2025 EDITION

MODUL PRAKTIKUM STRUKTUR DATA

DOSEN PEMBIMBING : WAHYU EKO SULISTIONO, S.T., M.SC

PROGRAM STUDI TEKNIK INFORMATIKA

JURUSAN TEKNIK ELEKTRO
FAKULTAS TEKNIK
UNIVERSITAS LAMPUNG

DAFTAR ISI

KATA PENGANTAR

Puji syukur kita panjatkan ke hadirat Allah SWT atas segala limpahan rahmat dan karunia-Nya sehingga penyusunan Modul Praktikum Struktur Data ini dapat diselesaikan dengan baik. Modul ini disusun sebagai salah satu penunjang kegiatan praktikum pada mata kuliah Struktur Data di Program Studi Teknik Informatika Universitas Lampung.

Modul ini berisi materi-materi pokok yang disusun secara sistematis untuk membantu mahasiswa memahami konsep, penerapan, dan implementasi struktur data dalam pemrograman. Dengan adanya modul ini, diharapkan mahasiswa dapat lebih mudah mempelajari teori yang telah disampaikan di perkuliahan dan mengaplikasikannya secara langsung melalui latihan praktikum.

Penyusun menyadari bahwa modul ini masih jauh dari kata sempurna. Oleh karena itu, kritik dan saran yang membangun dari para dosen, asisten praktikum, serta mahasiswa sangat diharapkan demi penyempurnaan modul di masa mendatang.

Akhir kata, semoga Modul Praktikum Struktur Data ini dapat memberikan manfaat yang sebesar-besarnya bagi mahasiswa dalam memahami konsep struktur data serta menjadi bekal dalam mengembangkan kemampuan pemrograman di masa depan.

Bandar Lampung, September 2025

Tim Penyusun Modul Praktikum

TATA TERTIB PRAKTIKUM

1. Mahasiswa yang diizinkan mengikuti praktikum adalah yang telah terdaftar dan memenuhi syarat yang telah ditentukan.
2. Praktikan dilarang mengoperasikan alat tanpa seizin asisten.
3. Praktikum dilaksanakan sesuai jadwal dan praktikan harus hadir 15 menit sebelum praktikum dimulai.
4. Praktikan harus berpakaian rapi dan tidak diperkenankan memakai kaos oblong.
5. Praktikan dilarang membuat kegaduhan selama berada dalam laboratorium dan wajib menjaga kebersihan di dalam maupun di halaman laboratorium.
6. Praktikan wajib mengerjakan tugas pendahuluan dari setiap percobaan sebelum mengikuti praktikum.
7. Ketidakhadiran peserta dalam suatu praktikum harus atas sepengetahuan asisten yang bersangkutan dan wajib diganti di lain hari. Apabila praktikan tidak hadir pada salah satu percobaan maka praktikan dinyatakan tidak lulus praktikum.
8. Praktikan harus melaksanakan asistensi kepada asisten yang bersangkutan selama penulisan laporan minimal 2 kali dan maksimal 3 kali.
9. Pelanggaran terhadap tata tertib akan diberikan sanksi: “TIDAK DIPERKENANKAN MENGIKUTI PRAKTIKUM”

PENGETAHUAN WAJIB

1. Memahami cara kerja dan mampu mengoperasikan komputer.
2. Mengetahui konsep file (berkas), dapat melakukan operasi dasar file (copy, paste,delete, dsb).
3. Dapat memanipulasi file menggunakan text editor.
4. Memahami konsep dasar pemrograman komputer.
5. Memahami syntax dari bahasa pemrograman yang digunakan.
6. Mampu menggunakan CLI (Command Line Interface).
7. Dapat membedakan antara huruf besar, huruf kecil, koma (,), titik koma (;), titik dua (:), kutip tunggal (‘), kutip ganda (“), dan lainnya yang diperlukan.

I PERCOBAAN 1: VARIABEL STRUKTUR DATA

1.1 Tujuan Percobaan

1. Dapat membuat dan memahami penggunaan array
2. Dapat membuat dan memahami penggunaan linked list
3. Dapat membuat dan memahami penggunaan vector

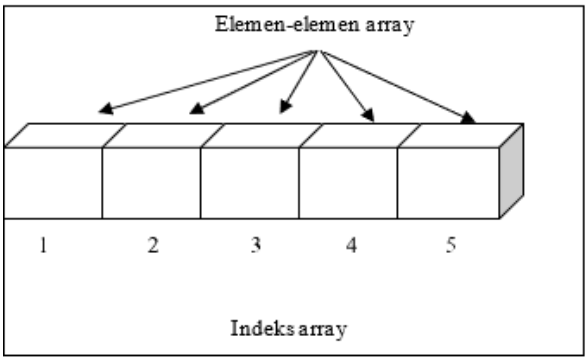
1.2 Tinjauan Pustaka

1.2.1 Pengertian Array

Array merupakan salah satu bentuk struktur data dalam pemrograman yang digunakan untuk menyimpan sejumlah elemen dengan tipe data yang sama (homogen). Keberadaan array memberikan kemudahan dalam proses penyimpanan maupun pengelolaan data, sebab banyak nilai dapat dimuat hanya dalam satu variabel terstruktur. Tanpa array, jika terdapat sejumlah besar data—misalnya dua puluh nilai—maka programmer harus mendeklarasikan dua puluh variabel berbeda. Hal tersebut tentu kurang efisien, baik dalam penulisan kode maupun dalam proses pemanggilan data. Dengan menggunakan array, keseluruhan nilai tersebut dapat disimpan dalam satu variabel tunggal yang diorganisasikan melalui indeks.

Setiap elemen dalam array menempati posisi tertentu yang disebut indeks. Indeks ini berfungsi sebagai acuan untuk mengakses maupun memodifikasi nilai pada elemen tersebut. Pada umumnya, indeks array dimulai dari angka nol, sehingga elemen pertama berada pada indeks 0, elemen kedua pada indeks 1, dan seterusnya hingga jumlah elemen yang ditentukan.

Kelebihan utama dari array terletak pada efisiensinya, baik dalam penyimpanan maupun akses data. Struktur ini memungkinkan pelaksanaan berbagai operasi, seperti perhitungan matematis, pengurutan data, pencarian elemen, penambahan maupun penghapusan data, serta penggantian nilai elemen, dengan lebih sederhana, sistematis, dan cepat.



Gambar 1.1 Ilustrasi Array

Deklarasi array dilakukan dengan menentukan tipe data dari elemen yang akan disimpan serta jumlah elemen yang dibutuhkan. Dengan cara ini, array berfungsi sebagai suatu objek yang dapat menampung banyak data sejenis hanya dalam satu variabel.

Setelah array didefinisikan, ia dapat diberikan nilai awal (inisialisasi), baik langsung pada saat deklarasi maupun diisi kemudian sesuai dengan kebutuhan program. Setiap elemen array memiliki posisi tertentu yang disebut indeks, yang dimulai dari 0 hingga (jumlah elemen – 1). Indeks inilah yang digunakan untuk mengakses, membaca, maupun mengubah nilai dari elemen-elemen yang ada di dalam array.

Dalam praktik pemrograman, pengolahan data pada array umumnya dilakukan dengan memanfaatkan struktur perulangan, seperti for, sehingga seluruh elemen dapat diproses secara berurutan dan teratur. Melalui proses deklarasi, inisialisasi, dan pemanfaatan indeks, array menjadi struktur data yang efektif untuk menyimpan sekaligus mengelola sekumpulan data dengan efisien.

1.2.2 Jenis-jenis Array

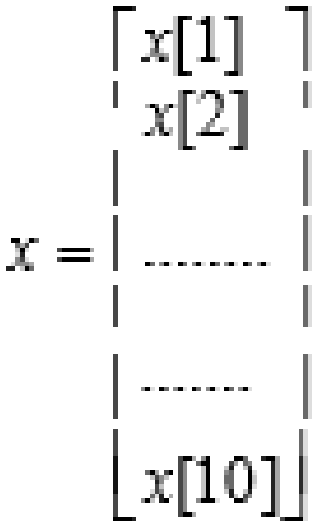
1. Array Satu Dimensi

Array satu dimensi merupakan bentuk struktur data yang digunakan untuk menyimpan sejumlah elemen dengan tipe data yang sama dalam susunan linier. Setiap elemen memiliki posisi tertentu yang dikenal dengan istilah indeks, di mana indeks dimulai dari 0 untuk elemen pertama, 1 untuk elemen kedua, dan seterusnya.

Penggunaan array satu dimensi sangat umum dalam pengelolaan data yang tersusun secara berurutan. Dengan adanya indeks, akses maupun manipulasi elemen menjadi lebih mudah

dilakukan, sehingga array ini sering dimanfaatkan untuk menyelesaikan berbagai permasalahan yang memerlukan penyimpanan dan pengolahan data secara teratur.

```
int x[10];
```



Gambar 1.2 Ilustrasi Array 1 Dimensi

2. Array Multidimensi

Array multidimensi merupakan jenis array yang memiliki dua dimensi atau lebih. Struktur data ini menyusun elemen-elemen dalam bentuk berlapis sehingga menyerupai matriks atau tabel. Setiap elemen pada array multidimensi diidentifikasi menggunakan koordinat yang ditentukan oleh indeks dari masing-masing dimensi. Sebagai contoh, array dua dimensi dapat dianalogikan sebagai sebuah tabel dengan baris dan kolom, sedangkan array tiga dimensi dapat digambarkan sebagai kumpulan matriks yang tersusun secara berlapis. Dengan memanfaatkan array multidimensi, penyimpanan serta pengolahan data menjadi lebih sistematis dan terorganisasi. Beberapa penerapan array multidimensi yang umum ditemui antara lain pada representasi matriks, pengolahan citra atau gambar, serta data yang bersifat berlapis. Bentuk yang paling sering digunakan adalah array dua dimensi atau matriks, misalnya dalam operasi penjumlahan matrix.

```
int y[10][10];

int z[10][10][10];
```

Gambar 1.3 Bentuk Array 2 Dimensi

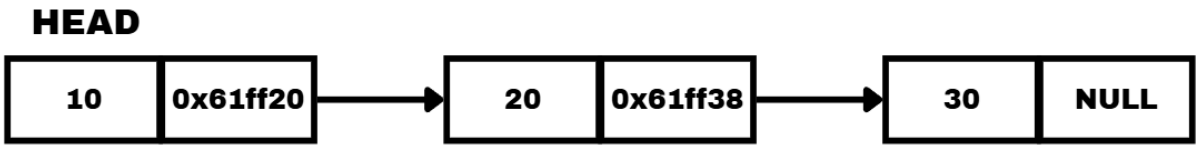
1.2.3 Pengertian Linked List

Linked List merupakan salah satu struktur data linier di mana elemen-elemennya tidak disimpan pada lokasi memori yang berdekatan, melainkan saling terhubung melalui pointer. Struktur ini membentuk rangkaian node yang saling terkait, di mana setiap node berisi data serta alamat yang menunjuk ke node berikutnya (atau node sebelumnya, pada tipe tertentu).

Secara umum, sebuah node dalam linked list terdiri dari dua bagian utama, yaitu:

1. Data, yang menyimpan nilai aktual atau informasi yang dimiliki oleh node.
2. Pointer, yang berfungsi menyimpan alamat memori dari node berikutnya atau node sebelumnya dalam urutan.

Dengan mekanisme tersebut, linked list dapat direpresentasikan sebagai rangkaian node yang saling terhubung, di mana setiap node menunjuk ke node lain sesuai dengan urutan yang telah ditentukan.



Gambar 1.4 Ilustrasi Linked List

Salah satu kelemahan utama dari linked list adalah waktu akses data yang bersifat linier terhadap jumlah node. Artinya, semakin banyak node yang ada, maka semakin lama pula waktu yang dibutuhkan untuk mengakses data, karena setiap node hanya dapat dijangkau secara

berurutan. Untuk mencapai node tertentu, kita harus melewati node-node sebelumnya terlebih dahulu, sehingga membuat linked list kurang efisien dalam proses akses acak.

Beberapa operasi dasar yang dapat dilakukan pada linked list antara lain:

1. Menambahkan Node

   Proses penambahan simpul dapat dilakukan berdasarkan posisi tertentu, misalnya:

   - Menambahkan simpul di belakang simpul terakhir.
   - Menambahkan simpul sebagai simpul pertama.

2. Menghapus Node

   Penghapusan simpul dapat dilakukan pada posisi depan, belakang, maupun tengah. Untuk menghapus simpul di tengah, diperlukan pointer yang menunjuk ke simpul sebelum simpul yang akan dihapus.

3. Mencari Node

   Proses pencarian dilakukan dengan menelusuri node satu per satu, sama halnya dengan membaca isi simpul. Namun, ditambahkan pengujian untuk menentukan apakah data yang dicari ada atau tidak.

4. Mencetak data Node

   Pencetakan dapat dilakukan dengan membaca isi simpul secara maju (dari node pertama ke node terakhir) atau secara mundur (dari node terakhir ke node pertama, pada jenis linked list tertentu seperti *doubly linked list*).

1.2.4 Jenis-jenis Linked List

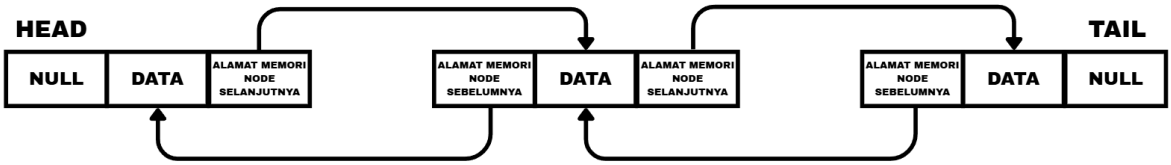
1. Singly linked list

   Singly linked list adalah linked list unidirectional. Single-linked list seperti sistem kereta api, yang menghubungkan setiap gerbong ke kerbong berikutnya.Jadi, kita hanya dapat melintasinya dalam satu arah, yaitu dari simpul kepala ke simpul ekor.

Gambar 1.4 Single Linked List

2. Double linked list

Double linked list adalah linked list dua arah. Jadi, kita bisa melintasinya secara dua arah, maju dan mundur. Dalam sebuah doubly linked list terdapat 2 pointer yang teridiri dari pointer HEAD yang menunjuk ke node selanjutnya pada linked list itu sendiri dan pointer TAIL yang menunjuk ke node sebelumnya pada linked list. Pada doubly linked list pointer HEAD pada node pertama selalu bernilai NULL, karena ini adalah data pertama dalam linked list tersebut. Begitu juga dengan pointer TAIL yang selalu NULL pada node terakhir yang menjadi penanda data terakhir dalam linked list.



Gambar 1.5 Doubly Linked List

1.2.5 Pengertian Vector

Vector merupakan salah satu struktur data fundamental yang digunakan dalam pemrograman untuk menyimpan sekumpulan elemen dengan tipe data yang sama secara berurutan. Berbeda dengan array statis yang ukurannya harus ditentukan sejak awal, vector bersifat dinamis sehingga kapasitasnya dapat bertambah atau berkurang secara otomatis sesuai kebutuhan. Implementasi vector pada umumnya menggunakan array sebagai penyimpanan internal, dan ketika kapasitas penuh, sistem akan melakukan realokasi dengan strategi tertentu seperti penggandaan ukuran agar operasi penambahan tetap efisien.

Keunggulan utama vector terletak pada efisiensinya dalam mengakses data. Setiap elemen dapat diakses langsung menggunakan indeks dengan waktu konstan. Namun, untuk operasi penambahan atau penghapusan elemen di bagian tengah atau awal vector, kinerja cenderung lebih lambat karena membutuhkan pergeseran elemen lain. Hal ini membuat vector paling sesuai digunakan untuk kasus-kasus di mana penambahan data banyak dilakukan di bagian akhir dan akses acak (random access) sangat dibutuhkan.

Dalam kehidupan nyata, vector memiliki banyak penerapan. Pada aplikasi e-commerce, data produk yang jumlahnya dinamis dapat disimpan dalam bentuk vector sehingga produk baru bisa ditambahkan atau dihapus dengan mudah. Pada sistem informasi akademik, daftar mahasiswa dalam sebuah kelas dapat direpresentasikan dengan vector karena jumlah mahasiswa dapat berubah setiap semester. Dalam bidang pengolahan citra digital, nilai intensitas piksel disimpan dalam vector sehingga gambar dapat diproses untuk keperluan filtering, kompresi, atau deteksi objek. Selain itu, vector juga berperan penting dalam kecerdasan buatan, khususnya pada pemrosesan bahasa alami, di mana kata atau dokumen direpresentasikan dalam bentuk vector numerik multidimensi (word embedding) untuk menghitung kemiripan antar kata atau dokumen.

1.3 Percobaan

1.3.1 Percobaan I-1: Array

```cpp
#include <iostream>
using namespace std;
void menu () {
    cout << "1. Tampilkan address array \n";
    cout << "2. Tampilkan address dari semua index array \n";
    cout << "3. Masukkan nilai kedalam semua index array \n";
    cout << "4. Keluar \n";
}
int main () {
    int a[5];
    int choice;
    bool running = true;
    while (running) {
        menu ();
        cin >> choice;
        switch (choice) {
            case 1 :
                cout << &a << endl;
                break;
            case 2 :
                for (int i=0; i<5; i++){
                    cout << &a[i] << endl;
                }
                break;
            case 3 :
                for (int i =0; i<5; i++) {
                    cin >> a[i];
                }
                    break;
            case 4 :
                running = false;
                return 0;
        }
    }
}
```

Array1D.cpp

```cpp
#include <iostream>
using namespace std;
void menu () {
    cout << "1. Tampilkan address array \n";
    cout << "2. Tampilkan address dari semua index array \n";
    cout << "3. Masukkan nilai kedalam semua index array \n";
```

```
    cout << "4. Keluar \n";
}
int main () {
    int b[3][2];
    int choice;
    bool running = true;
    while (running) {
        menu ();
        cin >> choice;
        switch (choice) {
            case 1 :
                cout << &b << endl;
                break;
            case 2 :
                for (int i=0; i<3; i++){
                    for (int j=0; j < 2; j++){
                        cout << &b[i][j] << endl;
                    }
                }
                break;
            case 3 :
                for (int i =0; i<3; i++) {
                    for (int j=0; j < 2; j++){
                        cin >> b[i][j];
                    }
                }
                    break;
            case 4 :
                running = false;
                return 0;
        }
    }
}
```

Array2D.cpp

1.3.2 Percobaan I-2: Linked List

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

struct node{
    int data;
    node *next;
} *start, *newptr, *save, *ptr, *rear;

node *create_new_node(int);
void insert_at_beg(node *);
void insert_at_end(node *);
```

```
void display(node *);
void delete_node();
int main() {
    start = rear = NULL;
    int c;
    char choice = 'y';
    int insert;
    while (choice == 'y' || choice == 'Y') {
        cout << "Nilai baru = ";
        cin >> c;
        cout << "Membuat node baru" << endl;
        newptr = create_new_node(c);
        if(newptr != NULL) {
            cout << "Berhasil membuat node baru" << endl;
        }
        else {
            cout << "Node baru tidak dapat dibuat";
            exit(1);
        }
        cout << "1. Depan \n";
        cout << "2. Belakang \n";
        cout << "Masukkan dimana ?";
        cin >> insert;
        switch (insert) {
            case 1 :
            insert_at_beg(newptr);
            cout << "Node dimasukkan di awal list" << endl;
            break;
            case 2 :
            insert_at_end(newptr);
            cout << "Node dimasukkan di akhir list" << endl;
            break;
        }
        cout << "List :";
        display(start);
        cout << "Mau membuat node baru? (y/n)";
        cin >> choice;
    }
    do {
        cout << "List:";
        display(start);
        cout << "Mau menghapus node pertama? (y/n)";
        cin >> choice;
        if(choice == 'y' || choice == 'Y'){
            delete_node();
        }
    } while (choice== 'y' || choice == 'Y');
```

```
    return 0;
}

node *create_new_node(int n){
    ptr = new node;
    ptr->data = n;
    ptr->next = NULL;
    return ptr;
}

void insert_in_beg(node *np) {
    if(start == NULL){
        start = rear = np;
    }
    else{
        save = start;
        start = np;
        np->next = start;
    }
}
void insert_in_end(node *np) {
    if(start == NULL){
        start = rear = np;
    }
    else{
        rear->next = np;
        rear = np;
    }
}

void delete_node() {
    if(start == NULL){
        cout << "Underflow!!!" << endl;
    }
    else{
        ptr = start;
        start = start->next;
    }
}
```

SingleLinkedList.cpp

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

struct node{
    int data;
    node *prev;
```

```
    node *next;
} *start, *newptr, *ptr, *rear;

node *create_new_node(int);
void insert_node(node *);
void traverseForward(node *);
void traverseBackward(node *);

int main() {
    start = rear = NULL;
    int c;
    char choice = 'y';
    while (choice == 'y' || choice == 'Y') {
        cout << "Nilai baru = ";
        cin >> c;
        cout << "Membuat node baru" << endl;
        newptr = create_new_node(c);
        if(newptr != NULL) {
            cout << "Berhasil membuat node baru" << endl;
        }
        else {
            cout << "Node baru tidak dapat dibuat";
            exit(1);
        }
        insert_node(newptr);
        cout << "Node dimasukkan ke list" << endl;
        cout << "Mau membuat node baru? (y/n)";
        cin >> choice;
    }
    int direction;
    cout << "1. Maju" << endl;
    cout << "2. Mundur" << endl;
    cout << "Arah transversal yang mana?";
    cin >> direction;
    switch (direction){
        case 1 :
        cout << "Traversal maju: ";
        traverseForward(start);
        break;
        case 2 :
        cout << "Transversal mundur: ";
        traverseBackward(rear);

    }
    return 0;
}
```

```
node *create_new_node(int n){
    ptr = new node;
    ptr->data = n;
    ptr->next = NULL;
    return ptr;
}

void insert_node(node *np) {
    if(start == NULL){
        start = rear = np;
    }
    else{
        rear->next = np;
        rear = np;
    }
}
void traverseForward(node* head) {
    node* current = head;
    while (current != NULL) {
        cout << current->data << " ";
        current = current->next;
    }
    cout << endl;
}

void traverseBackward(node* tail) {
    node* current = tail;
    while (current != NULL) {
        cout << current->data << " ";
        current = current->prev;
    }
    cout << endl;
}
```

DoublyLinkedList.cpp

1.3.3 Percobaan I-3 : Vector

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

struct Vector {
    int* data;
    int capacity;
    int length;
};

void init(Vector& v) {
    v.capacity = 2;
    v.length = 0;
    v.data = new int[v.capacity];
}

void resize(Vector& v, int newCap) {
    int* newData = new int[newCap];
    for (int i = 0; i < v.length; i++) {
        newData[i] = v.data[i];
    }
    delete[] v.data;
    v.data = newData;
    v.capacity = newCap;
}

void push_back(Vector& v, int value) {
    if (v.length == v.capacity) {
        resize(v, v.capacity * 2);
    }
    v.data[v.length] = value;
    v.length++;
}

void pop_back(Vector& v) {
    if (v.length > 0) {
        v.length--;
    }
}

int get(Vector& v, int index) {
    if (index >= 0 && index < v.length) {
        return v.data[index];
    }
    return -1;
}
```

```
void set(Vector& v, int index, int value) {
    if (index >= 0 && index < v.length) {
        v.data[index] = value;
    }
}

int size(Vector& v) {
    return v.length;
}

void display(Vector& v) {
    cout << "[";
    for (int i = 0; i < v.length; i++) {
        cout << v.data[i];
        if (i < v.length - 1) cout << ", ";
    }
    cout << "]\n";
}

void clear(Vector& v) {
    delete[] v.data;
    v.data = nullptr;
    v.capacity = 0;
    v.length = 0;
}

int main() {
    Vector v;
    init(v);
    push_back(v, 10);
    push_back(v, 20);
    push_back(v, 30);

    cout << "Isi vector: ";
    display(v);

    cout << "Elemen ke-1 = " << get(v, 1) << endl;

    set(v, 1, 99);
    cout << "Set elemen ke-1 jadi 99: ";
    display(v);

    pop_back(v);
    cout << "Setelah pop_back: ";
    display(v);

    cout << "Ukuran vector sekarang = " << size(v) << endl;
```

```
    clear(v);
    return 0;
}
```

Vector.cpp

1.4 Tugas Akhir

1. Buatlah program yang mengimplementasikan percobaan – percobaan diatas. Upload kode dari tugas tersebut ke akun Github masing-masing.

II PERCOBAAN 2: SORTING

3.1 Tujuan Percobaan

1. Dapat memahami fungsi dan jenis jenis Sorting
2. Dapat menggunakan berbagai jenis sorting

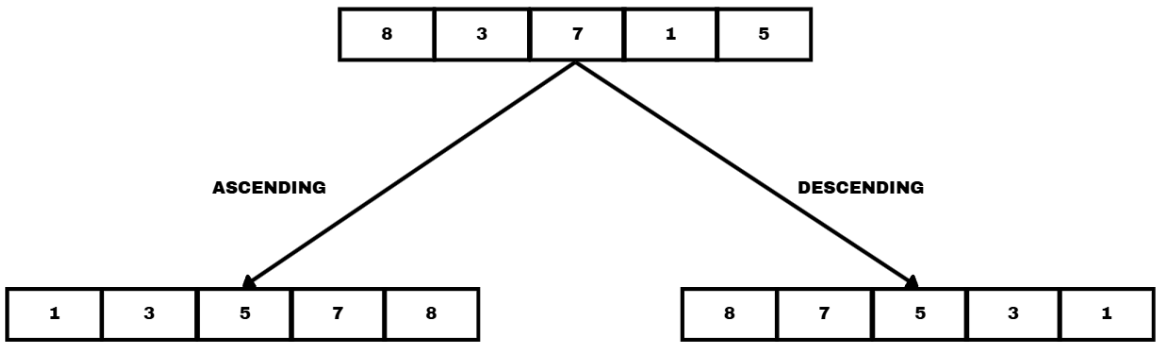
2.2 Tinjauan Pustaka

2.2.1 Pengertian Sorting

Dalam kehidupan sehari-hari, kegiatan mengurutkan sesuatu merupakan hal yang sangat umum dilakukan. Contohnya, saat upacara, peserta disusun dari yang paling pendek di barisan depan hingga yang paling tinggi di bagian belakang. Begitu pula dalam penyimpanan dokumen di filling cabinet yang diurutkan berdasarkan waktu pembuatan, atau penataan kartu kehadiran siswa yang disusun sesuai urutan abjad nama. Semua aktivitas tersebut pada dasarnya memiliki konsep yang sama dengan proses sorting dalam pemrograman.

Sorting adalah proses pengurutan sekumpulan data yang awalnya acak atau tidak beraturan menjadi teratur berdasarkan aturan tertentu. Pengurutan data umumnya dibedakan menjadi dua jenis, yaitu:

1. Ascending, yaitu pengurutan dari data terkecil ke data terbesar.
2. Descending, yaitu pengurutan dari data terbesar ke data terkecil.



Gambar 2.1 Sorting Ascending dan Descending

Dalam berbagai kasus pemrograman, seperti pada aplikasi kamus elektronik, buku telepon, maupun sistem informasi perpustakaan, pengurutan data menjadi hal yang penting untuk mempermudah proses pencarian. Pengurutan dapat dilakukan dengan dua cara, yaitu melalui proses *sorting* setelah data dimasukkan, atau dengan menerapkan mekanisme penyimpanan data secara terurut sejak awal saat data diinput ke dalam program.

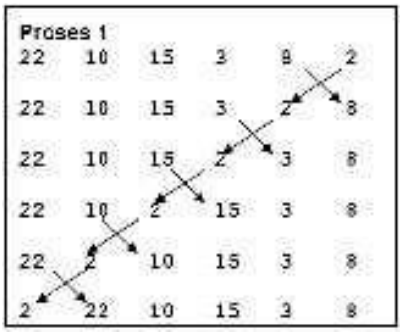
Kondisi ini menunjukkan bahwa metode pengurutan (*sorting*) memiliki peran yang sangat penting dan sering digunakan dalam pengembangan aplikasi. Oleh karena itu, pemahaman mengenai teknik pengurutan data menjadi salah satu bagian mendasar yang harus dikuasai ketika mempelajari struktur data.

2.2.2 Jenis-jenis Sorting

Terdapat banyak metode pengurutan data, namun pada modul ini kita akan membahas 4 jenis sorting yaitu :

1. Bubble Sort

Bubble Sort adalah salah satu algoritma pengurutan yang paling sederhana. Algoritma ini bekerja dengan cara menelusuri data secara berulang dari elemen pertama hingga elemen terakhir, kemudian membandingkan setiap pasangan elemen yang bersebelahan. Jika pasangan elemen tersebut tidak sesuai dengan urutan yang diinginkan, maka kedua elemen akan ditukar posisinya. Proses ini terus dilakukan hingga seluruh data berada dalam urutan yang benar.



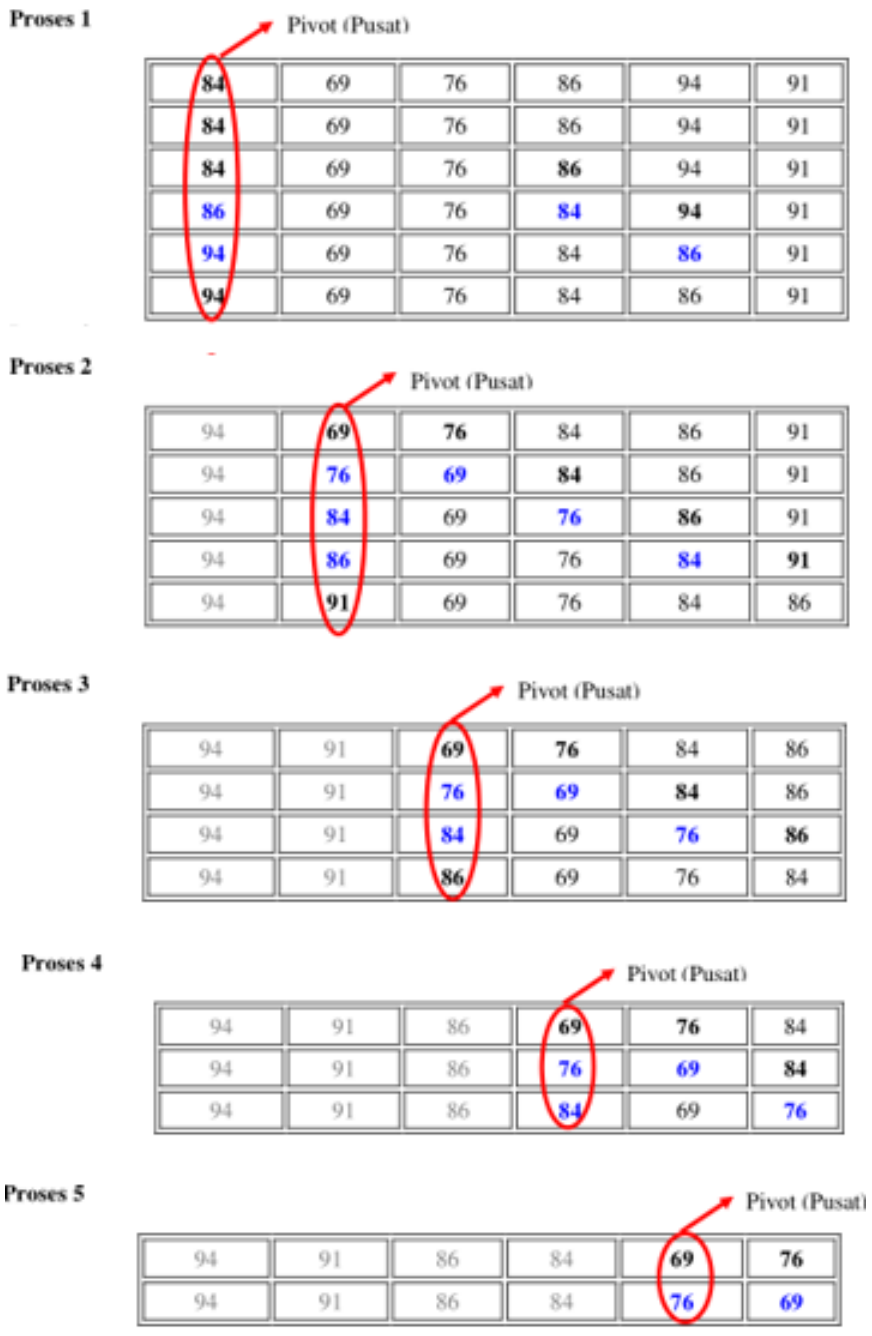
Gambar 2.2 Visualisasi Bubble Sort

Sebagai ilustrasi, misalkan terdapat data array {9, 6, 7, 3, 5} yang akan diurutkan dari nilai terkecil ke terbesar. Proses bubble sort dimulai dari elemen pertama pada index[0], yaitu 9, yang dibandingkan dengan elemen pada index[1], yaitu 6. Karena 9 lebih besar daripada 6, maka keduanya ditukar sehingga array berubah menjadi {6, 9, 7, 3, 5}. Selanjutnya, nilai 9 pada

index[1] dibandingkan dengan index[2] yaitu 7, dan karena 9 lebih besar, kembali dilakukan pertukaran sehingga array menjadi {6, 7, 9, 3, 5}. Proses ini terus dilanjutkan dengan membandingkan elemen secara berurutan dan menukar posisi jika elemen sebelumnya lebih besar dari elemen setelahnya. Setelah seluruh perulangan selesai dilakukan, data array akhirnya tersusun rapi menjadi {3, 5, 6, 7, 9}.

2. Exchange Sort

Exchange Sort memiliki konsep yang hampir sama dengan Bubble Sort, hanya saja cara perbandingan antar elemennya berbeda. Pada Exchange Sort, sebuah elemen akan dibandingkan langsung dengan semua elemen lainnya di dalam array, lalu dilakukan pertukaran apabila ditemukan nilai yang lebih kecil atau lebih besar sesuai kebutuhan. Dengan kata lain, selalu ada satu elemen yang berperan sebagai titik acuan (pivot) selama proses perbandingan. Sementara itu, Bubble Sort bekerja dengan cara membandingkan elemen yang bersebelahan, dimulai dari elemen pertama atau terakhir, lalu bergeser satu per satu sehingga elemen berikutnya menjadi acuan baru untuk dibandingkan dengan elemen di sebelahnya hingga seluruh data terurut.



Gambar 2.3 Visualisasi Exchanged Sort

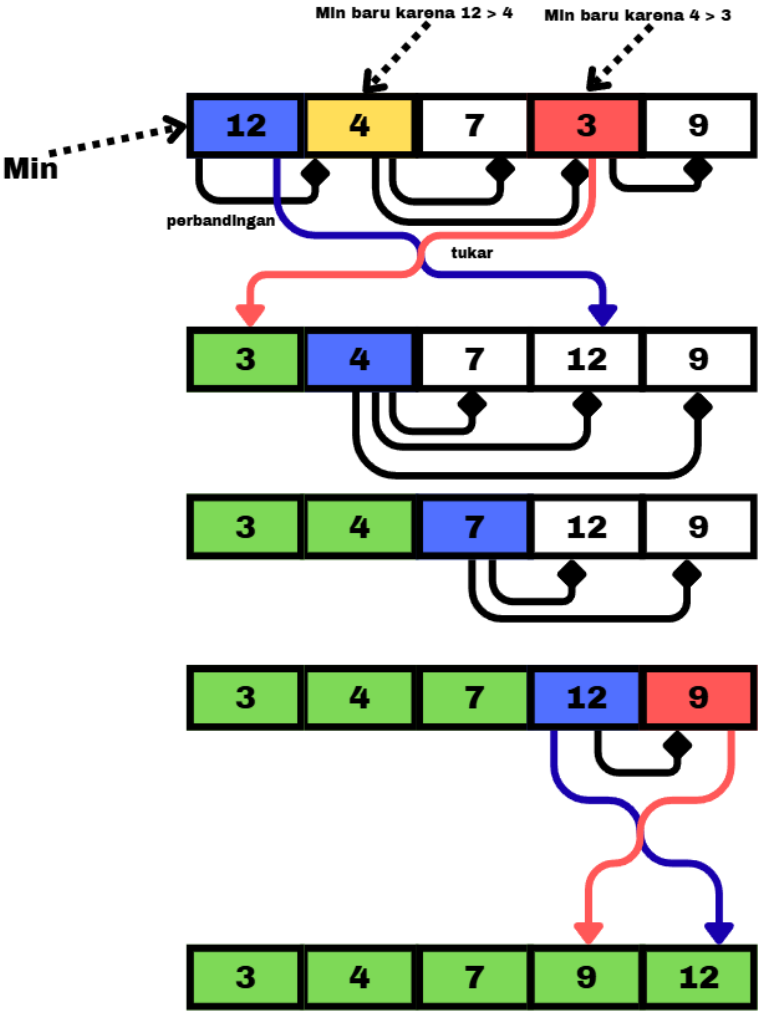
3. Selection Sort

Selection Sort merupakan salah satu algoritma pengurutan sederhana yang bekerja dengan cara mencari nilai terkecil atau terbesar dari kumpulan data pada setiap perulangan, kemudian menempatkan nilai tersebut pada posisi yang sesuai. Proses ini diulangi hingga seluruh elemen data berada dalam urutan yang benar. Analogi dalam kehidupan sehari-hari dapat dilihat pada penyusunan barisan peserta dengan tinggi badan yang harus diurutkan. Pertama, dicari peserta dengan tinggi badan paling rendah untuk ditempatkan di posisi awal barisan. Selanjutnya, dari peserta yang tersisa, kembali dicari yang paling rendah untuk ditempatkan di posisi kedua. Proses ini terus dilanjutkan hingga akhirnya peserta dengan tinggi badan paling tinggi berada di posisi terakhir.

Sebagai contoh pada pengurutan array {12, 4, 7, 3, 9} dengan algoritma Selection Sort, langkah pertama adalah menentukan elemen pertama yaitu 12 sebagai nilai minimum (min). Kemudian bandingkan 12 dengan 4, karena 4 < 12 maka set nilai min menjadi 4. Selanjutnya bandingkan 4 dengan 7, karena 7 > 4 maka nilai min tetap 4. Lalu bandingkan 4 dengan 3, karena 3 < 4 maka nilai min berubah menjadi 3. Terakhir bandingkan 3 dengan 9, karena 9 > 3 maka nilai min tetap 3. Setelah perulangan pertama selesai, nilai min adalah 3, sehingga dilakukan pertukaran 3 dengan 12 sebagai elemen pertama array. Urutan data menjadi {3, 4, 7, 12, 9}.

Pada perulangan kedua, elemen 3 tidak lagi dibandingkan. Maka ditetapkan 4 sebagai nilai min. Dibandingkan dengan 7 (7 > 4 sehingga tetap 4), lalu dengan 12 (12 > 4 tetap 4), kemudian dengan 9 (9 > 4 tetap 4). Karena nilai min tetap 4, maka tidak ada pertukaran. Urutan array masih {3, 4, 7, 12, 9}.

Pada perulangan ketiga, elemen 7 ditetapkan sebagai nilai min, dibandingkan dengan 12 (12 > 7 tetap 7), lalu dengan 9 (9 > 7 tetap 7). Karena tidak ada nilai yang lebih kecil, tidak ada pertukaran. Urutan tetap {3, 4, 7, 12, 9}.
Pada perulangan keempat, elemen 12 ditetapkan sebagai nilai min, lalu dibandingkan dengan 9. Karena 9 < 12, maka nilai min berubah menjadi 9 dan dilakukan pertukaran antara 12 dan 9. Hasil akhirnya adalah array terurut {3, 4, 7, 9, 12}.



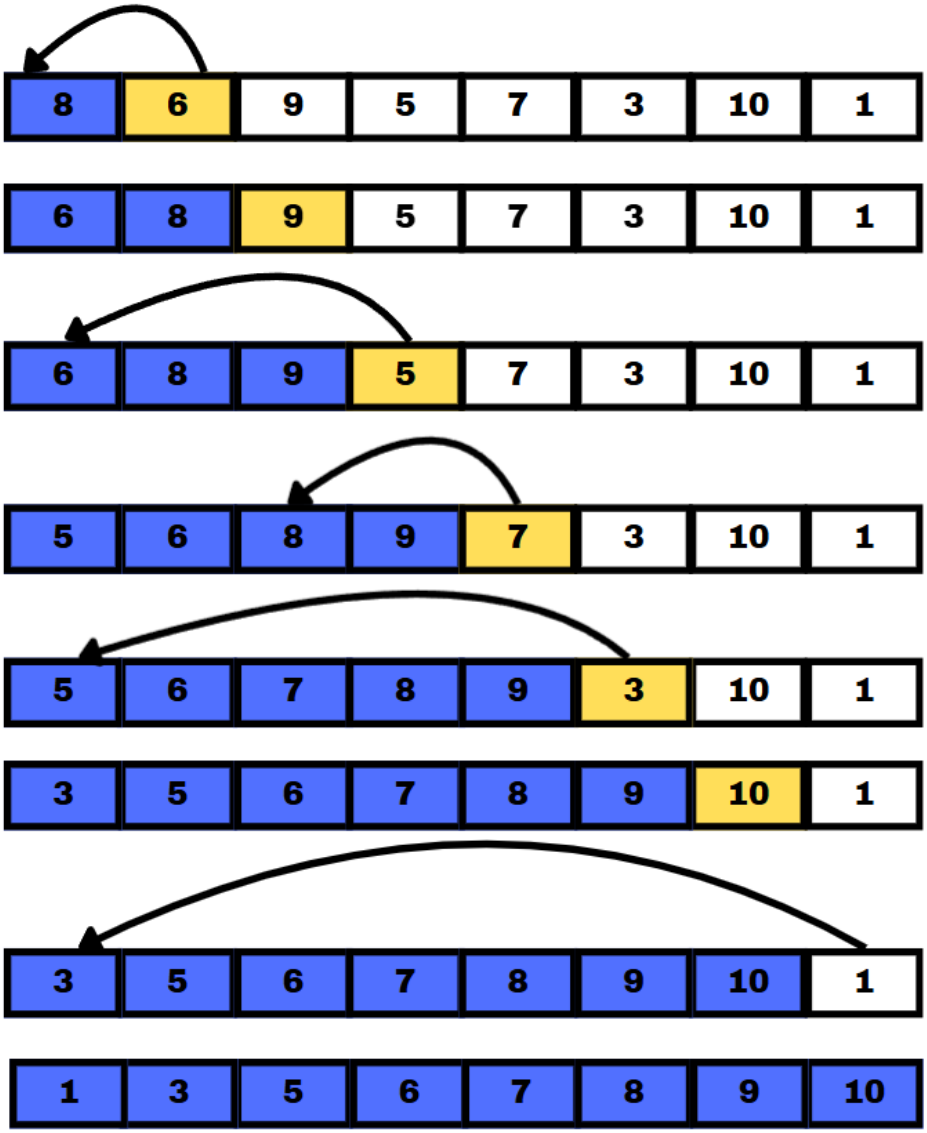
Gambar 2.4 Visualisasi Exchange Sort

4. Insertion Sort

Insertion Sort merupakan salah satu algoritma pengurutan sederhana yang bekerja dengan cara memulai proses dari elemen kedua pada array. Elemen ini kemudian dibandingkan dengan elemen sebelumnya (elemen pertama). Jika elemen kedua lebih kecil, maka posisinya ditukar dengan elemen pertama. Tahapan yang sama dilakukan secara berulang pada elemen-elemen berikutnya, dengan cara menyisipkan elemen tersebut ke posisi yang sesuai di dalam urutan yang sudah terbentuk, hingga seluruh elemen dalam array tersusun rapi.

Analogi dalam kehidupan sehari-hari dapat digambarkan melalui penyusunan kartu permainan. Misalnya ketika kita ingin mengurutkan kartu bridge. Semua kartu awalnya berserakan di atas meja. Kita mengambil kartu pertama dan meletakkannya sebagai acuan. Saat mengambil kartu

berikutnya, kita meletakkannya pada posisi yang sesuai, apakah di depan kartu pertama, di antara kartu yang sudah tersusun, atau di bagian akhir tumpukan. Proses ini diulangi hingga seluruh kartu berada dalam urutan yang benar, mulai dari As, angka 1 hingga 10, lalu Jack, Queen, dan King.



Gambar 2.5 Ilustrasi Insertion Sort

2.3 Percobaan

2.3.1 Percobaan II-1: Bubble Sort

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

void tukar(int *x, int *y);

int main() {
    int n;
    int arr[1005];

    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                tukar(&arr[j], &arr[j + 1]);
            }
        }
    }

    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}

void tukar(int *x, int *y) {
    int z = *x;
    *x = *y;
    *y = z;
}
```

Bubblesort.cpp

2.3.2 Percobaan II-2: Exchange Sort

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

void tukar(int *x, int *y);
```

```cpp
int main() {
    int n;
    int arr[1005];

    cout << "Masukkan jumlah elemen: ";
    cin >> n;

    cout << "Masukkan elemen array: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (arr[i] > arr[j]) {
                tukar(&arr[i], &arr[j]);
            }
        }
    }

    cout << "Array setelah diurutkan: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}

void tukar(int *x, int *y) {
    int z = *x;
    *x = *y;
    *y = z;
}
```

ExchangeSort.cpp

2.3.3 Percobaan II-3: Insertion Sort

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

int main() {
    int n;
    int arr[1005];

    cout << "Masukkan jumlah elemen: ";
    cin >> n;

    cout << "Masukkan elemen array: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    for (int i = 1; i < n; i++) {
        int temp = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > temp) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = temp;
    }

    cout << "Array setelah diurutkan (Insertion Sort): ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

InsertionSort.cpp

2.3.4 Percobaan II-4: Selection Sort

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

void tukar(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main() {
    int n;
    int arr[1005];

    cout << "Masukkan jumlah elemen: ";
    cin >> n;

    cout << "Masukkan elemen array: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    for (int i = 0; i < n - 1; i++) {
        int pos = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[pos]) {
                pos = j;
            }
        }
        if (pos != i) {
            tukar(&arr[i], &arr[pos]);
        }
    }

    cout << "Array setelah diurutkan (Selection Sort): ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

SelectionSort.cpp

2.4 Tugas Akhir

1. Buatlah program yang mengimplementasikan percobaan – percobaan diatas. Upload kode dari tugas tersebut ke akun Github masing-masing.

III PERCOBAAN 3: SEARCHING

3.1 Tujuan Percobaan

1. Dapat memahami fungsi dan jenis jenis searching
2. Dapat menggunakan berbagai jenis searching

3.2 Tinjauan Pustaka

3.2.1 Pengertian Searching

Searching atau pencarian dalam struktur data adalah proses algoritmik yang digunakan untuk memeriksa, menemukan, dan menentukan posisi suatu elemen (target) dari sekumpulan data. Elemen-elemen tersebut dapat tersimpan dalam berbagai struktur data, seperti array, list, linked list, tree, dan lain sebagainya.

Berdasarkan operasinya, terdapat dua algoritma pencarian yang umum digunakan, yaitu Sequential Search dan Binary Search. Pencarian merupakan proses yang sangat penting dalam pemrograman, sebab data yang telah disimpan dalam media penyimpanan hampir pasti akan dicari kembali untuk berbagai kebutuhan, seperti pembuatan laporan, proses pembaruan (*update*), perbaikan (*correction*), maupun penghapusan (*delete*).

Dalam pelaksanaan pencarian, dibutuhkan input berupa data yang akan dicari, yang disebut sebagai kunci pencarian. Kunci ini harus memiliki tipe data yang sama dengan field atau atribut tempat data tersebut disimpan. Hasil dari proses pencarian adalah indeks atau posisi data dalam struktur. Dari pencarian tersebut, terdapat dua kemungkinan hasil: pertama, data ditemukan dan indeksnya dapat diketahui; atau kedua, data tidak ditemukan dalam struktur data yang bersangkutan.

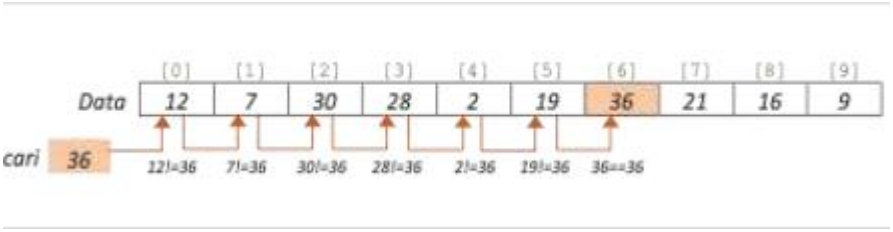
3.2.2 Jenis-jenis Searching

1.Sequential Searching

Sequential Search adalah algoritma pencarian paling sederhana yang menggunakan metode *brute force*, yaitu dengan memeriksa setiap elemen secara berurutan hingga data yang dicari ditemukan. Sebagai contoh, misalkan kita ingin mencari nilai **25** dari data array {40, 12, 18, 7, 25, 33, 50, 2, 15, 9}. Proses pencarian dimulai dengan membandingkan 25 dengan elemen

pertama pada index[0] yaitu 40. Karena tidak sama, pencarian dilanjutkan ke elemen berikutnya: index[1] = 12, index[2] = 18, index[3] = 7, hingga akhirnya ditemukan kecocokan pada index[4] dengan kondisi 25 == 25.

Kelebihan algoritma Sequential Search adalah data tidak harus dalam keadaan terurut, sehingga dapat digunakan pada kumpulan data yang acak. Namun, kelemahan utamanya adalah efisiensi waktu pencarian. Jika data yang dicari berada di akhir array, maka seluruh elemen harus diperiksa terlebih dahulu (*worst case*). Sebaliknya, jika data berada di awal array, maka pencarian dapat selesai dengan cepat (*best case*). Selain itu, semakin banyak jumlah data, semakin lama pula waktu yang dibutuhkan karena setiap elemen harus dicek satu per satu.



Gambar 3.1 Sequential Searching

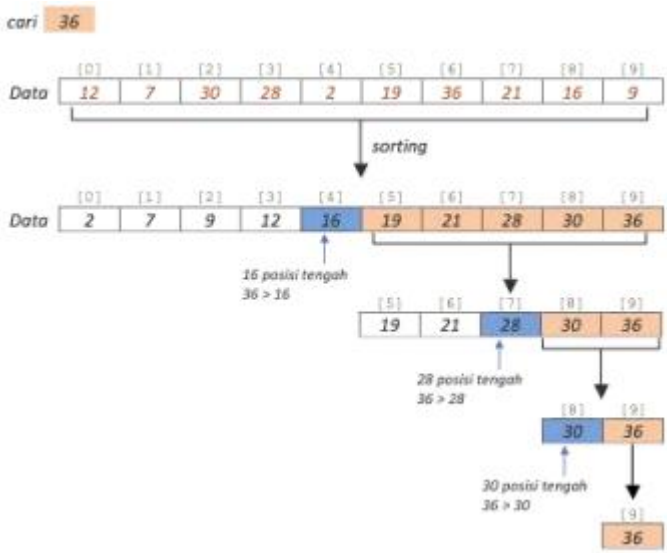
2.Binary Searching

Binary Search adalah algoritma pencarian yang memiliki syarat utama yaitu data harus dalam keadaan terurut terlebih dahulu. Proses pencarian dimulai dengan menentukan indeks tengah dari array. Nilai pada elemen tengah tersebut kemudian dibandingkan dengan kunci pencarian (data yang dicari).

Jika kunci pencarian lebih besar daripada nilai elemen tengah, maka pencarian dilanjutkan hanya pada bagian kanan array, sementara bagian kiri diabaikan. Sebaliknya, jika kunci pencarian lebih kecil daripada nilai elemen tengah, maka pencarian dilanjutkan hanya pada bagian kiri array, dan bagian kanan diabaikan. Proses ini terus diulangi dengan mencari elemen tengah dari bagian array yang tersisa, lalu membandingkannya kembali dengan kunci pencarian.

Langkah-langkah ini dilakukan secara berulang hingga ditemukan elemen yang sesuai dengan kunci pencarian atau hingga semua elemen telah diperiksa. Dengan cara ini, Binary Search jauh

lebih efisien dibandingkan Sequential Search, karena setiap langkah dapat mengeliminasi setengah dari jumlah data yang masih tersisa.



Gambar 3.2 Binary Searching

Misalkan tersedia array terurut [5, 8, 12, 20, 25, 30, 40, 50] dan kita ingin mencari angka 25 di dalamnya. Proses pencarian dimulai dengan menentukan indeks awal 0 dan indeks akhir 7. Dari sini diperoleh indeks tengah yaitu 3, dengan nilai 20. Karena nilai 25 lebih besar daripada 20, maka pencarian tidak lagi dilakukan pada bagian kiri array, melainkan berlanjut ke bagian kanan.

Selanjutnya, indeks awal bergeser menjadi 4 dan indeks akhir tetap 7 sehingga indeks tengah yang baru adalah 5 dengan nilai 30. Nilai ini dibandingkan dengan 25, dan karena 25 lebih kecil daripada 30, maka pencarian dialihkan ke sisi kiri dari bagian ini. Pada tahap berikutnya, indeks awal dan indeks akhir sama-sama berada pada posisi 4 sehingga indeks tengah bernilai 4 dengan nilai 25. Karena nilai ini sama dengan kunci pencarian, maka data berhasil ditemukan pada indeks ke-4.

3.3 Percobaan

3.3.1 Percobaan III-1: Sequential Searching

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

int main() {
    const int x = 15;
    int data[x] = {3,8,1,9,0,6,7,5,2,2,1,6,2,5,12};
    int target;
    int i = 0;
    int counter = 0;

    cout << "Masukkan angka yang ingin dicari : ";
    cin >> target;

    while (i < x) {
        if (data[i] == target) {
            counter++;
        }
        i++;
    }

    if (counter > 0) {
        cout << "Angka " << target << " ditemukan sebanyak " << counter << "
kali." << endl;
    }
    else {
        cout << "Angka " << target << " tidak ditemukan." << endl;
    }

    return 0;
}
```

SequentialSearching.cpp

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

int main() {
    const int x = 11;
    int data[x] = {3, 8, 1, 9, 0, 6, 7, 5, 12, 2};
    int target;
    int i = 0;
    int flag = 0;
```

```
    cout << "Masukkan angka yang ingin dicari : ";
    cin >> target;

    data[x] = target;

    while (data[i] != target) {
        i++;
    }

    if (i < x) {
        flag = 1;
    }

    if (flag == 1) {
        cout << "Ditemukan pada indeks ke-" << i << endl;
    } else {
        cout << "Tidak ditemukan" << endl;
    }

    return 0;
}
```

SequentialSearchingSentinel.cpp

3.3.2 Percobaan III-2: Binary Searching

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

int main() {
    int n;
    int target;
    int arr[1005];
    cout << "Masukkan jumlah elemen: ";
    cin >> n;
    cout << "Masukkan elemen (urut menaik): ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }
    cout << "Masukkan angka yang ingin dicari: ";
    cin >> target;

    int l = 0, r = n - 1, pos = -1;
    while (l <= r) {
        int m = l + (r - l) / 2;
        cout << "median :" << m << endl;
```

```
        if (arr[m] == target) {
            pos = m;
            break;
        }
        else if (arr[m] < target) {
            cout << "Mencari di kiri" << endl;
            l = m + 1;
        }
        else {
            cout << "Mencari di kanan" << endl;
            r = m - 1;
        }
    }
    if (pos != -1) {
        cout << "Ditemukan pada pencarian ke-" << pos+1 << endl;
    }
    else {
        cout << "Tidak ditemukan" << endl;
    }
    return 0;
}
```

BinarySearching.cpp

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

int main(){
const int n = 12;
int data[n] = {5, 12, 19, 23, 31, 37, 45, 52, 68, 74, 89, 95};
int low = 0;
int high = n - 1;
int pos;
int target;
cout << "Masukkan nilai yang ingin dicari" << endl;
cin >> target;

while(target > data[low] && target <= data[high]) {
        pos = (target-data[low])/(data[high]-data[low]) * (high-low) + low;
cout << pos << endl;
        if(target > data[pos]) {
            low = pos+1;
        }
        else if (target < data[pos]) {
            high = pos-1;
        }
```

```
        else {
        low = pos;
        }
    }

    if(target == data[low]) {
        cout << "Ketemu pada pencarian ke " << low+1 << endl;
    }
    else {
        cout << "Tidak ketemu" << endl; return 0;
    }
}
```

BinarySearchingInterpolation.cpp

3.4 Tugas Akhir

1. Buatlah program yang mengimplementasikan percobaan – percobaan diatas. Upload kode dari tugas tersebut ke akun Github masing-masing.

IV PERCOBAAN 4: STACK AND QUEUE

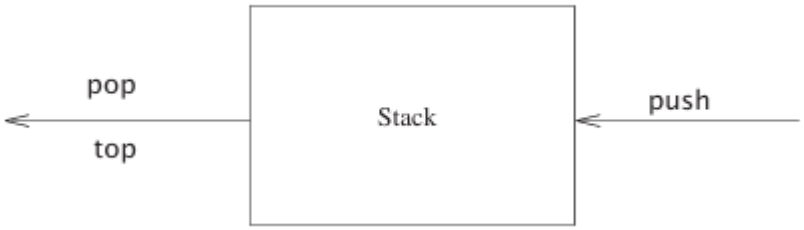
4.1 Tujuan Percobaan

1. Dapat membuat dan memahami stack
2. Dapat membuat dan memahami queue

4.2 Tinjauan Pustaka

4.2.1 Stack

Stack adalah struktur data linier yang menggunakan prinsip Last In First Out, dimana elemen yang terakhir dimasukkan ke dalam stack juga menjadi elemen pertama kali yang akan dibuang dari stack. Misalkan terdapat sebuah tumpukan piring di kantin. Piring pertama yang diletakkan di meja adalah piring A, kemudian di atasnya ditumpuk lagi piring B, disusul piring C, piring D, piring E, dan terakhir piring F. Jika seseorang ingin mengambil piring dari tumpukan tersebut, maka piring yang paling atas, yaitu piring F, harus diambil terlebih dahulu, kemudian piring E, piring D, piring C, piring B, dan terakhir piring A. Apabila kita ingin langsung mengambil piring C tanpa memindahkan piring-piring di atasnya, maka tumpukan akan berantakan.

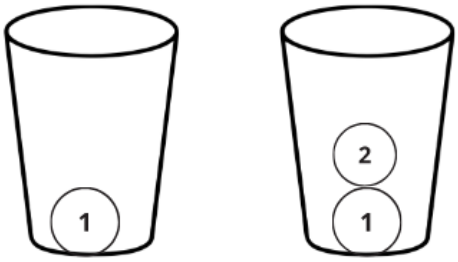


Gambar 4.1 Visualisasi Stack

a. Operasi dalam stack

1. Push

Operasi push adalah operasi dalam stack yang digunakan untuk menambah elemen di depan stack. Perhatikan contoh pada gambar dibawah :



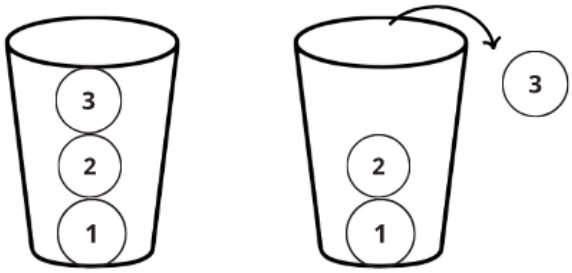
Gambar 4.2 Push

Sebelum dilakukan operasi *push*, misalkan elemen teratas dari Stack adalah nilai 30. Ketika sebuah elemen baru dengan nilai 45 dimasukkan ke dalam Stack melalui operasi *push*, maka elemen tersebut ditempatkan di atas nilai 30. Dengan demikian, setelah operasi ini, elemen teratas Stack berubah menjadi 45.Proses *push* dalam Stack melibatkan beberapa tahapan, yaitu menerima data yang akan disimpan, menaikkan penunjuk (*pointer*) Stack yang menunjukkan posisi atas (*top*), kemudian menyimpan data baru tersebut di posisi atas.

Namun, dalam praktiknya, operasi *push* berpotensi menimbulkan masalah Stack Overflow, yaitu kondisi ketika kapasitas Stack sudah penuh sehingga tidak ada lagi ruang untuk menambahkan elemen baru. Dalam pemrograman, kondisi ini perlu ditangani dengan baik, misalnya dengan memberikan pesan kesalahan atau mekanisme penanganan khusus agar program tetap berjalan sebagaimana mestinya.

2. Pop

Pop adalah kebalikan dari push. Pop adalah operai yang berguna untuk mengambil satu elemen dari stack. Perhatikan ilustrasi pop dibawah :



Gambar 4.3 Pop

Aktivitas dalam operasi pop adalah mengakses sekaligus menghapus elemen teratas dari Stack. Setelah elemen tersebut dihapus, pointer Stack diturunkan sehingga menunjuk ke posisi baru sebagai elemen teratas.

Dalam praktiknya, operasi pop bisa saja dilakukan ketika Stack dalam keadaan kosong. Kondisi ini akan menimbulkan kesalahan yang dikenal dengan istilah Stack Underflow, yaitu situasi ketika tidak ada elemen yang dapat dihapus. Oleh karena itu, sebelum melakukan operasi pop, sangat penting bagi pemrogram untuk melakukan pengecekan apakah Stack masih memiliki elemen atau sudah kosong, agar program dapat berjalan dengan benar tanpa menimbulkan error.

3. Operasi Full

Operasi Full digunakan untuk memeriksa apakah Stack sudah penuh atau belum. Suatu Stack dianggap penuh apabila posisi penunjuk teratas (S.Top) sama dengan kapasitas maksimum Stack (MaxS). Jika kondisi ini terpenuhi, maka fungsi akan mengembalikan nilai true. Sebaliknya, jika S.Top belum mencapai MaxS, maka fungsi mengembalikan nilai false.

4. Operasi Empty

Operasi Empty digunakan untuk memeriksa apakah Stack dalam keadaan kosong. Stack dinyatakan kosong apabila nilai S.Top sama dengan 0. Jika kondisi ini terpenuhi, fungsi akan

mengembalikan nilai true. Namun jika S.Top tidak sama dengan 0, artinya masih ada elemen dalam Stack, maka fungsi akan mengembalikan nilai false.

5. Operasi Clear

Operasi Clear berfungsi untuk mengosongkan seluruh elemen dalam Stack. Setelah operasi ini dilakukan, Stack akan berada dalam kondisi kosong dengan penunjuk Top kembali bernilai 0.

4.2.2 Queue

Setelah membahas Stack, abstraksi data sederhana berikutnya adalah Queue. Sama seperti Stack, konsep Queue dapat dengan mudah dijumpai dalam kehidupan sehari-hari. Sebagai contoh, dapat diilustrasikan dengan barisan orang yang menunggu giliran masuk ke sebuah teater.

Perbedaan mendasar antara Queue dan Stack terletak pada sifat akses datanya. Queue terbuka pada kedua ujungnya, di mana elemen baru selalu disisipkan pada satu sisi (bagian belakang) dan elemen dihapus dari sisi lainnya (bagian depan). Dengan cara ini, urutan elemen yang keluar akan sama dengan urutan saat elemen tersebut dimasukkan. Jika Stack bekerja dengan prinsip LIFO (Last In, First Out), maka Queue mengikuti prinsip FIFO (First In, First Out). Hal ini berarti data yang pertama kali masuk ke dalam Queue akan menjadi data yang pertama kali diproses, mendahului elemen-elemen lain yang datang setelahnya.

Contoh nyata dari Queue dalam kehidupan sehari-hari dapat dilihat pada jalur lalu lintas satu arah, di mana kendaraan yang lebih dahulu masuk akan keluar lebih dahulu pula. Contoh lainnya adalah barisan pembayaran di loket tiket pesawat maupun tiket kapal laut. Konsep ini juga banyak diterapkan pada berbagai sistem, baik konvensional maupun berbasis komputer, untuk membantu pengaturan proses agar lebih terstruktur serta memudahkan pengawasan, perbaikan, dan evaluasi. Karena alasan inilah Queue digunakan secara luas pada sistem dengan tingkat kompleksitas rendah hingga tinggi.

Dalam konteks struktur data, Queue merupakan sebuah objek atau lebih spesifik dikenal sebagai Abstract Data Type (ADT) yang mendukung beberapa operasi dasar. Operasi-operasi tersebut antara lain:

1. EnQueue: operasi untuk menambahkan elemen ke bagian belakang Queue. Artinya, setiap elemen baru yang masuk harus ditempatkan di posisi akhir.
2. DeQueue: operasi untuk menghapus elemen dari bagian depan Queue. Elemen yang masuk lebih dahulu akan dikeluarkan lebih dahulu sesuai prinsip FIFO.
3. IsEmpty: operasi untuk memeriksa apakah Queue dalam keadaan kosong, yaitu tidak ada elemen yang tersimpan.
4. IsFull: operasi untuk memeriksa apakah Queue sudah penuh, yaitu tidak dapat lagi menampung elemen baru.
5. Peek: operasi untuk melihat nilai elemen pada bagian depan Queue tanpa menghapusnya.

4.3 Percobaan

4.3.1 Percobaan IV-1: Stacked

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

const int MAX = 100;
int st[MAX];
int topIdx = -1;
bool isEmpty();
bool isFull();
void push(int x);
void pop();
void display();
void peek();

int main() {
    int pilih, val;
    do {
        cout << "\n=== STACK (Array) ===\n";
        cout << "1. Push\n2. Pop\n3. Peek\n4. Tampilkan\n5. Keluar\n";
        cout << "Pilih: ";
        cin >> pilih;
        if (pilih == 1) { cout << "Nilai: "; cin >> val; push(val); }
        else if (pilih == 2) pop();
        else if (pilih == 3) peek();
        else if (pilih == 4) display();
    } while (pilih != 5);
    return 0;
```

```
}

bool isEmpty() {
    return topIdx == -1;
}
bool isFull()  {
    return topIdx == MAX - 1;
}

void push(int x) {
    if (isFull()) {
        cout << "Stack penuh\n";
        return;
    }
    st[++topIdx] = x;
    cout << "Push " << x << " berhasil\n";
}

void pop() {
    if (isEmpty()) {
        cout << "Stack kosong\n";
        return;
    }
    cout << "Pop " << st[topIdx--] << " berhasil\n";
}

void peek() {
    if (isEmpty()) {
        cout << "Stack kosong\n";
        return;
    }
    cout << "Elemen teratas: " << st[topIdx] << '\n';
}
void display() {
    if (isEmpty()) {
        cout << "Stack kosong\n";
        return;
    }
    cout << "Isi stack (atas ke bawah): ";
    for (int i = topIdx; i >= 0; --i) {
        cout << st[i] << " ";
    }
    cout << '\n';
}
```

StackArray.cpp

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

struct Node {
    int data;
    Node* next;
};
Node* topPtr = nullptr;

bool isEmpty();
void push (int x);
void pop();
void peek();
void display();

int main() {
    int pilih, val;
    do {
        cout << "\n=== STACK (Linked List) ===\n";
        cout << "1. Push\n2. Pop\n3. Peek\n4. Tampilkan\n5. Keluar\n";
        cout << "Pilih: ";
        cin >> pilih;
        if (pilih == 1) { cout << "Nilai: "; cin >> val; push(val); }
        else if (pilih == 2) pop();
        else if (pilih == 3) peek();
        else if (pilih == 4) display();
    } while (pilih != 5);

    while (!isEmpty()) pop();
    return 0;
}

bool isEmpty() { return topPtr == nullptr; }

void push(int x) {
    Node* n = new Node{x, topPtr};
    topPtr = n;
    cout << "Push " << x << " berhasil\n";
}

void pop() {
    if (isEmpty()) {
        cout << "Stack kosong\n";
        return;
    }
```

```
    Node* temp = topPtr;
    cout << "Pop " << temp->data << " berhasil\n";
    topPtr = topPtr->next;
    delete temp;
}

void peek() {
    if (isEmpty()) {
        cout << "Stack kosong\n";
        return;
    }
    cout << "Elemen teratas: " << topPtr->data << '\n';
}

void display() {
    if (isEmpty()) {
        cout << "Stack kosong\n";
        return;
    }
    cout << "Isi stack (atas ke bawah): ";
    for (Node* cur = topPtr; cur != nullptr; cur = cur->next) cout << cur-
>data << " ";
    cout << '\n';
}
```

StackLinkedList.cpp

4.3.2 Percobaan IV-2: Queue

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

const int MAXN = 100;
int q[MAXN];
int frontIdx = -1, rearIdx = -1;

bool isEmpty() { return frontIdx == -1; }
bool isFull() { return (rearIdx + 1) % MAXN == frontIdx; }

void enqueue(int x) {
    if (isFull()) {
        cout << "Queue penuh\n";
        return;
    }
    if (isEmpty()) {
        frontIdx = rearIdx = 0;
    } else {
```

```
        rearIdx = (rearIdx + 1) % MAXN;
    }
    q[rearIdx] = x;
    cout << "Enqueue " << x << " berhasil\n";
}

void dequeue() {
    if (isEmpty()) {
        cout << "Queue kosong\n";
        return;
    }
    cout << "Dequeue " << q[frontIdx] << " berhasil\n";
    if (frontIdx == rearIdx) {
        frontIdx = rearIdx = -1;
    } else {
        frontIdx = (frontIdx + 1) % MAXN;
    }
}

void peek() {
    if (isEmpty()) {
        cout << "Queue kosong\n";
        return;
    }
    cout << "Elemen depan: " << q[frontIdx] << '\n';
}

void display() {
    if (isEmpty()) {
        cout << "Queue kosong\n";
        return;
    }
    cout << "Isi queue (depan ke belakang): ";
    int i = frontIdx;
    while (true) {
        cout << q[i] << " ";
        if (i == rearIdx) break;
        i = (i + 1) % MAXN;
    }
    cout << '\n';
}

int main() {
    int pilih, val;
    do {
```

```
        cout << "\n=== QUEUE (Array) ===\n";
        cout << "1. Enqueue\n2. Dequeue\n3. Peek\n4. Tampilkan\n5. Keluar\n";
        cout << "Pilih: ";
        cin >> pilih;
        if (pilih == 1) { cout << "Nilai: "; cin >> val; enqueue(val); }
        else if (pilih == 2) dequeue();
        else if (pilih == 3) peek();
        else if (pilih == 4) display();
    } while (pilih != 5);
    return 0;
}
```

QueueArray.cpp

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

struct Node {
    int data;
    Node* next;
};
Node* frontPtr = nullptr;
Node* rearPtr = nullptr;

bool isEmpty() { return frontPtr == nullptr; }

void enqueue(int x) {
    Node* n = new Node{x, nullptr};
    if (isEmpty()) {
        frontPtr = rearPtr = n;
    } else {
        rearPtr->next = n;
        rearPtr = n;
    }
    cout << "Enqueue " << x << " berhasil\n";
}

void dequeue() {
    if (isEmpty()) {
        cout << "Queue kosong\n";
        return;
    }
    Node* temp = frontPtr;
    cout << "Dequeue " << temp->data << " berhasil\n";
    frontPtr = frontPtr->next;
    if (frontPtr == nullptr) rearPtr = nullptr;
```

```
    delete temp;
}

void peek() {
    if (isEmpty()) {
        cout << "Queue kosong\n";
        return;
    }
    cout << "Elemen depan: " << frontPtr->data << '\n';
}

void display() {
    if (isEmpty()) {
        cout << "Queue kosong\n";
        return;
    }
    cout << "Isi queue (depan ke belakang): ";
    for (Node* cur = frontPtr; cur != nullptr; cur = cur->next) {
        cout << cur->data << " ";
    }
    cout << '\n';
}

int main() {
    int pilih, val;
    do {
        cout << "\n=== QUEUE (Linked List) ===\n";
        cout << "1. Enqueue\n2. Dequeue\n3. Peek\n4. Tampilkan\n5. Keluar\n";
        cout << "Pilih: ";
        cin >> pilih;
        if (pilih == 1) { cout << "Nilai: "; cin >> val; enqueue(val); }
        else if (pilih == 2) dequeue();
        else if (pilih == 3) peek();
        else if (pilih == 4) display();
    } while (pilih != 5);

    while (!isEmpty()) dequeue();
    return 0;
}
```

QueueLinkedList.cpp

4.4 Tugas Akhir

1. Buatlah program yang mengimplementasikan percobaan – percobaan diatas. Upload kode dari tugas tersebut ke akun Github masing-masing.

V PERCOBAAN 5: BINARY SEARCH TREE

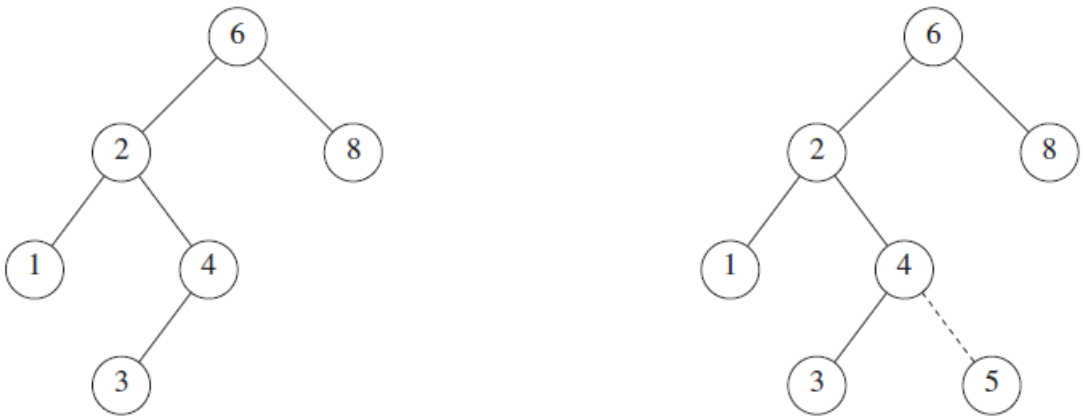
5.1 Tujuan Percobaan

1. Dapat memahami konsep binary search tree
2. Dapat menggunakan binary search tree

5.2 Tinjauan Pustaka

5.2.1 Pengertian BST

Binary Search Tree (BST) adalah salah satu bentuk struktur data pohon biner yang memiliki karakteristik khusus sehingga mampu melakukan operasi pencarian, penambahan, maupun penghapusan elemen dengan tingkat efisiensi yang tinggi. Ciri utama dari BST terletak pada aturan penempatan nilainya, yaitu setiap simpul memiliki nilai yang lebih besar dibandingkan seluruh simpul yang berada pada subpohon kiri, serta memiliki nilai yang lebih kecil dibandingkan seluruh simpul yang berada pada subpohon kanan.



Gambar 5.1 Visualisasi Binary Search Tree

Operasi pada Pohon Pencari Biner

1. Insert pada pohon percari biner

Insert pada BST dilakukan dengan memasukkan sebuah item data baru berdasarkan nilai kunci yang dimilikinya. Proses dimulai dari simpul akar. Jika pohon masih kosong, maka simpul baru tersebut akan menjadi akar. Jika pohon sudah berisi data, maka nilai kunci baru dibandingkan

dengan nilai pada simpul akar. Apabila kunci lebih kecil, simpul baru ditempatkan pada subpohon kiri; sebaliknya, jika lebih besar, simpul baru ditempatkan pada subpohon kanan. Proses perbandingan ini berlanjut secara rekursif hingga simpul baru menemukan posisi yang sesuai..

2. Searching pada pohon pencari biner

Searching pada BST bekerja dengan prinsip yang sama seperti proses penyisipan, menggunakan pendekatan rekursif. Pencarian dimulai dari simpul akar, kemudian dibandingkan dengan kunci pencarian. Jika simpul saat ini kosong, maka data tidak ditemukan. Jika nilainya sama dengan kunci pencarian, maka data berhasil ditemukan. Jika nilai kunci pencarian lebih besar daripada simpul saat ini, maka pencarian dilanjutkan ke subpohon kanan. Sebaliknya, jika lebih kecil, pencarian diarahkan ke subpohon kiri. Dengan demikian, pencarian tidak perlu dilakukan pada seluruh simpul, melainkan hanya pada jalur tertentu.

3. Deletion pada pohon pencari biner

Deletion pada BST adalah operasi penghapusan simpul yang dapat memengaruhi struktur pohon. Jika simpul yang dihapus adalah simpul daun (tidak memiliki anak), maka simpul tersebut cukup dihapus langsung. Jika simpul memiliki satu anak, maka anak tersebut akan menggantikan posisi simpul yang dihapus. Sedangkan jika simpul memiliki dua anak, maka posisinya digantikan dengan simpul pengganti, biasanya berupa simpul dengan nilai minimum dari subpohon kanan atau nilai maksimum dari subpohon kiri, sehingga sifat BST tetap terjaga.

5.2.2 Istilah Dalam BST

Prefix, infix, dan postfix merupakan tiga metode berbeda dalam menuliskan atau merepresentasikan suatu ekspresi matematika maupun logika. Jika diaplikasikan pada struktur pohon, ketiga notasi ini menunjukkan perbedaan urutan peletakan antara operator dan operand di dalam pohon.

1. Prefix

   Notasi prefix, atau dikenal juga sebagai *Polish Notation*, menempatkan operator sebelum operand. Pada struktur pohon, operator berada pada simpul akar, sedangkan operand ditempatkan pada simpul-simpul anaknya.

2. Infix

   Notasi infix merupakan bentuk yang paling umum digunakan dalam penulisan ekspresi matematika. Operator diletakkan di antara operand, namun penerapannya pada pohon memerlukan perhatian terhadap prioritas operator serta penggunaan tanda kurung untuk menentukan urutan evaluasi.

3. Postfix

   Notasi postfix, atau Reverse *Polish Notation*, menempatkan operator setelah operand. Dalam pohon, operator biasanya ditempatkan pada simpul anak paling kanan, sedangkan operand berada pada simpul-simpul anak lainnya.

5.3 Percobaan

5.3.1 Percobaan V-1: Binary Search Tree

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

struct Node {
    int key;
    Node* left;
    Node* right;
    Node(int k): key(k), left(nullptr), right(nullptr) {}
};

Node* insertNode(Node* root, int key) {
    if (!root) return new Node(key);
    if (key < root->key) root->left = insertNode(root->left, key);
    else if (key > root->key) root->right = insertNode(root->right, key);
    return root;
}

bool searchNode(Node* root, int key) {
    if (!root) return false;
    if (root->key == key) return true;
    if (key < root->key) return searchNode(root->left, key);
    return searchNode(root->right, key);
}

void inorder(Node* root) {
    if (!root) return;
    inorder(root->left);
    cout << root->key << " ";
    inorder(root->right);
}

void preorder(Node* root) {
    if (!root) return;
    cout << root->key << " ";
    preorder(root->left);
    preorder(root->right);
}

void postorder(Node* root) {
    if (!root) return;
    postorder(root->left);
    postorder(root->right);
```

```
    cout << root->key << " ";
}

int findMin(Node* root) {
    if (!root) return -1;
    while (root->left) root = root->left;
    return root->key;
}

int findMax(Node* root) {
    if (!root) return -1;
    while (root->right) root = root->right;
    return root->key;
}

int countNodes(Node* root) {
    if (!root) return 0;
    return 1 + countNodes(root->left) + countNodes(root->right);
}

long long sumNodes(Node* root) {
    if (!root) return 0;
    return root->key + sumNodes(root->left) + sumNodes(root->right);
}

int main() {
    Node* root = nullptr;
    int pilih, x;

    do {
        cout << "\n=== BST (Dasar) ===\n";
        cout << "1. Insert\n2. Search\n3. Inorder\n4. Preorder\n5.
Postorder\n";
        cout << "6. Min\n7. Max\n8. Count nodes\n9. Sum nodes\n10. Keluar\n";
        cout << "Pilih: ";
        cin >> pilih;

        if (pilih == 1) {
            cout << "Masukkan nilai: "; cin >> x;
            root = insertNode(root, x);
        } else if (pilih == 2) {
            cout << "Cari nilai: "; cin >> x;
            cout << (searchNode(root, x) ? "Ditemukan\n" : "Tidak
ditemukan\n");
        } else if (pilih == 3) {
```

```
            inorder(root); cout << "\n";
        } else if (pilih == 4) {
            preorder(root); cout << "\n";
        } else if (pilih == 5) {
            postorder(root); cout << "\n";
        } else if (pilih == 6) {
            cout << "Min: " << findMin(root) << "\n";
        } else if (pilih == 7) {
            cout << "Max: " << findMax(root) << "\n";
        } else if (pilih == 8) {
            cout << "Jumlah node: " << countNodes(root) << "\n";
        } else if (pilih == 9) {
            cout << "Jumlah nilai: " << sumNodes(root) << "\n";
        }
    } while (pilih != 10);

    return 0;
}
6
```

BTSDasar.cpp

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

struct Node {
    int key;
    Node* left;
    Node* right;
    Node(int k): key(k), left(nullptr), right(nullptr) {}
};

Node* insertNode(Node* root, int key) {
    if (!root) return new Node(key);
    if (key < root->key) root->left = insertNode(root->left, key);
    else if (key > root->key) root->right = insertNode(root->right, key);
    return root;
}

Node* findMinNode(Node* root) {
    while (root && root->left) root = root->left;
    return root;
}

Node* deleteNode(Node* root, int key) {
    if (!root) return nullptr;
    if (key < root->key) root->left = deleteNode(root->left, key);
    else if (key > root->key) root->right = deleteNode(root->right, key);
    else {
        if (!root->left && !root->right) {
            delete root; return nullptr;
        } else if (!root->left) {
            Node* t = root->right; delete root; return t;
        } else if (!root->right) {
            Node* t = root->left; delete root; return t;
        } else {
            Node* succ = findMinNode(root->right);
            root->key = succ->key;
            root->right = deleteNode(root->right, succ->key);
        }
    }
    return root;
}

int height(Node* root) {
    if (!root) return -1;
```

```
    int hl = height(root->left);
    int hr = height(root->right);
    return 1 + (hl > hr ? hl : hr);
}

void levelOrder(Node* root) {
    if (!root) { cout << "(kosong)\n"; return; }
    queue<Node*> q;
    q.push(root);
    while (!q.empty()) {
        Node* cur = q.front(); q.pop();
        cout << cur->key << " ";
        if (cur->left) q.push(cur->left);
        if (cur->right) q.push(cur->right);
    }
    cout << "\n";
}

bool findSuccessor(Node* root, int key, int &ans) {
    Node* cur = root;
    Node* succ = nullptr;
    while (cur) {
        if (key < cur->key) { succ = cur; cur = cur->left; }
        else if (key > cur->key) cur = cur->right;
        else break;
    }
    if (!cur) return false;
    if (cur->right) succ = findMinNode(cur->right);
    if (!succ) return false;
    ans = succ->key; return true;
}

bool findPredecessor(Node* root, int key, int &ans) {
    Node* cur = root;
    Node* pred = nullptr;
    while (cur) {
        if (key > cur->key) { pred = cur; cur = cur->right; }
        else if (key < cur->key) cur = cur->left;
        else break;
    }
    if (!cur) return false;
    if (cur->left) {
        Node* t = cur->left;
        while (t->right) t = t->right;
        pred = t;
```

```
    }
    if (!pred) return false;
    ans = pred->key; return true;
}

int main() {
    Node* root = nullptr;
    int pilih, x;

    do {
        cout << "\n=== BST (Lanjutan) ===\n";
        cout << "1. Insert\n2. Delete\n3. Level-order\n4. Height\n5.
Successor\n6. Predecessor\n7. Keluar\n";
        cout << "Pilih: ";
        cin >> pilih;

        if (pilih == 1) {
            cout << "Masukkan nilai: "; cin >> x;
            root = insertNode(root, x);
        } else if (pilih == 2) {
            cout << "Hapus nilai: "; cin >> x;
            root = deleteNode(root, x);
        } else if (pilih == 3) {
            levelOrder(root);
        } else if (pilih == 4) {
            cout << "Tinggi pohon: " << height(root) << "\n";
        } else if (pilih == 5) {
            cout << "Cari successor dari: "; cin >> x;
            int ans;
            if (findSuccessor(root, x, ans)) cout << "Successor: " << ans <<
"\n";
            else cout << "Tidak ada successor (mungkin kunci tidak ada atau
yang terbesar)\n";
        } else if (pilih == 6) {
            cout << "Cari predecessor dari: "; cin >> x;
            int ans;
            if (findPredecessor(root, x, ans)) cout << "Predecessor: " << ans
<< "\n";
            else cout << "Tidak ada predecessor (mungkin kunci tidak ada atau
yang terkecil)\n";
        }
    } while (pilih != 7);
    return 0;
}
```

BSTLanjut.cpp

5.4 Tugas Akhir

1. Buatlah program yang mengimplementasikan percobaan – percobaan diatas. Upload kode dari tugas tersebut ke akun Github masing-masing.

VI PERCOBAAN 6: HASH MAP

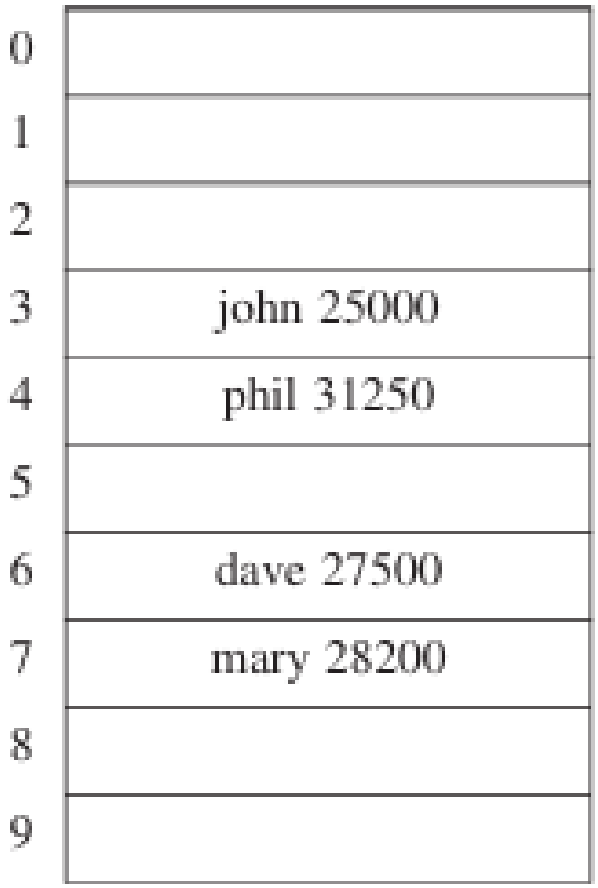
6.1 Tujuan Percobaan

1. Dapat memahami konsep hash map
2. Dapat menggunakan hash map

6.2 Tinjauan Pustaka

6.2.1 Pengertian Hash Map

Hash Map adalah salah satu struktur data asosiatif yang sangat penting dalam pemrograman. Struktur ini menyimpan data dalam bentuk pasangan kunci-nilai (key-value pairs), sehingga setiap kunci unik digunakan untuk mengakses nilai yang terkait dengannya. Konsep utamanya adalah memanfaatkan fungsi hash untuk mengubah kunci menjadi sebuah indeks dalam tabel hash. Indeks inilah yang menentukan lokasi penyimpanan nilai pada tabel, sehingga proses pencarian maupun penyisipan dapat dilakukan dengan sangat cepat.



Gambar 6.1 Visualiasi Hash Mapping

Berbeda dengan struktur data lain seperti array atau linked list, Hash Map tidak mengandalkan posisi berurutan, melainkan pemetaan melalui hash function. Hal ini memungkinkan akses data dengan kompleksitas waktu rata-rata O(1), jauh lebih efisien dibandingkan pencarian linier yang memerlukan O(n). Karena itu, Hash Map sangat cocok digunakan pada aplikasi yang membutuhkan operasi pencarian data berulang dengan kecepatan tinggi.

Karakteristik Hash Map

1. Penyimpanan pasangan kunci-nilai

   Setiap data disimpan dalam bentuk pasangan kunci dan nilai. Kunci harus unik agar tidak terjadi duplikasi data, sedangkan nilai bisa berupa tipe data apa pun.
2. Fungsi hash

   Fungsi hash bertugas mengubah kunci menjadi indeks dalam tabel hash. Kualitas fungsi hash sangat menentukan performa Hash Map, karena fungsi hash yang baik akan meminimalisasi tabrakan (*collision*).
3. Tidak terurut

   Berbeda dengan struktur map pada C++ yang disimpan terurut berdasarkan kunci (biasanya dengan binary search tree), unordered_map tidak menyimpan data dalam urutan tertentu. Keuntungannya adalah operasi penyisipan, penghapusan, dan pencarian dapat lebih cepat.
4. Penanganan collision

   Karena dua kunci yang berbeda bisa saja menghasilkan indeks yang sama, maka dibutuhkan metode penanganan tabrakan, misalnya dengan separate chaining atau open addressing.

Kelebihan Hash Map

1. Waktu akses, pencarian, dan penghapusan rata-rata sangat cepat (O(1)).
2. Dapat menyimpan data dalam jumlah besar dengan efisiensi tinggi.
3. Sangat fleksibel karena dapat menyimpan berbagai jenis data baik sebagai kunci maupun nilai.

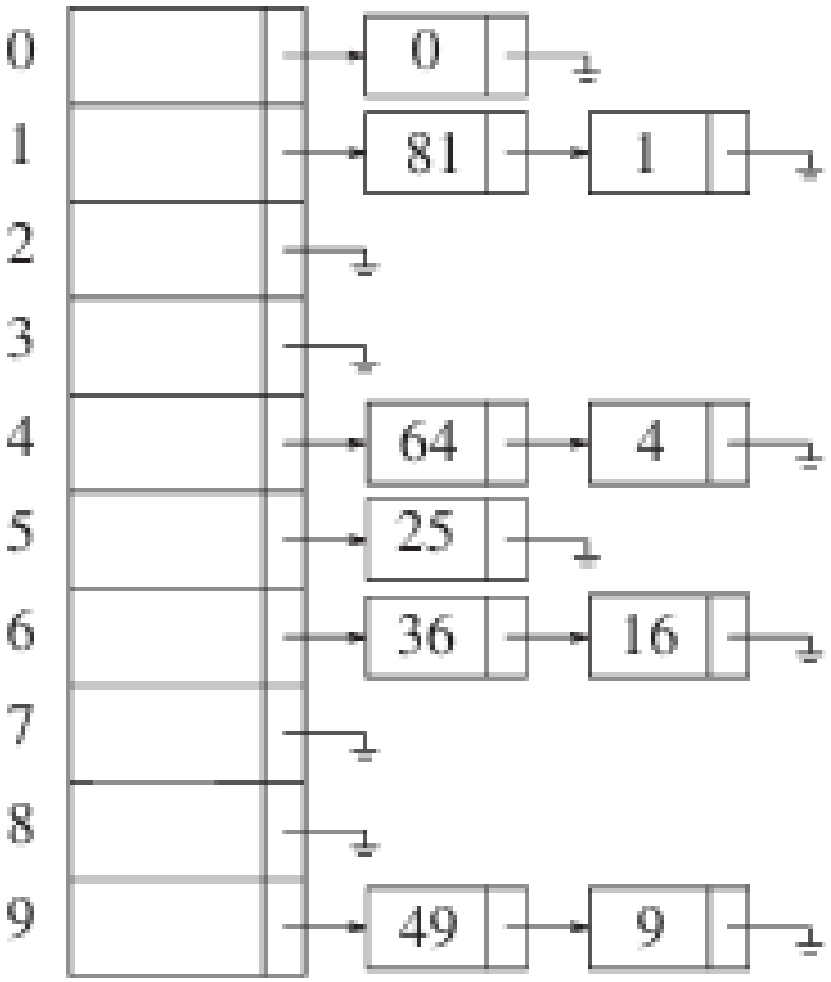
Kekurangan Hash Map

1. Tidak menjamin urutan elemen karena data disimpan berdasarkan hasil fungsi hash.
2. Membutuhkan fungsi hash yang baik, karena fungsi hash yang buruk akan meningkatkan jumlah collision dan menurunkan performa.
3. Memerlukan lebih banyak ruang memori dibandingkan struktur data lain seperti array atau linked list.

6.2.2 Jenis-jenis Hash Map

1. Separate Chaining

Separate chaining adalah metode penanganan tabrakan (*collision resolution*) pada tabel hash dengan cara menempatkan struktur data tambahan di setiap slot tabel. Biasanya, struktur yang digunakan adalah linked list. Dengan cara ini, setiap indeks tabel hash tidak hanya menyimpan satu elemen, tetapi dapat menyimpan sekumpulan elemen yang memiliki nilai hash sama.



Gambar 6.2 Visualisasi Separate Chaining

- Cara kerja:

  Jika sebuah kunci dipetakan ke indeks tertentu dan slot tersebut sudah terisi, maka elemen baru akan disimpan di dalam linked list pada indeks tersebut. Untuk pencarian, sistem cukup menelusuri linked list pada slot terkait hingga menemukan kunci yang diinginkan.
- Kelebihan:
  1. Implementasi relatif sederhana dan mudah dipahami.
  2. Tidak ada batasan jumlah elemen yang bisa disimpan pada satu indeks, sehingga performa masih dapat dijaga meskipun jumlah data melebihi ukuran tabel (*load factor* > 1).
  3. Mendukung operasi penghapusan dengan mudah, karena penghapusan pada linked list cukup mengubah pointer.

- Kekurangan:
  1. Membutuhkan memori tambahan untuk pointer dalam linked list.
  2. Jika banyak tabrakan terjadi, panjang linked list bisa bertambah dan menyebabkan waktu pencarian meningkat menjadi O(n) pada kasus terburuk.
  3. Lokalisasi data kurang efisien karena elemen-elemen bisa tersebar di memori.

2. Open Addresing

Open addressing adalah salah satu metode penyelesaian tabrakan (collision resolution) pada hash map yang banyak digunakan karena tidak memerlukan struktur data tambahan seperti linked list, sehingga lebih hemat ruang memori. Konsep dasar dari open addressing adalah seluruh data harus disimpan langsung ke dalam tabel hash, dan ketika sebuah kunci mengalami tabrakan, sistem tidak meletakkan elemen di luar tabel, melainkan melakukan pencarian slot kosong lain di dalam tabel itu sendiri. Pencarian slot kosong ini mengikuti suatu pola tertentu yang ditentukan oleh fungsi resolusi tabrakan. Dengan demikian, keberhasilan penyimpanan data dalam hash map open addressing sangat bergantung pada pemilihan fungsi hash yang baik serta strategi penanganan tabrakan yang tepat.

Salah satu strategi yang paling sederhana adalah linear probing, di mana pencarian slot kosong dilakukan secara berurutan dengan menambahkan offset satu per satu hingga

menemukan sel yang masih kosong. Strategi ini mudah diimplementasikan, tetapi menimbulkan masalah yang disebut primary clustering, yaitu terbentuknya deretan blok sel yang penuh secara berurutan. Jika sudah ada cluster yang panjang, maka setiap kunci baru yang masuk dan mengalami tabrakan akan semakin memperbesar cluster tersebut, sehingga memperburuk kinerja pencarian. Untuk mengatasi hal ini, dikembangkan strategi quadratic probing, yang menggunakan langkah pencarian dengan pola kuadrat (misalnya 1, 4, 9, dan seterusnya). Dengan cara ini, pertumbuhan cluster dapat diperlambat karena slot yang diperiksa menyebar lebih jauh. Akan tetapi, quadratic probing masih dapat menimbulkan masalah secondary clustering, yakni ketika dua kunci memiliki hash awal yang sama, keduanya akan menempuh jalur pencarian identik sehingga tetap berpotensi berbenturan berulang kali.

Strategi lain yang lebih efektif adalah double hashing, yang memanfaatkan dua fungsi hash berbeda. Jika terjadi tabrakan, maka langkah pencarian berikutnya dihitung berdasarkan fungsi hash kedua, sehingga distribusi lokasi pencarian lebih acak dan masalah clustering dapat diminimalisasi. Double hashing dianggap lebih mendekati kinerja pencarian acak dan biasanya mampu memberikan distribusi data yang merata di dalam tabel, asalkan ukuran tabel dipilih sebagai bilangan prima untuk memastikan semua slot dapat diakses. Dengan demikian, double hashing umumnya lebih disukai pada situasi di mana kinerja dan pemerataan distribusi data sangat penting, meskipun perhitungannya lebih kompleks karena memerlukan dua fungsi hash.

Karakteristik penting dari open addressing adalah bahwa kinerjanya sangat dipengaruhi oleh load factor (λ), yaitu perbandingan antara jumlah elemen yang disimpan dengan ukuran tabel. Jika load factor terlalu tinggi, misalnya mendekati 1, maka peluang tabrakan semakin besar dan pencarian slot kosong membutuhkan lebih banyak percobaan, sehingga waktu rata-rata operasi meningkat signifikan. Untuk mempertahankan efisiensi, load factor pada tabel hash open addressing biasanya dijaga di bawah 0,5. Hal ini membuat operasi pencarian, penyisipan, maupun penghapusan tetap dapat dilakukan dengan kompleksitas rata-rata O(1). Namun, bila tabel terlalu penuh atau terjadi terlalu banyak operasi campuran antara penyisipan dan penghapusan, maka performa dapat menurun drastis, bahkan

mendekati O(n). Karena itu, dalam praktiknya digunakan mekanisme rehashing, yaitu memperbesar ukuran tabel (biasanya dua kali lipat dari ukuran lama) ketika tabel sudah terlalu penuh, kemudian menghitung ulang posisi semua elemen agar distribusi kembali merata.

6.3 Percobaan

6.3.1 Percobaan VI-1: Hash Map

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

const int SIZE = 10;

struct Node {
    int key;
    int value;
    Node* next;
};

void initTable(Node* table[]) {
    for (int i = 0; i < SIZE; i++) {
        table[i] = nullptr;
    }
}

int hashFunction(int key) {
    return (key % SIZE + SIZE) % SIZE;
}

void insert(Node* table[], int key, int value) {
    int index = hashFunction(key);
    Node* cur = table[index];
    while (cur != nullptr) {
        if (cur->key == key) {
            cur->value = value;
            return;
        }
        cur = cur->next;
    }
    Node* baru = new Node;
    baru->key = key;
    baru->value = value;
    baru->next = table[index];
    table[index] = baru;
}

Node* search(Node* table[], int key) {
    int index = hashFunction(key);
    Node* cur = table[index];
    while (cur != nullptr) {
```

```
        if (cur->key == key) {
            return cur;
        }
        cur = cur->next;
    }
    return nullptr;
}

void removeKey(Node* table[], int key) {
    int index = hashFunction(key);
    Node* cur = table[index];
    Node* prev = nullptr;
    while (cur != nullptr) {
        if (cur->key == key) {
            if (prev == nullptr) {
                table[index] = cur->next;
            } else {
                prev->next = cur->next;
            }
            delete cur;
            return;
        }
        prev = cur;
        cur = cur->next;
    }
}

void display(Node* table[]) {
    cout << "\nIsi Hash Table:\n";
    for (int i = 0; i < SIZE; i++) {
        cout << i << ": ";
        Node* tmp = table[i];
        while (tmp != nullptr) {
            cout << "(" << tmp->key << "," << tmp->value << ") -> ";
            tmp = tmp->next;
        }
        cout << "NULL\n";
    }
}

int main() {
    Node* table[SIZE];
    initTable(table);

    insert(table, 1, 100);
```

```
    insert(table, 11, 200);
    insert(table, 21, 300);
    insert(table, 2, 400);

    display(table);

    Node* hasil = search(table, 11);
    if (hasil != nullptr) {
        cout << "\nKey 11 ditemukan dengan value = " << hasil->value << endl;
    } else {
        cout << "\nKey 11 tidak ditemukan\n";
    }

    removeKey(table, 11);
    cout << "\nSetelah menghapus key 11:\n";
    display(table);

    return 0;
}
```

HashMapSeparateChaining.cpp

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

const int SIZE = 10;

struct Node {
    int key;
    int value;
    Node* next;
};

void initTable(Node* table[]) {
    for (int i = 0; i < SIZE; i++) {
        table[i] = nullptr;
    }
}

int hashFunction(int key) {
    return (key % SIZE + SIZE) % SIZE;
}

void insert(Node* table[], int key, int value) {
    int index = hashFunction(key);
    Node* cur = table[index];
```

```
    while (cur != nullptr) {
        if (cur->key == key) {
            cur->value = value;
            return;
        }
        cur = cur->next;
    }
    Node* baru = new Node;
    baru->key = key;
    baru->value = value;
    baru->next = table[index];
    table[index] = baru;
}

Node* search(Node* table[], int key) {
    int index = hashFunction(key);
    Node* cur = table[index];
    while (cur != nullptr) {
        if (cur->key == key) {
            return cur;
        }
        cur = cur->next;
    }
    return nullptr;
}

void removeKey(Node* table[], int key) {
    int index = hashFunction(key);
    Node* cur = table[index];
    Node* prev = nullptr;
    while (cur != nullptr) {
        if (cur->key == key) {
            if (prev == nullptr) {
                table[index] = cur->next;
            } else {
                prev->next = cur->next;
            }
            delete cur;
            return;
        }
        prev = cur;
        cur = cur->next;
    }
}
```

```cpp
void display(Node* table[]) {
    cout << "\nIsi Hash Table:\n";
    for (int i = 0; i < SIZE; i++) {
        cout << i << ": ";
        Node* tmp = table[i];
        while (tmp != nullptr) {
            cout << "(" << tmp->key << "," << tmp->value << ") -> ";
            tmp = tmp->next;
        }
        cout << "NULL\n";
    }
}

int main() {
    Node* table[SIZE];
    initTable(table);

    insert(table, 1, 100);
    insert(table, 11, 200);
    insert(table, 21, 300);
    insert(table, 2, 400);

    display(table);

    Node* hasil = search(table, 11);
    if (hasil != nullptr) {
        cout << "\nKey 11 ditemukan dengan value = " << hasil->value << endl;
    } else {
        cout << "\nKey 11 tidak ditemukan\n";
    }

    removeKey(table, 11);
    cout << "\nSetelah menghapus key 11:\n";
    display(table);

    return 0;
}
```

HashMapOpenAddressing.cpp

6.4 Tugas Akhir

1. Buatlah program yang mengimplementasikan percobaan – percobaan diatas. Upload kode dari tugas tersebut ke akun Github masing-masing.

Daftar Pustaka

Weiss, M.A. 2014. *Data Structures and Algorithm Analysis in C++* (4th ed.). Boston: Pearson Education.

Erkamim, M., Abdurrohim, I., Yuliyanti, S., Karim, R., Rahman, A., Admira, T.M.A., & Ridwan, A. 2024. *Buku Ajar Algoritma dan Struktur Data*. Jambi: PT Sonpedia Publishing Indonesia.

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

Ginting, S.H.N., Effendi, H., Kumar, S., Marsisno, W., Sitanggang, Y.R.U., Anwar, K., Santiari, N.P.L., Setyowibowo, S., Sigar, T.R., Atho'illah, I., Setyantoro, D., & Smrti, N.N.E. 2023. *Pengantar Struktur Data*. Medan: PT Mifandi Mandiri Digital.