Capabilities for Safe Parallel Array Programming. *Art, Science, and Engineering of Programming*, *4*(1). https://doi.org/10.22152/programming-journal.org/2020/4/1

Beazley, D. M. (2023). *Python Cookbook* (3rd ed.). O'Reilly Media.

Beizer, B. (1995). *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley \& Sons.

Bitner, J. R., & Reingold, E. M. (1975). Interpolation Search-A Log Log N Search. *Communications of the ACM*, *18*(11), 651–656. https://doi.org/10.1145/361002.361007

Brassard, G., & Bratley, P. (1996). *Fundamentals of Algorithmics*. Prentice Hall.

Buana, K. S., & Setiawan, H. (2021). *Pemrograman Terstruktur*. Syiah Kuala University Press. https://books.google.co.id/books?id=aLVsEAAAQBAJ

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

Craig, A. (2019). Binary Search. In *Polynomial Time Algorithms*. Taylor \& Francis. https://taylorandfrancis.com/knowledge/Engineering_and_technology/Computer_science/Binary_search/

Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. (2006). *Algorithms*. McGraw-Hill.

Downey, A. B. (2012). *Think Python: How to Think Like a Computer Scientist* (2nd ed.). O'Reilly Media.

Downey, A. B. (2015). *Think Python: How to Think Like a Computer Scientist* (2nd ed.). O'Reilly Media.

EeksforGeeks. (2025). *Introduction to Divide and Conquer Algorithm*. geeksforgeeks.org.

Evi Lestari Pratiwi. (2020). *Konsep Dasar Algoritma dan Pemrograman Dengan Bahasa Java* (A. Pratomo (ed.); 1st ed.). Poliban Press.

Fowler, M. (2010). *Refactoring: Improving the Design of Existing Code* (2nd ed.). Addison-Wesley Professional.

Ginting, S. H. N., Hansi Effendi, S. K., Marsisno, W., Sitanggang, Y. R. U., Anwar, K., Santiari, N. P. L., Setyowibowo, S., Sigar, T. R., Atho'illah, I., Setyantoro, D., & Smrti, N. N. E. (2023). *Struktur Data*. PT. Mifandi Mandiri Digital.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data Structures and Algorithms in Python*. Wiley.

Gries, D., & Schneider, F. B. (1993). *A Logical Approach to Discrete Math*. Springer-Verlag.

Grus, J. (2022). *Data Science from Scratch*. O'Reilly Media.

Hafiz, S., Ginting, N., Effendi, H., Kumar, S., Marsisno, W., Ria, Y., Sitanggang, U., Anwar, K., Putu, N., Santiari, L., Setyowibowo, S., Sigar, R., Atho'illah, I., Setyantoro, D., Nyoman, N., & Smrti, E. (2024). *Pengantar Struktur Data* (Vol. 1, Issue 01). Penerbit Mifandi Mandiri Digital. https://jurnal.mifandimandiri.com/index.php/penerbitmmd/article/view/39

Harabor, D., & Koenig, S. (2021). Faster Goal Bounding for Jump Point Search. *Journal of Artificial Intelligence Research*, *70*, 1035–1070. https://www.jair.org/index.php/jair/article/download/12255/26657/26035

Kadir, A. (2018). *Dasar Pemrograman Python 3*. Andi Offset.

Kadir, A. (2019a). *Algoritma dan Pemrograman dengan Python*. Elex Media Komputindo.

Kadir, A. (2019b). *Logika Pemrograman Python*. Elex Media Computindo.

Khesya, N. (2021). *Mengenal Flowchart Dan Pseudocode Dalam Algoritma Dan Pemrograman*. https://doi.org/10.31219/OSF.IO/DQ45E

Khomsah, S. (2022). *Algoritma dan Pemograman Menggunakan Python*. Wawasan Ilmu.

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1: Fundamental Algorithms* (3rd ed.). Addison-Wesley.

Kumakiri, M., Bei, L., Tsuji, T., & Higuchi, K. (2006). Flexibly resizable multidimensional arrays. *ICDEW 2006 - Proceedings of the 22nd International Conference on Data Engineering Workshops*, 83–88. https://doi.org/10.1109/ICDEW.2006.62

Kurniawan, R. (2018). *Artikel Struktur Data 135100062*.

Lambert, K. A. (2019). *Fundamentals of Python: First Programs* (2nd ed.). Cengage Learning.

Lubanovic, B. (2019). *Introducing Python: Modern Computing in Simple Packages*. O'Reilly Media.

Lutz, M., & Ascher, D. (2009). *Learning Python* (5th ed.). O'Reilly Media.

Makor, L., Kloibhofer, S., Leopoldseder, D., Bonetta, D., Stadler, L., & Mössenböck, H. (2022). Automatic Array Transformation to Columnar Storage at Run Time. *MPLR 2022 - Proceedings of the 19th International Conference on Managed Programming*

*Languages and Runtimes*, 16–28. https://doi.org/10.1145/3546918.3546919

Marsellus Oton Kadang. (2021). *Buku Bahan Ajar Algoritma dan Pemrograman* (A. K. Muzakkir (ed.); 1st ed., Vol. 4, Issue 1). Humanities Genius.

Martelli, A. (2017). *Python in a Nutshell*. O'Reilly Media.

Moffat, A., & Mackenzie, J. (2022). Immediate-Access Indexing Using Space-Efficient Extensible Arrays. *ACM International Conference Proceeding Series*. https://doi.org/10.1145/3572960.3572984

Nugroho, A. (2019). *Struktur Data*. https://doi.org/10.31219/OSF.IO/6D3QJ

Nurhasanah, T., Hidayat, T., Nurjayadi, N., Muzawi, R., Faizin, N., & Herwin, H. (2015). Analisa Perbandingan Algoritma Pencarian (Searching Algorithm). *Jurnal Teknologi Informasi Dan Industri*, *1*(1), 51–60. https://ejurnal.istp.ac.id/index.php/jtii/article/download/268/228

Oliphant, T. (2020). *Guide to NumPy* (2nd ed.). NumPy Developers.

Putri, M. P., Barovih, G., Azdy, R. A., Yuniansyah, Saputra, A., Sriyeni, Y., Rini, A., & Admojo, F. T. (2022). *Algoritma dan Struktur Data* (1st ed.). WIDINA BHAKTI PERSADA BANDUNG.

Python Software Foundation. (2024). *Downloads \& Documentation*.

Ramadhan, H. (2022). Perbandingan Algoritma Binary Search dan Sequential Search untuk Pencarian Persediaan Stok Barang Berbasis Web. *JIPI (Jurnal Ilmiah Penelitian Dan Pembelajaran Informatika)*, *7*(2), 294–302. https://jurnal.stkippgritulungagung.ac.id/index.php/jipi/article/download/2646/1164

Ray, S., Chatterjee, H. S., Mahato, S. N., & Roy, N. K. (2021). Linear Search. *IETE Journal of Research*. https://taylorandfrancis.com/knowledge/Engineering_and_technology/Computer_science/Linear_search/

Rumapea, Y. Y. (2017). Analisis Perbandingan Metode Algoritma

Quick Sort dan Merge Sort dalam Pengurutan Data terhadap Jumlah Langkah dan Waktu. *Methodika*, *3*(2), 5–9. https://doi.org/10.46880/MTK.V3I2.54

Sandria, Y. A., Nurhayoto, M. R. A., Ramadhani, L., & Harefa, R. S. (2022). *Penerapan Algoritma Selection Sort untuk Melakukan Pengurutan Data dalam Bahasa Pemrograman PHP*.

Saptadi, A. H., & Sari, D. W. (2012). *Analisis algoritma insertion sort, merge sort dan implementasinya dalam bahasa pemrograman c++*. *4*(November), 10–17.

Sari, N., & Rosadi, D. (2017). Analisis Perbandingan Algoritma Sequential Search dan Binary Search dalam Pencarian Data. *Jurnal ELKOM*, *9*(2), 1–8. https://journal.stekom.ac.id/index.php/elkom/article/download/1033/848/3259

Severance, C. (2016). *Python for Everybody: Exploring Data Using Python 3*. CreateSpace Independent Publishing Platform.

Singh, S. K., & Singh, A. K. (2014). Two Way Linear Search Algorithm. *International Journal of Computer Applications*, *107*(21), 6–8. https://www.researchgate.net/publication/284494241_Two_way_Linear_Search_Algorithm

Sofianti, H. A., Manullang, Y. V, Tampubolon, N. A., Naibaho, L. H., & Gunawan, I. (2025). Implementasi Struktur Data Array dan Linked List dalam Pengelolaan Data Mahasiswa. *Menulis: Jurnal Penelitian Nusantara*, *1*(6), 871–877. https://doi.org/10.59435/MENULIS.V1I6.417

Soroush, E., Balazinska, M., & Wang, D. (2011). ArrayStore: A storage manager for complex parallel array processing. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 253–264. https://doi.org/10.1145/1989323.1989351

Sweigart, A. (2015). *Automate the Boring Stuff with Python: Practical Programming for Total Beginners*. No Starch Press.

Sweigart, A. (2019). *Automate the Boring Stuff with Python* (2nd ed.). No Starch Press.

Sweigart, A. (2020). *Automate the Boring Stuff with Python (2nd ed.)*. No Starch Press.

Topperworld.in. (n.d.). *Bubble Sort*. Https://Topperworld.In/.

Tuersley, M. (2004). Data structures. *Cadalyst*, *21*(8), 56.

Van Rossum, G., & Drake, F. L. (2009). *Python Reference Manual*.

Wang, Q. (2025). *The Bathroom Model: A Realistic Approach to Hash Table Algorithm Optimization*. https://arxiv.org/abs/2502.10977

Winardi, S. (2023). *Pemrograman Python Untuk Pemula*. Deepublish Publisher.

Yanti, F., & Eriana, E. S. (2024). *Algoritma Sorting Dan Searching*. EUREKA MEDIA AKSARA.

Zelle, J. (2004). *Python Programming: An Introduction to Computer Science* (2nd ed.). Franklin, Beedle & Associates.

# Biodata Penulis

**Heru Saputra, M.Kom. CITSA, CAIM,** lahir di Banda Aceh pada 01 Mei 1984, merupakan dosen Heru Saputra. merupakan dosen tetap di Univeritas Swasta di kota Padang, Sumatera Barat. Menyelesaikan pendidikan S1 pada Jurusan Sistem Informasi dan melanjutkan S2 pada Jurusan Sistem Informasi di Universitas UPI YPTK Padang. Penulis menekuni beberapa bidang ilmu, seperti Blockchain, Machine Learning, VR/AR dan Artificial Intelligence. Penulis berperan aktif sebagai editor jurnal dan sebagai reviwer pada seminar nasional dan sebagai narasumber di berbagai acara seminal nasional. Penulis telah menyelesaikan beberapa buku yakni Teknologi cloud computing, Sistem Multimedia, Perancangan basis data, Pembelajaran berbasis multimedia, Teknologi jaringan nirkabel, Transformasi Digital: Memahami Internet Of Things, Sistem Basis Data: Konsep, Desain Dan Implementasi. E-mail: h3ru.saputra@gmail.com

**Shevti Arbekti Arman, M.Kom**., lahir di Lubuk Basung pada 16 September 1992, merupakan dosen tetap pada Program Studi Teknologi Informasi, Institut Teknologi dan Bisnis Ahmad Dahlan Jakarta. Ia menyelesaikan pendidikan sarjana dan magister di bidang Teknologi Informasi di Universitas Putra Indonesia "YPTK" Padang. Mata kuliah yang diampu mencakup bidang-bidang inti informatika seperti Sistem Pendukung Keputusan, Sistem Informasi Geografis, Rekayasa Perangkat Lunak, Struktur Data, Bahasa Pemrograman, hingga Multimedia dan Desain Antarmuka. Selain aktif mengajar, beliau juga melaksanakan penelitian dan pengabdian kepada masyarakat secara berkelanjutan, di antaranya di bidang sistem cerdas, multimedia interaktif, serta

teknologi pembelajaran. Beberapa hasil penelitiannya telah dipublikasikan di jurnal nasional dan dipresentasikan dalam forum ilmiah. Pada tahun 2024, beliau memperoleh hibah Penelitian Dosen Pemula (PDP) dari BIMA. Saat ini, Shevti dipercaya sebagai Sekretaris Program Studi Teknologi Informasi di kampus tempatnya mengabdi.

Email: shevtiarman@gmail.com

**Muhammad Fairuzabadi, S.Si., M.Kom**., lahir di Bulukumba, Sulawesi Selatan, pada 26 September 1974. Meraih gelar Sarjana Ilmu Komputer dari Fakultas MIPA Universitas Kristen Immanuel (UKRIM) Yogyakarta pada tahun 1998 dan menyelesaikan program Magister Ilmu Komputer di Universitas Gadjah Mada pada tahun 2006. Saat ini, bekerja sebagai dosen pada Program Sarjana Informatika Fakultas Sains dan Teknologi Universitas PGRI Yogyakarta (UPY). Telah menulis 25 buku dengan topik sistem informasi, artificial intelligence (AI), data science, dan deep learning. Aktif dalam penelitian dan pengabdian kepada masyarakat, dengan fokus pada pengembangan sistem informasi, sistem pakar, dan data science.

Email: fairuz@upy.ac.id

**Ir. Ali Impron, S.Kom., M.Kom.** (Dosen Program Studi Informatika**,** Fakultas Teknik dan Pertanian**,** Universitas Muhammadiyah Sampit) lahir di Grobogan, Jawa Tengah, pada tahun 1983. Penulis adalah seorang dosen dan profesional IT dengan pengalaman lebih dari 17 tahun dalam mengelola layanan IT, internet service provider, dan industri pertambangan. Penulis memiliki keahlian dalam infrastruktur IT, IoT, Data Center, dan Bussiness Intelligence. Saat ini, penulis menjabat sebagai Head of Information Technology di PT. Darma Henwa Tbk, di mana penulis bertanggung jawab untuk mengelola tim IT dan memastikan layanan IT berkualitas tinggi.

Penulis menyelesaikan gelar Sarjana Komputer Prodi Teknik Informatika dari STMIK AKAKOM Yogyakarta, gelar Insinyur dari Prodi Program Profesi Insinyur Institut Teknologi Indonesia dan gelar Magister Teknologi Informasi dari Universitas Teknologi Digital Indonesia, Yogyakarta. Saat ini, penulis sedang menempuh Program Doktor Ilmu Teknik di Universitas Negeri Yogyakarta. Selain perannya di dunia industri, penulis juga tercatat sebagai dosen tetap di Universitas Muhammadiyah Sampit.
Selama karirnya, penulis telah memperoleh berbagai sertifikasi internasional yang mengakui keahliannya di bidang teknologi informasi, termasuk Microsoft Certified IT Professional (MCITP), Cisco Certified Network Associate (CCNA), dan Juniper Networks Certified Associate (JNCIA). Penulis memiliki minat besar dalam penelitian dan integrasi produk IT terbaru dengan proses bisnis, serta memiliki pemahaman mendalam tentang administrasi sistem, keamanan IT, dan jaringan.

Email: ali.impron@gmail.com

**Sugeng Winardi, S.Kom., MT**, lahir di Sleman, Yogyakarta pada 17 Januari 1968. Meraih gelar Sarjana Komputer tahun 2002 dari Program Studi Teknik Informatika STMIK Akakom Yogyakarta dan selanjutnya menyelesaikan Program Magister pada tahun 2012 di Program Studi Magister Teknik Informatika Universitas Atma Jaya Yogyakarta. Selain berperan sebagai pengajar di Program Studi Sistem Informasi Universitas Respati Yogyakarta, juga menjadi konsultas di Perusahaan yang bergerak di bidang IT dan ikut sebagai tenaga ahli yang mengerjakan beberapa proyek pengembangan sistem informasi, master plan dan blue print Teknologi Informasi di instansi pemerintahan daerah, kementerian serta lembaga. Penulis aktif dalam penelitian, pengabdian dan kegiatan sosial yang lainnya.

Email : sugengw@respati.ac.id.

**Ester Lumba, S.Si., M.Kom**., lahir di Palu, Sulawesi Tengah, pada 07 Agustus 1971. Meraih gelar Sarjana Ilmu Komputer dari Fakultas MIPA Universitas Kristen Immanuel (UKRIM) Yogyakarta pada tahun 1998 dan menyelesaikan program Magister Ilmu Komputer di Universitas Budi Luhur Jakarta pada tahun 2008. Saat ini, bekerja sebagai dosen pada beberapa perguruan tinggi swasta di JABODETABEK. Menjadi Trainer independent di instansi pemerintah maupun swasta. Terlibat dalam pengembangan aplikasi perangkat lunak sebagai sistem analis dan project manager untuk aplikasi bisnis.

Email: estlumba@gmail.com

**Firdiyan Syah, S.Kom., M.Kom**., lahir di Yogyakarta pada 31 Juli 1990. Memiliki pengalaman di bidang pengolahan citra digital, kecerdasan buatan, serta pengembangan aplikasi berbasis teknologi. Dengan latar belakang pendidikan Teknik Informatika dari STMIK AMIKOM Yogyakarta dan Universitas Amikom Yogyakarta, telah mengajar selama lebih dari lima tahun di bidang Signal and Image Processing, Technopreneur, dan Human-Computer Interaction. Selain itu, aktif dalam penelitian yang berfokus pada metode deteksi wajah, pengembangan sistem berbasis Android, serta penerapan kecerdasan buatan dalam berbagai aspek teknologi. Saat ini, ia bekerja sebagai dosesn di Universitas PGRI Yogyakarta dan telah mempublikasikan berbagai karya ilmiah di jurnal nasional dan internasional. Selain penelitian, ia juga aktif dalam pengabdian kepada masyarakat melalui berbagai pelatihan digital, seperti pembuatan video pembelajaran untuk sekolah luar biasa dan peningkatan kualitas produk bagi pengrajin lokal melalui digital marketing.

Email: firdiyan@upy.ac.id

**Faqihuddin Al Anshori, S.T., M.Kom,** lahir di Yogyakarta pada 25 Agustus 1989. Merupakan dosen tetap di Univeritas PGRI Yogyakarta di kota Yogyakarta, Darah Istimewa Yogyakarta. Menyelesaikan pendidikan S1 pada Jurusan Teknik Informatika dan melanjutkan S2 pada Jurusan Magister Teknik Inofrmatika di Universitas Ahmad Dahlan Yogyakarta. Saat ini penulis sedang menyelasaikan S3 pada jurusan Pendidikan Teknologi Kejuruan di Universitas Negeri Yogyakarta. Penulis merupakan seorang peneliti di bidang Sistem Informasi yang memiliki spesialisasi dalam peramalan (forecasting) berbasis Artificial Intelligence. Fokus utama saya adalah merancang model-model prediktif yang dapat mengidentifikasi pola, tren, dan proyeksi masa depan dari data sistem informasi, terutama dalam konteks pendidikan, SDM, dan otomasi keputusan strategis. Keahlian saya terletak pada integrasi metode machine learning seperti LSTM, Decision Tree, dan model time series dengan aplikasi nyata di dunia pendidikan dan kerja. Penulis juga terlibat di pengurusan keagamaan serta sosial kemasyarakatan melalui BADKO TKA dan TPA Kec.Depok, Sleman selain itu penulis menjadi Guru Ngaji/TPA di lingkungan Kota Yogyakarta dan Kec. Depok. E-mail: faqihuddinalanshori@upy.ac.id

**Nurirwan Saputra, S.Kom., M.Eng.**, lahir di Serang, Banten, pada 2 Mei 1988. Ia meraih gelar Sarjana Komputer (S.Kom.) dari Jurusan Teknik Informatika, Universitas Islam Indonesia (UII) Yogyakarta pada periode 2007–2012. Selanjutnya, ia melanjutkan pendidikan magister di Program Studi Magister Teknologi Informasi, Universitas Gadjah Mada (UGM), dan menyelesaikannya pada tahun 2015 dengan gelar M.Eng.

Sejak tahun 2015, Nurirwan telah berkarier sebagai dosen di Universitas Pendidikan Yogyakarta (UPY). Ia mengajar berbagai mata kuliah, di antaranya Algoritma dan Pemrograman, Struktur Data, Pemrograman Berorientasi Objek, dan Information Retrieval System. Selain mengajar, ia aktif dalam penelitian

dengan fokus pada bidang Analisis Sentimen dan Natural Language Processing.

Email: nurirwan@upy.ac.id

**Marsellus Oton Kadang, S.Kom., M.T**., lahir di Tikala, Toraja Utara, Sulawesi Selatan, pada 28 Januari 1974. Meraih gelar Sarjana Komputer (S.Kom) dari Jurusan Teknik Informatika Sekolah Tinggi Manajemen Informatika dan Komputer (STMIK) Dipanegara Makassar pada tahun 1999 dan menyelesaikan program Magister Teknik Elektro Konsentrasi Informatika di Universitas Hasanuddin pada tahun 2008. Pada Tahun 2005 terangkat sebagai Pegawai Negeri Sipil (PNS) Kopertis Wilayah IX Sulawesi dan dipekerjakan sebagai dosen DPK di UNDIPA Makassar dari tahun 2005 sampai sekarang. Selain berperan sebagai pengajar, aktif sebagai tenaga ahli dalam berbagai proyek pengembangan sistem informasi, aktif dalam penelitian dan pengabdian kepada masyarakat.

Email: mkadang2000@gmail.com

**Widi Hastomo**, adalah seorang penulis dan peneliti yang aktif di bidang kecerdasan buatan. Dengan latar belakang akademik dan pengalaman profesional, ia telah berkontribusi dalam berbagai penelitian dan publikasi, khususnya terkait bidang deep learning dan computer vision. Selain itu, ia juga terlibat dalam pengembangan teknologi dan inovasi yang mendukung digital health AI, serta aktif dalam berbagai forum akademik dan profesional. Komitmennya terhadap penelitian dan pengembangan terus mendorongnya untuk mengeksplorasi ide-ide baru yang berdampak bagi kemajuan ilmu pengetahuan dan teknologi.
Untuk diskusi lebih lanjut atau kolaborasi, silakan hubungi melalui email: Widie.has@gmail.com

# STRUKTUR DATA dan ALGORITMA DALAM PYTHON

## Panduan Praktis

Buku "Struktur Data dan Algoritma dalam Python: Panduan Praktis" adalah referensi komprehensif yang ditujukan untuk mahasiswa, dosen, dan praktisi teknologi informasi. Buku ini dirancang untuk membantu pembaca memahami konsep dan implementasi struktur data dan algoritma dengan pendekatan praktis menggunakan Python. Materi yang dibahas mencakup dasar-dasar algoritma, struktur data fundamental seperti array, list, set, dan dictionary, serta struktur data lanjutan seperti stack, queue, dan linked list. Python dipilih karena sintaksnya yang sederhana, fleksibel, dan sangat mendukung pembelajaran konsep algoritmik bagi semua level pengguna.

Setiap bab dalam buku ini dilengkapi dengan penjelasan teoretis, ilustrasi, dan contoh kode Python yang bisa langsung dijalankan. Pendekatan ini memastikan pembaca tidak hanya memahami konsepnya, tetapi juga dapat mengimplementasikannya dalam berbagai konteks pengembangan sistem nyata. Di akhir buku, ada pembahasan mengenai teknik pencarian dan pengurutan data, serta studi kasus nyata dalam bidang data science dan machine learning. Dengan demikian, buku ini berfungsi sebagai jembatan antara teori akademik dan kebutuhan industri saat ini, memberikan landasan yang kuat untuk berinovasi dalam teknologi informasi.

yash media.
Follow Us:
yashmedia.id

IKAPI

TEKNOLOGI
ISBN 978-634-7327-38-3
9 786347 327390