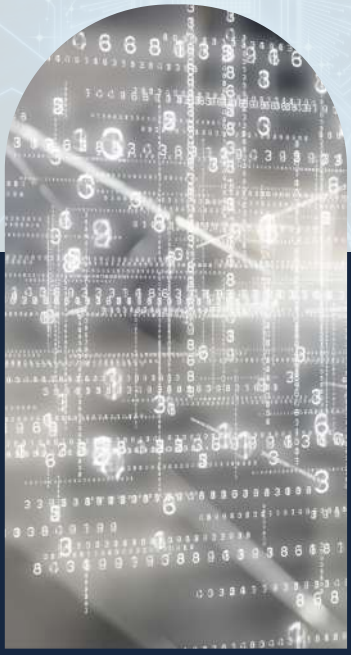




ALGORITMA DAN STRUKTUR DATA

Gusti Eka Yuliastuti, Rudiansyah, Khairunnisa Samosir,
Budanis Dwi Meilani, Sulistyowati,
Sugiarto Cokrowibowo, Putri Ayu Maharani,
Ellbert Hutabri, Kusnaeni

ISBN 978-623-125-499-3

ALGORITMA DAN STRUKTUR DATA

Gusti Eka Yuliastuti
Rudiansyah
Khairunnisa Samosir
Budanis Dwi Meilani
Sulistyowati
Sugiarto Cokrowibowo
Putri Ayu Maharani
Ellbert Hutabri
Kusnaeni

GET PRESS INDONESIA

ALGORITMA DAN STRUKTUR DATA

Penulis : Gusti Eka Yuliastuti
Rudiansyah
Khairunnisa Samosir
Budanis Dwi Meilani
Sulistyowati
Sugiarto Cokrowibowo
Putri Ayu Maharani
Ellbert Hutabri
Kusnaeni

ISBN : 978-623-125-499-3

Editor : Mila Sari, S.ST., M.Si.
Desain Sampul dan Tata Letak : Tri Putri Wahyuni, S.Pd.

Penerbit: GET PRESS INDONESIA
Anggota IKAPI No. 033/SBA/2022

Redaksi:
Jl. Palarik RT 01 RW 06, Kelurahan Air Pacah
Kecamatan Koto Tangah, Padang, Sumatera Barat
website: www.getpress.co.id
email: adm.getpress@gmail.com

Cetakan pertama, Desember 2024

KATA PENGANTAR

Puji syukur kehadirat Tuhan Yang Maha Esa atas segala rahmat dan karunia-Nya sehingga buku ini yang berjudul Algoritma dan Struktur Data dapat tersusun dengan baik. Buku ini disusun untuk memberikan pemahaman mendalam mengenai berbagai konsep dasar dan implementasi struktur data, mulai dari array satu dimensi dan multidimensi, record, pointer, hingga struktur yang lebih kompleks seperti linked list, antrean (queue), algoritma pencarian, dan sorting. Setiap bab disusun secara sistematis, dimulai dari konsep dasar hingga implementasi yang lebih lanjut, dengan harapan dapat menjadi panduan bagi pembaca dalam memahami serta menguasai algoritma dan struktur data secara menyeluruh. Semoga buku ini bermanfaat bagi para mahasiswa, praktisi, maupun siapa saja yang tertarik memperdalam ilmu di bidang algoritma dan struktur data.

Padang, November 2024

Penulis

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

BAB 1
PENGANTAR ARRAY 1-DIMENSI

Oleh Gusti Eka Yuliastuti

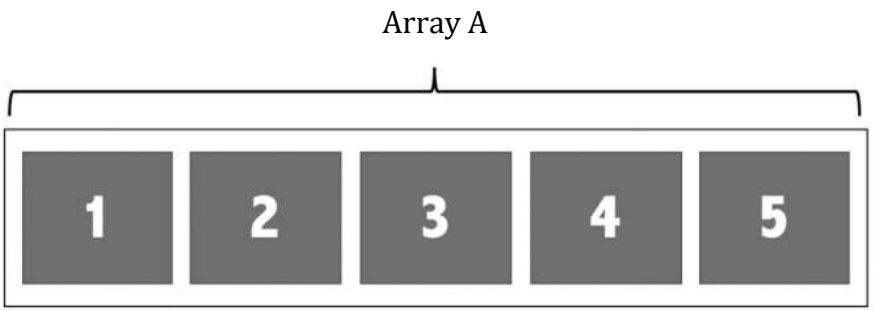
1.1 Pendahuluan

Struktur data merupakan objek yang digunakan program komputer untuk mengatur lebih dari satu informasi pada satu waktu. Objek dalam tanda kurung siku ”[ ]” adalah struktur data yang disebut sebagai array (Sindar, 2018). Array dapat diartikan sebagai suatu kumpulan data yang berjajar dalam satu baris. Array merupakan tipe struktur data yang memiliki komponen dalam jumlah yang tetap (Tarigan, 2022). Fungsi dari array yakni untuk mempermudah dalam hal penyimpanan data.

Array juga merupakan salah satu tipe data terstruktur yang dapat digunakan pada beberapa bahasa pemrograman seperti PHP, Pascal, Java, C++, Python, dan lain sebagainya (Putri et al., 2022). Dalam bahasa pemrograman, array merupakan variabel sejenis yang berada pada satu baris sedemikian rupa sehingga alamatnya saling bersambung atau bersebelahan (Soetanto, 2021).

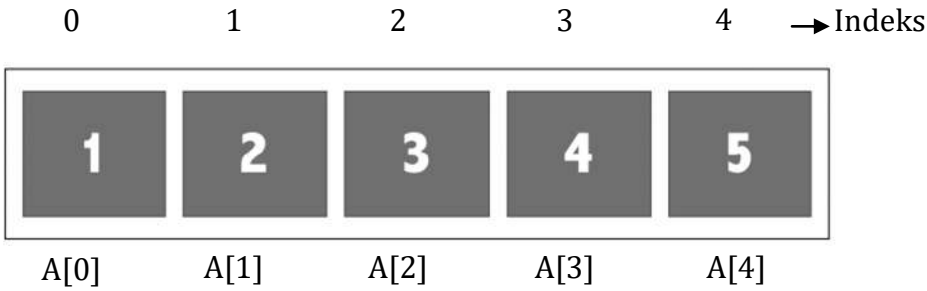
Terdapat berbagai jenis array yang dapat digunakan dalam bahasa pemrograman apapun seperti array satu dimensi (contoh: list dan vector) , array dua dimensi (tabel dan matriks) dan array multi dimensi (matriks 3 dimensi) (Siregar et al., 2017). Array satu dimensi adalah struktur data yang menyimpan sekumpulan elemen dari tipe data yang sama dalam urutan linier atau satu baris. Setiap elemen dalam array diakses

menggunakan indeks yang menunjukkan urutan posisi elemen tersebut dalam array (Sindar, 2018). Array satu dimensi merupakan jenis array yang paling sederhana dan sering digunakan.



Gambar 1. 1. Contoh Array Satu Dimensi
Sumber: (Waluyanti, 2018)

Pada Gambar 1.1 menunjukkan bahwa A merupakan suatu array yang memiliki 5 buah elemen di dalamnya. Kelima elemen tersebut disediakan untuk menampung nilai dengan tipe data yang ditentukan. Elemen dengan urutan pertama biasanya diberi indeks 0, elemen berikutnya diberi indeks 1 dan seterusnya hingga elemen terakhir dalam array. Indeks array selalu dimulai dari 0 sehingga untuk array dengan panjang n akan memiliki indeks terakhir n-1. Indeks pada array digunakan untuk menandai tiap nilai yang tersimpan dalam elemen. Ilustrasi terkait pemberian indeks pada elemen dalam array seperti ditunjukkan pada Gambar 1.2. Dalam satu blok memori akan dialokasikan untuk array menampung semua elemennya. Tiap elemen array dapat diakses dalam waktu yang konstan dengan menggunakan indeks elemen tertentu (Santoso, 2021).



Gambar 1. 2. Indeks pada Array Satu Dimensi
Sumber: (Waluyanti, 2018)

1.2 Tipe Data Array

Tipe data array merupakan tipe data yang berisi kumpulan tipe data yang lain. Tipe data array mampu menampung data yang banyak dengan tipe data yang sama. Dengan penggunaan array, proses penyimpanan data ke dalam variabel menjadi lebih efisien dan mudah meskipun jumlah datanya banyak (Tarigan, 2022).

Implementasi pada hampir semua bahasa pemrograman biasanya variabel array harus terdiri dari satu tipe data saja. Array tidak dapat dibuat dari berbagai macam tipe data. Misalnya isian dari array adalah bilangan bulat maka hanya bisa menggunakan tipe data integer. Adapun ciri khas tipe data array adalah dengan penggunaan tanda kurung siku ”[ ]“ (Sindar, 2018).

Pada dasarnya array dapat menyimpan berbagai tipe data tergantung dari bahasa pemrograman apa yang digunakan (Waluyanti, 2018). Berikut adalah beberapa tipe data umum yang dapat dimasukkan ke dalam array seperti ditunjukkan pada Tabel 1.1.

Tabel 1. 1. Tipe Data Array

No.	Tipe Data	Fungsi	Contoh
1	Integer	Untuk menyimpan bilangan bulat	[1,2,3,4,5]
2	Float	Untuk menyimpan bilangan desimal	[1.5, 2.8, 4.2]
3	String	Untuk menyimpan teks atau satu kata	[kucing, kelinci, hamster]
4	Character	Untuk menyimpan satu karakter	[‘a’, ‘b’, ‘c’]
5	Boolean	Untuk menyimpan nilai benar atau salah	[true, false]
6	Array atau List	Untuk menyimpan array lain atau bisa disebut array multidimensional	[[1,2],[3,4],[5,6]]
7	Object	Untuk menyimpan sebuah objek (berlaku pada beberapa bahasa pemrograman seperti javascript, Python dan Java)	[{“nama”: “gusti”}, {“nama”: “eka”}]
8	Null atau Undefined	Pada beberapa bahasa pemrograman mendukung nilai null masuk ke dalam array	[10, 15, 20, null]

Pada bahasa pemrorgraman yang mendukung array dengan tipe data campuran, seperti Python dan JavaScript, dapat memungkinkan berbagai tipe data berada dalam satu array yang sama. Sedangkan pada bahasa pemrograman lain biasanya membatasi array untuk menyimpan hanya satu tipe data tertentu.

1.3 Deklarasi dan Inisialisasi Array

Array perlu dideklarasikan seperti layaknya sebuah variabel. Tujuan dari deklarasi dan inisialisasi ini yakni untuk menyediakan ruang memori serta menetapkan nilai awalnya (Santoso, 2021). Terdapat berbagai cara dalam mendeklarasikan array tergantung pada bahasa pemrograman yang digunakan. Berikut beberapa contoh deklarasi array pada beberapa bahasa pemrograman.

1. Bahasa Pemrograman C++
   Pada C++, array dideklarasikan dengan menentukan tipe data elemen dan ukuran array di dalam kurung siku seperti ditunjukkan pada Gambar 1.3 (Sindar, 2018).

```cpp
int arr[5];          // Deklarasi array integer dengan 5 elemen
float scores[10];    // Deklarasi array float dengan 10 elemen
char letters[26];    // Deklarasi array char dengan 26 elemen
```

Gambar 1. 3. Deklarasi Array pada C++

   Untuk inisialisasi saat deklarasi array di C++ seperti ditunjukkan pada Gambar 1.4.

```cpp
int arr[5] = {1, 2, 3, 4, 5};  // Inisialisasi array integer dengan nilai
```

Gambar 1. 4. Inisialisasi Array pada C++

2. Bahasa Pemrograman Java
   Pada Java, array dideklarasikan dengan mendefinisikan tipe data dan menambahkan tanda kurung siku setelah tipe data seperti ditunjukkan pada Gambar 1.5 (Putri et al., 2022).

```java
int[] arr = new int[5];          // Deklarasi array integer dengan 5 elemen
double[] values = new double[10]; // Deklarasi array double dengan 10 elemen
char[] letters = new char[26];    // Deklarasi array char dengan 26 elemen
```

Gambar 1. 5. Deklarasi Array pada Java

Untuk inisialisasi saat deklarasi array di Java seperti ditunjukkan pada Gambar 1.6.

```java
int[] arr = {1, 2, 3, 4, 5};     // Inisialisasi array integer dengan nilai
```

Gambar 1. 6. Inisialisasi Array pada Java

3. Bahasa Pemrograman Python
   Pada Python, array biasanya diimplementasikan sebagai list dan dapat dideklarasikan dengan menggunakan tanda kurung siku seperti ditunjukkan pada Gambar 1.7.

```python
arr = [0] * 5         # Deklarasi list dengan 5 elemen, semua elemen bernilai 0
scores = [0.0] * 10  # Deklarasi list dengan 10 elemen float, semua elemen bernilai 0.0
letters = [''] * 26  # Deklarasi list dengan 26 elemen string kosong
```

Gambar 1. 7. Deklarasi Array pada Python

Untuk inisialisasi saat deklarasi array di Python seperti ditunjukkan pada Gambar 1.8.

```python
arr = [1, 2, 3, 4, 5]   # Inisialisasi list dengan nilai
```

Gambar 1. 8. Inisialisasi Array pada Python

4. Bahasa Pemrograman JavaScript
   Pada JavaScript, array dideklarasikan menggunakan tanda kurung siku dan dapat dinisialisasi langsung dengan nilai seperti ditunjukkan pada Gambar 1.9.

```javascript
let arr = new Array(5);    // Deklarasi array dengan 5 elemen undefined
let scores = new Array(10); // Deklarasi array dengan 10 elemen undefined
let letters = new Array(26); // Deklarasi array dengan 26 elemen undefined
```

Gambar 1. 9. Deklarasi Array pada JavaScript

   Untuk inisialisasi saat deklarasi array di Python seperti ditunjukkan pada Gambar 1.10.

```javascript
let arr = [1, 2, 3, 4, 5]; // Inisialisasi array dengan nilai
```

Gambar 1. 10. Inisialisasi Array pada JavaScript

5. Bahasa Pemrograman C#
   Pada C#, array dideklarasikan dengan mendefinisikan tipe data dan menambahkan tanda kurung siku seperti ditunjukkan pada Gambar 1.11.

```csharp
int[] arr = new int[5];         // Deklarasi array integer dengan 5 elemen
double[] values = new double[10]; // Deklarasi array double dengan 10 elemen
char[] letters = new char[26];   // Deklarasi array char dengan 26 elemen
```

Gambar 1. 11. Deklarasi Array pada C#

   Untuk inisialisasi saat deklarasi array di C# seperti ditunjukkan pada Gambar 1.12.

```csharp
int[] arr = {1, 2, 3, 4, 5};    // Inisialisasi array integer dengan nilai
```

Gambar 1. 12. Inisialisasi Array pada C#

6. Bahasa Pemrograman PHP
   Pada PHP, array dideklarasikan menggunakan tanda kurung siku atau fungsi array() seperti ditunjukkan pada Gambar 1.13.

```swift
var arr: [Int] = Array(repeating: 0, count: 5)   // Deklarasi array integer dengan 5 eleme
var scores: [Double] = Array(repeating: 0.0, count: 10) // Deklarasi array double dengan 1
var letters: [Character] = Array(repeating: " ", count: 26) // Deklarasi array char dengan
```

Gambar 1. 13. Deklarasi Array pada PHP

Untuk inisialisasi saat deklarasi array di PHP seperti ditunjukkan pada Gambar 1.14.

```swift
var arr = [1, 2, 3, 4, 5]   // Inisialisasi array integer dengan nilai
```

Gambar 1. 14. Inisialisasi Array pada PHP

1.4 Pengaksesan Elemen Array

Pengaksesan elemen array satu dimensi melibatkan penggunaan indeks untuk mendapatkan nilai pada posisi tertentu dalam array (Sindar, 2018). Berikut beberapa contoh pengaksesan elemen array dalam berbagai bahasa pemrograman.

1. Bahasa Pemrograman C++
   Pada C++, array dideklarasikan dengan tipe data elemen dan ukuran array di dalam kurung siku. Indeks pada array dimulai dari 0. Contoh pengaksesan elemen dalam array seperti ditunjukkan pada Gambar 1.15.

```cpp
#include <iostream>
using namespace std;

int main() {
    // Mendefinisikan array
    int array[] = {10, 20, 30, 40, 50};

    // Mengakses elemen pertama
    cout << array[0] << endl;  // Output: 10

    // Mengakses elemen ketiga
    cout << array[2] << endl;  // Output: 30

    // Mengakses elemen terakhir
    cout << array[sizeof(array)/sizeof(array[0]) - 1] << endl;  // Output: 50

    return 0;
}
```

Gambar 1. 15. Pengaksesan Elemen Array pada C++

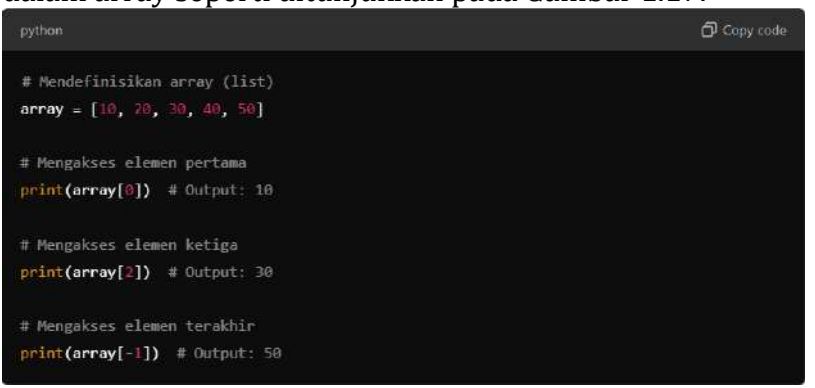
2. Bahasa Pemrograman Java

   Pada Java, array dideklarasikan dengan tipe data elemen dan elemen dimulai dari 0. Contoh pengaksesan elemen dalam array seperti ditunjukkan pada Gambar 1.16.

```java
public class Main {
    public static void main(String[] args) {
        // Mendefinisikan array
        int[] array = {10, 20, 30, 40, 50};

        // Mengakses elemen pertama
        System.out.println(array[0]);  // Output: 10

        // Mengakses elemen ketiga
        System.out.println(array[2]);  // Output: 30

        // Mengakses elemen terakhir
        System.out.println(array[array.length - 1]);  // Output: 50
    }
}
```

Gambar 1. 16. Pengaksesan Elemen Array pada Java

3. Bahasa Pemrograman Python
   Pada Python, array umumnya diimplementasikan sebagai list dan indeks dimulai dari 0. Contoh pengaksesan elemen dalam array seperti ditunjukkan pada Gambar 1.17.



Gambar 1. 17. Pengaksesan Elemen Array pada Python

4. Bahasa Pemrograman JavaScript
   Pada JavaScript, array menggunakan indeks yang dimulai dari nol. Contoh pengaksesan elemen dalam array seperti ditunjukkan pada Gambar 1.18.

Gambar 1. 18. Pengaksesan Elemen Array pada JavaScript

5. Bahasa Pemrograman C#
   Pada C#, array menggunakan indeks yang dimulai dari nol. Contoh pengaksesan elemen dalam array seperti ditunjukkan pada Gambar 1.19.

```csharp
using System;

class Program {
    static void Main() {
        // Mendefinisikan array
        int[] array = {10, 20, 30, 40, 50};

        // Mengakses elemen pertama
        Console.WriteLine(array[0]);  // Output: 10

        // Mengakses elemen ketiga
        Console.WriteLine(array[2]);  // Output: 30

        // Mengakses elemen terakhir
        Console.WriteLine(array[array.Length - 1]);  // Output: 50
    }
}
```

Gambar 1. 19. Pengaksesan Elemen Array pada C#

6. Bahasa Pemrograman PHP
   Pada PHP, array menggunakan indeks yang dimulai dari nol. Contoh pengaksesan elemen dalam array seperti ditunjukkan pada Gambar 1.20.

```php
<?php
// Mendefinisikan array numerik
$array = array(10, 20, 30, 40, 50);

// Mengakses elemen pertama
echo $array[0];  // Output: 10

// Mengakses elemen ketiga
echo $array[2];  // Output: 30

// Mengakses elemen terakhir
echo $array[count($array) - 1];  // Output: 50
?>
```

Gambar 1. 20. Pengaksesan Elemen Array pada PHP

Hal yang Perlu Diperhatikan, yakni:

1. **Indeks Dimulai dari 0:** Untuk semua bahasa pemrograman yang digunakan seperti pada contoh secara default menggunakan indeks yang dimulai dari 0. Artinya, elemen pertama berada di indeks 0, elemen kedua di indeks 1, dan begitu seterusnya.
2. **Indeks Negatif:** Pada bahasa pemrograman Python, kita dapat menggunakan indeks negatif untuk mengakses elemen dari akhir array (misalnya array[-1] untuk elemen terakhir pada array tersebut).
3. **Pengecekan Batas:** Selalu pastikan bahwa indeks yang akan diakses berada dalam rentang valid untuk array tersebut. Misalnya, mengakses elemen dengan indeks negatif atau indeks yang lebih besar dari ukuran array akan menghasilkan kesalahan atau nilai error.

1.5 Modifikasi Elemen Array

Modifikasi elemen array atau mengubah nilai elemen pada array satu dimensi dapat dilakukan dengan cara yang sama seperti pengaksesan elemen. Namun, setelah sintaksis pengaksesan elemen, kita perlu memberikan nilai baru atau merujuk pada indeks elemen tertentu yang akan diubah nilainya. Berikut beberapa contoh modifikasi elemen array dalam berbagai bahasa pemrograman.

1. Bahasa Pemrograman C++
   Pada C++, kita dapat mengakses elemen array menggunakan indeks dan mengubah nilainya. Contoh modifikasi elemen dalam array seperti ditunjukkan pada Gambar 1.21.

```cpp
#include <iostream>
using namespace std;

int main() {
    // Membuat array
    int arr[] = {1, 2, 3, 4, 5};

    // Mengubah nilai pada indeks ke-2
    arr[2] = 10;

    // Output
    for(int i = 0; i < 5; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}
```

Gambar 1. 21. Modifikasi Elemen Array pada C++

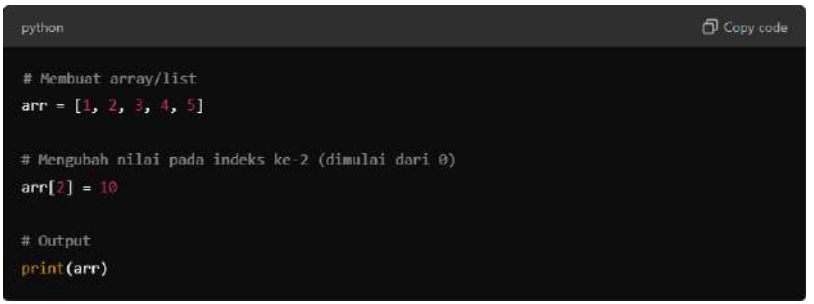
2. Bahasa Pemrograman Java

   Pada Java, kita dapat mengakses elemen array menggunakan indeks dan mengubah nilainya. Contoh modifikasi elemen dalam array seperti ditunjukkan pada Gambar 1.22.

```java
public class Main {
    public static void main(String[] args) {
        // Membuat array
        int[] arr = {1, 2, 3, 4, 5};

        // Mengubah nilai pada indeks ke-2
        arr[2] = 10;

        // Output
        for (int i : arr) {
            System.out.print(i + " ");
        }
    }
}
```

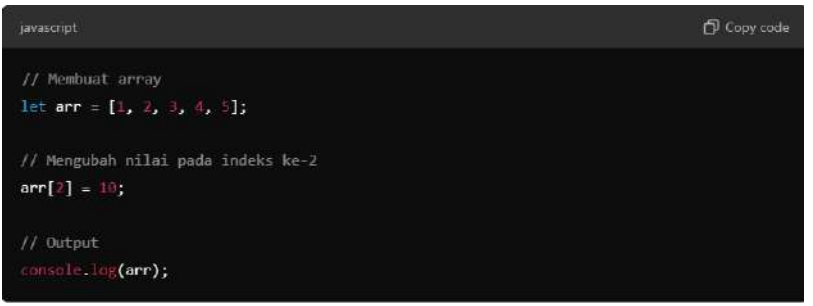
Gambar 1. 22. Modifikasi Elemen Array pada Java

3. Bahasa Pemrograman Python
   Pada Python, kita dapat mengakses elemen array menggunakan indeks dan mengubah nilainya. Contoh modifikasi elemen dalam array seperti ditunjukkan pada Gambar 1.23.



Gambar 1. 23. Modifikasi Elemen Array pada Python

4. Bahasa Pemrograman JavaScript
   Pada JavaScript, kita dapat mengakses elemen array menggunakan indeks dan mengubah nilainya. Contoh modifikasi elemen dalam array seperti ditunjukkan pada Gambar 1.24.



Gambar 1. 24. Modifikasi Elemen Array pada JavaScript

5. Bahasa Pemrograman C#
   Pada C#, kita dapat mengakses elemen array menggunakan indeks dan mengubah nilainya. Contoh modifikasi elemen dalam array seperti ditunjukkan pada Gambar 1.25.

```csharp
using System;

class Program {
    static void Main() {
        // Membuat array
        int[] arr = {1, 2, 3, 4, 5};

        // Mengubah nilai pada indeks ke-2
        arr[2] = 10;

        // Output
        foreach (int i in arr) {
            Console.Write(i + " ");
        }
    }
}
```

Gambar 1. 25. Pengaksesan Elemen Array pada C#

6. Bahasa Pemrograman PHP

   Pada PHP, kita dapat mengakses elemen array menggunakan indeks dan mengubah nilainya. Contoh modifikasi elemen dalam array seperti ditunjukkan pada Gambar 1.26.

```php
<?php
// Membuat array
$arr = [1, 2, 3, 4, 5];

// Mengubah nilai pada indeks ke-2
$arr[2] = 10;

// Output
print_r($arr);
?>
```

Gambar 1. 26. Modifikasi Elemen Array pada PHP

1.6 Operasi Dasar pada Array

Selain deklarasi, inisialisasi dan modifikasi, terdapat beberapa operasi dasar yang dapat dilakukan pada array satu dimensi antara lain:

1. **Penghapusan elemen (delete)** → Elemen dalam array dapat dihapus menggunakan indeks tertentu. Dalam bahasa yang tidak mendukung array dinamis, ini memerlukan pembuatan array baru tanpa ada elemen yang dihapus.
2. **Penambahan elemen (insert atau append)** → Elemen baru dapat ditambahkan ke dalam array pada indeks tertentu (hanya mungkin di bahasa yang mendukung array dinamis, seperti Python atau Java). Insert merupakan penyisipan atau penambahan elemen pada posisi tertentu, sedangkan append merupakan penambahan elemen pada akhir array.
3. **Penghitungan panjang array (length)** → Operasi ini mengembalikan hasil penghitungan jumlah elemen yang ada dalam array (Sindar, 2018).
4. **Pencarian elemen (search)** → Pencarian ini dilakukan untuk menemukan indeks dengan nilai tertentu dalam array.
5. **Traversal array (iterasi)** → Traversal merupakan proses mengakses setiap elemen dalam array satu per satu dengan cara mengunjungi setiap elemen dalam array untuk melakukan operasi, seperti mencetak nilai atau menghitung total. Ini sering dilakukan menggunakan sintaksis seperti for atau while.
6. **Pengurutan elemen (sort)** → Pengurutan elemen berarti menyusun atau mengatur elemen pada array dalam urutan tertentu. Contoh: urutan menaik (ascending) atau menurun (descending).
7. **Penggabungan array (concatenation)** → Dua array atau lebih dapat digabungkan menjadi satu array.

8. **Pembalik urutan (reverse)** → Operasi ini dapat membalik urutan elemen-elemen dalam array.
9. **Pencacahan array (slice)** → Operasi ini memungkinkan pengambilan subset dari array dengan batas indeks tertentu.
10. **Penyalinan elemen (copy)** → Operasi dasar ini dapat digunakan untuk menyalin elemen dari satu array ke array lain.

DAFTAR PUSTAKA

Putri, M. P., Barovih, G., Azdy, R. A., Yuniansyah, Y., Saputra, A., Sriyeni, Y., Rini, A., & Admojo, F. T. (2022). *Algoritma dan Struktur Data*. Widina.

Santoso, J. T. (2021). STRUKTUR DATA dan ALGORITMA (Bagian 1). In *Penerbit Yayasan Prima Agus Teknik* (Issue Bagian 1). http://penerbit.stekom.ac.id/index.php/yayasanpat/article/view/288

Sindar, A. (2018). *Struktur Data dan Algoritma dengan C++* (Issue December). CV. AA Rizky.

Siregar, I., Tarigan, U., Siregar, D. A., Sari, R. M., Tambunan, M. M., & Nasution, T. H. (2017). Determination of Optimal Production Planning in Palm Oil Company (Case Study). *International Conference on Industrial Engineering and Applications*.

Soetanto, H. (2021). *Struktur Data Dasar-Dasar Pemrograman*. Universitas Budi Luhur. http://dx.doi.org/10.31219/osf.io/rtupn

Tarigan, W. (2022). *Algoritma Pemrograman dan Struktur Data*.

Waluyanti, S. (2018). Bab 8 Array 1 Dimensi. In *Modul Digital Mata Kuliah: Algoritma dan Pemrograman* (Issue 2008, pp. 8–10). Sistem Informasi UHW Perbanas.

BAB 2
PENGANTAR ARRAY MULTI-DIMENSI

Oleh Rudiansyah

2.1 Pendahuluan

Dalam pemrograman, sering kali kita perlu menyimpan dan memanipulasi data dalam bentuk tabel atau matriks. Misalnya, ketika kita ingin menyimpan data nilai siswa dalam berbagai mata pelajaran, atau merepresentasikan koordinat dalam ruang dua dimensi atau tiga dimensi. Untuk itu, array satu dimensi tidak cukup efisien atau fleksibel. Oleh karena itu, array multidimensi menyediakan struktur yang lebih sesuai untuk menangani data tersebut. Array multidimensi adalah struktur data yang digunakan untuk menyimpan sejumlah elemen yang diorganisir dalam lebih dari satu dimensi. Jika array satu dimensi menyimpan elemen dalam bentuk daftar linier (seperti array atau list biasa), array multidimensi memungkinkan penyimpanan data dalam bentuk tabel atau grid, yang lebih kompleks. Contoh paling umum dari array multidimensi adalah array dua dimensi, yang digunakan untuk merepresentasikan data dalam bentuk matriks atau tabel dengan baris dan kolom. Array multidimensi adalah array yang memiliki lebih dari satu dimensi, yang paling sederhana adalah array dua dimensi, tetapi dapat berkembang menjadi tiga dimensi atau lebih. Array multidimensi sangat penting dalam pemrograman karena dapat

digunakan untuk mewakili data yang lebih kompleks, seperti matriks, tabel, atau bahkan gambar dan volume.

Contoh Aplikasi Array Multidimensi:

1. Matriks dan Aljabar Linier: Array dua dimensi digunakan untuk operasi matematika seperti matriks, yang sering diterapkan dalam grafik komputer, pemrosesan sinyal, dan pengenalan pola.
2. Pengolahan Gambar: Gambar digital dapat diwakili sebagai array dua dimensi dari piksel, di mana setiap elemen menyimpan informasi warna pada titik tertentu dalam gambar.
3. Data Geospasial atau 3D: Dalam simulasi fisik atau rendering grafis, kita sering kali bekerja dengan data tiga dimensi, di mana koordinat X, Y, dan Z memerlukan array tiga dimensi untuk menyimpannya.

Definisi dan Struktur Array Multidimensi

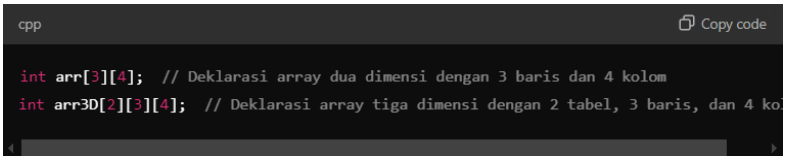
Secara formal, array multidimensi adalah array yang terdiri dari array lainnya sebagai elemen-elemennya. Misalnya, array dua dimensi adalah array yang setiap elemennya adalah array satu dimensi, sedangkan array tiga dimensi adalah array yang setiap elemennya adalah array dua dimensi.

- **Array Dua Dimensi**: Array dua dimensi adalah susunan elemen dalam bentuk matriks (baris dan kolom). Dalam konteks ini, kita mengakses elemen menggunakan dua indeks: satu untuk baris dan satu untuk kolom. Misalnya, arr[2][3] akan merujuk pada elemen di baris ketiga dan kolom keempat.
- **Array Tiga Dimensi**: Array tiga dimensi bisa dianggap sebagai sekumpulan matriks, atau array dua dimensi yang ditumpuk satu di atas yang lain. Untuk mengakses elemen, kita memerlukan tiga indeks, seperti arr[2][3][4].

Deklarasi dan Inisialisasi Array Multidimensi

Deklarasi array multidimensi mengikuti pola yang mirip dengan array satu dimensi, tetapi dengan tambahan tanda kurung yang menunjukkan dimensi tambahan. Contoh deklarasi array dua dimensi dan tiga dimensi dalam beberapa bahasa pemrograman:

- **C++**:



- **Python**:

Manfaat dan Keterbatasan

Manfaat utama array multidimensi adalah kemampuan untuk menyimpan dan mengorganisasi data yang lebih kompleks dalam struktur yang lebih mudah dikelola, seperti tabel, grafik, dan data spasial. Namun, array multidimensi juga memiliki beberapa keterbatasan:

1. **Penggunaan Memori**: Array multidimensi membutuhkan blok memori yang besar, terutama untuk array dengan dimensi yang tinggi, yang dapat menyebabkan pemborosan memori jika tidak digunakan dengan bijak.
2. **Ukuran Tetap**: Sebagian besar bahasa pemrograman membutuhkan kita untuk menentukan ukuran array pada saat deklarasi, yang tidak dapat diubah setelahnya, kecuali menggunakan array dinamis.

2.2 Konsep Array Multi Dimensi

Array multidimensi bisa dianggap sebagai sebuah array yang berisi array lain di dalamnya. Dalam array satu dimensi, kita hanya memiliki satu indeks untuk mengakses elemen-elemen di dalam array. Namun, dalam array dua dimensi atau lebih, kita memerlukan lebih banyak indeks untuk menentukan posisi elemen.

Contoh dasar array multidimensi adalah **array dua dimensi**, yang dapat dianggap sebagai tabel dengan baris dan kolom. Array tiga dimensi dapat dianggap sebagai tumpukan dari beberapa tabel dua dimensi, sementara array dengan lebih dari tiga dimensi semakin sulit untuk divisualisasikan tetapi masih mengikuti prinsip yang sama.

Deklarasi Array Multidimensi:

- **Array dua dimensi** dapat dinyatakan sebagai array dari array satu dimensi. Misalnya, dalam C++:

```cpp
int arr[3][4];  // Deklarasi array dua dimensi dengan 3 baris dan 4 k
```

- **Array tiga dimensi** dideklarasikan seperti berikut:

```cpp
int arr[2][3][4];  // Deklarasi array tiga dimensi dengan 2 tabel, 3
```

2.3 Dasar- Dasar Array Multidimensi

- **Definisi Array Multidimensi**: Menjelaskan lebih mendalam tentang apa itu array multidimensi, perbedaan antara array satu dimensi, dua dimensi, dan tiga dimensi.

- **Deklarasi dan Inisialisasi Array Multidimensi**: Contoh deklarasi dan inisialisasi array multidimensi dalam berbagai bahasa pemrograman (seperti C++, Python, Java).
- **Notasi Indeks**: Menjelaskan bagaimana elemen dalam array multidimensi diakses menggunakan indeks dalam berbagai dimensi.

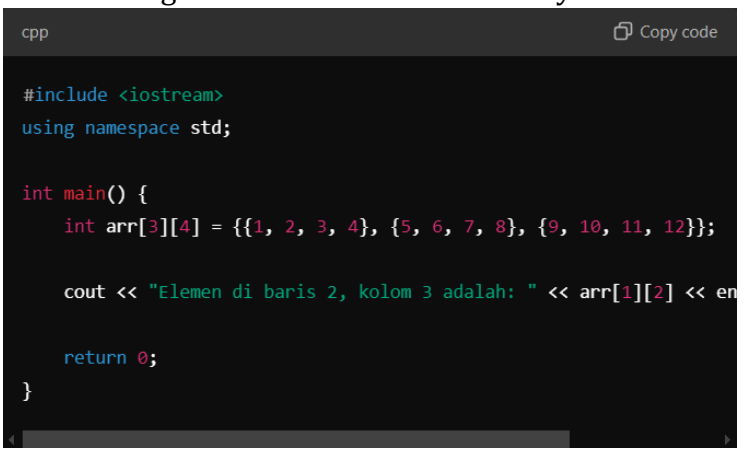
2.4 Operasi pada Array Multidimensi

- **raversal (Penelusuran Array Multidimensi)**: Cara menelusuri elemen-elemen dalam array multidimensi menggunakan loop bersarang.
- **Pengisian dan Pengaksesan Elemen**: Bagaimana mengisi elemen array multidimensi dan mengaksesnya melalui indeks yang sesuai.
- **Pengurutan (Sorting) Array Multidimensi**: Algoritma pengurutan pada array multidimensi (dapat mencakup teknik flattening array dan pengurutan elemen).
- **Pencarian (Searching) pada Array Multidimensi**: Algoritma pencarian dalam array multidimensi, misalnya **linear search** dan **binary search** jika array sudah diurutkan.
- **Perubahan Ukuran Array**: Jika membahas bahasa pemrograman dengan array dinamis, bahas bagaimana mengubah ukuran array multidimensi (misalnya dengan menggunakan struktur data yang mendukung pertumbuhan dinamis).

2.5 Pengaksesan Elemen pada Array Multidimensi

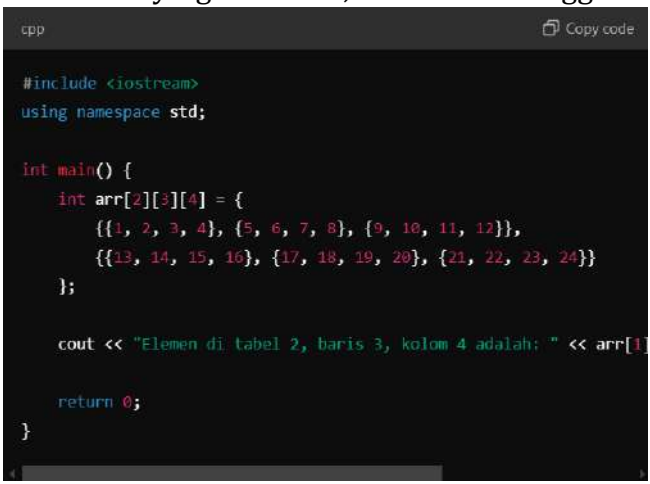
Untuk mengakses elemen pada array multidimensi, kita perlu menentukan indeks untuk setiap dimensi. Misalnya, dalam array dua dimensi, indeks pertama merujuk ke baris, dan indeks kedua merujuk ke kolom.

Contoh Pengaksesan Elemen dalam Array Dua Dimensi:



Pada contoh di atas, arr[1][2] mengakses elemen di baris kedua dan kolom ketiga, yang bernilai 7. Ini karena indeks array dimulai dari 0.

Contoh Pengaksesan Elemen dalam Array Tiga Dimensi:
Untuk array tiga dimensi, kita harus menggunakan tiga indeks:



Pada contoh ini, kita mengakses elemen 24, yang terletak di tabel kedua, baris ketiga, kolom keempat.

2.6 Keuntungan dan Penggunaan Array Multidimensi

Array multidimensi memiliki berbagai aplikasi praktis dalam pemrograman, terutama dalam kasus di mana kita perlu bekerja dengan data terstruktur dalam bentuk tabel, grafik, atau data lebih kompleks lainnya.

Penggunaan utama array multidimensi:

1. **Matriks dalam Matematika:** Array dua dimensi sangat sering digunakan untuk merepresentasikan matriks dalam aljabar linear. Contohnya:
    - Matriks transformasi dalam grafik komputer.
    - Operasi matriks dalam kalkulus numerik.

2. **Representasi Gambar atau Pixel:** Dalam pengolahan gambar digital, gambar sering kali direpresentasikan sebagai array dua dimensi, di mana setiap elemen (atau pixel) menyimpan nilai warna. Dalam gambar RGB (Red, Green, Blue), gambar dapat disimpan sebagai array tiga dimensi, di mana dua dimensi pertama mewakili koordinat pixel, dan dimensi ketiga menyimpan nilai untuk warna merah, hijau, dan biru.
3. **Data 3D atau Volume:** Array tiga dimensi digunakan untuk merepresentasikan data 3D, seperti volume dalam simulasi fisika atau data medis dalam pemindaian MRI.
4. **Pengolahan Data Multitabel:** Array multidimensi sering digunakan dalam pengelolaan data yang lebih kompleks, seperti pengolahan data dalam sistem database yang memerlukan pengelompokan data dalam berbagai kategori atau atribut.

2.7 Keterbatasan Array Multidimensi

Meskipun array multidimensi menawarkan banyak keuntungan, ada beberapa keterbatasan yang perlu diperhatikan:

1. **Ukuran Tetap:** Sebagian besar bahasa pemrograman mengharuskan kita untuk menentukan ukuran array pada saat deklarasi, dan ukuran ini tidak dapat diubah setelahnya. Hal ini membatasi fleksibilitas array.
2. **Efisiensi Memori:** Karena array multidimensi memerlukan blok memori yang besar dan berurutan, penggunaan array multidimensi yang besar dapat menyebabkan penggunaan memori yang berlebihan dan inefisiensi dalam pemrograman.
3. **Kompleksitas Pengaksesan:** Meskipun akses elemen dalam array multidimensi relatif cepat, kompleksitas

indeks yang banyak bisa membuat kode lebih sulit dibaca dan di-debug, terutama untuk array dengan dimensi yang lebih tinggi.

2.8 Operasi pada Array Multidimesi

1. Operasi Traversal (Penelusuran): Traversal atau penelusuran array multidimensi melibatkan penggunaan loop bersarang (nested loops) untuk mengakses setiap elemen dalam array. Dalam array dua dimensi, kita memerlukan dua loop untuk mengakses elemen-elemen dalam baris dan kolom.

   Contoh traversal array dua dimensi:

```cpp
#include <iostream>
using namespace std;

int main() {
    int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};

    // Traversal array dua dimensi
    for (int i = 0; i < 2; i++) {  // Loop untuk baris
        for (int j = 0; j < 3; j++) {  // Loop untuk kolom
            cout << arr[i][j] << " ";  // Mencetak elemen
        }
        cout << endl;
    }

    return 0;
}
```

2. Pengurutan (Sorting): Mengurutkan array multidimensi lebih rumit dibandingkan dengan array satu dimensi. Biasanya, array dua dimensi diubah terlebih dahulu menjadi array satu dimensi sebelum diterapkan algoritma pengurutan. Ini bisa dilakukan dengan menyalin elemen-elemen array multidimensi ke dalam array satu dimensi, mengurutkannya, lalu menyalin kembali hasilnya.

3. Pencarian (Searching): Pencarian dalam array multidimensi biasanya dilakukan dengan traversal elemen satu per satu (linear search). Dalam beberapa kasus, algoritma pencarian yang lebih kompleks, seperti binary search, dapat diterapkan pada array dua dimensi yang telah diurutkan.

DAFTAR PUSTAKA

Deitel, H. M., & Deitel, P. J. (2016). C++ How to Program (10th ed.). Pearson Education.

Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., & Stein, Clifford. (2009). *Introduction to Algorithms* (3rd ed.)

Sedgewick, Robert & Wayne, Kevin. (2011). *Algorithms* (4th ed.)

Kernighan, B. W., & Ritchie, D. M. (1988). The C Programming Language (2nd ed.). Prentice Hall.

BAB 3
STRUKTUR DATA RECORD DAN POINTER

Oleh Khairunnisa Samosir

3.1 Pendahuluan

Struktur data adalah cara untuk mengorganisasi dan menyimpan data dalam suatu sistem komputer agar dapat digunakan secara efisien. Dua konsep penting dalam struktur data adalah record dan pointer.(khairunisa samosir, 2024).

3.2 Struktur Data Record

Record adalah tipe data yang digunakan untuk mengelompokkan elemen-elemen data yang berbeda menjadi satu kesatuan. Setiap elemen dalam record, yang sering disebut field, dapat memiliki tipe data yang berbeda, seperti integer, string, atau tipe lainnya. Record sangat berguna dalam konteks di mana informasi terkait perlu dikelompokkan, seperti data mahasiswa yang memiliki nama, NIM, dan umur.

Contoh penggunaan:Mengelola data mahasiswa, karyawan, atau entitas lain yang memerlukan informasi terstruktur.

Contoh Implementasi:

Bahasa C:

```c
typedef struct {
    char nama[50];
    int nim;
    int umur;
} Mahasiswa;
```

Gambar 3. 1. Bahasa C

Bahasa Python:

```python
class Mahasiswa:
    def __init__(self, nama, nim, umur):
        self.nama = nama
        self.nim = nim
        self.umur = umur

```

Gambar 3. 2. Bahasa Python

Kegunaan:

- Mengelola data kompleks seperti data pengguna, produk, atau informasi akademis.
- Memudahkan pengelompokan data terkait dalam aplikasi yang memerlukan pemrosesan data berstruktur.

3.3 Pointer

Pointer adalah variabel yang menyimpan alamat memori dari variabel lain. Dengan menggunakan pointer, kita dapat mengakses dan memanipulasi data secara langsung di memori. Pointer sangat penting dalam pemrograman, terutama untuk

struktur data dinamis seperti linked list, tree, dan pengelolaan memori.

Contoh penggunaan: Mengelola struktur data dinamis dan mengoptimalkan penggunaan memori.

Ciri-ciri:

1. Menyimpan Alamat: Pointer menyimpan alamat dari variabel lain, bukan nilai.
2. Dinamis: Memungkinkan pengalokasian dan dealokasi memori secara dinamis.
3. Menyediakan Referensi: Memungkinkan fungsi untuk memodifikasi variabel di luar lingkupnya.

Contoh Implementasi:

Bahasa C:

```c
int a = 10;
int *p = &a; // p adalah pointer yang menunjuk ke alamat a

```

Gambar 3. 3. Bahasa C

Kegunaan:

1. Mengelola memori untuk struktur data dinamis (misalnya, linked list, tree).
2. Menghindari salinan data besar untuk meningkatkan efisiensi.
3. Mengimplementasikan array dinamis dan parameter fungsi yang lebih fleksibel.

Contoh Program Menggunakan Record dan Pointer

Contoh: Daftar Mahasiswa Menggunakan Linked List

Berikut adalah contoh program dalam C yang menggabungkan struktur data record dan pointer untuk membuat daftar mahasiswa menggunakan linked list:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Definisi struktur untuk mahasiswa
typedef struct {
    char nama[50];
    int nim;
    int umur;
} Mahasiswa;

// Definisi struktur node untuk linked list
typedef struct Node {
    Mahasiswa data;
    struct Node* next;
} Node;

// Fungsi untuk membuat node baru
Node* buatNode(Mahasiswa m) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = m;
    newNode->next = NULL;
    return newNode;
}

// Fungsi untuk menambahkan node di akhir linked list
void tambahNode(Node** head, Mahasiswa m) {
    Node* newNode = buatNode(m);
    if (*head == NULL) {
        *head = newNode;
    } else {
        Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
```

```
        temp->next = newNode;
    }
}

// Fungsi untuk menampilkan daftar mahasiswa
void tampilkanList(Node* head) {
    Node* current = head;
    while (current != NULL) {
        printf("Nama: %s, NIM: %d, Umur: %d\n", current->data.nama, current->data.nim, current->data.umur);
        current = current->next;
    }
}

// Fungsi utama
int main() {
    Node* head = NULL;
    Mahasiswa m1 = {"Budi", 123456, 20};
    Mahasiswa m2 = {"Siti", 654321, 22};
    Mahasiswa m3 = {"Joko", 789012, 21};

    // Menambahkan mahasiswa
    tambahNode(&head, m1);
    tambahNode(&head, m2);
    tambahNode(&head, m3);

    // Menampilkan daftar mahasiswa
    printf("Daftar Mahasiswa:\n");
    tampilkanList(head);
```

```
    // Dealokasi memori untuk semua node
    Node* temp;
    while (head != NULL) {
        temp = head;
        head = head->next;
        free(temp);
    }

    return 0;
}
```

Gambar 3. 4. Contoh Program

Penjelasan Program

Struktur Data:

Mahasiswa: Menyimpan nama, NIM, dan umur mahasiswa.

Node: Menyimpan data mahasiswa dan pointer ke node berikutnya.

Fungsi:

buatNode: Membuat node baru dengan data mahasiswa.

tambahNode: Menambahkan node baru ke akhir linked list.

tampilkanList: Menampilkan semua mahasiswa dalam linked list.

Fungsi Utama:

Menambahkan beberapa mahasiswa ke dalam linked list.

Menampilkan daftar mahasiswa.

Menangani dealokasi memori untuk semua node.

3.3 Langkah-langkah Memahami Struktur Data Record

FONT SUB-SUBBAB: Heading 3

1. Definisi dan Konsep Dasar:

   Pahami apa itu record sebagai kumpulan field yang berisi berbagai tipe data.

   Kenali kegunaan record dalam pengelompokan data terkait.

2. Implementasi Record:

   Pilih bahasa pemrograman (misalnya, C, Python).

   Buat definisi record menggunakan struct (C) atau class (Python).

Contoh C:

```c
typedef struct {
    char nama[50];
    int nim;
    int umur;
} Mahasiswa;

```

3. Akses dan Manipulasi Field:
    - Pelajari cara mengakses field menggunakan notasi titik.
    - Implementasikan fungsi untuk mengisi dan menampilkan data record.
4. Kelebihan dan Kekurangan:
    - Diskusikan kelebihan (organisasi, tipe beragam) dan kekurangan (ukuran statik) dari penggunaan record.

Langkah-langkah Memahami Pointer

1. Definisi dan Konsep Dasar:

    Pahami apa itu pointer sebagai variabel yang menyimpan alamat memori dari variabel lain.

2. Deklarasi dan Inisialisasi:

    Pelajari cara mendeklarasikan pointer dan menginisialisasinya dengan alamat variabel.

    Contoh C :

```c
int a = 10;
int* p = &a; // p menyimpan alamat a

```

3. Penggunaan Pointer dalam Fungsi:
   Implementasikan fungsi yang menggunakan pointer untuk mengubah nilai variabel di luar lingkup fungsi.
   Contoh C:

```c
void ubahNilai(int* p) {
    *p = 100; // Mengubah nilai
}

```

3.4 Kelebihan dan Kekurangan Record

1. **Kelebihan:**

- Mengelompokkan data yang terkait.
- Menyediakan struktur yang jelas dan mudah digunakan.

2. **Kekurangan:**

- Ukuran statik dapat mengakibatkan pemborosan memori.
- Tidak mendukung operasi dinamis secara langsung.

3.5 Keuntungan Menggunakan Record dan Pointer

1. Keuntungan Record

- **Organisasi Data:** Memungkinkan penyimpanan data yang terorganisir, memudahkan akses dan manipulasi.

- **Keterkaitan Data:** Memungkinkan pengelompokan data terkait dalam satu unit.

2. Keuntungan Pointer

- **Efisiensi Memori:** Menghindari duplikasi data dan memungkinkan pengelolaan memori yang lebih efisien.
- **Akses Dinamis:** Memungkinkan alokasi memori dinamis sesuai kebutuhan saat runtime.

3.6 Praktik Terbaik dalam Menggunakan Record dan Pointer

1. Manajemen Memori

- **Dealokasi Memori:** Pastikan untuk selalu menggunakan free (C) setelah selesai menggunakan memori yang dialokasikan dengan malloc untuk menghindari kebocoran memori.
- **Inisialisasi Pointer:** Selalu inisialisasi pointer sebelum digunakan untuk menghindari dereferensi pointer yang tidak valid.

2. Struktur Kode yang Jelas

- **Penamaan yang Jelas:** Gunakan nama yang deskriptif untuk struktur dan field agar kode mudah dipahami.
- **Modularisasi:** Pisahkan fungsi-fungsi terkait ke dalam file yang berbeda untuk menjaga kode terorganisir.

Contoh Kasus Penggunaan

1. Penggunaan Linked List dalam Manajemen Data

Misalkan kita ingin mengelola data pelanggan dalam sebuah aplikasi. Kita dapat menggunakan linked list untuk

menambahkan, menghapus, dan menampilkan data pelanggan secara dinamis.

```c
typedef struct {
    char nama[50];
    int id;
} Pelanggan;

typedef struct Node {
    Pelanggan data;
    struct Node* next;
} Node;

// Fungsi untuk menambah pelanggan
void tambahPelanggan(Node** head, Pelanggan p) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = p;
    newNode->next = *head; // Menambahkan di depan
    *head = newNode;
}
```

2. Implementasi Binary Search Tree (BST)

BST adalah struktur data yang memungkinkan pencarian data secara efisien. Kita dapat menggunakan record dan pointer untuk membangun BST.

```c
typedef struct Node {
    Mahasiswa data;
    struct Node* left;
    struct Node* right;
} TreeNode;

// Fungsi untuk menambahkan node ke BST
TreeNode* insert(TreeNode* node, Mahasiswa m) {
    if (node == NULL) {
        TreeNode* newNode = (TreeNode*)malloc(sizeof(TreeNode));
        newNode->data = m;
        newNode->left = newNode->right = NULL;
        return newNode;
    }
    if (m.nim < node->data.nim) {
        node->left = insert(node->left, m);
    } else {
        node->right = insert(node->right, m);
    }
    return node;
}
```

Kesimpulan

Struktur data record dan pointer adalah fondasi penting dalam pemrograman yang memungkinkan pengelolaan data yang efisien dan terorganisir. Pemahaman yang mendalam tentang keduanya akan memperkuat kemampuan Anda dalam membangun aplikasi yang kompleks dan berfungsi dengan baik.

DAFTAR PUSTAKA

Weiss, M. A. (2014). Data Structures and Algorithm Analysis in C. Pearson.

King, K. N. (2009). C Programming: A Modern Approach. W. W. Norton & Company.

Sedgewick, R., & Wayne, K. (2011). Algorithms, Part I. Addison-Wesley.

Karumanchi, N. (2011). Data Structures and Algorithms Made Easy. CareerMonk Publications.

GeeksforGeeks. (n.d.). Data Structures. Diakses dari [GeeksforGeeks](https://www.geeksforgeeks.org/data-structures/).

Coursera. (n.d.). Data Structures and Algorithms Specialization. Diakses dari [Coursera](https://www.coursera.org/specializations/data-structures-algorithms).

BAB 4
FUNGSI DAN PENGIRIMAN PARAMETER DALAM FUNGSI

Oleh Budanis Dwi Meilani

4.1 Pendahuluan

Di dalam penyelesaian suatu program seringkali proses besar diselesaikan dengan membagi menjadi beberapa proses yang lebih kecil. Pembagian proses ini biasa kita sebut dengan sub proses atau sub program. Setiap sub proses atau sub program melakukan kegiatan yang spesifik yang berbeda dengan sub proses yang lain, sehingga tidak terjadi overlapping tugas.

Subproses atau subprogram terbagi menjadi dua bentuk yaitu prosedur (*procedure*) dan fungsi (*function*). Dimana perbedaan dari prosedur dan fungsi adalah :

a. Prosedur adalah subproses / subprogram yang tidak membawa nilai balik.
b. Fungsi adalah subproses / subprogram yang membawa nilai balik.

Di dalam pembahasan di bab ini kami akan memakai istilah fungsi untuk subprogram atau subproses dan tidak membedakan antara prosedur dan fungsi.

Sebuah fungsi berisikan suatu kegiatan atau sejumlah pernyataan yang dikemas dalam sebuah nama. Nama ini dapat dipanggil beberapa kali di dalam program pada tempat yang berbeda-beda. Tujuan dari pembuatan fungsi adalah memudahkan dalam pembuatan program dan menghemat ukuran program.

4.2 Manfaat Fungsi

Pembuatan fungsi di dalam program sangatlah disarankan dan banyak manfaatnya diantaranya adalah :

a. Memudahkan dalam proses menyelesaikan masalah. Karena sumua proses sudah dikelompokan menjadi subproses berdasarkan nama fungsi masing-masing.
b. Memudahkan dalam pembacaan program. Pada saat merujuk pada suatu proses maka langsung menuju pada fungsi yang melakukan proses tersebut.
c. Memudahkan dalam menelusuri alur program. Pada saat pemanggilan fungsi maka akan langsung ke proses fungsi tersebut.
d. Memudahkan dalam proses *debugging* yaitu mencari atau memperbaiki kesalahan yang ada di dalam program yang dibuat.
e. Memudahkan dalam pengembangan sistem. Karena semua proses sudah tersusun berdasarkan subproses-subproses atau fungsi

4.3 Parameter

Parameter adalah suatu nilai atau data yang dikirimkan ke fungsi untuk di proses di dalam fungsi. Paramater juga bisa disebut sebagai argumen, keduanya sama saja hanya berbeda

istilah. Keberadaan parameter tidak harus ada, tergantung subproses apa yang dibuat. Di dalam fungsi boleh tidak mempunyai parameter atau memiliki parameter satu atau lebih dengan tipe data yang beragam.

Sebuah parameter di dalam fungsi menjadi variabel lokal dimana parameter tersebut di deklarasikan. Tipe data untuk parameter mengikuti aturan aturan yang berlaku pada variabel.

4.4 Nilai Balik

Nilai Balik adalah data yang dihasilkan dari fungsi yang dikirim ke subproses yang memanggil fungsi. Nilai balik tidak harus ada tergantung subproses apa yang akan dibuat. Subproses tidak harus memiliki nilai balik atau memiliki satu nilai balik dengan tipe data yang beragam.

Gambar 4. 1. Fungsi, Parameter dan Nilai Balik

Pada gambar 4.1. dijelaskan bahwa parameter atau argument digunakan untuk membawa inputan data ke dalam fungsi untuk diproses. Sedangkan nilai balik digunakan untuk membawa hasil proses atau nilai ke luar fungsi dan diberikan kepada pemanggil fungsi.

4.5 Pemanggilan Fungsi

Untuk pemanggilan Fungsi harus diperhatikan definisi fungsi yang dibuat. Perbedaan parameter dan nilai balik menjadi acuan dalam pemanggilan fungsi.

Format Pendefinisian Fungsi :

Tipedata_nilai_balik ***Nama_fungsi*** *(tipedata* ***nama_parameter****)*

{

Statement;

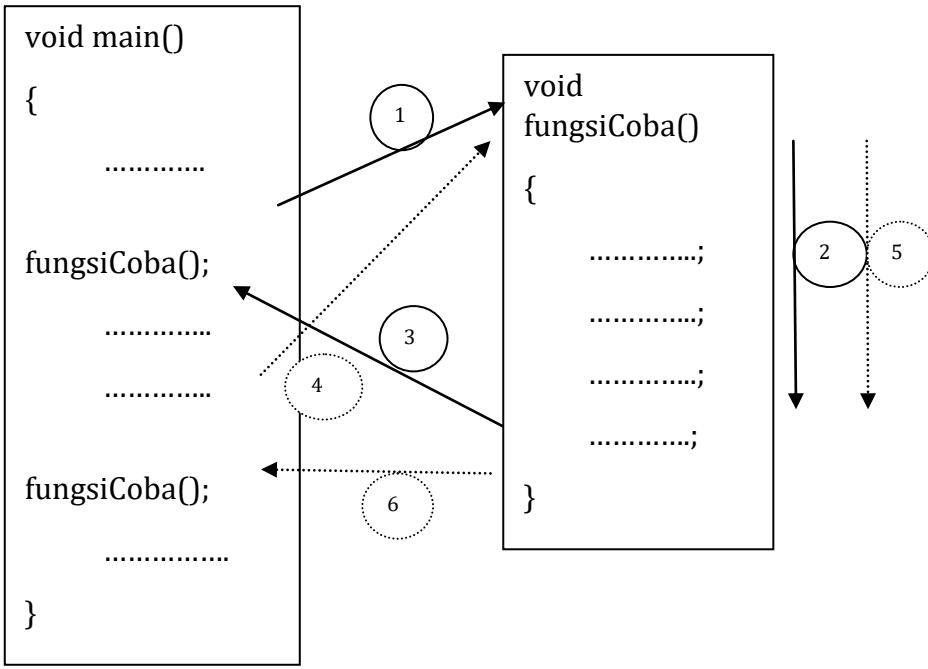
Return (nilai_balik);

}

Format Pemanggilan Fungsi :

Nama_Fungsi(nama_parameter);

Untuk jelasnya bagaimana pemanggilan fungsi bisa dilihat pada gambar 4.2.



Gambar 4. 2. Diagram Pemanggilan Fungsi

Pada gambar 4.2. adalah proses pemanggilan fungsi dari fungsi utama (program utama) ke fungsi dengan nama fungsiCoba. **Void main()** adalah fungsi utama dan **void fungsiCoba()** adalah fungsi yang didefinisikan atau yang dibuat. Maka proses dimulai dari pembacaan atau menjalankan fungsi utama main(). Kemudian di dalam main() baca proses sampai menuju ke :

Langkah 1 : pemanggilan fungsiCoba() maka menuju ke definisi fungsiCoba().

Langkah 2 : membaca dan menjalankan proses pada fungsiCoba() sampai selesai.

Langkah 3 : kembali ke main() melanjutkaan proses.

Langkah 4 : pemanggilan fungsiCoba() dan kembali menuju ke definisi fungsiCoba().

Langkah 5 : membaca dan menjalankan proses pada fungsiCoba() sampai selesai.

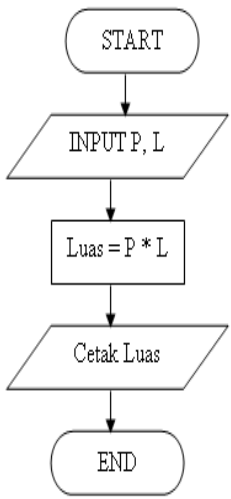
Langkah 6 : kembali ke main() melanjutkaan proses. Atau selesai.

4.6 Cara membuat Fungsi

Secara sintaksis, pembuatan fungsi di dalam bahasa pemrograman mungkin bisa berbeda-beda tetapi secara konsep adalah sama. Yang harus diperhatikan dalam membuat fungsi adalah penentuan parameter dan nilai balik.

Studi kasus :

Buatlah program untuk menghitung luas persegi panjang. Kerjakan dengan menggunakan fungsi.



Gambar 4. 3. Flowchart Menghitung Luas Persegi Panjang

Gambar 4.3 adalah flowchart untuk menghitung luas persegi panjang, dimana ada dua input yaitu p (panjang) dan l (lebar), satu output yaitu luas. Dari flowchart di atas akan dibuatkan fungsi untuk proses tersebut.

Jawaban 1: fungsi tanpa parameter dan tanpa nilai baik (disebut prosedure).

Kita pikirkan kalau kita akan membuat fungsi tanpa parameter dan tanpa nilai balik maka tidak ada nilai yang diinputkan ke fungsi dan tidak ada nilai yang dihasilkan dari fungsi.

Pendeklarasian fungsi sebagai berikut :

void LuasPP( ) ;

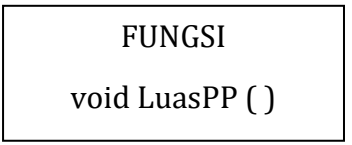
artinya void berarti tidak ada nilai balik dan di dalam kurung kosong artinya tidak ada parameter.

Pemanggilan fungsinya sebagai berikut :

LuasPP( );

```
void LuasPP()
{
    int p, l, luas;
    cout << " Input panjang = ";
    cin >> p;
    cout << " Input lebar = ";
    cin >> l;
    luas = p * l;
    cout << " Luas = " << luas << endl;
}

int main()
{
    LuasPP();
}
```

Gambar 4. 4. subprg tanpa parameter dan tanpa nilai balik



Gambar 4. 5. Diagram Fungsi void LuasPP ( )

Jawaban 2: fungsi dengan parameter dan tanpa nilai baik (disebut prosedure).

Kita pikirkan kalau kita akan membuat fungsi dengan parameter dan tanpa nilai balik maka ada nilai yang diinputkan ke fungsi dan tidak ada nilai yang dihasilkan dari fungsi. Nilai yang diinputkan ke fungsi disebut parameter. Untuk menentukan jumlah parameter dengan melihat berapa jumlah nilai yang akan diinputkan untuk menghasilkan output dalam fungsi. Untuk Luas persegi panjang ada 2 nilai yang harus diinputkan yaitu panjang dan lebar, maka 2 nilai tersebut masukkan menjadi prameter.

Pendeklarasian fungsi sebagai berikut :

void LuasPP(int p, int l) ;

artinya void berarti tidak ada nilai balik dan di dalam kurung terdapat 2 parameter yaitu parameter p (panjang) dan parameter l (lebar) dengan tipe data integer (bilangan bulat).

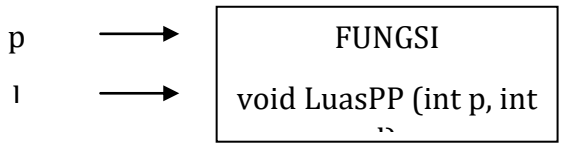
Pemanggilan fungsinya sebagai berikut :

LuasPP(10,5);

untuk pemanggilan fungsi ini harus membawa nilai sebanyak 2 nilai dimana 10 diberikan ke p dan 5 diberikan ke l.

```
void LuasPP(int p, int l)
{
    int luas;
    luas = p * l;
    cout << " Luas = " << luas << endl;
}

int main()
{
    LuasPP(10,5);
}
```

Gambar 4. 6. subprg dengan parameter dan tanpa nilai balik



Gambar 4. 7. Diagram Fungsi void LuasPP (int p, int l)

Jawaban 3: fungsi tanpa parameter dan dengan nilai baik (disebut function).

Kita pikirkan kalau kita akan membuat fungsi tanpa parameter dan dengan nilai balik maka tidak ada nilai yang diinputkan ke fungsi dan ada nilai yang dihasilkan dari fungsi. Nilai balik adalah nilai hasil proses di dalam fungsi yang dibawa ke subprogram yang memanggil fungsi. Maka terdapat satu nilai balik yaitu hasil dari perhitungan luas.

Pendeklarasian fungsi sebagai berikut :

int LuasPP( )

artinya int berarti terdapat nilai balik dengan tipe data integer dan di dalam kurung kosong artinya tidak ada parameter.

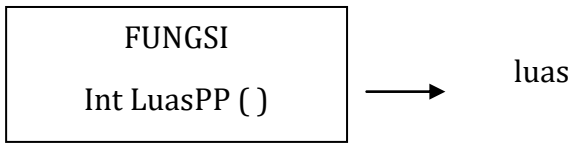
Pemanggilan fungsinya sebagai berikut :

Hasil = LuasPP();

untuk pemanggilan fungsi ini tidak membawa nilai tetapi menghasilkan nilai balik dan hasil nilai balik diberikan ke variabel Hasil.

```
int LuasPP()
{
    int p, l, luas;
    cout << " Input panjang = ";
    cin >> p;
    cout << " Input lebar = ";
    cin >> l;
    luas = p * l;
    return(luas);
}

int main()
{
    int Hasil;
    Hasil = LuasPP();
    cout << " Luas = " << Hasil << endl;
}
```

Gambar 4. 8. subprg tanpa parameter dan dengan nilai balik



Gambar 4. 9. Diagram Fungsi int LuasPP ( )

Jawaban 4: fungsi dengan parameter dan dengan nilai baik (disebut function).

Kita pikirkan kalau kita akan membuat fungsi dengan parameter dan dengan nilai balik maka ada nilai yang diinputkan ke fungsi dan ada nilai yang dihasilkan dari fungsi. Nilai yang diinputkan ke fungsi disebut parameter. Untuk menentukan jumlah parameter dengan melihat berapa jumlah nilai yang akan diinputkan untuk menghasilkan output dalam fungsi. Untuk Luas persegi panjang ada 2 nilai yang harus diinputkan yaitu panjang dan lebar, maka 2 nilai tersebut masukkan menjadi parameter. Nilai balik adalah nilai hasil proses di dalam fungsi yang dibawa ke subprogram yang memanggil fungsi. Maka terdapat satu nilai balik yaitu hasil dari perhitungan luas.

Pendeklarasian fungsi sebagai berikut :

int LuasPP(int p, int l)

artinya int terdapat nilai balik dengan tipe data integer dan di dalam kurung terdapat 2 parameter yaitu parameter p (panjang) dan parameter l (lebar) dengan tipe data integer (bilangan bulat).

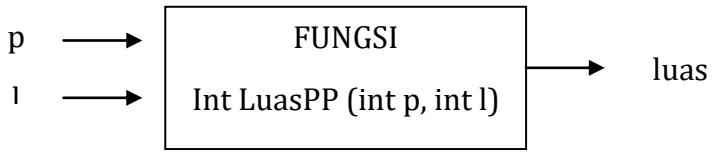
Pemanggilan fungsinya sebagai berikut :

Hasil = LuasPP(10,5);

untuk pemanggilan fungsi ini harus membawa nilai sebanyak 2 nilai dimanaa 10 diberikan ke p dan 5 diberikan ke l dan menghasilkan nilai dan hasil nilai balik diberikan ke variabel Hasil.

```
int LuasPP(int p, int l)
{
    int luas;
    luas = p * l;
    return(luas);
}

int main()
{
    int Hasil;
    Hasil = LuasPP(10,5);
    cout << " Luas = " << Hasil << endl;
}
```

Gambar 4. 10. subprg dengan parameter dan dengan nilai balik



Gambar 4. 11. Diagram Fungsi int LuasPP (int p, int l)

Pada fungsi yang mempunyai nilai balik akan terdapat kata kunci return. Return ini untuk memberitahukaan variabel yang menjadi nilai balik.

DAFTAR PUSTAKA

Stroustrup, Bjarne.*Programming principle and practice using C++*. Amerika: Penerbit. Pearson education, Inc.2014.

Hariyanto, Eko & Sulistianingsih, Indri. 2019. Dasar Pemrograman Java. Medan : Fakultas Ekonomi Universitas Panca Budi.

Kadir, Abdul. 2014. Buku Pertama Belajar Pemrograman Java untuk Pemula. Yogyakarta: Mediakom.

Rinaldi Munir Leony Lidya, Algortima dan Pemrograman dalam bahasa Pascal, C, dan C++, Informatika, 2016

H.M. Deitel, P.J. Deitel , S.E. Santry, Java How To Program, Late Objects, 11th Edition, Deitel & Associates, Inc, 2017.

Abdul Kadir, Algoritma & Pemrograman Menggunakan C & C++, Andi Offset, 2012

BAB 5
PENGENALAN DAN IMPLEMENTASI LINKED LIST

Oleh Sulistyowati

5.1 Pendahuluan

Dalam kehidupan sehari-hari, kita sudah seringkali mendengar istilah list (senarai), yaitu kumpulan dari sejumlah data. Sebagai contoh, kita akan memesan menu makanan atau minuman di sebuah restoran. Menu pertama, kedua, dan seterusnya bisa kita tuliskan dalam list daftar pesanan, seperti pada Gambar 5.1.

Gambar 5. 1. Daftar menu pesanan

Jika menu makanan atau minuman pada list tersebut sudah disajikan oleh pramusaji, maka menu tersebut akan dihapus/dicoret dari list daftar pesanan. Dan kita juga bisa

menambahkan beberapa menu makanan atau minuman lainnya ke dalam list daftar pesanan, seperti pada Gambar 5.2.

Gambar 5. 2. Daftar menu pesanan setelah perubahan

Pengolahan data yang kita lakukan menggunakan komputer seringkali mirip dengan ilustrasi di atas, sering melibatkan penyimpanan dan manipulasi data dalam bentuk yang terorganisir. Salah satu cara umum untuk menyimpan sekumpulan data adalah menggunakan larik (array). Larik adalah struktur data yang menyimpan elemen-elemen data dengan tipe data yang sama di lokasi memori yang berdekatan, yang memungkinkan akses cepat ke setiap elemen berdasarkan indeksnya. Namun, larik memiliki keterbatasan, yaitu ukurannya tetap (statis). Artinya, jika kita ingin menambah data baru melebihi batas larik, kita harus membuat larik baru dengan ukuran lebih besar. Jika pengolahan data kita membutuhkan fleksibilitas lebih, seperti penambahan atau penghapusan data secara dinamis, kita bisa menggunakan struktur data lain seperti **Linked List** yang lebih fleksibel dalam mengelola ukuran data. Artinya untuk menyimpan sejumlah data yang banyaknya selalu

berubah dan tidak dapat diketahui secara pasti kapan proses penambahan atau penghapusan data akan berhenti.

5.1.1 Array vs Linked List

Array dan Linked List memiliki kesamaan, yaitu untuk penyimpanan data dengan jumlah yang banyak. Namun terdapat beberapa perbedaan mendasar antara Array dan Linked List, antar lain (Geeksforgeeks, 2024) & (Standish, 1995) :

1. **Struktur dan Penyimpanan Memori**
   - Array :
     - ✓ Array menyimpan elemen-elemen dalam blok memori yang berdekatan. Setiap elemen diakses menggunakan indeks.
     - ✓ Ukuran array tetap, artinya kita harus menentukan ukuran array saat membuatnya, dan tidak bisa mengubah ukuran tersebut setelahnya.
   - Linked List :
     - ✓ Linked list terdiri dari serangkaian node yang masing-masing menyimpan data dan pointer ke node berikutnya (pada single linked list).
     - ✓ Elemen-elemen di linked list tidak disimpan di lokasi memori yang berdekatan, tetapi terhubung melalui pointer.
     - ✓ Ukuran linked list bersifat dinamis, dan node dapat ditambahkan atau dihapus tanpa batasan ukuran
2. **Akses Elemen**
   - Array : Akses acak dimungkinkan karena semua elemen tersimpan di lokasi memori yang bersebelahan. Ini berarti elemen dapat diakses dalam waktu konstan menggunakan indeks.

- Linked List : Akses sekuensial diperlukan. Misal untuk mengakses elemen ke-3, maka kita harus mulai dari node pertama dan melakukan traversal ke node berikutnya hingga mencapai elemen tersebut. Ini berarti akses memerlukan waktu linear tergantung pada posisi elemen.

3. Penambahan dan Penghapusan Elemen

- Array :
  - ✓ Penamambahan di akhir array relatif cepat jika ada ruang tersisa. Namun, jika array penuh, alokasi ulang memori diperlukan dan bisa memakan waktu.
  - ✓ Penambahan atau penghapusan di tengah array lambat karena memerlukan penggeseran elemen-elemen lainnya untuk memberi ruang atau menutup celah.
- **Linked List** :
  - ✓ Penambahan atau penghapusan node di posisi mana pun (awal, tengah, atau akhir) cepat, selama kita memiliki akses ke node tersebut (misalnya dengan traversal sebelumnya).
  - ✓ Namun, jika node tidak diketahui, traversal dari awal hingga node yang diinginkan akan memakan waktu.

4. Penggunaan Memori

- **Array** :
  - ✓ Array memerlukan blok memori yang berkelanjutan. Karena ukurannya tetap, jika kita mengalokasikan array yang terlalu besar, bisa terjadi pemborosan memori jika tidak semua elemen digunakan.
  - ✓ Tidak ada overhead memori untuk pointer seperti pada linked list.

- **Linked List**:
  - ✓ Linked list lebih hemat memori dalam hal hanya menyimpan elemen yang diperlukan karena ukurannya dinamis.
  - ✓ Namun, setiap node membutuhkan memori tambahan untuk menyimpan pointer ke node berikutnya, yang meningkatkan overhead memori.

4. Ukuran

- **Array** : Ukuran array tetap setelah dideklarasikan. Jika diperlukan menambah lebih banyak elemen, maka harus membuat array baru yang lebih besar dan menyalin elemen-elemen yang sudah ada.
- **Linked List** : Linked list bersifat dinamis, sehingga kita bisa menambah atau mengurangi node sesuai kebutuhan tanpa batasan ukuran (kecuali keterbatasan memori total).

6. Pencarian Elemen

- **Array** : Pencarian elemen di array bergantung pada metode pencarian yang digunakan.
- **Linked List** : Pencarian di linked list harus dilakukan secara linier dari awal hingga akhir (traversal), karena tidak ada cara untuk langsung mengakses elemen dengan indeks.

7. Kapan Menggunakan

- **Array** :
  - ✓ Cocok digunakan jika diketahui jumlah elemen yang ingin disimpan dari awal dan membutuhkan akses acak yang cepat.

- ✓ Cocok untuk aplikasi yang memerlukan banyak operasi baca dan jarang menambahkan atau menghapus elemen di tengah-tengah data.

- **Linked List** :
  - ✓ Cocok digunakan jika tidak diketahui jumlah elemen yang akan disimpan atau jika sering menambah atau menghapus elemen di berbagai posisi dalam koleksi data.
  - ✓ Linked list lebih efisien untuk pengelolaan data yang sifatnya dinamis, dimana perubahan ukuran sering terjadi.

5.2 Pointer

Sebelum kita membahas lebih lanjut tentang Linked list, maka kita pelajari terlebih dahulu tentang pointer. Pointer adalah variabel khusus yang menyimpan alamat memori dari variabel lain, bukan nilai sebenarnya dari variabel tersebut (Kadir, 2001). Suatu pointer dalam pemrograman dimaksudkan untuk menunjuk ke suatu Alamat memori. Misalnya didefinisikan X adalah sebuah pointer dan Y adalah variabel yang terletak dalam memori dengan alamat 0xfff2, pointer X dapat diatur agar menunjuk ke variabel Y (Kadir, 2001). Dengan menggunakan pointer, kita dapat mengakses dan memanipulasi nilai yang disimpan di Lokasi/alamat memori lain secara langsung (alamat ini dinyatakan dalam bentuk heksadesimal, seperti 0x100, 0x104).

Bentuk umum dari pernyataan variabel pointer :

```
tipe_data *nama_variabel;
```

Dimana :

- ✓ tipe_data adalah tipe dasar pointer
- ✓ nama_variabel adalah nama variabel pointer

- ✓ * adalah operator memori yang fungsinya untuk mengembalikan nilai variabel pada alamatnya yang ditentukan oleh operand

Contoh :

float *ABC;

Statement diatas mendeklarasikan sebuah variabel bernama ABC sebagai pointer, dimana alamat memori yang ditunjuk oleh ABC dimaksudkan untuk berisi data bertipe float.

Perbedaan antara variabel pointer dengan variabel biasa dapat dilihat pada Tabel 5.1.

Tabel 5. 1. Perbedaan varibel pointer dan variabel biasa

VARIABEL POINTER	VARIABEL BIASA
Berisi alamat memori dari suatu variabel tertentu	Berisi data/nilai
Membutuhkan operator khusus ”&” yang menunjuk alamat dari suatu variabel tertentu dan operator “*” yang membaca nilai dari alamat variabel yang ditunjuk oleh pointer tersebut	Operasi yang bisa dilakukan adalah : • operasi aritmatik, yaitu : + , - , * , / , % • operator rasional, yaitu : < , > , <= , >= , == , != • operator logika, yaitu : && , || , ! • operator assignment, yaitu : += , -= , *= , /= , %=
Pengalokasiannya bersifat dinamis	Pengalokasiannya bersifat statis
Deklarasi :	Deklarasi :

tipe_data *nama_var;	tipe_data nama_var;

Operator pada pointer ada dua macam, yaitu :

1. **Operator Deference (&)**
   Operator ini digunakan untuk mengetahui alamat memori tempat penyimpanan data.

   Contoh :

   ```
   int *X;          //deklarasi variabel pointer
   int Y = 10;      //inisialisasi variabel Y dengan nilai 10
   X = &Y;          //inisialisasi variabel pointer X dengan nilai
                      alamat memori dimana variabel Y disimpan
   cout << X;
   ```

   Misal jika variabel Y disimpan di dalam memori pada alamat 0x0012ff50, maka hasil yang ditampilkan adalah 0x0012ff50.

2. **Operator Reference (*)**
   Operator ini digunakan untuk mengakses nilai/data pada sebuah alamat memori yang ditunjuk oleh variabel pointer.

   Contoh :

   ```
   int *X;
   int Y = 100 , Z;
   X = &Y;
   Z = *X;          //inisialisasi variabel Z dengan nilai yang disimpan
                      di alamat memori yang disimpan oleh pointer X
   cout << Z;
   ```

   Maka hasilnya adalah : 100

5.3 Linked List (Senarai Berantai)

Linked List adalah salah satu struktur data yang terdiri dari serangkaian elemen yang saling terhubung dengan bantuan pointer, di mana setiap elemen disebut sebagai **simpul atau node** (Aho, dkk, 1987). Setiap node dalam linked list berisi dua bagian utama:

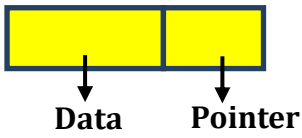
1. **Medan informasi / data**

   Adalah bagian yang menyimpan nilai data atau informasi.

2. **Medan penyambung / pointer (atau reference)**

   Adalah bagian yang menyimpan alamat memori dari simpul berikutnya atau referensi ke node berikutnya dalam urutan.

Bentuk simpul atau node dalam linked list dapat dilihat pada Gambar 5.3.

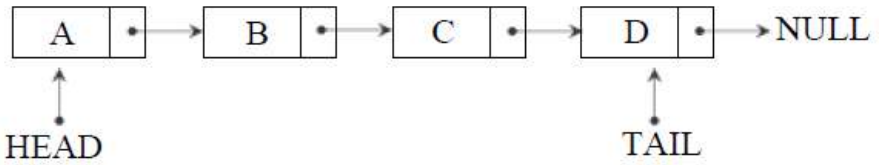


Gambar 5. 3. Ilustrasi simpul atau node

5.3.1 Single Linked List

Single Linked List adalah sebuah linked list dimana setiap satu simpul/node hanya memiliki sebuah variabel pointer yang digunakan untuk menunjuk pada simpul/node berikutnya.

Ilustrasi Single Linked List seperti pada Gambar 5.4.



Gambar 5. 4. Ilustrasi Single Linked List

Single Linked List diatas terdiri dari 4 simpul. Simpul A disebut **kepala/head** dan simpul D disebut **ekor/tail**. Tanda panah merupakan pengait/pointer yang menghubungkan satu simpul dengan simpul yang lainnya. Pointer yang dimiliki oleh simpul D, karena tidak mengait ke simpul lain maka menuju ke NULL.

```
struct tipe_data pointer
{
        tipe_data nama_medan_info;
        tipe_data_pointer
        *nama_medan_penyambung;
};
```

Bentuk umum deklarasi single linked list :

Keterangan :

- tipe_data_pointer = tipe data buatan yang berupa simpul atau node
- nama_medan_info = nama medan informasi
- nama_medan_penyambung = nama medan penyambung atau pointer

- nama_variabel_pointer = nama variabel yang bertipe pointer

Contoh deklarasi single linked list :

```
struct simpul
{
        int info;
        simpul *next;
};
simpul *X;
```

Dari contoh diatas, berarti telah dibuat sebuah tipe data buatan berupa struktur yang diberi nama "simpul". Dimana field yang terbentuk ada 2, yaitu "info" → untuk menyatakan medan informasi dan "next" → untuk menyatakan medan penyambung (harus bertipe pointer). Kemudian dideklarasikan sebuah variabel pointer yaitu X yang bertipe data simpul.

Untuk membuat simpul/mengalokasikan sebuah simpul dalam memori, menggunakan statement :

```
new nama_simpul;
```

Contoh :

```
X = new simpul;
```

Dengan perintah tersebut maka akan terbentuk sebuah simpul baru yang ditunjuk oleh pointer X, seperti ilustrasi pada Gambar 5.5.

Gambar 5. 5. Ilustrasi pembetukan sebuah simpul

Terdapat beberapa operasi pada Single Linked List, antara lain :

1. **Membentuk Simpul Awal**

   Pada saat pertama kali, linked list selalu dalam keadaan kosong. Hal ini dikarenakan belum ada simpul yang terbentuk.

   Sehingga perlu dibentuk sebuah simpul baru. Untuk membentuk simpul awal dalam linked list, kita perlu membuat node pertama yang berisi data dan pointer ke node berikutnya yang nilainya pada awalnya kosong atau NULL. Namun sebelumnya, pada saat pendeklarasian kita perlu menambahkan deklarasi untuk beberapa variabel pointer lain yang dibutuhkan, seperti deklarasi berikut :

```
struct simpul
{
    int info;
    simpul *next;
};
simpul *BARU, *HEAD, *TAIL;
```

   Variabel pointer “BARU” digunakan untuk menunjuk ke simpul yang baru dibuat atau dibentuk, variabel pointer “HEAD” untuk menunjuk ke simpul yang berada di posisi paling depan pada linked list, dan variabel pointer “TAIL” untuk menunjuk ke simpul yang berada pada posisi paling belakang pada linked list.

Pada keadaaan single linked list masih kosong, maka pointer head dan tail bernilai NULL karena tidak ada simpul yang ditunjuk. Sehingga perlu dilakukan inisialisasi tahap awal :

HEAD = NULL;

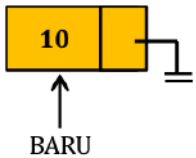
TAIL = NULL;

Ilustrasi keadaan single linked list saat masih kosong seperti pada Gambar 5.6.

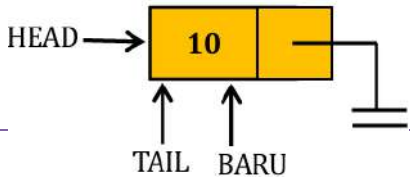
Gambar 5. 6. Ilustrasi single linked list saat kosong

Selanjutnya kita mulai membentuk simpul awal dengan algoritma sebagai berikut :

1. Buat simpul baru yang ditunjuk oleh pointer BARU dan masukkan nilai datanya :



2. Karena single linked masih kosong, maka simpul BARU otomatis berada pada posisi paling depan sekaligus paling belakang. Sehingga pointer HEAD menunjuk ke simpul BARU dan pointer juga menunjuk ke simpul BARU :



Statement untuk membuat simpul awal pada single linked list :

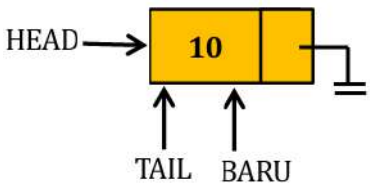
```
BARU = new simpul;
BARU -> info = 10;  //misal nilai medan informasi =10
BARU -> next = NULL;
HEAD = BARU;
TAIL = BARU;
```

2. Menyisipkan Simpul di Depan Single Linked List

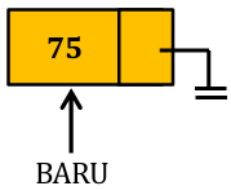
Menyisipkan simpul di depan linked list artinya menambahkan simpul baru pada posisi sebelum simpul yang paling depan dalam single linked list, sehingga simpul baru tersebut menjadi simpul pertama.

Algoritma penyisipan simpul di depan single linked list adalah sebagai berikut :

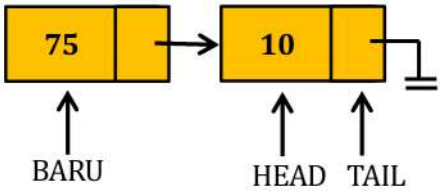
1. Kondisi single linked list sebelum penyisipan :



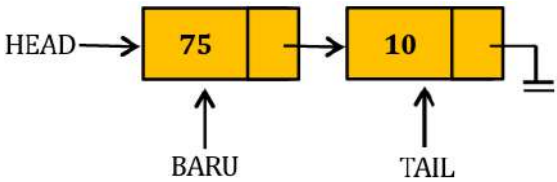
2. Buat simpul baru dan masukkan nilai datanya :



3. Sisipkan simpul baru ke dalam single linked list, pointer pada simpul BARU menunjuk/menyambung ke simpul paling depan :



4. Karena sekarang posisi simpul paling depan adalah simpul BARU, maka pointer HEAD menunjuk ke simpul BARU :



Statement penyisipan simpul di depan single linked list :

```
BARU = new simpul;
BARU -> info = 75;
BARU -> next = NULL;
if (HEAD == NULL)
        TAIL = BARU;
else
        BARU -> next = HEAD;
HEAD = BARU;
```

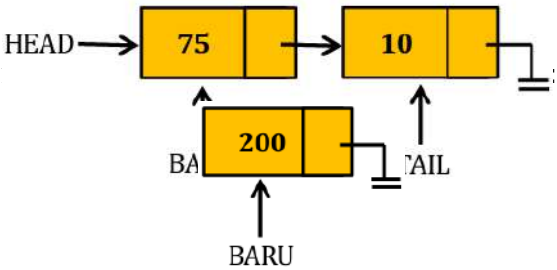
3. Menyisipkan Simpul di Belakang Single Linked List

Menyisipkan simpul di belakang linked list artinya menambahkan simpul baru pada posisi setelah simpul yang paling belakang dalam single linked list, sehingga simpul baru tersebut menjadi simpul terakhir.

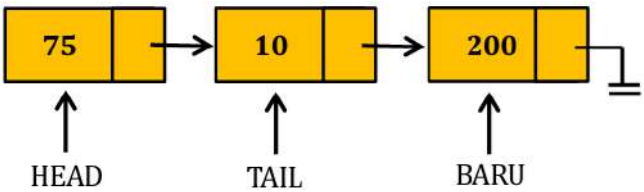
Algoritma penyisipan simpul di belakang single linked list adalah sebagai berikut :

1. Kondisi single linked list sebelum penyisipan :

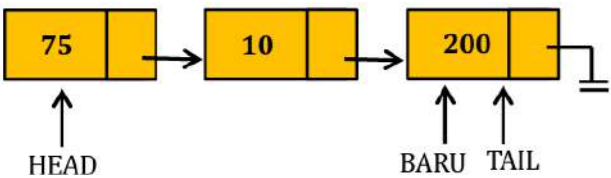
2. Buat s



3. Sisipkan simpul baru ke dalam single linked list, pointer pada simpul paling belakang (HEAD) menunjuk/menyambung ke simpul BARU :



4. Karena sekarang posisi simpul paling belakang adalah simpul BARU, maka pointer TAIL menunjuk ke simpul BARU :



Statement penyisipan simpul di belakang single linked list :

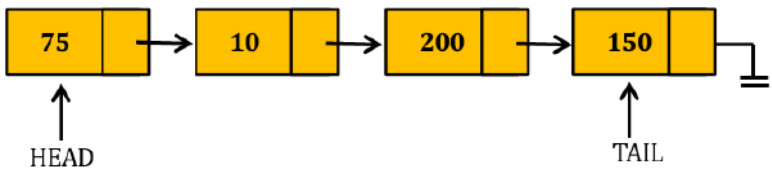
```
BARU = new simpul;
BARU -> info = 200;
BARU -> next = NULL;
if (HEAD == NULL)
        HEAD = BARU;
else
        TAIL -> next = BARU;
TAIL = BARU;
```

4. Menyisipkan Simpul di Tengah Single Linked List

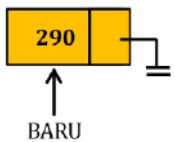
Menyisipkan simpul di tengah linked list artinya menambahkan simpul baru pada posisi setelah simpul tertentu. Menyisipkan simpul di tengah linked list melibatkan penambahan simpul baru di antara dua simpul yang sudah ada. Langkah-langkah ini memerlukan pencarian simpul yang ada di posisi yang diinginkan, dan kemudian menyisipkan simpul baru dengan mengatur ulang pointer.

Algoritma penyisipan simpul di tengah single linked list adalah sebagai berikut :

1. Kondisi single linked list sebelum penyisipan :



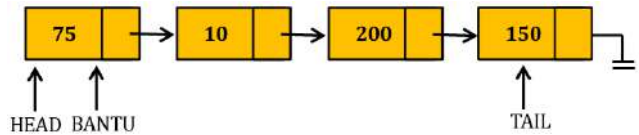
2. Buat simpul baru dan masukkan nilai datanya :



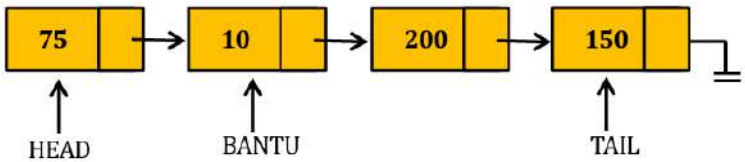
3. Cari posisi penyisipan. Mulai dari head dan cari simpul yang berada sebelum posisi tempat simpul baru akan disisipkan. Misalnya, jika ingin menyisipkan simpul di posisi ke 3, maka cari simpul di posisi ke 2.

4. Untuk mencari posisi yang diinginkan, maka dibutuhkan sebuah pointer BANTU. Pointer BANTU inilah yang akan bergerak dari head dan akan berhenti pada posisi – 1.

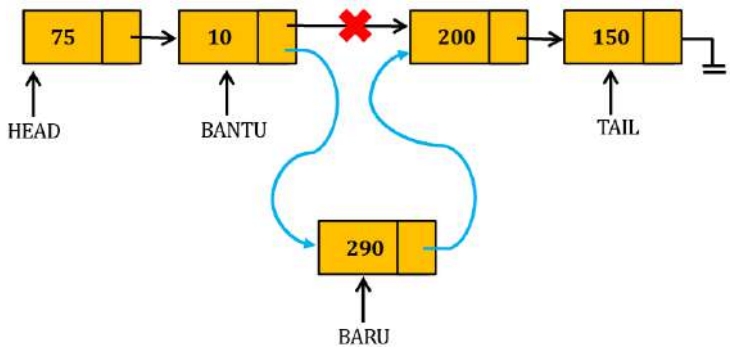
   Semula arahkan dahulu pointer BANTU di posisi head :



   Kemudian gerakkan pointer BANTU ke simpul berikutnya sampai menemukan posisi – 1 (karena kita akan menyisipkan simpul baru pada posisi = 3, pointer BANTU harus berhenti pada posisi simpul ke 2) :



5. Atur pointer, yaitu pointer simpul BARU menunjuk ke simpul setelah BANTU dan pointer dari simpul BANTU menunjuk ke simpul BARU :



Sehingga sekarang simpul dengan data 10 menyambung ke simpul dengan data 290, dan simpul dengan data 290 menyambung ke simpul dengan data 200. Simpul baru berhasil disisipkan pada posisi ke-3 dalam single linked list.

Statementnya menyisipkan simpul di tengah single linked list :

```
BARU = new simpul;
BARU -> info = 290;
BARU -> next = NULL;
Posisi = 3;
if (head==NULL)
{
        HEAD = BARU;
        TAIL = BARU;
}
else
{
        BANTU = HEAD;
        Letak = 1;
        while ((Letak != Posisi-1) && (BANTU != NULL))
        {
                BANTU = BANTU -> next;
                Letak++;
        }
                BARU -> next = BANTU -> next;
                BANTU -> next = BARU;
        }
```

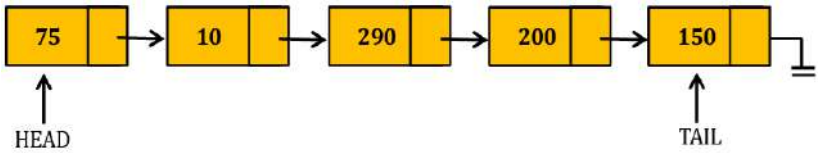
5. Menghapus Simpul di Depan Single Linked List

Menghapus simpul di depan single linked list (yaitu simpul head) adalah operasi menghapus simpul yang berada pada posisi paling depan dalam sebuah single linked list. Operasi ini melibatkan pengubahan pointer head untuk menunjuk ke

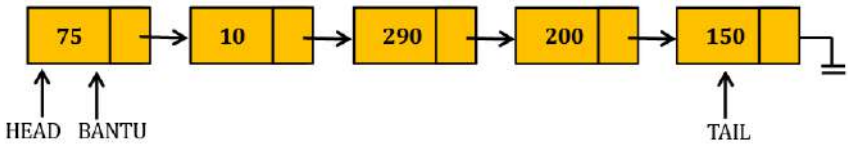
simpul berikutnya, kemudian membebaskan memori dari simpul yang dihapus.
Algoritma menghapus simpul di depan single linked list :

1. Periksa apakah single linked list kosong.
   Jika single linked list kosong (head == NULL), maka tidak ada simpul yang bisa dihapus, jadi operasi dihentikan.

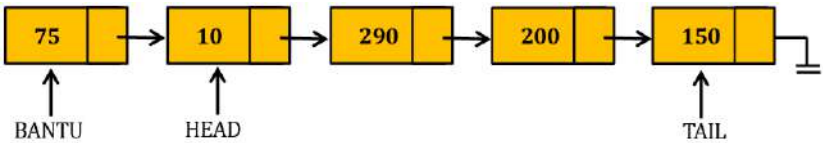
2. Jika single linked list tidak kosong, maka lakukan langkah 3 sampai 5. Misal kondisi single linked list sebelum penghapusan :



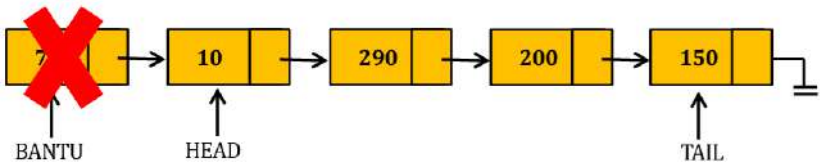
3. Tunjuk sebuah pointer bantuan yang menunjuk ke posisi simpul di head :



4. Lalu pindahkan pointer head ke simpul berikutnya :



5. Hapus simpul yang di depan dan bebaskan memori simpul lama :



Statement untuk menghapus simpul di depan single linked list :

```
if (HEAD == NULL)
    cout<<“TIDAK ADA SIMPUL YANG DIHAPUS“;
else
{
    BANTU = HEAD;
    HEAD = BANTU -> next;
    DELETE BANTU;
}
```

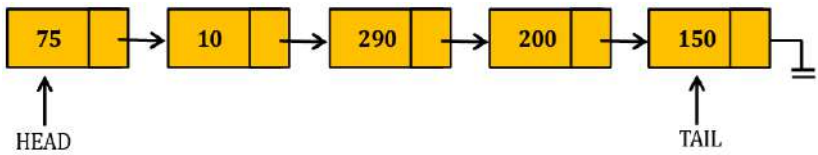
6. Menghapus Simpul di Belakang Single Linked List

Menghapus simpul di belakang single linked list adalah proses menghapus simpul yang berada pada posisi paling akhir dalam single linked list. Berbeda dengan penghapusan simpul di depan, kita perlu mencari simpul kedua terakhir, mengubah pointer next-nya menjadi NULL, dan kemudian membebaskan memori simpul terakhir.

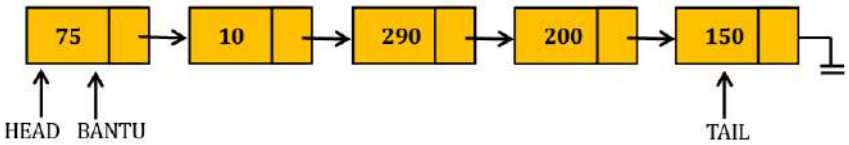
Algoritma menghapus simpul di belakang single linked list :

1. Periksa apakah single linked list kosong :
   Jika single linked list kosong (head == NULL), maka tidak ada simpul yang bisa dihapus, jadi operasi dihentikan.

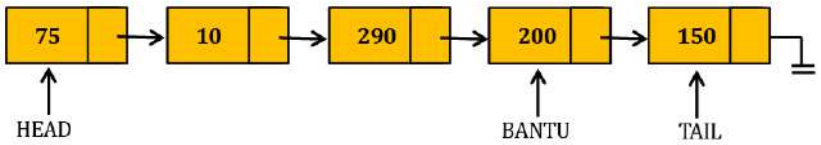
2. Jika single linked list tidak kosong, maka lakukan Langkah 3 sampai 8. Misal kondisi single linked list sebelum penghapusan :



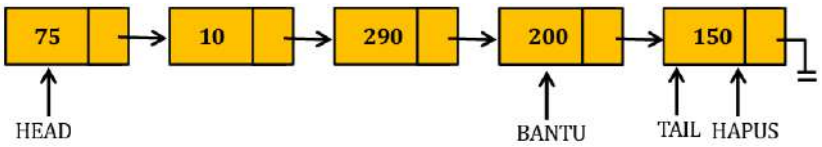
3. Cari 2 simpul yang terakhir. Dimulai dengan menempatkan pointer bantuan (BANTU) pada simpul di head :



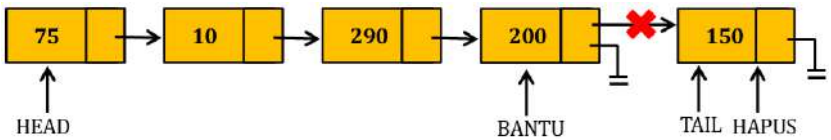
4. Kemudian gerakkan pointer BANTU ke simpul berikutnya sampai pointer BANTU berada pada posisi simpul sebelum simpul yang terakhir :



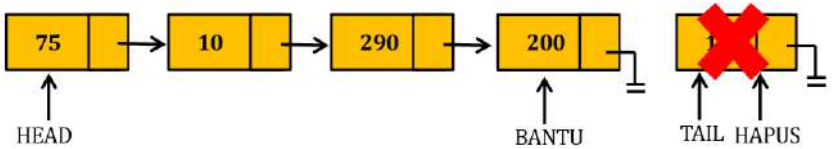
5. Tentukan pointer HAPUS untuk menunjuk ke TAIL :



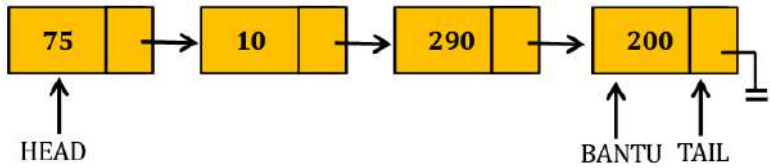
6. Set pointer penunjuk pada simpul BANTU menjadi NULL :



7. Hapus simpul yang di belakang (HAPUS) dan bebaskan memori simpul lama :



8. Arahkan pointer TAIL ke simpul BANTU :



Statement menghapus simpul di belakang single linked list :

```
if (HEAD == NULL)
    cout<<"TIDAK ADA SIMPUL YANG DIHAPUS";
else
{
    BANTU = HEAD;
    while (BANTU -> next -> next != NULL)
            BANTU = BANTU -> next;
    HAPUS = TAIL;
    TAIL = BANTU;
    TAIL -> NEXT == NULL;
    DELETE HAPUS;
}
```

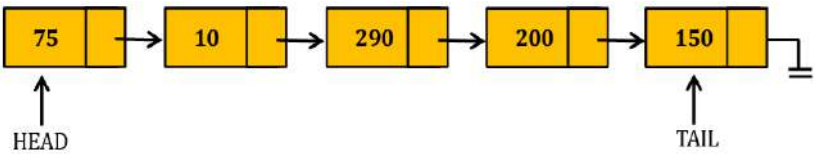
7. Menghapus Simpul di Tengah Single Linked List

Untuk menghapus simpul di tengah single linked list, kita harus menemukan simpul sebelum simpul yang ingin dihapus, lalu mengubah penunjuk (pointer) next dari simpul tersebut agar melewati simpul yang dihapus dan menunjuk ke simpul setelahnya. Setelah itu, kita membebaskan memori dari simpul yang dihapus.

Algoritma menghapus simpul di Tengah single linked list :

1. Periksa apakah single linked list kosong :
   Jika single linked list kosong (head == NULL), maka tidak ada simpul yang bisa dihapus, jadi operasi dihentikan.

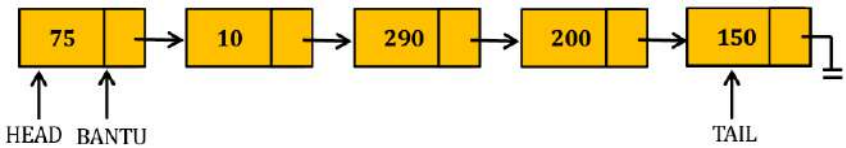
2. Jika single linked list tidak kosong, maka lakukan Langkah 3 sampai 8. Misal kondisi single linked list sebelum penghapusan :



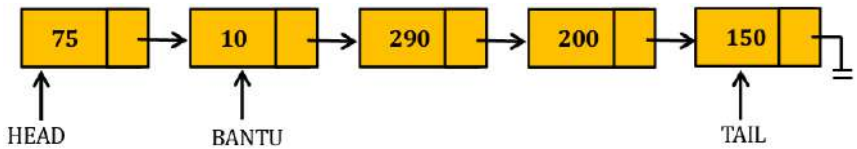
3. Tentukan posisi simpul yang akan dihapus. Temukan simpul sebelum simpul yang akan dihapus, yaitu mulai dari head dan cari simpul yang berada sebelum simpul yang ingin dihapus. Misal kita akan menghapus simpul pada posisi ke-3, maka kita harus menemukan simpul pada posisi ke-2.

4. Untuk mencari posisi yang diinginkan, maka dibutuhkan sebuah pointer BANTU. Pointer BANTU inilah yang akan bergerak dari head dan akan berhenti pada posisi – 1.

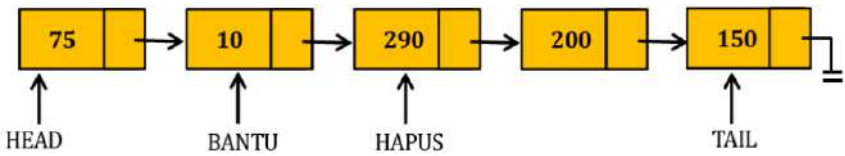
   Semula arahkan dahulu pointer BANTU di posisi head :



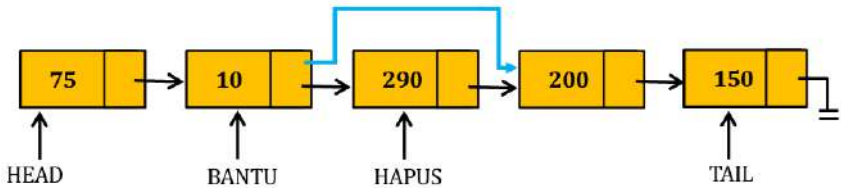
5. Kemudian gerakkan pointer BANTU ke simpul berikutnya sampai menemukan posisi – 1 (karena kita akan menghapus simpul pada posisi = 3, pointer BANTU harus berhenti pada posisi simpul ke 2) :



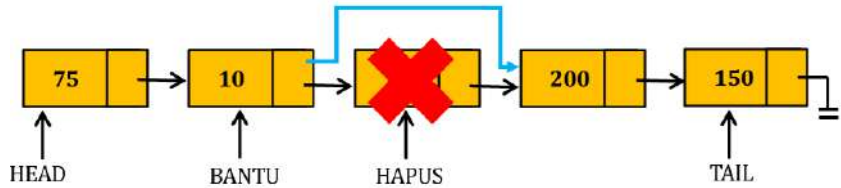
6. Tentukan pointer HAPUS untuk menunjuk ke simpul setelah BANTU :



7. Atur pointer, yaitu pointer simpul BANTU menunjuk ke simpul setelah HAPUS :



8. Hapus simpul HAPUS dan bebaskan memori simpul lama :



Statement penghapusan simpul di tengah single linked list :

```
Posisi = 3;
if (HEAD == NULL)
   cout<<“TIDAK ADA SIMPUL YANG DIHAPUS“;
else
{
   BANTU = HEAD;
   Letak = 1;
   while ((Letak != Posisi-1) && (BANTU != NULL))
            {
                  BANTU = BANTU -> next;
                  Letak++;
            }
            HAPUS = BANTU -> next;
            BANTU -> next = HAPUS -> next;
            DELETE HAPUS;
}
```

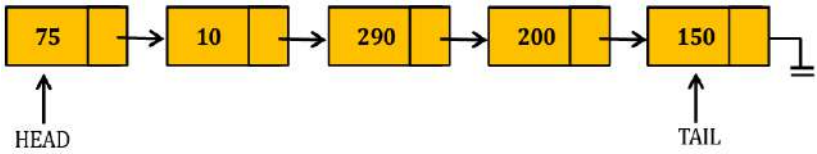
8. Mencetak Simpul Dalam Single Linked List

Untuk mencetak simpul dalam single linked list, maka harus menelusuri setiap simpul mulai dari head hingga simpul terakhir. Kita mencetak nilai setiap simpul saat menelusurinya, kemudian berhenti saat pointer next dari simpul menunjuk ke NULL (simpul terakhir).

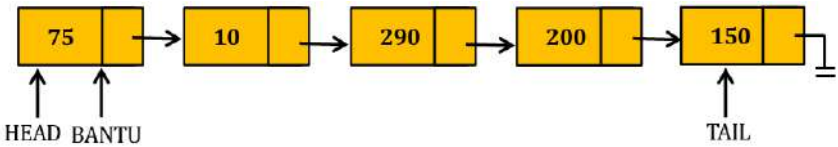
Algoritma mencetak simpul dalam single linked list :

1. Periksa apakah single linked list kosong :
   Jika single linked list kosong (head == NULL), maka tidak ada simpul yang bisa dicetak, jadi operasi dihentikan.

2. Jika single linked list tidak kosong, maka lakukan langkah 3 sampai 8. Misal kondisi single linked list sekarang :



3. Tentukan pointer BANTU untuk menunjuk ke simpul HEAD :



4. Cetak nilai data pada simpul yang ditunjuk oleh pointer BANTU. Kemudian gerakkan pointer BANTU ke simpul berikutnya dan cetak lagi nilai data simpulnya. Begitu seterusnya sampai pointer BANTU bernilai NULL.

Statement mencetak simpul dalam single linked list :

```
if (HEAD == NULL)
    cout<<"SINGLE LINKED LIST KOSONG";
else
{
    BANTU = HEAD;
    while (BANTU != NULL)
    {
            cout<<BANTU -> info;
            BANTU = BANTU -> next;
    }
}
```

DAFTAR PUSTAKA

Kadir, Abdul. (2001). *Pemrograman C++ : Membahas Pemrograman Berorientasi Objek Menggunakan Turbo C++ dan Borland C++.* Yogyakarta : Andi Offset.

Aho, Alfred V.; Hopcroft, John E.; Ullman, Jeffrey D. (1987). *Data Structures and Algorithm*. Addison Wesley Publishing Company.

Geeksforgeeks. (2024). *Linked List vs Array*. Diakses pada laman : https://www.geeksforgeeks.org/linked-list-vs-array

Standish, Thomas. (1995). *Data Structures, Algorithms, and Software Principles in C*. Pearson Education.

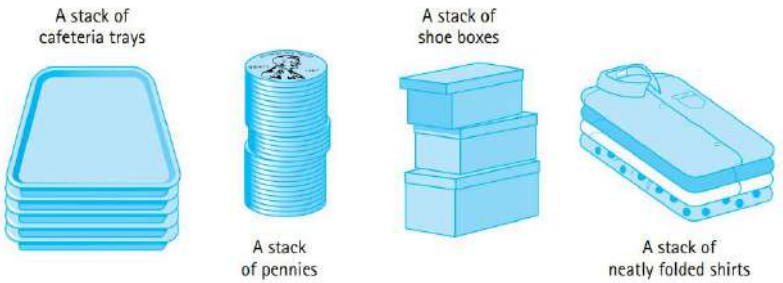
BAB 6
STACK DENGAN ARRAY

Oleh Sugiarto Cokrowibowo

6.1 Struktur Data Stack

Stack atau sering juga disebut tumpukan merupakan struktur data linear dinamis. Elemen-elemen di dalam *stack* dapat bertambah dan berkurang. Urutan operasi penambahan dan pengurangan elemen-elemen di dalam *stack* menggunakan mekanisme *Last In First Out* (LIFO) yaitu elemen akan dimasukkan melalui satu pintu dan keluar melalui pintu yang sama sedemikian sehingga elemen yang terakhir dimasukkan akan menjadi elemen pertama yang akan dikeluarkan dari stack. Perhatikan gambar 6.1 yang merupakan representasi tumpukan dari beberapa objek atau benda. Defenisi pada tingkat abstrak, *stack* adalah kelompok elemen homogen yang terurut. Penghapusan elemen yang ada dan penambahan elemen baru hanya dapat terjadi di bagian atas stack. Misalnya, jika kemeja biru favorit Anda berada di bawah kemeja merah yang sudah pudar dan lama di tumpukan kemeja, pertama-tama Anda mengambil kemeja merah dari bagian atas tumpukan. Kemudian Anda mengeluarkan kemeja biru, yang sekarang berada di bagian atas tumpukan. Kemeja merah kemudian dapat diletakkan kembali di bagian atas tumpukan. Atau bisa juga dibuang! (Dale et al., 2018)(Karumanchi, 2010).

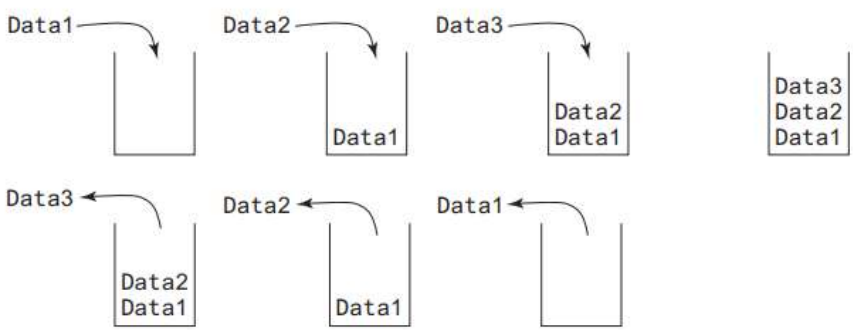
Tumpukan dapat dianggap sebagai struktur yang "terurut" karena elemen-elemen muncul secara berurutan sesuai dengan lamanya mereka berada dalam tumpukan. Elemen-elemen yang telah berada dalam tumpukan lebih awal akan berada di bagian terbawah sedangkan yang terakhir akan berada di bagian teratas. Pada setiap saat, jika diberikan dua elemen dalam tumpukan, yang satu lebih tinggi daripada yang lain atau dengan kata lain dapat dinyatakan bahwa terdapat urutan hierarki atau urutan level posisi di dalam *stack* (tumpukan) Misalnya, kemeja merah berada di bagian atas tumpukan daripada kemeja biru (Dale et al., 2018). Stack digunakan saat kita memiliki tumpukan item, dan hanya bekerja dengan item teratasnya. Item di atas selalu merupakan item yang terakhir dimasukkan ke dalam tumpukan (Filho, 2017). Sehingga mekanisme *Last In First Out* (LIFO) yang telah disebutkan di awal dapat juga disebut sebagai *First In Last Out* (FILO) (Gersting, 2014).



Gambar 6. 1. Stack di dunia nyata
(Sumber : Object-oriented data structures using Java. Figure 2.2 (Dale et al., 2018))

Ilustrasi untuk urutan operasi LIFO ini diperlihatkan pada gambar 6.2. dimana elemen data dimasukkan ke dalam *stack* satu persatu dimulai dari Data1 yang menempati posisi paling dasar di *stack*. Kemudian saat Data2 dimasukkan ke dalam *stack* maka Data2 akan menempati posisi di atas Data1. Demikian pula

seterusnya, saat Data3 dimasukkan ke dalam *stack* maka Data3 akan menempati posisi di atas Data2. Sehingga saat ini *stack* berisi Data1, Data2, dan Data3 yang terurut dari bawah (bagian dasar *stack*) sampai ke atas. Hal sebaliknya terjadi saat hendak mengeluarkan elemen data dari dalam *stack.* Maka elemen yang pertama kali dikeluarkan adalah elemen yang paling terakhir dimasukkan ke dalam *stack* (*Last In First Out*)dalam hal ini adalah Data3. Jadi data pertama yang dapat kita keluarkan adalah Data3, selanjutnya di dalam *stack* akan tersisa elemen Data2 dan Data1. Data berikutnya yang dapat kita keluarkan setelah mengeluarkan Data3 adalah Data2 dan di dalam *stack* tersisa elemen Data1. Data1 merpakan elemen terakhir yang dapat kita keluarkan dari dalam *stack*. Dan diketahui bersama bahwa ternyata Data1 ini merupakan elemen yang paling pertama dimasukkan ke dalam *stack* (*Fist In*) dan sekarang menjadi elemen yang terakhir dikeluarkan dari *stack* (*Last Out*) (Daniel Liang, 2019).



Gambar 6. 2. Penanganan LIFO pada *stack*
(Sumber : Introduction to Java™ Programming and Data Structures, Comprehensive Version, Eleventh Edition, Global Edition. Figure 10.11 (Daniel Liang, 2019))

Daftar berikut berisi aplikasi penggunaan dan kasus penggunaan untuk struktur data berbasis tumpukan:

- rekursi
- pelacakan pemanggilan fungsi
- evaluasi ekspresi
- pembalikan karakter
- servicing hardware interrupts
- penyelesaian masalah kombinatorial menggunakan algoritma backtracking (Campesato, 2023).

6.2 Operasi-Operasi Pada Stuktur Data Stack

Secara konsep pada struktur data *stack* hanya elemen data paling atas (*top element*) yang dapat terlihat (visible) dari luar dan dapat diakses. Sehingga secara umum operasi-operasi pada struktur data *stack* ini berkaitan atau dilakukan pada *top element*. Seperti operasi *peek* yang digunakan untuk memeriksa (inspect) elemen terakhir (*top*) yang ada pada *stack*. Operasi lainnya yang juga melibatkan *top element* adalah operasi *pop* yang digunakan untuk mengambil/menghapus elemen (retrieve) dari *stack*. Selain operasi *peek* dan operasi *pop* yang melibatkan *top element* juga terdapat operasi *push* yang digunakan untuk menambahkan elemen terakhir yang akan berada di posisi *top* (KOFFMAN & WOLFGANG, 2016). Masih banyak operasi yang dapat dibuat untuk melengkapi struktur data *stack* sesuai kebutuhan programer, namun berikut ini operasi-operasi standar yang umumnya ada pada struktur data *stack*.

Tabel 6. 1. Operasi Standar pada struktur data *stack*.

Operasi	Deskripsi fungsi
isEmpty( )	Mengembalikan nilai **TRUE** jika *stack* dalam keadaan kosong (belum ada elemen yang dimasukkan ke *stack*).
isFull( )	Mengembalikan nilai **TRUE** jika *stack* dalam keadaan penuh atau **MAX_SIZE** telah tercapai.
size( )	Mengembalikan banyaknya elemen yang tersimpan di dalam *stack*.
peek( )	Mengembalikan *top element* yang ada di dalam *stack*
pop( )	Mengembalikan *top element* yang ada di dalam *stack* serta menghapus *top element* dari dalam *stack*.
push(E element)	Menambahkan elemen baru ke dalam *stack*. Elemen yang baru ditambahkan ini akan menjadi *top element* yang baru.

Sumber: (KOFFMAN & WOLFGANG, 2016)(Campesato, 2023)

6.3 Membuat Struktur Data Stack Menggunakan Array

Pada pembahasan berikut ini akan dibuat struktur data *stack* yang berisi operasi-operasi standar sturktur data *stack* menggunakan *array*. Sebenarnya pembuatan struktur data *stack* menggunakan *array* termasuk pilihan yang kurang efisien dikarenakan struktur *array* yang umumnya tidak dinamis dalam hal ukuran (*size*) sehingga dibeberapa bahasa pemrograman tingkat tinggi seperti java lebih direkomendasikan untuk

menggunakan struktur *vector, array list,* maupun *linked list* alih-alih menggunakan array. Namun pada kasus kita bekerja menggunakan bahasa pemrograman tingkat rendah atau mungkin kita akan menggunakan program untuk *hardware* dengan spesifikasi rendah dan hanya *array* yang tersedia maka kita harus dapat berimprofisasi untuk menggunakan *array* sebagai struktur data dasar untuk membangun *stack*. Hal yang penting perlu diingat pula bahwa dalam hal panjang elemen (*size*) atau banyaknya elemen yang dapat disimpan didalam *stack* sifatnya adalah dinamis artinya jumlah elemennya bisa berkurang dan bertambah menggunakan operasi *pop* dan *push*, sedangkan pada *array* panjang elemennya bersifat statis. Sekali panjang elemen array ditetapkan maka panjangnya tidak akan berubah kecuali *array* tersebut diinisialisasi kembali. Solusi untuk masalah ini adalah kita dapat menetapkan *MAX_SIZE* untuk *array* dengan nilai yang sangat besar misalnya 100. Tapi bagaimana jika *MAX_SIZE* tercapai, misalnya kita hendak menambahkan elemen ke 101. Jika dipaksakan hal ini tentu akan mengakibatkan *RUNTIME ERROR*. Untuk menghindari terjadinya *ERROR* maka kita dapat melakukan inisialisasi kembali terhadap *array* yang digunakan misalnya dengan melipatgandakan *MAX_SIZE* menjadi 200 lalu kemudian memasukkan kembali elemen-elemen yang telah tersimpan sebelumnya ke dalam *array* baru yang saat ini *MAX_SIZE* telah bernilai 200 (dapat menampung 200 elemen). Operasi memperbesar *MAX_SIZE* ini terus dilakukan setiap hendak menambahkan elemen baru di saat size *array* kembali mencapai MAX_SIZE (Schildt & Coward, 2024)(Lowe, 2023)(Weiss, 2012).

Berikut ini kita akan membuat struktur data *stack* menggunakan bahasa pemrograman Java. Pada bahasa pemrograman Java struktur data diterapkan atau dibuat dalam bentuk *class* dan operasi-operasi struktur datanya dibuat dalam bentuk method. pada pembahasan berikut ini akan

diperlihatkan langkah-langkah pembuatan struktur data *stack* menggunakan *array* di Java.

6.3.1 Membuat *Class* untuk Struktur Data *Stack*

Langkah awal yang dilakukan untuk membuat struktur data *stack* adalah membuat *class*. Di Java sebuah *class* adalah cetak biru untuk membuat objek. Dari *class* **Stack** ini nanti kita dapat membuat tak berhingga objek *stack*. *Class* digunakan untuk membuat stuktur data yang tentunya akan berisi data berikut operasi-operasi untuk mengelola data yang tersimpan di dalamnya.

Pada contoh penerapan *stack* kali ini akan digunakan array yang mula-mula memiliki *MAX_SIZE* = 10. *MAX_SIZE* ini akan diperpanjang sesuai kebutuhan nantinya. Selanjutnya diinisialisasi pula *array* bertipe **int** yang diberi nama *array* data. Tipe **int** ini digunakan jika kita hendak menyimpan elemen-elemen array pertipe **int** (bilangan bulat), jika elemen-elemen yang akan disimpan bertipe lain (misalnya: double, boolean, String, Mahasiswa, Buku, long, short) maka tipe **int** ini harus disesuaikan atau dapat juga dideklarasikan menggunakan tipe *generic* (tipe *generic* tidak akan dibahas pada subbab ini).

Berikut ini deklarasi *class* untuk struktur data *stack*:

```
public class Stack {
   private int MAX_SIZE = 10;
   private int[]data = new int[MAX_SIZE];
   private int top = -1;

}
```

Nilai awal untuk variabel top adalah -1. Nilai ini dapat digunakan untuk mengindikasikan apakah *stack* dalam keadaan kosong (*isEmpty*) atau tidak. Variabel-variabel **MAX_SIZE**, **data**, dan **top** diberi visibilitas **private** yang artinya bahwa keriga variabel ini tidak dapat terlihat dari luar *class stack* dan *object* yang dibuat dari *class stack*. Pada langkah berikutnya akan dibuat operasi-operasi untuk mengelola nilai variabel-variabel yang ada pada *stack*. Operasi-operasi ini dibuat menggunakan *method* di Java.

6.3.2 Membuat Operasi isEmpty( )

Operasi **isEmpty( )** digunakan untuk memeriksa apakah *stack* dalam keadaan kosong atau tidak atau dengan kata lain menyatakan apakah terdapat elemen di dalam *stack*. Operasi **isEmpty( )** ini akan mengembalikan nilai true jika *stack* dalam keadaan kosong. Kondisi kosong atau tidaknya *stack* dapat dideteksi dari nilai variabel **top** jika nilai variabel **top** = -1 atau lebih kecil dari 0 maka *stack* sedang dalam keaddan kosong atau tidak ada elemen yang disimpan di dalam *array* yang kita gunakan untuk membuat *stack*, sebagaimana kita ketahui indeks *array* terkecil adalah 0.

Berikut ini *source code* untuk membuat operasi **isEmpty( )**:

```
public boolean isEmpty( ){
   if(top<0){
      return true;
   }else{
      return false;
   }
}
```

Operasi **isEmpty( )** akan mengembalikan nilai **true** jika variabel **top** bernilai lebih kecil dari 0. Sebaliknya jika nilai variabel **top** bernilai lebih besar atau sama dengan 0 maka operasi **isEmpty( )** akan mengembalikan nilai **false**.

6.3.3 Membuat Operasi isFull( )

Method **isFull( )** adalah kebalikan dari operasi **isEmpty( )**. Operasi **isFull( )** digunakan untuk memeriksa apakah array yang digunakan untuk membuat *stack* telah terisi penuh atau belum. Kondisi *full* ini bisa dideteksi dari nilai variabel **top**. Karena index array paling kecil adalah 0 maka index *array* terakhir akan bernilai = **MAX_SIZE** – 1. Nah jika niali variabel **top** sudah bernilai = **MAX_SIZE** – 1 atau lebih besar lagi, maka kondisi ini mengindikasikan bahwa *stack* sudah berada dalam kondisi *full*. Dari kondisi full ini nantinya kita dapat melakukan penanganan misalnya kita hendak menampahkan elemen baru sementara *stack* sudah berada dalam kondisi *full* maka harus kita perbesar terlebih dahulu size dari *array* yang digunakan. Tapi itu nanti, saat ini kita hanya akan mendeteksi apakah *stack* dalam keadaan *full* atau tidak berdasarkan nilai dari variabel **top**.

```
public boolean isFull( ){
   if(top>=MAX_SIZE-1){
      return true;
   }else{
      return false;
   }
}
```

6.3.4 Membuat Operasi size( )

Operasi size( ) digunakan untuk menyatakan panjang atau banyaknya elemen yang tersimpan di dalam *stack*. Hal ini dapat dilakukan menggunakan operasi sederhana dan masih menggunakan nilai variabel **top**. Banyaknya elemen yang tersimpan di dalam *stack* dapat diperoleh dari nilai variabel **top + 1**. Jika *stack* dalam keadaan kosong maka nilai size adalah 0.

```
public int size( ){
   if(isEmpty( )){
      return 0;
   }else{
      return top + 1;
   }
}
```

6.3.5 Membuat Operasi peek( )

Operasi peek( ) digunakan untuk memeriksa atau melihat elemen yang berada pada posisi **top** dalam hal ini elemen yang terakhir dimasukkan. Untuk menampilkan elemen data pada posisi top sebenarnya cukup menggunakan kode **data[top]** untuk menampilkan elemen array data pada indeks = **top**. Namun, programmer perlu mengantisipasi jika nilai **top** masih sama dengan **-1** dalam hal ini *stack* masih dalam keadaan kosong (*empty*). Dalam kondisi kosong tersebut maka program harus tetap berjalan dan ditangani dengan memberikan nilai kembalian *default* **-999** (hanya contoh nilai *default* yang dapat dirubah). Apa yang sebenarnya akan terjadi jika kita memaksakan program untuk mengakses nilai **top = -1**?. Hal ini akan mengakibatkan pemanggilan elemen pada indeks -1: data[-

1]. Hal ini dapat mengakibatkan error **Index Out Of Bound** karena tidak ada indeks elemen data yang lebih kecil dari 0.

Berikut ini *source code* untuk membuat operasi **peek( )**:

```
public int peek() {
   if (isEmpty()) {
      System.err.println("Stack is Empty");
      return -999;
   } else {
      return data[top];
   }
}
```

6.3.6 Membuat Operasi pop( )

Mirip dengan operasi peek( ), pada operasi pop( ) dilakukan pembacaan data pada posisi **top** sekaligus menghapusnya dari stack. Jadi setelah dilakukan operasi pop( ) maka elemen pada posisi **top** akan terhapus dan indeks **top** berkurang satu (*decrement*) akibat dari operasi **top--**.

Berikut ini *source code* untuk membuat operasi **pop( )**:

```
public int pop() {
   if (isEmpty()) {
      System.err.println("Stack is Empty");
      return -999;
   } else {
```

```
        int value = data[top];
        data[top] = 0;
        top--;
        return value;
    }
}
```

6.3.7 Membuat Operasi push(E element)

Operasi push(E element) digunakan untuk menambahkan elemen baru ke dalam *stack*. Elemen yang ditambahkan ini akan menempati posisi **top** yang baru. Jika *stack* telah berada dalam kondisi full (MAX_SIZE telah tercapai) sebelum elemen baru ini ditambahkan ke dalam *stack* maka terlebih dahulu harus dilakukan perluasan ukuran/kapasitas *MAX_SIZE* dari *stack*. Dalam contoh program berikut kita menambahkan 10 kapasitas elemen baru ke array data. Mekanisme penambahan kapasitasnya adalah dengan membuat array data yang baru (int[ ] new_data) dengan kapasitas MAX_SIZE yang telah diperbesar, kemudian melakukan copy-array untuk memindahkan semua elemen array **data** yang sebelumnya ke **new_data**. Disini digunakan mekanisme *copy-array* satu demi satu elemen ke *array* yang baru ini untuk menghilangkan pengaruh tipe *reference* dari tipe array pada variabel **data** sebelumnya. Setelah semua elemen **data** di-copy ke **new_data** selanjutnya variabel **data** kita inisialisasikan kembali menggunakan array **new_data**. Proses ini nampaknya agak rumit dan merepotkan, tapi demikianlah adanya seorang programmer sesungguhnya senantiasa berpikir rumit dan mengantisipasi error yang mungkin akan terjadi.

Berikut ini *source code* untuk membuat operasi **push(E element)**:

```
public void push(int element) {
   if (isFull()) {
      MAX_SIZE = MAX_SIZE + 10;
      int[] new_data = new int[MAX_SIZE];
      for (int i = 0; i < data.length; i++) {
         new_data[i] = data[i];
      }
      data = new_data;
   }
   top++;
   data[top] = element;
}
```

Pada operasi push(E element) ini terdapat sebuah parameter input **element** yang dalam contoh program kali ini bertipe **int**, maka deklarasi operasinya ditulis menjadi **push(int element)**. Operasi push( ) ini digunakan untuk menambahkan elemen ke dalam *stack* dan elemen yang baru ditambahkan tersebut akan menempati posisi teratas di *stack* (menjadi elemen **top** yang baru)

6.3.8 Membuat Operasi toString( )

Operasi atau *method toString( )* ini sebenarnya bukan operasi standar yang harus ada pada struktur data *stack* namun disini dibutuhkan untuk menampilkan atau mengkonversi isi elemen-elemen yang ada di dalam *stack* ke dalam format/tipe

data *String* untuk kemudian dicetak ke jendela *console* menggunakan *method System.out.println( )* nanti saat pengujian struktur data *stack*. Operasi *toString( )* dibuat dengan bantuan *class* **StringBuffer**. Berikut ini *source code method toString( )* yang kita buat:

```
public String toString() {
   StringBuffer sb = new StringBuffer();
   sb.append("[");
   if (isEmpty() == false) {
      for (int i = 0; i <= top; i++) {
         if (i > 0) {
            sb.append(", ");
         }
         sb.append(data[i]);
      }
   }
   sb.append("]");
   return sb.toString();
}
```

6.3.9 Source Code lengkap untuk class Stack

Setelah menambahkan operasi-operasi standar kedalam *class* **Stack**, berikut ini source code lengkap untuk *class* **Stack** yang kita buat menggunakan **array** menggunakan bahasa pemrograman **Java**. *Class* **Stack** ini sekaligus sebagai bentuk

penerapan langsung dari abstraksi Struktur Data Stack ke dalam bahasa pemrograman.

```
public class Stack {


   private int MAX_SIZE = 10;
   private int[] data = new int[MAX_SIZE];
   private int top = -1;


   public boolean isEmpty() {
      if (top < 0) {
         return true;
      } else {
         return false;
      }
   }


   public boolean isFull() {
      if (top >= MAX_SIZE - 1) {
         return true;
      } else {
         return false;
      }
   }
```

```
public int size() {
   if (isEmpty()) {
      return 0;
   } else {
      return top + 1;
   }
}

public int peek() {
   if (isEmpty()) {
      System.err.println("Stack is Empty");
      return -999;
   } else {
      return data[top];
   }
}

public int pop() {
   if (isEmpty()) {
      System.err.println("Stack is Empty");
      return -999;
```

```
    } else {
        int value = data[top];
        data[top] = 0;
        top--;
        return value;
    }
}

public void push(int element) {
    if (isFull()) {
        MAX_SIZE = MAX_SIZE + 10;
        int[] new_data = new int[MAX_SIZE];
        for (int i = 0; i < data.length; i++) {
            new_data[i] = data[i];
        }
        data = new_data;
    }
    top++;
    data[top] = element;
}

public String toString() {
```

```
    StringBuffer sb = new StringBuffer();
    sb.append("[");
    if (isEmpty() == false) {
       for (int i = 0; i <= top; i++) {
          if (i > 0) {
             sb.append(", ");
          }
          sb.append(data[i]);
       }
    }
    sb.append("]");
    return sb.toString();
  }

}
```

Selanjutnya struktur data **Stack** yang telah dibuat ini akan diuji penggunaan operasi-operasinya menggunakan *class* Main (*class* program) yang diberi nama *class* **Test**.

6.3.10 Menguji Struktur Data Stack

Untuk menguji *class* **Stack** yang telah dibuat, mula-mula kita siapkan sebuac *class* program yang memiliki *method* **main( )** sebagai berikut:

```
public class Test {


   public static void main(String[] args) {


   }
}
```

Langkah selanjutnya di dalam *method* **main( )** dibuat sebuah *object* dari *class* **Stack** (menginstan *class*) menggunakan *keyword* **new**.

```
public class Test {


   public static void main(String[] args) {
      Stack myStack = new Stack();
   }
}
```

Object yang dibuat diberi nama **myStack**. Melalui *object* **myStack** ini nantinya akan dipanggil semua operasi (*method*) yang sebelumnya telah dipersiapkan di dalam *class* **Stack**.

Pada saat awal tentu **myStack** masih dalam keadaan kosong sehingga jika dilakukan pemanggilan operasi *isEmpty( )* pada saat awal *object* **myStack** dibuat maka akan mengembalikan nilai **true** sedangkan jika operasi *isFull( )* yang dipanggil maka akan mengembalikan nilai **false**.

```
        boolean kosongkah = myStack.isEmpty();
        boolean fullkah = myStack.isFull();

System.out.println("==================================
==");
        System.out.println("Kosong        = "+kosongkah);
        System.out.println("Full          = "+fullkah);
        System.out.println("Elements      = "+myStack);
        System.out.println("Top Elements  = "+myStack.peek( ));
        System.out.println("Size          = "+myStack.size( ));
        System.out.println("_____________________________________");
```

Saat kondisi awal dan **myStack** masih kosong hasil *print* di jendela *console* akan terlihat sebagai berikut:

```
run:
==================================
Kosong          = true
Full            = false
Elements        = [ ]
Stack is Empty
Top Elements    = -999
Size                    = 0
_____________________________________
BUILD SUCCESSFUL (total time: 0 seconds)
```

Pesan kesalahan ***Stack is Empty*** ditampilkan mengindikasikan bahwa **myStack** masih dalam keadaan kosong.

Selanjutnya akan dilakukan penambahan beberapa elemen ke dalam **myStack** sebagai berikut:

```
    myStack.push(2);
    myStack.push(3);
    myStack.push(4);
    myStack.push(5);
    myStack.push(6);
    myStack.push(7);
    myStack.push(8);
    myStack.push(9);
    kosongkah = myStack.isEmpty();
    fullkah = myStack.isFull();

System.out.println("==================================
==");
    System.out.println("Kosong        = "+kosongkah);
    System.out.println("Full          = "+fullkah);
    System.out.println("Elements      = "+myStack);
    System.out.println("Top Elements  = "+myStack.peek());
    System.out.println("Size          = "+myStack.size());
    System.out.println("_____________________________________");
```

Namapak di jendela *console* saat program dijalankan maka akan menampilkan nilai operasi *isFull( )* dan *isEmpty( )* kedua bernilai **false**, *size* dari **myStack** akan bernilai sama dengan 8 karena sudah terdapat sebanyak 8 elemen di dalam **myStack** dan **top elemen** bernilai 9 karena elemen yang terakhir dimasukkan dan berada di posisi **top** adalah 9.

```
==================================
Kosong          = false
Full            = false
Elements        = [2, 3, 4, 5, 6, 7, 8, 9]
Top Elements    = 9
Size            = 8

_____________________________________
```

Selanjutnya **myStack** akan terus diisi hingga melampaui batas MAX**_SIZE** = **10** untuk mengetahui apakah **myStack** dapat beradaptasi jika kondisi **top elemen** telah berada diposisi **MAX_SIZE** dimana sebelumnya telah dipersiapkan operasi untuk memperluas **size** *array* yang digunakan di dalam *stack*.

```
    Stack myStack = new Stack();


    boolean kosongkah = myStack.isEmpty();
    boolean fullkah = myStack.isFull();

System.out.println("==================================
==");
    System.out.println("Kosong        = "+kosongkah);
```

```
        System.out.println("Full          = "+fullkah);
        System.out.println("Elements      = "+myStack);
        System.out.println("Top Elements  = "+myStack.peek());
        System.out.println("Size          = "+myStack.size());
        System.out.println("_____________________________");

        myStack.push(2);
        myStack.push(3);
        myStack.push(4);
        myStack.push(5);
        myStack.push(6);
        myStack.push(7);
        myStack.push(8);
        myStack.push(9);
        kosongkah = myStack.isEmpty();
        fullkah = myStack.isFull();

System.out.println("================================
==");
        System.out.println("Kosong        = "+kosongkah);
        System.out.println("Full          = "+fullkah);
        System.out.println("Elements      = "+myStack);
        System.out.println("Top Elements  = "+myStack.peek());
```

```
        System.out.println("Size          = "+myStack.size());
        System.out.println("_____________________________________");


        myStack.push(11);
        myStack.push(12);
        myStack.push(13);
        myStack.push(14);
        myStack.push(15);
        kosongkah = myStack.isEmpty();
        fullkah = myStack.isFull();

System.out.println("==================================
==");
        System.out.println("Kosong        = "+kosongkah);
        System.out.println("Full          = "+fullkah);
        System.out.println("Elements      = "+myStack);
        System.out.println("Top Elements  = "+myStack.peek());
        System.out.println("Size          = "+myStack.size());
        System.out.println("_____________________________________");
```

Nampak bahwa **size** dari **myStack** saat ini sudah mencapai angka 13 dan tidak terjadi **error**. Artinya *array* di dalam **myStack** berhasil diperluas ukurannya yang semula adalah sama dengan 10 (**MAX_SIZE = 10**)saat ini telah menjadi sama dengan 20 (**MAX_SIZE = MAX_SIZE + 10)**.

```
==================================
Kosong                  = false
Full                    = false
Elements                = [2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15]
Top Elements            = 15
Size                    = 13

_____________________________
```

Selanjutnya kita dapat menguji operasi pop( ) untuk mengambil atau penghapus elemen-elemen yang ada di dalam **myStack**. Kali ini penghapusan elemen akan dilakukan di dalam perulangan **while** (*loop* **while**) hingga **myStack** kembali berada dalam keadaan kosong.

```
System.out.println("OPERASI
POP============================");
      while (myStack.isEmpty() == false) {
         int value = myStack.pop();
         kosongkah = myStack.isEmpty();
         fullkah = myStack.isFull();

System.out.println("==================================
==");
         System.out.println("Kosong        = " + kosongkah);
         System.out.println("Full          = " + fullkah);
         System.out.println("Elements      = " + myStack);
         System.out.println("POP           = " + value);
```

```
        System.out.println("Size          = " + myStack.size());
        System.out.println("_____________________________________");
    }
    System.out.println("AKHIR DARI OPERASI POP================");
```

Saat perulangan **while** untuk operasi *pop( )* ini dijalankan maka operasi *pop( )* akan terus dipanggil selama nilai operasi *isEmpty( )* bernilai **false**.

Source code lengkap untuk pengujian program diperlihatkan sebagai berikut:

```
public class Test {


  public static void main(String[] args) {
    Stack myStack = new Stack();


    boolean kosongkah = myStack.isEmpty();
    boolean fullkah = myStack.isFull();

System.out.println("================================");
    System.out.println("Kosong        = " + kosongkah);
    System.out.println("Full          = " + fullkah);
    System.out.println("Elements      = " + myStack);
    System.out.println("Top Elements  = " + myStack.peek());
```

```
        System.out.println("Size          = " + myStack.size());
        System.out.println("_____________________________________");


        myStack.push(2);
        myStack.push(3);
        myStack.push(4);
        myStack.push(5);
        myStack.push(6);
        myStack.push(7);
        myStack.push(8);
        myStack.push(9);
        kosongkah = myStack.isEmpty();
        fullkah = myStack.isFull();

System.out.println("==================================
==");
        System.out.println("Kosong        = " + kosongkah);
        System.out.println("Full          = " + fullkah);
        System.out.println("Elements      = " + myStack);
        System.out.println("Top Elements  = " + myStack.peek());
        System.out.println("Size          = " + myStack.size());
        System.out.println("_____________________________________");
```

```
        myStack.push(11);
        myStack.push(12);
        myStack.push(13);
        myStack.push(14);
        myStack.push(15);
        kosongkah = myStack.isEmpty();
        fullkah = myStack.isFull();

System.out.println("==========================================");
        System.out.println("Kosong        = " + kosongkah);
        System.out.println("Full          = " + fullkah);
        System.out.println("Elements      = " + myStack);
        System.out.println("Top Elements  = " + myStack.peek());
        System.out.println("Size          = " + myStack.size());
        System.out.println("_____________________________________");

        System.out.println("OPERASI POP=========================");
        while (myStack.isEmpty() == false) {
            int value = myStack.pop();
            kosongkah = myStack.isEmpty();
            fullkah = myStack.isFull();
```

```
System.out.println("==========================================");
        System.out.println("Kosong        = " + kosongkah);
        System.out.println("Full          = " + fullkah);
        System.out.println("Elements      = " + myStack);
        System.out.println("POP           = " + value);
        System.out.println("Size          = " + myStack.size());
        System.out.println("_____________________________________");
    }
    System.out.println("AKHIR DARI OPERASI POP================");
  }
}
```

Sedakan output lengkap programnya di jendela *console* adalah sebagai berikut:

```
run:
==================================
Kosong        = true
Full          = false
Elements      = []
Stack is Empty
Top Elements    = -999
Size          = 0
```

```
_____

==================================

Kosong        = false

Full          = false

Elements      = [2, 3, 4, 5, 6, 7, 8, 9]

Top Elements    = 9

Size          = 8

_____

==================================

Kosong        = false

Full          = false

Elements      = [2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15]

Top Elements    = 15

Size          = 13

_____

OPERASI
POP=======================================

==================================

Kosong        = false

Full          = false

Elements      = [2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14]

POP           = 15

Size          = 12
```

```
_____

==================================

Kosong        = false

Full          = false

Elements      = [2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13]

POP           = 14

Size          = 11

_____

==================================

Kosong        = false

Full          = false

Elements      = [2, 3, 4, 5, 6, 7, 8, 9, 11, 12]

POP           = 13

Size          = 10

_____

==================================

Kosong        = false

Full          = false

Elements      = [2, 3, 4, 5, 6, 7, 8, 9, 11]

POP           = 12

Size          = 9

_____
```

```
==================================
Kosong        = false
Full          = false
Elements      = [2, 3, 4, 5, 6, 7, 8, 9]
POP           = 11
Size          = 8

_____________________________
==================================
Kosong        = false
Full          = false
Elements      = [2, 3, 4, 5, 6, 7, 8]
POP           = 9
Size          = 7

_____________________________
==================================
Kosong        = false
Full          = false
Elements      = [2, 3, 4, 5, 6, 7]
POP           = 8
Size          = 6

_____________________________
==================================
```

```
Kosong      = false
Full        = false
Elements    = [2, 3, 4, 5, 6]
POP         = 7
Size        = 5
_____________________________
=================================
Kosong      = false
Full        = false
Elements    = [2, 3, 4, 5]
POP         = 6
Size        = 4
_____________________________
=================================
Kosong      = false
Full        = false
Elements    = [2, 3, 4]
POP         = 5
Size        = 3
_____________________________
=================================
Kosong      = false
```

```
Full          = false
Elements      = [2, 3]
POP           = 4
Size          = 2

_____________________________
=================================
Kosong        = false
Full          = false
Elements      = [2]
POP           = 3
Size          = 1

_____________________________
=================================
Kosong        = true
Full          = false
Elements      = []
POP           = 2
Size          = 0

_____________________________
AKHIR DARI OPERASI POP============================
BUILD SUCCESSFUL (total time: 0 seconds)
```

DAFTAR PUSTAKA

Campesato, O. (2023). *DATA STRUCTURES IN JAVA*. MERCURY LEARNING AND INFORMATION.

Dale, N., Joyce, D. T., & Weems, C. (2018). *Object-oriented data structures using Java* (4th ed.). Jones & Bartlett Learning. https://doi.org/10.5860/choice.49-0319

Daniel Liang, Y. (2019). *Introduction to Java™ Programming and Data Structures, Comprehensive Version, Eleventh Edition, Global Edition*. Pearson. www.pearsonglobaleditions.com/liang

Filho, W. V. F. (2017). *Computer Science Distilled*. CODE ENERGY LLC.

Gersting, J. L. (2014). *Mathematical Structures for Computer Science*. W. H. Freeman and Company.

Karumanchi, N. (2010). *Data Structures And Algorithms Made Easy*. CareerMonk.

KOFFMAN, E. B., & WOLFGANG, P. A. T. (2016). *Data Structures Abstraction and Design Using Java Third Edition*. Wiley.

Lowe, D. (2023). *Java® All-in-One For Dummies®, 7th Edition Published* (7th ed.). John Wiley & Sons.

Schildt, H., & Coward, D. (2024). *Java The Complete Reference Thirteenth Edition*. McGraw Hill.

Weiss, M. A. (2012). *Data Structures and Algorithm Analysis in Java* (3rd ed.). Pearson.

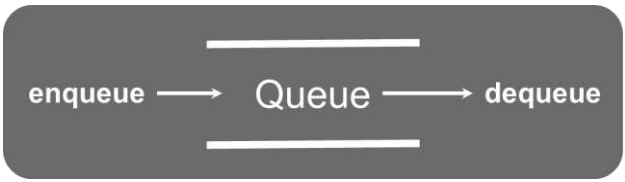
BAB 7
QUEUE DENGAN ARRAY

Oleh Putri Ayu Maharani

7.1 Pendahuluan

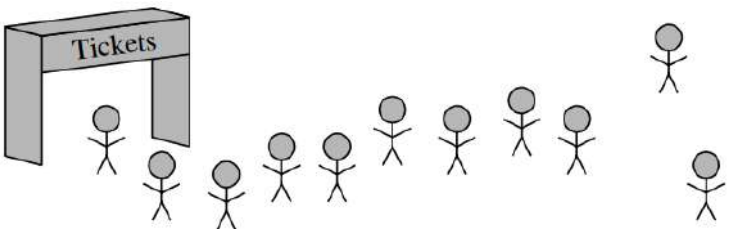
Struktur data fundamental berikutnya adalah antrean (queue). Struktur ini mirip dengan tumpukan (stack), karena keduanya adalah kumpulan objek, namun berbeda dalam cara memasukkan dan mengeluarkan item. Antrean dapat diibaratkan seperti antrean orang yang menunggu untuk menaiki wahana di taman hiburan. Orang yang menunggu masuk ke antrean dari belakang, sedangkan orang yang berada di depan antrean adalah yang pertama menaiki wahana. (Goodrich, et al., 2013)

Antrean adalah kumpulan item yang diurutkan berdasarkan prinsip "*first-in first-out*" (FIFO), di mana penambahan item baru terjadi di ujung yang disebut "*rear*" dan penghapusan item terjadi di ujung lainnya yang disebut "*front*". Artinya, item pertama yang masuk akan menjadi item pertama yang keluar, mirip dengan konsep "*first-come, first-served*". Berbeda dengan tumpukan yang menggunakan prinsip *last-in first-out* (LIFO). Prinsip FIFO dalam antrean yaitu *enqueue* dan *dequeue*. *Enqueue* adalah penambahan item baru di belakang antrean, sedangkan *dequeue* adalah penghapusan item dari depan antrean (lihat Gambar 7.1). (Miller & David, 2013)

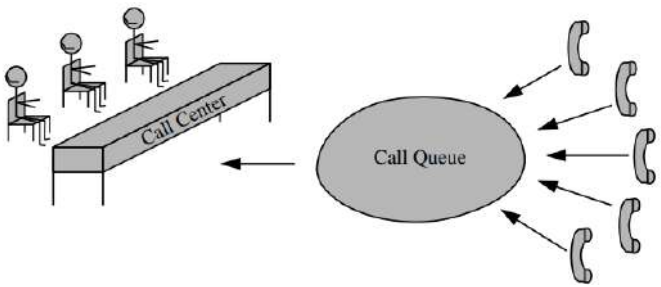


Gambar 7. 1. Ilustrasi prinsip *enqueue* dan *dequeue*

Terdapat banyak aplikasi antrean, contohnya pada kehidupan sehari-hari adalah antrean di bioskop, antrean di kasir, antrean di kafetaria/restoran, penanganan panggilan ke pusat layanan pelanggan, dan lain sebagainya seperti diilustrasikan pada Gambar 7.2. (Goodrich, et al., 2013)



(a)



(b)

Gambar 7. 2. Ilustrasi FIFO antrean dalam kehidupan sehari-hari. (a) orang-orang mengantri dengan berbaris untuk membeli tiket; (b) antrean panggilan telepon ke pusat layanan pelanggan.
Sumber : (Goodrich, et al., 2013)

Dalam dunia komputer juga terdapat banyak contoh umum untuk antrean. Misalnya di dalam sebuah laboratorium komputer terdapat 30 komputer yang terhubung dengan 1 printer. Ketika para siswa ingin mencetak tugas, tugas mereka akan mengantre dengan tugas cetak lainnya yang sedang menunggu. Tugas pertama yang masuk akan menjadi tugas pertama yang diproses. Tugas yang terakhir masuk dalam antrean harus menunggu hingga semua tugas lain dicetak terlebih dahulu. (Hetland, 2014)

Selain antrean pada printer, contoh lain yang menggunakan prinsip antrean adalah sistem operasi yang menggunakan berbagai jenis antrean untuk mengelola proses-proses di dalam komputer. Seperti proses penjadwalan didasarkan pada alogritma antrean yang berusaha menjalankan program secepat mungkin untuk melayani sebanyak mungkin pengguna. Contoh lainnya yaitu ketika kita mengetik pada *keyboard*, terkadang penekanan tombol terjadi lebih cepat daripada karakter yang muncul pada layar komputer. Hal ini disebabkan karena komputer sedang menjalankan tugas lain pada saat itu. Saat terjadi penekanan tombol, proses tersebut masuk ke dalam *buffer* seperti antrean hingga akhirnya ditampilkan pada layar sesuai urutan. (Baka, 2017)

7.2 Tipe Data Abstrak Antrean

Antrean didefinisikan sebagai kumpulan objek yang diurutkan, di mana akses dan penghapusan item hanya diperbolehkan pada item pertama dalam antrean, sedangkan penambahan item hanya diperbolehkan di bagian belakang. Pembatasan ini dilakukan untuk menerapkan aturan bahwa proses penambahan dan penghapusan objek sesuai dengan prinsip FIFO. Oleh karena itu, untuk mendukung hal ini program akan mendefinisikan ujung belakang dengan "*rear*", sebagai

posisi dimana objek baru ditambahkan, dan ujung lainnya dengan "*front*", sebagai posisi objek yang akan dihapus. (Cormen, et al., 2022)

Tipe data abstrak atau abstract data type (ADT) dari antrean ditentukan oleh operasi-operasi berikut:

a. Queue(): fungsi untuk membuat antrean baru yang kosong. Fungsi ini tidak memerlukan parameter dan akan mengembalikan antrean kosong apabila dipanggil.
b. enqueue(item): fungsi untuk menambahkan item baru ke belakang antrean. Fungsi ini memerlukan parameter berupa nilai dari item yang akan ditambahkan dan tidak mengembalikan apa pun ketika dipanggil.
c. dequeue(): fungsi untuk menghapus item di posisi depan antrean. Fungsi ini tidak memerlukan parameter dan mengembalikan item yang telah dihapus, serta memodifikasi isi antrean.
d. is_empty(): fungsi untuk memeriksa apakah antrean kosong. Fungsi ini tidak memerlukan parameter dan mengembalikan nilai boolean ketika dipanggil.
e. len(queue): fungsi untuk mengembalikan jumlah item dalam antrean. Dalam Python, fungsi ini diimplementasikan menggunakan metode __len__.

Item-item yang ditambahkan ke dalam antrean dapat memilki tipe yang bervariasi. Sebagai contoh, kita membuat sebuah antrean dengan nama *queue* dan isinya kosong. Tabel 2.1 menunjukkan hasil dari serangkaian operasi antrean. Posisi antrean yang paling depan berada di sebelah kiri. Angka 10 adalah item yang pertama dimasukkan ke dalam antrian sehingga menjadi item pertama yang akan dikembalikan oleh operasi *dequeue*. (Miller & David, 2013)

Tabel 6. 2. Contoh operasi antrean

Operasi	Isi Antrean *front* ← *queue* ← *rear*	Return Value
queue.is_empty()	[]	True
queue.enqueue(10)	[10]	-
queue.enqueue('ilkom')	[10, 'ilkom']	-
queue.enqueue(True)	[10, 'ilkom', True]	-
len(queue)	[10, 'ilkom', True]	3
queue.is_empty()	[10, 'ilkom', True]	False
queue.dequeue()	['ilkom', True]	10
queue.dequeue()	[True]	'ilkom'
queue.enqueue(3.14)	[True, 3.14]	-
len(queue)	[True, 3.14]	2

Contoh soal:

Misalkan terdapat serangkaian operasi rangkaian sebagai berikut:

```
q = Queue()
q.enqueue('Bab')
q.enqueue(8)
q.enqueue('Queue')
q.dequeue()
```

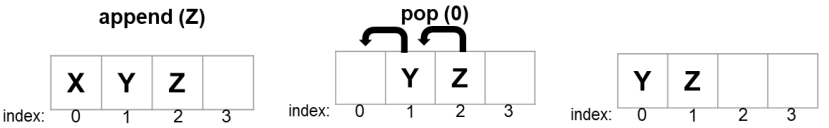
Bagaimana isi dari antrean tersebut?

a. ['Bab', 8]
b. [8, 'Queue']
c. ['Bab', 'Queue']

d. [‘Bab’, 8, ‘Queue’]

7.3 Implementasi Antrean dengan Array

Untuk ADT antrean kita mungkin saja menggunakan ADT tumpukan, seperti menambahkan item ke dalam antrean dengan memanggil append(item) untuk menambahkannya ke akhir array dan menghapus item dari antrean dengan menggunakan sintaks pop(0) untuk menghapus item pertama dalam array, namun terdapat beberapa pertimbangan terkait efisiensi. (Heineman, et al., 2009)

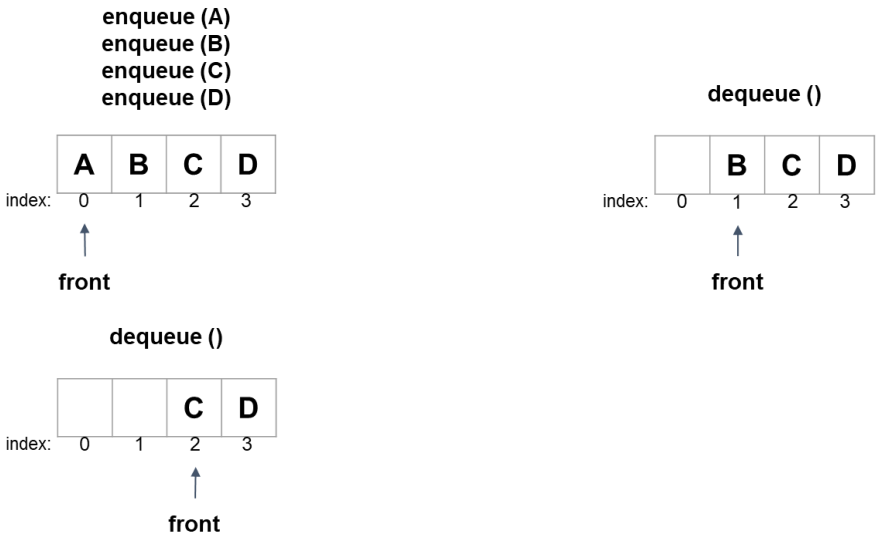


Gambar 7. 3. Ilustrasi efisiensi dari fungsi pop(0) untuk ADT antrean

Cara tersebut mudah untuk diimplementasikan, tapi ketika pop dipanggil pada array dengan indeks selain indeks default (0), sebuah *loop* akan dieksekusi untuk menggeser semua item setelah indeks yang dihapus ke arah kiri untuk mengisi celah yang disebabkan oleh operasi pop seperti pada Gambar 7.3. Sehingga pemanggilan fungsi pop akan mengakibatkan performa terburuk dengan kompleksitas waktu $\Theta(n)$.

Strategi ini dapat ditingkatkan dengan menghindari untuk memanggil pop(0) sepenuhnya. Sebagai gantinya dapat mendefinisikan item yang akan dihapus (*dequeue*) dari array dengan referensi None, dan menggunakan variabel front untuk menyimpan indeks posisi item yang saat ini berada di antrean terdepan. Dengan algoritma *dequeue* seperti ini maka operasi akan berjalan dalam kompleksitas waktu $\Theta(1)$. Setelah beberapa

operasi *dequeue*, pendekatan ini menghasilkan antrean seperti yang digambarkan pada Gambar 7.4.



Gambar 7. 4. Pendekatan algoritma *dequeue* dengan kompleksitas waktu Θ(1)

Namun demikian, pendekatan ini masih memiliki kekurangan. Pada tumpukan, panjang array sama dengan ukuran dari tumpukan meskipun array dasarnya mungkin sedikit lebih besar, berbeda dengan antrean. Sebuah antrean dapat memiliki jumlah item yang sedikit namun disimpan dalam array yang sangat besar. Hal tersebut terjadi ketika penambahan item baru ke dalam antrean secara terus-menerus dan kemudian menghapus item lain dari antrean yang mengakibatkan pergeseran referensi front ke arah kanan. Seiring waktu, ukuran array dasar akan bertambah menjadi Θ(m), Dimana *m* adalah jumlah total operasi antrean sejak antrean dibuat, bukan jumlah item yang saat ini ada dalam antrean. Pendekatan ini akan berdampak buruk pada aplikasi yang memiliki antrean

berukuran kecil tetapi digunakan untuk jangka waktu yang lama. Contohnya pada daftar tunggu di restoran yang maksimal entrinya adalah 30 pada satu waktu, namun dalam jangka waktu satu hari (atau seminggu), jumlah total entri bisa jauh lebih besar.

7.3.1 Menggunakan Array Sirkuler

Untuk mengembangkan pendekatan algoritma antrean yang lebih baik, kita dapat menggunakan array secara sirkuler. Dimana referensi bagian depan antrean (front) tetap bergeser ke kanan, kemudian item yang baru ditambahkan ke dalam antrean akan mengisi posisi kosong di depan referensi front. Asumsikan bahwa array yang mendasari suatu antrean memiliki panjang yang tetap *N*, yang lebih besar dari jumlah item sebenarnya dalam antrean. Item-item baru dimasukkan ke dalam antrean akan ditempatkan setelah bagian front hingga indeks *N-1*, kemudian melanjutkan pada indeks 0, 1, dan seterusnya hingga indeks front. Gambar 7.5 mengilustrasi pendekatan antrean ini dengan item pertama adalah *E* dan item terakhir adalah *M*.

Gambar 7. 5. Pemodelan sebuah antrean menggunakan array sirkuler

Menerapkan array sirkuler ini tidaklah sulit. Ketika menghapus item dari antrean (*dequeue*) dan memindahkan posisi front, kita dapat menggunakan aritmatika $f = (f + 1)\ \%\ N$. Dimana operator % dalam Python menunjukkan operator modulo, yang mengambil sisa dari perhitungan pembagian. Misalnya, 11 dibagi 2 memiliki hasil bagi 5 dengan

sisa 1, yaitu $\frac{11}{2} = 5\frac{1}{2}$. Jadi dalam Python, 11//2 menghitung hasil bagi yaitu 5, sementara 11%2 menghitung sisa bagi yaitu 1. Operator modulo sangat ideal untuk menerapkan array sirkuler. Sebagai contoh, apabila kita memiliki array dengan ukuran 15 dan indeks front adalah 12, maka kita dapat posisi penempatan item baru dengan menghitung (12+1) % 15, yang hasilnya adalah 12 karena 12 dibagi 15 adalah 0 maka sisa baginya adalah 12. Demikian pula untuk menambahkan item setelah indeks 13 akan menghasilkan indeks 14. Namun, ketika kita akan menambahkan item baru setelah indeks 14 (indeks terakhir dalam array (*N-1*)), maka hasil dari (14 + 1) % 15 adalah 0 (karena 15 dibagi 15 memiliki sisa bagi 0), sehingga item baru tersebut akan ditempatkan pada indeks 0.

7.3.2 Implementasi Antrean dalam Python

Implementasi ADT antrean menggunakan array sirkuler dalam bahasa pemrograman Python disajikan dalam Fragmen Kode 6.1, dimana kelas antrean menggunakan ketiga variable berikut:

_data: referensi ke sebuah array dengan kapasitas tetap.

_size: bilangan bulat yang mewakili jumlah item yang terdapat dalam antrean saat ini (berbeda dengan ukuran array).

_front: bilangan bulat yang mewakili indeks posisi item pertama dalam antrean (dengan asumsi bahwa antrean tidak kosong).

Pertama-tama, kita mereservasi sebuah array berukuran sedang untuk menyimpan data, walaupun antrean pada awalnya masih berukuran nol. Selanjutnya kita menginisialisasi referensi front ke indeks nol. Ketika fungsi front atau dequeue dipanggil saat belum ada item dalam antrean, program akan

memunculkan pesan bahwa antrean kosong seperti pada Fragmen Kode berikut: (Ramalho, 2015)

```
class ArrayQueue:
  DEFAULT_CAPACITY = 10 #kapasitas array untuk antrean baru

  def __init__(self):
    #membuat antrean kosong
    self._data = [None] ArrayQueue.DEFAULT_CAPACITY
    self._size = 0
    self._front = 0

  def __len__(self):
    #mengembalikan jumlah item di dalam antrean
    return self._size

  def is_empty(self):
    #mengembalikan nilai True apabila antrean kosong
    return self._size == 0

  def first(self):
    #mengembalikan item yang berada di posisi paling depan antrean
    tanpa menghapusnya
    #memunculkan pesan pengecualian jika antrean kosong
    if self.is_empty():
      raise Empty('Antrean kosong')
    return self._data[self._front]

  def dequeue(self):
    #menghapus item pertama dalam antrean dan mengembalikan
    item yang dihapus
    #memunculkan pesan pengecualian jika antrean kosong
    if self.is_empty():
      raise Empty('Antrean kosong')
    answer = self._data[self._front]
    self._data[self._front] = None
    self._front = (self._front + 1)%len(self._data)
    self._size = -1
```

```
34
35          return answer
36
37      def enqueue(self, e):
38          #menambahkan item di posisi paling akhir
39          #jika jumlah item antrean sama dengan ukuran array, maka ukuran
            array dilipatgandakan
40          if self._size == len(self._data):
41              self._resize(2 * len(self._data))
42          available = (self._front + self._size) % len(self._data)
43          self._data[available] = e
44          self._size += 1
45
46      def _resize(self, cap):
47          #menambah ukuran array dengan kapasitas baru
48          old = self._data              #menyimpan data awal
49          self._data = [None]*cap #mengalokasikan array dengan kapasitas
            baru
50          walk = self._front
51      for k in range(self. size):
52          self._data[k] = old[walk]
53          walk = (1 + walk) % len(old)
54      self._front = 0 #front ditempatkan di indeks 0
```

Fragmen Kode 7.1 Implementasi antrean menggunakan array

7.3.2.1 Menambahkan dan Menghapus Item dalam Antrean

Untuk menambahkan item baru ke belakang antrean (*enqueue*) kira perlu menentukan indeks yang tepat untuk menempatkan item baru tersebut. Meskipun posisi paling belakang antrean tidak disimpan secara eksplisit, tapi kita dapat menghitung posisi indeks untuk menempatkan item baru sebagaimana rumus pada baris ke-42 dalam Fragmen Kode 7.1:

```
available = (self._front + self._size) % len(self._data)
```

Perhatikan bahwa kita menggunakan ukuran antrean sebelum penambahan item baru. Contohnya, apabila kita memiliki antrean dengan kapasitas 15, jumlah item dalam antrean saat ini ada 3, dan item pertama berada pada indeks 7, maka ketiga item tersebut berada pada indeks 7, 8, dan 9. Saat item baru akan ditambahkan maka akan ditempatkan pada indeks (front + size) % len(data) = (7 + 3) % 15 = 10. Dalam kasus array sirkuler, jika antrean hipotetis kita memiliki 3 item dengan item pertama pada indeks 13 maka perhitungan (13 + 3) % 15 akan menghasilkan 1, yang mana tepat karena ketiga item tersebut berada pada indeks 13, 14, 0, dan item yang baru akan ditempatkan pada indeks 1.

Untuk fungsi *dequeue*, `self._front` saat ini menunjuk ke indeks dari item yang akan dihapus. Di sini kita juga menyimpan item yang akan dihapus ini untuk dikembalikan nilainya saat fungsi *dequeue* dipanggil sebelum item tersebut dihapus dari antrean dengan menetapkan `answer = self._data[self._front]`. Setelah itu data `self._front` dihapus sebagaimana pada baris 31:

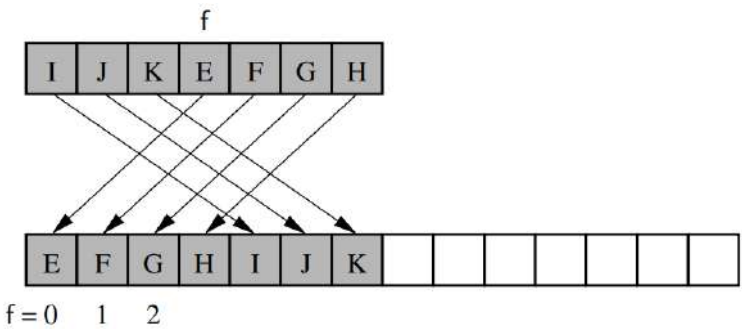
```
self._data[self._front] = None
```

Hal ini memungkinkan Python mendapatkan kembali ruang yang tidak digunakan. Secara internal, Python melacak jumlah referensi ke setiap objek. Apabila jumlah tersebut mencapai nol maka objek tersebut tidak dapat diakses lagi, sehingga memori dapat dikembalikan dan digunakan kembali. Setelah itu, fungsi *dequeue* memperbarui nilai `self._front` menggunakan kode pada baris 32:

```
self._front = (self._front + 1)%len(self._data)
```

7.3.2.2 Memperbesar Ukuran Antrean

Ketika fungsi *enqueue* dipanggil, dan pada saat yang sama ukuran antrean sama dengan ukuran array dasar, kita dapat menggunakan teknik standar untuk menggandakan kapasitas penyimpanan array tersebut. Namun mengubah ukuran antrean perlu memperhatikan banyak hal. Setelah membuat referensi sementara ke array lama, `old = self. data`, kita mengalokasikan array baru dengan ukuran dua kali lipat dan menyalin referensi dari array lama ke array yang baru. Saat mentransfer isi antrean, kode program juga menyelaraskan ulang bagian depan antrean ke indeks 0 pada array baru, sebagaimana ditunjukkan pada Gambar 7.6.



Gambar 7. 6. Mengubah ukuran antrean sambil menyelaraskan kembali item *front* pada indeks 0.

7.3.2.3 Menganalisis Implementasi Antrean dengan Array

Tabel 7.2 menunjukkan kompleksitas waktu dari ADT antrean berbasis array. Dengan mengecualikan utilitas `_resize`, semua fungsi bergantung pada sejumlah operasi constant, seperti aritmatika, perbandingan, dan penugasan. Oleh karena itu, setiap fungsi memiliki waktu eksekusi terburuk $\Theta(1)$, kecuali untuk *enqueue* dan *dequeue*, yang memiliki kompleksitas waktu $\Theta(1)$ yang diamortisasi.

Tabel 7. 1. Performa antrean dengan array. Batas untuk *enqueue* dan *dequeue* diamortisasi karena perubahan ukuran array. Kompleksitas ruang adalah Θ(n), dimana *n* adalah jumlah item dalam antrian.

Operasi	Running Time
queue.enqueue(e)	$\Theta(1)$*
queue.dequeue()	$\Theta(1)$*
queue.first()	$\Theta(1)$
queue.is_empty()	$\Theta(1)$
len(queue)	$\Theta(1)$

*diamortisasi

Latihan soal:

1. Bagaimana isi dari antrean berikut jika dimulai dengan antrean yang kosong?

   enqueue(15), enqueue(2), dequeue(), enqueue(3), enqueue(5), dequeue(), dequeue(), enqueue(9), enqueue(1), dequeue(), enqueue(7), enqueue(10), dequeue(), dequeue(), enqueue(6), dequeue(), dequeue().

2. Apabila antrean pada latihan soal nomor 1 diterapkan pada kode program **ArrayQueue** dengan ukuran kapasitas inisial array adalah 30 dan ukurannya tidak diperbesar. Berapakah nilai akhir dari variabel `front`, dan terletak pada indeks ke berapa variabel `front` tersebut?

DAFTAR PUSTAKA

Baka, B., 2017. Python Data Structures and Algorithms. s.l.:Packt Publishing Ltd.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C., 2022. Introduction to algorithms. s.l.:MIT Press.

Goodrich, M. T., Tamassia, R. & Goldwasser, M. H., 2013. Data structures and algorithms in Python.. United States of America: John Wiley & Sons Ltd..

Heineman, G. T., Pollice, G. & Selkow, S., 2009. Algorithms in a Nutshell. s.l.:O'Reilly Media, Inc..

Hetland, M. L., 2014. Python Algorithms: mastering basic algorithms in the Python Language. s.l.:Apress.

Miller, B. N. & David, . L. R., 2013. Problem Solving with Algorithms and Data Structures. 3.0 ed. s.l.:Franklin Beedle & Associates.

Ramalho, L., 2015. Fluent Python: Clear, concise, and effective programming. s.l.:O'Reilly Media, Inc..

BAB 8
ALGORITMA SEARCHING

Oleh Ellbert Hutabri

8.1 Pendahuluan

Algoritma pencarian (*searching*) merupakan salah satu konsep dasar dalam ilmu komputer yang memiliki peran penting dalam berbagai aplikasi, mulai dari pengolahan data, manajemen basis data, hingga pengembangan perangkat lunak. Pencarian adalah proses untuk menemukan elemen tertentu dalam kumpulan data, baik itu dalam bentuk array, daftar terurut, atau struktur data kompleks seperti pohon atau graf.

Algoritma pencarian dapat menerima sebuah kata kunci untuk menemukan data informasi berdasarkan kata kunci tersebut. Sehingga efisiensi dari algoritma pencarian sering kali menjadi penentu utama dalam kinerja keseluruhan sebuah sistem, terutama ketika bekerja dengan dataset yang besar.

Berikut adalah beberapa pendapat ahli mengenai algoritma pencarian (searching):

1. Donald Knuth - Pionir Ilmu Komputer

   Menurut Knuth, algoritma pencarian adalah salah satu teknik paling mendasar dalam pemrograman komputer, khususnya dalam konteks sorting dan searching. Pencarian linier meskipun sederhana, cenderung tidak efisien untuk dataset besar, tetapi tetap memiliki nilai praktis dalam situasi di mana penyortiran tidak memungkinkan (Knuth, 1998).

2. Jon Bentley - Ahli Efisiensi Algoritma

   Bentley menyebutkan bahwa pencarian adalah salah satu bidang di mana efisiensi sangat penting, khususnya dalam sistem real-time. Algoritma pencarian biner sering kali menjadi pilihan terbaik untuk pengindeksan data terurut dalam database besar, karena memberikan kompromi yang baik antara kecepatan dan kompleksitas implementasi (Bentley, 2000).

3. Robert Sedgewick - Pengembang Teori Algoritma

   Sedgewick berpendapat bahwa meskipun pencarian biner optimal untuk data yang terurut, dalam praktik dunia nyata, pencarian eksponensial dapat memberikan hasil yang lebih baik pada kasus di mana pencarian dilakukan pada dataset yang sangat besar, seperti struktur pohon atau graf (Sedgewick, 2002).

4. Mark Allen Weiss - Ahli Algoritma dan Struktur Data

   Weiss menyatakan bahwa pencarian interpolasi merupakan alternatif yang menarik dari pencarian biner untuk dataset yang terdistribusi secara seragam. Algoritma ini bekerja lebih efisien dalam kasus distribusi seragam, tetapi menjadi kurang efektif jika distribusi data tidak rata (Weiss, 2014).

5. Michael T. Goodrich, Roberto Tamassia dan Michael H. Goldwasser - Ahli Struktur Data

   Goodrich dan Tamassia menekankan pentingnya memahami struktur data yang digunakan dalam konteks algoritma pencarian. Menurut mereka, pemilihan struktur data yang tepat, seperti hash tables atau binary search trees, sangat menentukan efisiensi algoritma pencarian yang diterapkan (Goodrich, Tamassia and Goldwasser, 2014).

6. Thomas H. Cormen - Pengarang Introduction to Algorithms

   Cormen dan tim berpendapat bahwa algoritma pencarian biner merupakan salah satu algoritma paling efisien untuk pencarian dalam data terurut, dengan kompleksitas waktu yang logaritmik. Namun, penting untuk mencatat bahwa algoritma ini hanya berlaku pada data yang sudah terurut dan memerlukan langkah penyortiran sebelum pencarian (Cormen *et al.*, 2022).

Berdasarkan pendapat para ahli di atas mengenai algoritma pencarian adalah efisiensi algoritma sangat bergantung pada karakteristik dataset dan konteks penggunaan. Pencarian linier adalah algoritma pencarian sederhana, tetapi tidak efisien untuk dataset besar. Algoritma pencarian biner sering dianggap sebagai pilihan yang optimal untuk data terurut, terutama dalam sistem real-time dan database besar, karena memberikan keseimbangan antara kecepatan dan kompleksitas. Namun, dalam dataset yang sangat besar atau kompleks, pendekatan eksponensial atau struktur pohon menjadi lebih efektif.

Secara umum algotirma pencarian dibagi menjadi dua, yaitu:

1. Algoritma Pencarian Linier

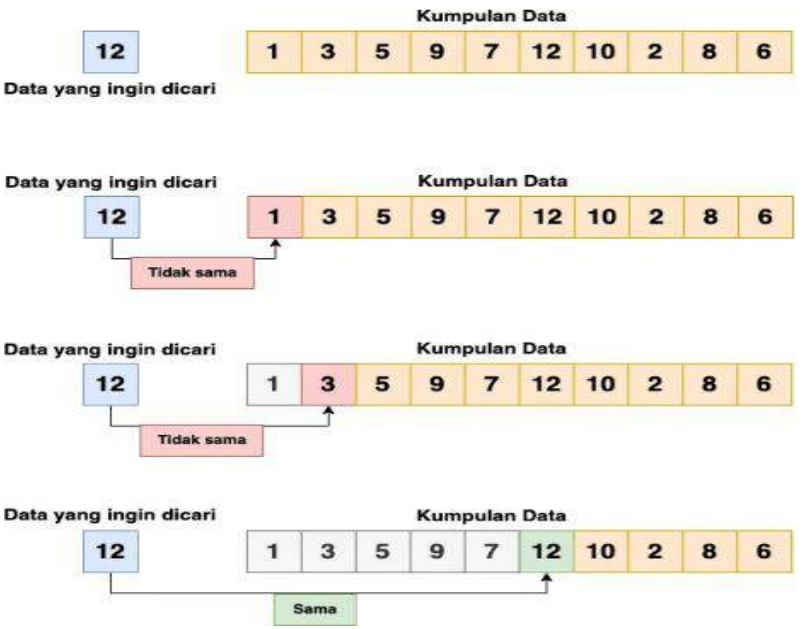
   Metode ini memeriksa setiap elemen secara berurutan sampai item yang dicari ditemukan atau dipastikan tidak ada dalam kumpulan data.
2. Algoritma Pencarian Biner

   Algoritma ini beroperasi pada data yang sudah terurut, dengan membagi kumpulan data menjadi dua dan memeriksa elemen di tengah. Berdasarkan hasil pemeriksaan, algoritma menentukan apakah pencarian akan dilanjutkan pada bagian kiri atau kanan. Proses ini berulang hingga item ditemukan atau seluruh daftar telah diperiksa.

8.2 Pencarian Linier

Lagoritma pencarian linier (linier search) dikenal juga dengan sequential search, adalah sebuah teknik pencarian data dengan mengunjungi setiap data secara satu persatu, apabila ditemukan kesamaan antara keyword dengan data yang ada maka aka ditampilkan pada output (Wahyuni, Andryana and Rahman, 2022).

Algoritma pencarian linier merupakan teknik pencarian sederhana di mana setiap elemen dalam daftar atau array diperiksa satu per satu hingga elemen yang diinginkan ditemukan. Algoritma ini efektif untuk data yang tidak terurut atau dalam kasus di mana tidak ada struktur khusus pada data.



Gambar 8. 1. Ilustrasi Cara Kerja Algoritma Pencarian

Cara kerja linier search adalah sebagai berikut:

1. Algoritma ini dapat diterapkan untuk mencari data baik dalam urutan menaik, menurun, maupun data yang tidak terurut.
2. Algoritma akan menerima input data dari pengguna, input yang diberikan disebut juga dengan nilai target.
3. Algoritma akan memulai pencarian dari elemen pertama yang ada dalam urutan data.
4. Algoritma melakukan pemeriksaan apakah elemen memiliki nilai yang sama dengan nilai target, jika elemen memiliki nilai yang sama maka pencarian akan berhenti, jika tidak maka akan di ulang sampai selesai diperiksa.

contoh:

```python
# Linear Search in Python

def linear_search(arr, target):

    for i, value in enumerate(arr):
        if value == target:
            return i
    return -1

array = [2, 4, 0, 1, 9]
x = 1

result = linear_search(array, x)

if result == -1:
    print("Element not found")
else:
    print(f"Element found at index: {result}")
```

Dari kode di atas dapat dijelaskan:

```
def linearSearch(array, n, x):
```

Fungsi liniersearch digunakan untuk melakukan pencarian linear di dalam array atau list. Berfungsi untuik menerima tiga parameter yaitu:

- array: List atau array yang akan dicari elemennya.
- n: Jumlah elemen di dalam array (panjang array).
- x: Elemen yang akan dicari di dalam array.

```
for i in range(0, n):
    if (array[i] == x):
        return i
return -1
```

Kode di atas merupakan proses pencarian linear:

- Pada bagian ini, terdapat perulangan for yang berjalan dari indeks 0 hingga n-1 (jumlah elemen di dalam array).
- Untuk setiap iterasi, elemen pada indeks i dari array (array[i]) dibandingkan dengan elemen yang dicari (x).
- Jika elemen yang dicari (x) ditemukan di dalam array (kondisi array[i] == x terpenuhi), fungsi akan mengembalikan indeks elemen tersebut (i).
- Jika tidak ada elemen yang cocok hingga akhir perulangan, fungsi akan mengembalikan -1, yang menunjukkan bahwa elemen tersebut tidak ditemukan.

```
array = [2, 4, 0, 1, 9]
x = 1
n = len(array)
```

Kode di atas merupakan inisialisasi variabel yang telah ditentukan:

- array berisi angka-angka yang akan dicari elemennya (misalnya [2, 4, 0, 1, 9]).
- x: Elemen yang akan dicari (dalam hal ini, angka 1).
- n: Panjang array, atau jumlah elemen dalam array, yang dihitung menggunakan len(array). Nilainya adalah 5 karena ada 5 elemen di dalam list.

```
result = linearSearch(array, n, x)

if(result == -1):
    print("Element not found")
else:
    print("Element found at index: ", result)
```

Fungsi dari variabel result adalah untuk menyimpan hasil yang dikembalikan oleh fungsi linear_search, berikut adalah prosesnya:

- Setelah memanggil fungsi linear_search(array, x), hasil dari pencarian elemen x dalam list array disimpan dalam variabel result.
- Fungsi linear_search akan mengembalikan indeks di mana elemen x ditemukan di dalam array, atau -1 jika elemen tidak ditemukan.
- Setelah nilai disimpan di result, program akan memeriksa apakah result bernilai -1 atau tidak:

- Jika result == -1, berarti elemen tidak ditemukan, dan pesan "Element not found" akan dicetak.
- Jika result != -1, berarti elemen ditemukan, dan program akan mencetak indeks tempat elemen ditemukan.

Alur Eksekusi Program:

1. Program menginisialisasi array [2, 4, 0, 1, 9] dan elemen yang dicari (x = 1).
2. Fungsi linearSearch dipanggil untuk mencari elemen 1 dalam array.
3. Program akan memeriksa setiap elemen array secara berurutan:
    - Apakah elemen ke-0 (2) adalah 1? Tidak.
    - Apakah elemen ke-1 (4) adalah 1? Tidak.
    - Apakah elemen ke-2 (0) adalah 1? Tidak.
    - Apakah elemen ke-3 (1) adalah 1? Ya.
4. Fungsi mengembalikan nilai 3 (indeks elemen 1).
5. Karena hasilnya bukan -1, program akan mencetak: "Element found at index: 3".

8.3 Big O Notation

Big-O Notation merupakan metode untuk mengubah semua langkah-langkah dalam sebuah algoritma ke dalam bentuk aljabar, dengan mengabaikan konstanta kecil serta koefisien yang tidak memiliki pengaruh signifikan terhadap kompleksitas keseluruhan dari masalah yang diselesaikan oleh algoritma tersebut.

Big-O Notation adalah notasi yang digunakan untuk menyatakan batas atas asimptotik dari kompleksitas waktu atau ruang sebuah algoritma. Ini menjelaskan pertumbuhan waktu eksekusi sebuah algoritma dalam kaitannya dengan ukuran input, dengan mengabaikan konstanta kecil dan faktor non-dominan (Subandijo, 2011).

Prinsip-prinsip Big-O Notation dapat diterapkan dalam ilmu komputer untuk mengklasifikasikan algoritma pemrograman berdasarkan tingkat kerumitannya. Dalam penerapannya, Big-O Notation mengukur seberapa lama waktu eksekusi dan jumlah ruang atau sumber daya yang dibutuhkan, seiring dengan meningkatnya jumlah data input.

Berikut adalah beberapa contoh cara membaca notasi Big-O (Chivers and Sleightholme, 2015) beserta penjelasannya:

1. O(log n) menunjukkan bahwa kompleksitas akan bertambah seiring dengan logaritma dari jumlah data. Jika sebuah algoritma memiliki kompleksitas ini, maka algoritma tersebut sangat efisien.
2. O(1) berarti kompleksitas tetap konstan, tidak peduli berapa banyak data yang diproses. Ini adalah kategori kompleksitas terbaik untuk sebuah algoritma.
3. O(n) menunjukkan bahwa kompleksitas akan meningkat sebanding dengan jumlah data, sering disebut sebagai kompleksitas linear.
4. O(n+2) atau O(2n+5) setara dengan O(n), karena konstanta tidak diperhitungkan dalam analisis kompleksitas.

Berikut adalah beberapa contoh penerapan Big-O Notation pada struktur data umum:

1. Array
    a. Akses (Access): O(1) - Karena array bersifat indeks langsung, waktu untuk mengakses elemen tertentu adalah konstan.
    b. Pencarian (Search): O(n) - Untuk pencarian elemen tertentu, Anda perlu memeriksa setiap elemen, yang berarti kompleksitasnya sebanding dengan jumlah elemen.
    c. Penyisipan (Insertion): O(n) - Jika elemen perlu disisipkan pada posisi tertentu, maka elemen lain mungkin perlu dipindahkan, sehingga membutuhkan waktu linear.
    d. Penghapusan (Deletion): O(n) - Menghapus elemen juga bisa memerlukan pemindahan elemen lainnya, menyebabkan kompleksitas linear.
2. Linked List
    a. Akses: O(n) - Untuk mengakses elemen di posisi tertentu, Anda harus menelusuri daftar dari awal hingga posisi yang diinginkan.
    b. Pencarian: O(n) - Setiap elemen perlu diperiksa satu per satu untuk menemukan elemen yang dicari.
    c. Penyisipan: O(1) - Jika disisipkan di awal (head) atau akhir (tail) dari linked list, waktu penyisipannya konstan.
    d. Penghapusan: O(1) - Menghapus elemen dari awal atau akhir memerlukan waktu konstan, tetapi penghapusan di tengah membutuhkan traversal terlebih dahulu (O(n)).

3. Stack dan Queue
    a. Akses: O(n) - Seperti Linked List, akses elemen tertentu memerlukan traversal.
    b. Pencarian: O(n) - Harus memeriksa setiap elemen satu per satu.
    c. Penyisipan (Push/Enqueue): O(1) - Penambahan elemen di stack (push) atau queue (enqueue) adalah operasi konstan.
    d. Penghapusan (Pop/Dequeue): O(1) - Menghapus elemen dari stack (pop) atau queue (dequeue) juga merupakan operasi konstan.
4. Hash Table
    a. Akses: O(1) - Dengan hashing yang baik, waktu akses ke elemen bisa sangat cepat, karena menggunakan indeks yang langsung.
    b. Pencarian: O(1) - Sama dengan akses, pencarian elemen biasanya bersifat konstan, kecuali jika terjadi collision.
    c. Penyisipan: O(1) - Menambahkan elemen baru pada posisi yang di-hash umumnya memerlukan waktu konstan.
    d. Penghapusan: O(1) - Menghapus elemen berdasarkan kunci juga umumnya memerlukan waktu konstan.
5. Binary Search Tree (BST)
    a. Akses: O(log n) - Jika BST seimbang, waktu untuk mengakses elemen tertentu adalah logaritmik.
    b. Pencarian: O(log n) - Pencarian elemen dalam BST yang seimbang juga bersifat logaritmik.
    c. Penyisipan: O(log n) - Menyisipkan elemen ke dalam BST seimbang membutuhkan waktu logaritmik.
    d. Penghapusan: O(log n) - Menghapus elemen dalam BST seimbang memerlukan waktu logaritmik.

e. Pencarian biner (Binary Search) merupakan algoritma pancarian data dengan

8.4 Pencarian Biner (*Binary Search*)

Algoritma Pencarian Biner (Binary Search) adalah metode pencarian yang diterapkan pada array yang sudah diurutkan dengan cara membagi interval pencarian menjadi dua secara berulang-ulang. Prinsip dari pencarian adalah array telah diurutkan, sehingga dapat mengurangi kompleksitas waktu menjadi O(log N) (Rubinstein-Salzedo, 2018). Teknik pencarian biner selalu dimulai dengan elemen tengah pada array, jika elemen array belum terurut maka perlu diurutkan terlebih dahulu.

Alur kerja pencarian biner adalah sebagai berikut:

a. Membagi eleman array menjadi dua bagian, sehingga didapatkan indeks tengah array yang disebut dengan mid, sehingga membagi kumpulan araay menjadi sub array kanan dan sub array kiri
b. Bandingkan elemen tengah dengan kata kunci yang dicari
c. Jika kata kunci tidak bernilai sama dengan elemen tengah, maka akan menggunakan aturan sebagai berikut:
    - Jika kata kunci lebih kecil dari mid, maka pencarian akan dilakukan pada sisi kiri mid
    - Jika kata kunci lebih besar dari mid, maka pencarian dilakukan pada sisi kanan mid
d. Proses akan terus dilakukan sampai seluruh elemen habis.

Terdapat araay A seperti gambar dibawah, data yang akan adicari adalah 23, langkahnya adalah sebagai berikut:

Gambar 8. 2. Array A

Cari elemen tengah array:

Gambar 8. 3. Proses Pencarian Elemen Tengah (Mid)

Elmen tengah pada array diatas adalah 16, karna 23 lebih besar dari 16 maka akan dilakukan pencarian pada sub array kanan. Pada sub array kanan, tentukan kembali elemen tengahnya.

Gambar 8. 4. Proses Pencarian Elemen Tengah (MID) Sub Array Kanan

Elemen tengah pada sub array kanan adalah 56, karna 23 lebih kecil dari 56 maka pencarian biner dilakupan pada sisi kiri array tengah, sehingga didapat nilai 23 dengan Low 5, Mid 5 dan High 9.

Gambar 8. 5. Kata Kunci Ditemukan

Contoh program pencarian biner:

```
# Fungsi untuk melakukan pencarian biner
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2  # Menentukan indeks tengah

        # Jika elemen tengah adalah target
        if arr[mid] == target:
            return mid

        # Jika target lebih besar, abaikan bagian kiri
        elif arr[mid] < target:
            low = mid + 1
```

```
        # Jika target lebih kecil, abaikan bagian kanan
        else:
            high = mid - 1

    # Jika target tidak ditemukan
    return -1

# Array yang diurutkan
arr = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]
target = 23

# Memanggil fungsi pencarian biner
result = binary_search(arr, target)

# Menampilkan hasil
if result != -1:
    print(f"Elemen ditemukan pada indeks {result}")
else:
    print("Elemen tidak ditemukan")
```

Output program ini akan menunjukkan bahwa elemen 23 ditemukan di indeks 5:

```
Elemen ditemukan pada indeks 5
```

Keuntungan Pencarian Biner:

a. Pencarian biner lebih cepat dibandingkan pencarian linear, terutama untuk array yang berukuran besar.
b. Lebih efisien daripada algoritma pencarian lainnya dengan kompleksitas waktu yang serupa, seperti pencarian interpolasi atau eksponensial.

c. Pencarian biner sangat ideal untuk mencari data dalam jumlah besar yang disimpan di penyimpanan eksternal, seperti hard drive atau cloud.

Kekurangan Pencarian Biner:

a. Array harus diurutkan terlebih dahulu.
b. Pencarian biner membutuhkan data yang tersimpan dalam lokasi memori yang berdekatan.
c. Elemen-elemen dalam array harus dapat dibandingkan dan diurutkan.

DAFTAR PUSTAKA

Bentley, J. (2000) *Programming Pearls , Second Edition*. Second Edi. New York City: Addison-Wesley Professional.

Chivers, I. and Sleightholme, J. (2015) 'An Introduction to Algorithms and the Big O Notation BT - Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77', in Chivers, I. and Sleightholme, J. (eds). Cham: Springer International Publishing, pp. 359–364. doi: 10.1007/978-3-319-17701-4_23.

Cormen, T. H. *et al.* (2022) *Introduction to algorithms, 4 Edition*, *The MIT Press Cambridge, Massachusetts London, England*. London: The MIT Press.

Goodrich, M. T., Tamassia, R. and Goldwasser, M. H. (2014) *Data Structures and Algorithms in Java*. Edited by S. Edition. United States: JohnWiley & Sons, Inc.

Knuth, D. E. (1998) *Art of Computer Programming - Volume 3 (Sorting & Searching)*. Canada: Addison Wesley Longman All.

Rubinstein-Salzedo, S. (2018) 'Big O Notation and Algorithm Efficiency', in.

Sedgewick, R. (2002) *Algorithms in C++*. Addison-Wesley (Algorithms in C++).

Subandijo, S. (2011) 'Efisiensi Algoritma dan Notasi O-Besar', *ComTech: Computer, Mathematics and Engineering Applications*, 2(2), p. 849. doi: 10.21512/comtech.v2i2.2835.

Wahyuni, W. S., Andryana, S. and Rahman, B. (2022) 'Penggunaan Algoritma Sequential Searching Pada Aplikasi Perpustakaan Berbasis Web', *JIPI (Jurnal Ilmiah Penelitian dan Pembelajaran Informatika)*, 7(2), pp. 294–302. doi: 10.29100/jipi.v7i2.2646.

Weiss, M. A. (2014) *Data and structures and Algorithm Analysis in C++*. Fourth Edi, *Pearson Education. Inc*. Fourth Edi. New Jersey. doi: 10.1201/9781420015133.pt5.

BAB 9
ALGORITMA SORTING

Oleh Kusnaeni

9.1 Pendahuluan

9.1.1 Pengertian Sorting

Sorting, atau pengurutan, adalah suatu proses dalam ilmu komputer yang bertujuan untuk menyusun elemen-elemen dalam suatu koleksi, seperti array atau daftar, sesuai dengan urutan tertentu(Sari *et al.*, 2018). Urutan ini dapat berupa urutan menaik (ascending) atau menurun (descending) berdasarkan suatu kriteria tertentu, seperti nilai numerik atau abjad. Sorting adalah konsep dasar yang digunakan dalam berbagai algoritma dan aplikasi, dan memiliki peran penting dalam meningkatkan efisiensi dalam pencarian, pengolahan, dan analisis data (Putri *et al.*, 2022).

9.1.2 Pentingnya Sorting dalam Ilmu Komputer

Sorting, atau pengurutan, adalah salah satu operasi paling fundamental dalam ilmu komputer. Sorting memungkinkan data untuk diorganisasikan dalam urutan yang dapat diakses dan diproses dengan lebih efisien, yang sangat penting dalam berbagai aplikasi seperti pencarian data, algoritma pengurutan, dan pengelolaan basis data. Algoritma sorting yang efisien dapat secara signifikan mengurangi waktu eksekusi dalam proses pencarian dan pengambilan data(Zutshi and Goswami, 2021).

Sorting juga menjadi dasar bagi berbagai algoritma lain, termasuk algoritma pencarian (*searching*), penggabungan (*merging*), dan pemfilteran (*filtering*). Selain itu, banyak struktur data seperti heap, binary search trees, dan hash tables, memanfaatkan konsep sorting dalam implementasi dan operasinya. Sorting juga memainkan peran penting dalam analisis dan pengolahan data besar (big data), di mana kecepatan dan efisiensi pengolahan data menjadi faktor utama(Yanti and Emi Sita Eriana, 2024).

9.2 Klasifikasi Algoritma Sorting

Algoritma sorting dapat diklasifikasikan berdasarkan berbagai kriteria, seperti kompleksitas waktu, kompleksitas ruang, metode pengurutan, dan stabilitas(Musiafa, 2022). Berikut ini adalah beberapa klasifikasi umum dari algoritma sorting:

1. Berdasarkan Metode Pengurutan
   Comparison-Based Sorting (Pengurutan Berdasarkan Perbandingan):
   a. Bubble Sort: Algoritma sederhana yang membandingkan setiap pasangan elemen yang berdekatan dan menukarnya jika berada dalam urutan yang salah.
   b. Selection Sort: Algoritma yang berulang kali menemukan elemen terkecil (atau terbesar) dari bagian yang tidak diurutkan dan menempatkannya di posisi akhir.
   c. Insertion Sort: Algoritma yang membangun array terurut satu per satu, dengan memasukkan elemen

dari bagian yang tidak diurutkan ke posisi yang benar dalam array terurut.

d. Merge Sort: Algoritma divide and conquer yang membagi array menjadi subarray yang lebih kecil, mengurutkannya, dan kemudian menggabungkannya kembali.
e. Quick Sort: Algoritma divide and conquer yang memilih elemen pivot dan mempartisi array sehingga elemen yang lebih kecil dari pivot berada di kiri, dan yang lebih besar di kanan.
f. Heap Sort: Algoritma yang mengubah array menjadi struktur data heap, dan kemudian secara berulang mengekstraksi elemen maksimum dan menempatkannya di posisi akhir.

Non-Comparison-Based Sorting (Pengurutan Tidak Berdasarkan Perbandingan):

a. Counting Sort: Algoritma yang menghitung kemunculan setiap elemen dan kemudian menempatkan elemen tersebut ke dalam urutan yang benar berdasarkan jumlah kemunculan.
b. Radix Sort: Algoritma yang mengurutkan elemen-elemen dengan memproses digit demi digit, mulai dari digit yang paling signifikan atau paling tidak signifikan.
c. Bucket Sort: Algoritma yang membagi elemen-elemen ke dalam beberapa "bucket" atau keranjang, mengurutkan setiap bucket secara individu, dan kemudian menggabungkannya.

- Berdasarkan Kompleksitas Waktu :
  $O(n^2)$ Sorting Algorithms:
  Bubble Sort, Selection Sort, Insertion Sort
  O(n log n) Sorting Algorithms:
  Merge Sort, Quick Sort, Heap Sort, Radix Sort (untuk input yang terbatas)

- Berdasarkan Stabilitas :
  Stable Sorting Algorithms:
  (Algoritma yang mempertahankan urutan relatif dari elemen-elemen yang setara): Merge Sort, Insertion Sort, Bubble Sort, Counting Sort, Radix Sort

  Unstable Sorting Algorithms:
  (Algoritma yang tidak mempertahankan urutan relatif dari elemen-elemen yang setara) : Quick Sort, Heap Sort, Selection Sort

- Berdasarkan Kompleksitas Ruang:
  In-Place Sorting Algorithms:
  (Algoritma yang memerlukan ruang tambahan yang minimal, biasanya O(1)): Quick Sort, Heap Sort, Insertion Sort, Selection Sort

  Not In-Place Sorting Algorithms:
  (Algoritma yang memerlukan ruang tambahan lebih dari O(1)) : Merge Sort, Counting Sort, Radix Sort.

- Berdasarkan Kompleksitas Ruang:
  Divide and Conquer Algorithms**:**
  Merge Sort, Quick Sort,
  Incremental Algorithms:
  Insertion Sort, Selection Sort,
  Distribution Algorithms:
  Counting Sort, Bucket Sort, Radix Sort

Dengan memahami klasifikasi ini, pengembang dapat memilih algoritma sorting yang paling sesuai dengan kebutuhan aplikasi mereka berdasarkan faktor seperti ukuran data, kebutuhan kecepatan, dan penggunaan memori.

9.3 Algoritma Sorting Sederhana

9.3.1 Bubble Sort

1. Pengertian Bubble Sort

   Bubble Sort adalah salah satu algoritma sorting yang paling sederhana dan mudah dipahami. Algoritma ini bekerja dengan cara berulang kali membandingkan pasangan elemen yang bersebelahan dalam suatu array dan menukarnya jika mereka berada dalam urutan yang salah(Abdullah *et al.*, 2023). Proses ini diulangi terus-menerus hingga tidak ada lagi elemen yang perlu ditukar, yang berarti array telah terurut.

2. Prosedur pengurutan Algoritma Bubble Sort:
   1. Inisialisasi: Mulai dari elemen pertama dalam array.
   2. Bandingkan: Bandingkan elemen saat ini dengan elemen berikutnya.
      a. Jika elemen saat ini lebih besar dari elemen berikutnya, tukar keduanya.
      b. Jika tidak, biarkan mereka tetap pada posisi mereka.
   3. Lanjutkan ke Pasangan Berikutnya: Pindah ke pasangan elemen berikutnya, dan ulangi proses perbandingan dan pertukaran.
   4. Iterasi Ulang: Setelah mencapai akhir array, kembali ke elemen pertama dan ulangi proses ini untuk seluruh array.

5. Pengulangan Berkurang: Setelah setiap iterasi lengkap, elemen terbesar diurutkan ke posisi yang benar di akhir array. Karena itu, pengulangan berikutnya dapat melewati elemen tersebut dan fokus hanya pada elemen-elemen yang tersisa.
6. Selesai: Proses berlanjut sampai seluruh array terurut, yang ditandai dengan tidak adanya pertukaran elemen dalam satu iterasi penuh.

3. Ilustrasi Algoritma Bubble Sort:

Misalkan kita memiliki array `[5, 3, 8, 4, 2]` dan ingin mengurutkannya menggunakan *Bubble Sort*:

a. Iterasi 1:
   - Bandingkan `5` dan `3`, tukar menjadi `[3, 5, 8, 4, 2]`
   - Bandingkan `5` dan `8`, tidak ada perubahan `[3, 5, 8, 4, 2]`
   - Bandingkan `8` dan `4`, tukar menjadi `[3, 5, 4, 8, 2]`
   - Bandingkan `8` dan `2`, tukar menjadi `[3, 5, 4, 2, 8]`
b. Iterasi 2:
   - Bandingkan `3` dan `5`, tidak ada perubahan `[3, 5, 4, 2, 8]`
   - Bandingkan `5` dan `4`, tukar menjadi `[3, 4, 5, 2, 8]`
   - Bandingkan `5` dan `2`, tukar menjadi `[3, 4, 2, 5, 8]`
   - `8` sudah di tempatnya, tidak perlu dibandingkan.
c. Iterasi 3:
   - Bandingkan `3` dan `4`, tidak ada perubahan `[3, 4, 2, 5, 8]`
   - Bandingkan `4` dan `2`, tukar menjadi `[3, 2, 4, 5, 8]`
   - `5` dan `8` sudah di tempatnya, tidak perlu dibandingkan.
d. Iterasi 4:
   - Bandingkan `3` dan `2`, tukar menjadi `[2, 3, 4, 5, 8]`

- `4`, `5`, dan `8` sudah di tempatnya, tidak perlu dibandingkan.

Pada iterasi ini, array telah diurutkan, dan algoritma berhenti.

4. Karakteristik Bubble Sort:
   a. Kompleksitas Waktu: $O(n^2)$ dalam kasus terburuk dan rata-rata, di mana n adalah jumlah elemen dalam array. Dalam kasus terbaik (ketika array sudah terurut), kompleksitas waktunya adalah $O(n)$.
   b. Kompleksitas Ruang: $O(1)$, karena Bubble Sort adalah algoritma in-place, artinya tidak memerlukan ruang tambahan yang signifikan selain untuk variabel sementara.
   c. Stabilitas: Bubble Sort adalah algoritma stabil, yang berarti elemen-elemen dengan nilai yang sama mempertahankan urutan relatif mereka setelah pengurutan.

   Bubble Sort sering digunakan sebagai pengantar untuk memahami konsep dasar sorting, meskipun dalam praktiknya tidak efisien untuk mengurutkan array besar.

9.3.2 Selection Sort

1. Pengertian Selection Sort

   Selection Sort adalah salah satu algoritma pengurutan sederhana yang bekerja dengan cara memilih elemen terkecil (atau terbesar, tergantung pada pengurutan) dari bagian yang belum diurutkan dari array dan menukarnya dengan elemen pada posisi saat ini(Rahayuningsih, 2016). Proses ini diulang hingga seluruh array terurut.

2. Prosedur pengurutan Algoritma Selection Sort:

   a. Inisialisasi: Mulai dari indeks pertama dari array sebagai elemen yang akan diurutkan.

   b. Pencarian Elemen Terkecil: Cari elemen terkecil di bagian yang belum diurutkan dari array.

   c. Pertukaran: Setelah elemen terkecil ditemukan, tukar elemen tersebut dengan elemen pertama dari bagian yang belum diurutkan.

   d. Pindah ke Indeks Berikutnya: Pindahkan indeks ke posisi berikutnya dan ulangi proses pencarian dan pertukaran hingga seluruh array terurut.

3. Ilustrasi Algoritma Selection Sort:

   Misalkan kita memiliki array `[64, 25, 12, 22, 11]` yang ingin kita urutkan dalam urutan naik.

   a. Iterasi Pertama:
   - Cari elemen terkecil dari array `[64, 25, 12, 22, 11]`.
   - Elemen terkecil adalah `11`.
   - Tukar `11` dengan elemen pertama, sehingga array menjadi `[11, 25, 12, 22, 64]`.

   b. Iterasi Kedua:
   - Cari elemen terkecil dari subarray `[25, 12, 22, 64]`.
   - Elemen terkecil adalah `12`.
   - Tukar `12` dengan elemen kedua, sehingga array menjadi `[11, 12, 25, 22, 64]`.

   c. Iterasi Ketiga:
   - Cari elemen terkecil dari subarray `[25, 22, 64]`.
   - Elemen terkecil adalah `22`.
   - Tukar `22` dengan elemen ketiga, sehingga array menjadi `[11, 12, 22, 25, 64]`.

d. Iterasi Keempat:
   - Cari elemen terkecil dari subarray `[25, 64]`.
   - Elemen terkecil adalah `25`.
   - Tidak ada pertukaran yang diperlukan karena `25` sudah berada pada posisi yang benar.

e. Iterasi Kelima:
   - Subarray terakhir `[64]` hanya memiliki satu elemen, sehingga tidak ada tindakan yang diperlukan.
   - Hasil akhirnya adalah array yang terurut `[11, 12, 22, 25, 64]`.

Pada iterasi ini, array telah diurutkan, dan algoritma berhenti.

4. Karakteristik Selection Sort:
   a. Kompleksitas Waktu: $\boldsymbol{o}(\boldsymbol{n^2})$pada semua kasus (terbaik, rata-rata, dan terburuk).
   b. Kompleksitas Ruang: $\boldsymbol{o}(\mathbf{1})$ (in-place, tidak memerlukan memori tambahan).
   c. Tidak Stabil: Selection Sort tidak mempertahankan urutan relatif elemen dengan nilai yang sama.
   d. Tidak Adaptif: Tidak ada perbaikan kinerja pada array yang sudah hampir terurut.

9.3.3 Insertion Sort

1. Pengertian Insertion Sort:

   Insertion Sort adalah algoritma pengurutan yang menggunakan metode penyisipan untuk mengurutkan elemen dalam sebuah array (Maulana, 2016). Algoritma ini bekerja dengan membagi array menjadi dua bagian: bagian yang sudah terurut dan bagian yang belum terurut. Berikut adalah konsep dasar dari Insertion Sort:

Prinsip Kerja Algoritma Insertion Sort:

1. Pembagian Array:
   - Array dibagi menjadi dua bagian:

   - Bagian yang sudah terurut (mulai dari indeks 0 hingga i-1).

   - Bagian yang belum terurut (dari indeks i hingga akhir array).

2. Iterasi dan Penyisipan:
   - Dimulai dengan elemen pertama dari bagian yang belum terurut, algoritma kemudian menyisipkan elemen tersebut ke dalam posisi yang benar di bagian yang sudah terurut.

   - Proses ini diulang untuk setiap elemen di bagian yang belum terurut, dengan menyisipkan elemen tersebut ke dalam posisi yang tepat pada bagian yang sudah terurut.

3. Penyisipan Elemen:
   - Untuk setiap elemen dari bagian yang belum terurut, algoritma membandingkannya dengan elemen-elemen yang sudah terurut.

   - Elemen tersebut akan dipindahkan ke posisi yang tepat dalam urutan yang sudah terurut, dengan memindahkan elemen-elemen yang lebih besar ke kanan untuk memberikan ruang.

2. Prosedur pengurutan Algoritma Insertion Sort:
   1. Mulai dari Elemen Kedua: Algoritma memulai dari elemen kedua (indeks 1) karena elemen pertama dianggap sudah terurut.

2. Pilih Elemen untuk Disisipkan: Pilih elemen dari bagian yang belum terurut yang akan disisipkan ke dalam bagian yang sudah terurut.
3. Bandingkan dan Geser:
   - Bandingkan elemen yang dipilih dengan elemen-elemen dalam bagian yang sudah terurut dari kanan ke kiri.
   - Geser elemen-elemen yang lebih besar ke kanan untuk membuat ruang bagi elemen yang dipilih.
4. Sisipkan Elemen: Sisipkan elemen yang dipilih pada posisi yang benar di bagian yang sudah terurut.
5. Ulangi: Ulangi proses ini untuk semua elemen di bagian yang belum terurut.

3. Ilustrasi Algoritma Insertion Sort:

   Misalkan kita memiliki array [3, 1, 4, 1, 5] dan kita ingin mengurutkannya menggunakan Insertion Sort.

   a. Langkah 1: Mulai dari elemen kedua (1), bandingkan dengan elemen pertama (3). Sisipkan 1 ke posisi yang tepat sehingga array menjadi [1, 3, 4, 1, 5].
   b. Langkah 2: Pertimbangkan elemen berikutnya (4). Bandingkan dengan elemen yang sudah terurut (1, 3) dan letakkan 4 di tempat yang sesuai, sehingga array tetap [1, 3, 4, 1, 5].
   c. Langkah 3: Pertimbangkan elemen berikutnya (1). Bandingkan dengan elemen yang sudah terurut dan sisipkan 1 di posisi yang benar, sehingga array menjadi [1, 1, 3, 4, 5].
   d. Langkah 4: Elemen terakhir (5) sudah berada di posisi yang benar, jadi array tetap [1, 1, 3, 4, 5].

4. Karakteristik Insertion Sort

a. Sederhana dan Mudah Diimplementasikan: Algoritma ini mudah dipahami dan diimplementasikan.
b. Adaptif: Algoritma ini bekerja lebih efisien pada array yang sudah hampir terurut.
c. Stabil: Menjaga urutan relatif elemen dengan nilai yang sama.

9.4 Algoritma Sorting yang Lebih Kompleks

9.4.1 Merger Sort

1. Pengertian Merger Sort

Merge Sort adalah algoritma pengurutan yang menggunakan pendekatan divide and conquer untuk mengurutkan elemen dalam sebuah array (Sena *et al.*, 2024). Algoritma ini membagi array menjadi bagian-bagian kecil, mengurutkan bagian-bagian tersebut, dan kemudian menggabungkannya kembali menjadi array yang terurut. Berikut adalah konsep dasar dari Merger Sort:

Prinsip Kerja Algoritma Merger Sort:

a. Pembagian (Divide): Array dibagi menjadi dua bagian yang kira-kira sama besar, secara rekursif, hingga setiap bagian hanya memiliki satu elemen atau kosong. Array dengan satu elemen sudah dianggap terurut.
b. Penggabungan (Merge): Setelah array dibagi menjadi bagian-bagian terkecil, algoritma mulai menggabungkan bagian-bagian tersebut kembali. Selama proses penggabungan, elemen-elemen digabungkan dengan cara yang terurut, menghasilkan array yang lebih besar yang sudah terurut.

2. Prosedur pengurutan Algoritma Insertion Sort:
    1. Pembagian Array:
        - Tentukan titik tengah dari array.
        - Bagi array menjadi dua subarray, satu dari awal hingga titik tengah dan satu lagi dari titik tengah hingga akhir.
    2. Pengurutan Subarray:
        - Terapkan Merge Sort secara rekursif pada kedua subarray yang lebih kecil hingga mencapai subarray dengan satu elemen.
    3. Penggabungan Subarray:
        - Gabungkan kedua subarray yang sudah terurut menjadi satu array yang terurut.
        - Proses ini dilakukan dengan membandingkan elemen-elemen dari kedua subarray dan menyusunnya dalam urutan yang benar.
    4. Rekursi:
        - Ulangi proses pembagian dan penggabungan hingga seluruh array terurut.

3. Ilustrasi Algoritma Insertion Sort:
   Misalkan kita memiliki array [38, 27, 43, 3, 9, 82, 10].

    - Langkah 1: Pembagian
        - Bagikan array menjadi [38, 27, 43, 3] dan [9, 82, 10].
        - Bagikan [38, 27, 43, 3] menjadi [38, 27] dan [43, 3].
        - Bagikan [9, 82, 10] menjadi [9] dan [82, 10].
    - Langkah 2: Pengurutan Subarray
        - Terapkan Merge Sort pada [38, 27] menjadi [27, 38] dan [43, 3] menjadi [3, 43].
        - Terapkan Merge Sort pada [82, 10] menjadi [10, 82].

- Langkah 3: Penggabungan
    - Gabungkan [27, 38] dan [3, 43] menjadi [3, 27, 38, 43].
    - Gabungkan [10, 82] dengan [3, 27, 38, 43] menjadi [3, 9, 10, 27, 38, 43, 82].

4. Kelebihan dan Kekurangan Merger Sort :
    a. Kelebihan Merger Sort
        - Kompleksitas Waktu yang Konsisten: Merge Sort memiliki kompleksitas waktu $O(n \log n)$ dalam kasus terburuk, rata-rata, dan terbaik. Ini membuatnya sangat efisien untuk array besar.
        - Stabil: Merge Sort adalah algoritma stabil, artinya elemen dengan nilai yang sama akan tetap dalam urutan yang sama seperti di array awal.
        - Adaptif: Tidak tergantung pada urutan awal elemen dalam array.
    b. Kekurangan Merger Sort:
        - Penggunaan Memori: Merge Sort membutuhkan ruang tambahan untuk menyimpan array sementara selama proses penggabungan. Kompleksitas ruangnya adalah $O(n)$
        - Overhead Rekursi: Meskipun efektif, penggunaan rekursi dalam Merge Sort dapat menyebabkan overhead pada stack jika ukuran array sangat besar.

9.4.2 Quick Sort

1. Pengertian Quick Sort

    Quick Sort adalah algoritma pengurutan yang menggunakan metode divide and conquer untuk mengurutkan elemen dalam array. Algoritma ini bekerja dengan memilih elemen pivot, mengatur elemen di sekitar pivot, dan kemudian secara rekursif menerapkan proses

yang sama pada subarray yang dihasilkan. Berikut adalah konsep dasar dari Quick Sort:

Prinsip Kerja:

1. Pemilihan Pivot:
   Pilih sebuah elemen dari array sebagai pivot. Pivot dapat dipilih dengan berbagai metode, seperti memilih elemen pertama, terakhir, tengah, atau menggunakan metode acak.
2. Partisi (Partition):
   Atur elemen-elemen di sekitar pivot sehingga elemen yang lebih kecil dari pivot berada di kiri pivot dan elemen yang lebih besar dari pivot berada di kanan pivot. Setelah partisi, pivot berada di posisinya yang benar dalam array terurut.
3. Rekursi:
   Terapkan Quick Sort secara rekursif pada subarray yang berada di kiri dan kanan pivot. Proses ini diulang hingga subarray memiliki satu elemen atau kosong, yang berarti sudah terurut.

2. Prosedur pengurutan Algoritma Quick Sort:
   a. Pemilihan Pivot:
      Pilih pivot dari array. Misalnya, jika memilih pivot sebagai elemen terakhir, maka pivot akan menjadi elemen terakhir dalam array.
   b. Partisi:
      Atur elemen-elemen sehingga semua elemen yang lebih kecil dari pivot berada di sebelah kiri dan semua elemen yang lebih besar berada di sebelah kanan. Ini biasanya dilakukan dengan dua indeks yang bergerak dari keduaujung array menuju pusat.

c. Rekursi pada Subarray
d. Terapkan Quick Sort pada subarray di kiri dan kanan pivot secara rekursif hingga subarray memiliki satu elemen atau kosong.

3. Ilustrasi Algoritma Quick Sort:
   Misalkan kita memiliki array [10, 7, 8, 9, 1, 5] dan kita menggunakan elemen terakhir sebagai pivot.

   - Langkah 1: Pilih pivot (5).
   - Langkah 2: Partisi array:
     - Elemen-elemen yang lebih kecil dari 5 (1) dan elemen-elemen yang lebih besar (10, 7, 8, 9) diposisikan di sekitar pivot.
     - Setelah partisi, array menjadi [1, 5, 8, 9, 7, 10] dengan pivot (5) berada di posisinya yang benar.
   - Langkah 3: Terapkan Quick Sort pada subarray [1] dan [8, 9, 7, 10].
   - Langkah 4: Ulangi proses pada subarray [8, 9, 7, 10]:
     - Pilih pivot (10).
     - Partisi array menjadi [8, 9, 7, 10] (karena 10 sudah di posisi yang benar).
     - Terapkan Quick Sort pada subarray [8, 9, 7].
   - Langkah 5: Terapkan Quick Sort pada [8, 9, 7]:
     - Pilih pivot (7).
     - Partisi array menjadi [7, 9, 8].
     - Terapkan Quick Sort pada [9, 8].
- Langkah 6: Setelah semua subarray terurut, array lengkapnya menjadi [1, 5, 7, 8, 9, 10].

4. Kelebihan dan Kekurangan Quick Sort :
    a. Kelebihan Quick Sort
        - Kompleksitas Waktu Rata-Rata yang Baik: Quick Sort memiliki kompleksitas waktu rata-rata $O(n \log n)$, membuatnya efisien untuk array besar.
        - Efisiensi Ruang: Quick Sort adalah algoritma **in-place** yang tidak memerlukan ruang tambahan yang signifikan selain untuk rekursi, sehingga kompleksitas ruangnya adalah $O(\log n)$ dalam kasus rata-rata.
        - Praktis dalam Implementasi: Quick Sort sering kali lebih cepat dibandingkan algoritma lain dalam praktik karena kinerjanya yang baik pada banyak jenis data.
    b. Kekurangan Quick Sort
        - Kompleksitas Waktu Terburuk: Dalam kasus terburuk, seperti ketika array sudah terurut atau hampir terurut dan pivot dipilih secara buruk, Quick Sort memiliki kompleksitas waktu $O(n^2)$. Namun, ini dapat dikurangi dengan pemilihan pivot yang lebih baik.
        - Tidak Stabil: Quick Sort tidak stabil, artinya elemen dengan nilai yang sama mungkin tidak mempertahankan urutan relatif mereka.
        - Overhead Rekursi: Penggunaan rekursi dapat menyebabkan overhead pada stack, terutama untuk array yang sangat besar.

DAFTAR PUSTAKA

Abdullah, M.F. *et al.* (2023) "Penggunaan Algoritma Bubble Sort dalam Pengurutan Nomor Induk Mahasiswa," *Jurnal Ilmu Komputer*, 2(1), pp. 14–19. doi:10.56211/helloworld.v2i1.206.

Maulana, R. (2016) "Analisa Perbandingan Kompleksitas Algoritma Selectionsort Dan Insertionsort," *Informatika*, 3(September), pp. 208–218.

Musiafa, Z. (2022) *Buku ajar STRUKTUR DATA DAN IMPLEMENTASI ALGORITMA (SDIA)*. 1st ed. Edited by Z. Musiafa. Jakarta: Indonesia.

Putri, M.P. *et al.* (2022) *Algoritma dan Struktur Data*. Edited by M.P. Putri. Bandung: Penerbit Widina Bhakti Persada Bandung.

Rahayuningsih, P.A. (2016) "Analisis Perbandingan Kompleksitas Algoritma Pengurutan Nilai (Sorting) Panny," *Jurnal Evolusi*, 4(2).

Sari, N. *et al.* (2018) "Analisis Algoritma Bubble Sort Secara Ascending Dan Descending Serta Implementasinya Menggunakan Bahasa Pemrograman Java Analisis Algoritma Bubble Sort Secara Ascending Dan Descending Serta Implementasinya Menggunakan Bahasa Pemrograman Java," *ADI Bisnis Digital Interdisiplin Jurnal*, 3(1), pp. 16–23. doi:https://doi.org/10.34306/abdi.v3i1.625.

Sena, M.B. *et al.* (2024) "Perbandingan kinerja algoritma sorting dalam pengurutan data menggunakan bahasa python," in *Prosiding Seminar Nasional Sains dan Teknologi Seri 02*, pp. 310–314.

Yanti, F. and Emi Sita Eriana, Mk. (2024) *Algoritma Sorting Dan Searching*. 1st ed. Edited by F. Yanti. EUREKA MEDIA AKSARA.

Zutshi, A. and Goswami, D. (2021) "Systematic review and exploration of new avenues for sorting algorithm," *International Journal of Information Management Data Insights*, 1(2), p. 100042. doi:10.1016/j.jjimei.2021.100042.

INDEKS

A

D

E

F

H

L

M

N

P

Q

R

S

GLOSARIUM

Array : struktur data yang menyimpan elemen-elemen data dengan tipe data yang sama di lokasi memori yang berdekatan, yang memungkinkan akses cepat ke setiap elemen berdasarkan indeksnya.

Array : Struktur data linear di mana semua elemen disusun secara berurutan.

Array : Struktur data yang menyimpan sekumpulan elemen dengan tipe data yang sama, di mana setiap elemen memiliki indeks yang menunjukkan posisinya dalam array.

Buffer : Area di memori utama yang digunakan untuk menyimpan atau menahan data sementara.

Dequeue : Menghapus atau mengeluarkan item pada antrean paling depan.

Elemen : Satuan data individu yang merupakan bagian dari struktur data seperti array atau list.

Enqueue : Menambahkan item baru ke belakang antrean.

FIFO : Prinsip "first-in first-out" dimana item yang pertama masuk akan menjadi item pertama yang keluar.

Fungsi / Function : Subproses yang mempunyai nilai balik

Graf (Graph): Struktur data yang terdiri dari simpul-simpul (nodes) yang dihubungkan oleh garis-garis (edges), digunakan untuk memodelkan hubungan antar objek.

High (Batas Atas): Indeks atau posisi akhir dari suatu struktur data, seperti array atau list, yang digunakan untuk menunjukkan batas atas dalam pencarian atau manipulasi data. Dalam pencarian biner, high adalah posisi akhir interval pencarian yang terus disesuaikan sampai ditemukan elemen yang dicari atau interval pencarian habis.

Larik : istilah lain dari array.

LIFO : Prinsip "*last-in first-out*" dimana item yang terakhir masuk akan menjadi item pertama yang keluar.

Linear Search (Pencarian Linier): Metode pencarian di mana setiap elemen dalam struktur data diperiksa satu per satu secara berurutan hingga elemen yang dicari ditemukan atau hingga semua elemen telah diperiksa.

Linked list : struktur data yang terdiri dari serangkaian elemen yang saling terhubung dengan bantuan pointer, di mana setiap elemen disebut sebagai simpul atau node

List: Struktur data yang menyimpan urutan elemen yang bisa memiliki tipe data berbeda dan ukurannya dapat dinamis, tidak seperti array yang biasanya memiliki ukuran tetap**.**

Low (Batas Bawah): Indeks atau posisi awal dari suatu struktur data, seperti array atau list, yang digunakan untuk menunjukkan batas bawah saat melakukan pencarian atau manipulasi data. Dalam pencarian biner, low adalah posisi awal interval pencarian.

Manajemen Memori : Proses pengelolaan dan pengaturan penggunaan memori dalam komputer untuk memastikan efisiensi dan efektivitas. Ini melibatkan alokasi, penggunaan, dan dealokasi memori selama eksekusi program.

Mid (Tengah): Posisi tengah dalam suatu struktur data, seperti array atau list, sering digunakan dalam algoritma pencarian biner untuk membagi data menjadi dua bagian.

Nilai Balik : Nilai yang diberikan ke subproses yang memanggil fungsi

Node : elemen dasar yang membentuk struktur data linked lisgt, yang terdiri dari dua komponen yaitu data dan pointer.

Parameter / argumen : Nilai yang diinputkan ke dalam subproses

Pointer : variabel khusus yang menyimpan alamat memori dari variabel lain, bukan nilai sebenarnya dari variabel tersebut.

Prosedure : Subproses yang tidak mempunyai nilai balik

Queue: Struktur data yang mengikuti prinsip FIFO (First In, First Out), yaitu elemen pertama yang dimasukkan adalah elemen pertama yang dikeluarkan. Operasi dasar pada queue meliputi enqueue (menambahkan elemen di bagian belakang) dan dequeue (menghapus elemen dari bagian depan). Contoh penggunaan queue adalah pada sistem antrean atau penjadwalan tugas dalam CPU.

Record dan Pointer : Dua konsep dasar dalam pemrograman dan struktur data yang berfungsi untuk mengelola dan mengorganisasi data secara efisien.

Searching (Pencarian): Proses menemukan elemen tertentu dalam suatu struktur data, seperti array, list, atau database, berdasarkan kriteria atau nilai yang ditentukan.

Senarai berantai : istilah lain dari linked list.

Simpul : istilah lain dari node.

Single linked list : sebuah linked list dimana setiap satu simpul/node hanya memiliki sebuah variabel pointer yang digunakan untuk menunjuk pada simpul/node berikutnya.

Stack: Struktur data yang mengikuti prinsip LIFO (Last In, First Out), yaitu elemen terakhir yang dimasukkan adalah elemen pertama yang dikeluarkan. Operasi dasar pada stack meliputi push (menambahkan elemen ke bagian atas) dan pop (menghapus elemen dari bagian atas). Contoh penggunaan stack adalah pada pengelolaan undo/redo di aplikasi.

Struktur Data Record : Tipe struktur data yang digunakan untuk mengelompokkan beberapa elemen yang memiliki tipe data berbeda ke dalam satu kesatuan logis. Record sering kali digunakan untuk merepresentasikan entitas dunia nyata yang memiliki atribut terkait.

Struktur Kode yang Jelas : Mengacu pada cara penulisan kode yang terorganisir, mudah dibaca, dan dipahami. Ini sangat penting dalam pengembangan perangkat lunak untuk memastikan bahwa kode dapat dikelola dengan baik, baik oleh penulisnya maupun oleh orang lain yang akan memelihara atau mengembangkan kode tersebut di masa depan.

BIODATA PENULIS

Gusti Eka Yuliastuti, S.Kom, M.Kom
Dosen Program Studi Teknik Informatika
Fakultas Teknik Elektro dan Teknologi Informasi
Institut Teknologi Adhi Tama Surabaya (ITATS)

Penulis lahir di Surabaya pada tahun 1994. Penulis memperoleh gelar Sarjana Program Studi Informatika pada tahun 2016 dan gelar Magister Program Studi Ilmu Komputer pada tahun 2018 dari Fakultas Ilmu Komputer Universitas Brawijaya Malang.

Penulis merupakan dosen tetap Jurusan Teknik Informatika Institut Teknologi Adhi Tama Surabaya (ITATS) sejak 2019. Selain aktif dalam bidang pendidikan, penulis juga aktif dalam bidang penelitian dan pengabdian. Beberapa hasil penelitian penulis dimuat pada jurnal nasional terakreditasi dan jurnal internasional bereputasi. Bidang penelitian penulis yakni pada Algoritma Evolusi.

BIODATA PENULIS

Rudiansyah, S.Kom., M.Kom
Dosen Program Studi Sistem Informasi
Fakultas Ilmu Komputer Universitas Sjakhyakirti Palembang

Penulis lahir di Palembang tanggal 20 November 1979. Penulis adalah dosen tetap pada Program Studi Sistem Informasi Fakultas Ilmu Komputer, Universitas Sjakhyakirti Palembang. Menyelesaikan pendidikan S1 pada Jurusan Sistem Informasi dan melanjutkan S2 pada Jurusan Teknik Informatika. Penulis menekuni bidang Menulis.

BIODATA PENULIS

Khairunnisa Samosir,S.Kom.,M.Kom
Dosen Program Studi Ilmu Komputer
Fakultas Teknik Universitas Graha Nusantara

Penulis Lahir di Tanjungbalai. Penulis di besarkan seorang Ayah Guru dan Ibu Guru. Penulis anak ke 5 dari 6 bersaudara. Alhamdulillah telah menerbitkan beberapa Buku yang sesuai Bidang Penulis. Mudah-mudahan diberi Kesempatan untuk menjadi Seorang Profesor dan bisa Lanjut S3.

BIODATA PENULIS

Budanis Dwi Meilani, ST., M.Kom
Dosen Program Studi Sistem Informasi
Fakultas Teknik Elektro dan Teknologi Informasi
Institut Teknologi Adhi Tama Surabaya

Ketertarikan penulis pada bidang ilmu komputer terutama Informatika dimulai pada tahun 1998 silam. Penulis menyelesaikan Pendidikan S1 Teknik Informatika di Institut Teknologi Adhi Tama Surabaya (ITATS) dan berhasil lulus tahun 1998. Penulis kemudian melanjutkan Pendidikan S2 Teknik Informatika di Institut Teknologi 10 Nopember Surabaya (ITS) dan berhasil lulus tahun 2008.

Penulis memiliki kepakaran dibidang Algorithma & Pemrograman, Pemrograman Berorientasi Obyek, Struktur Data, Data Mining dan Sistem Pendukung Keputusan. Saat ini penulis berkarir sebagai dosen tetap di Institut Teknologi Adhi Tama Surabaya (ITATS) pada program studi Sistem Informasi. Untuk mewujudkan karir menjadi dosen maka penulis juga aktif dalam penelitian penelitian yang berhubungan dengan bidang kepakaran. Penelitian yang dilakukan penulis mendapatkan dana dari berbagai sumber diantaranya dana yang didapat dari mandiri, institusi swasta, institusi tempat bekerja dan Kemenristek DIKTI. Hasil dari penelitian sudah banyak yang

penulis publikasikan berupa jurnal dan prosiding. Semoga kedepannya penulis bisa memberikan kontribusi poitif bagi bangsa dan negara.

Email Penulis: budanis@itats.ac.id.

BIODATA PENULIS

Sulistyowati, S.T., M.Kom.
Dosen Program Studi Sistem Informasi
Fakultas Teknik Elektro dan Teknologi Informasi
Institut Teknologi Adhi Tama Surabaya

Penulis lahir di Banyuwangi tanggal 5 April 1977. Penulis adalah dosen tetap pada Jurusan Sistem Informasi Fakultas Teknik Elektro dan Teknologi Informasi, Institut Teknologi Adhi Tama Surabaya mulai tahun 2000 sampai sekarang. Penulis menyelesaikan pendidikan S1 pada Jurusan Teknik Informatika di Institut Teknologi Adhi Tama Surabaya dan melanjutkan pendidikan S2 pada Jurusan Teknik Informatika di Institut Teknologi Sepuluh Nopember Surabaya. Bidang riset penulis adalah di bidang Riset Operasi, Rekayasa Perangkat Lunak, Sistem Pendukung Manajemen, dan Pemrograman. Kritik dan saran yang membangun dapat disampaikan melalui email sulis_tyowati@itats.ac.id.

BIODATA PENULIS

Sugiarto Cokrowibowo
Dosen Program Studi Informatika
Fakultas Teknik Universitas Sulawesi Barat

Penulis lahir di Palopo tanggal 24 Mei 1986. Penulis adalah dosen tetap pada Program Studi Informatika yang saat ini diberi amanah sebagai Wakil Dekan 1 Bidang Akademik dan Kemahasiswaan Fakultas Teknik Universitas Sulawesi Barat. Menyelesaikan pendidikan S1 pada Program Studi Matematika Universitas Hasanuddin dan melanjutkan S2 pada Program Studi Teknik Elektro Universitas Hasanuddin. Bidang kajian yang diminati oleh penulis, antara lain artificial intelligence; soft computing; internet of things; programming; algoritma dan struktur data. Penulis dapat dihubungi melalui surel: sugiarto.cokrowibowo.id@gmail.com penulis juga aktif membuat konten video pembelajaran algoritma dan pemrograman di channel Youtube FOSALGO https://www.youtube.com/@FOSALGO

BIODATA PENULIS

Putri Ayu Maharani, S.T., M.Sc.
Dosen Program Studi Ilmu Komputer
Jurusan Teknologi Produksi dan Industri
Institut Teknologi Bacharuddin Jusuf Habibie

Penulis lahir di Makassar tanggal 11 Juni 1994. Penulis adalah dosen tetap pada Program Studi Ilmu Komputer Jurusan Teknologi Produksi dan Industri, Institut Teknologi Bacharuddin Jusuf Habibie. Menyelesaikan pendidikan S1 pada Program Studi Teknik Informatika dan melanjutkan S2 pada Program Studi Ilmu Komputer. Penulis menekuni bidang *Software Engineering*.

BIODATA PENULIS

Ellbert Hutabri, S.Kom., M.Kom.
Dosen Program Studi Teknik Informatika
Fakultas Teknik dan Komputer Universitas Putera Batam

Penulis lahir di Sicincin pada tanggal 5 Oktober 1985. Saat ini, penulis merupakan dosen tetap di Program Studi Teknik Informatika, Fakultas Teknik dan Komputer, Universitas Putera Batam. Penulis menyelesaikan pendidikan S1 dan S2 di Jurusan Sistem Informasi. Selama kariernya, penulis telah aktif menulis dan menerbitkan sejumlah buku serta artikel yang berhubungan dengan ilmu komputer seperti Arsitektur Komputer dan Sistem E-commerce.

BIODATA PENULIS

Kusnaeni, M.Si.
Dosen Program Studi Matematika
Jurusan Sains

Penulis lahir di Makassar tanggal 16 Januari 1995. Penulis adalah dosen tetap pada Program Studi Sistem Energi Jurusan Sains, Institut Teknologi Bacharuddin Jusuf Habibie. Menyelesaikan pendidikan S1 pada Jurusan Matemtika Universitas Hasanuddin dan melanjutkan S2 pada Jurusan Statistika dan Sains Data Institut Pertanian Bogor. Penulis menekuni penelitian-penelitian pada bidang Sains Data.