MODUL PRAKTIKUM

STRUKTUR DATA

DISUSUN OLEH :

MOH. ERKAMIM, S.Kom., M.Kom

PROGRAM STUDI D4 SISTEM INFORMASI KOTA CERDAS

FAKULTAS TEKNIK

UNIVERSITAS TUNAS PEMBANGUNAN

SURAKARTA

HALAMAN PENGESAHAN

Matakuliah	: Struktur Data
Kode Dokumen	: VSI2311/02/2023
Jenis Bahan Ajar	: Modul Praktikum
Program Studi	: D4 Sistem Informasi Kota Cerdas
Fakultas	: Teknik
Perguruan Tinggi	: Universitas Tunas Pembangunan Surakarta
Penyusun	: Moh. Erkamim, S.Kom., M.Kom

Modul ini disusun sebagai bahan ajar atau petunjuk praktikum dalam mata kuliah **Struktur Data.**

Disahkan
Surakarta, 6 Februari 2023
Menyetujui
Ketua Program Studi
D4 Sistem Informasi Kota Cerdas

Moh. Erkamim, S.Kom., M.Kom

Mengesahkan
Dekan Fakultas Teknik

Dr. Tri Hartanto, S.T., M.Sc

DAFTAR ISI

HALAMAN PENGESAHAN ii
DAFTAR ISI iii
PENGANTAR iv
MODUL I ARRAY 1
MODUL II LINKED LIST 6
MODUL III STACK 13
MODUL IV QUEUE 18
MODUL V TREES 23
MODUL VI GRAPHS 28
MODUL VII SORTING 33
MODUL VIII SEARCHING 38
MODUL IX DYNAMIC PROGRAMMING 42
PROYEK AKHIR 46
SOURCE CODE TUGAS 49

PENGANTAR

Struktur data adalah konsep dasar dalam pemrograman yang memungkinkan kita untuk menyimpan, mengatur, dan mengakses data dengan efisien. Sebagai seorang pemrogram, pemahaman yang baik tentang struktur data akan membantu Anda dalam mengembangkan solusi perangkat lunak yang lebih efisien dan kuat. Modul ini akan memandu Anda melalui berbagai jenis struktur data yang umum digunakan, mulai dari array hingga graph, dan bagaimana mengimplementasikannya dalam bahasa pemrograman.

Mengapa Struktur Data Penting?

Pertanyaan pertama yang mungkin muncul adalah mengapa kita perlu belajar tentang struktur data. Struktur data adalah alat yang mendasar dalam pemrograman yang membantu kita mengatasi masalah kompleks dengan cara yang efisien. Misalnya, ketika Anda ingin mencari data tertentu dalam daftar besar, pemilihan struktur data yang tepat dapat membuat pencarian menjadi jauh lebih cepat daripada jika Anda hanya menggunakan array sederhana. Dengan pemahaman tentang struktur data, Anda dapat merancang program yang lebih efisien, ekonomis dalam penggunaan sumber daya, dan mudah dimengerti.

Pendekatan dalam Belajar Struktur Data

Pendekatan yang baik dalam belajar struktur data adalah dengan memahami konsep dasar, mengamati cara kerja struktur data, dan kemudian mengimplementasikannya secara praktek. Kami akan menggunakan bahasa pemrograman C++ sebagai alat untuk mengimplementasikan berbagai struktur data. Namun, konsep-konsep yang Anda pelajari dalam modul ini dapat diaplikasikan dalam bahasa pemrograman lain juga.

Latihan dan Pengembangan Kemampuan

Setiap bab modul ini akan dilengkapi dengan latihan-latihan praktikum yang dirancang untuk menguji pemahaman Anda. Melalui latihan ini, Anda akan mendapatkan pengalaman dalam mengimplementasikan struktur data yang telah Anda pelajari. Pengembangan kemampuan melalui latihan praktikum adalah kunci untuk menjadi pemrogram yang lebih baik.

Apa yang Akan Anda Pelajari?

Anda akan memulai perjalanan Anda dalam memahami struktur data dengan mempelajari konsep dasar seperti array, linked list, dan stack. Kami akan membahas keuntungan dan kerugian masing-masing struktur data dan kapan mereka paling efektif. Setiap bab akan mengajarkan Anda bagaimana mengimplementasikan struktur data tersebut dalam bahasa pemrograman C++ serta bagaimana memecahkan masalah dengan menggunakan struktur data tersebut.

Kapan Sebaiknya Menggunakan Struktur Data Tertentu?

Selain memahami bagaimana cara mengimplementasikan struktur data, Anda juga akan memahami kapan sebaiknya menggunakan struktur data tertentu dalam situasi tertentu. Pemilihan yang tepat dari struktur data dapat memiliki dampak besar pada kinerja dan keefisienan program Anda.

Eksplorasi yang Tak Terbatas

Ketika Anda memahami konsep dasar struktur data, Anda akan merasa lebih percaya diri dalam menghadapi berbagai jenis masalah pemrograman. Struktur data adalah alat yang sangat kuat, dan eksplorasi dalam dunia struktur data adalah tak terbatas. Modul ini adalah langkah awal Anda dalam menggali kekuatan dan potensi yang ada dalam struktur data.

Selamat belajar.

MODUL I ARRAY

1. TUJUAN

- Memahami konsep dasar array dalam bahasa pemrograman C++.
- Mampu melakukan operasi dasar pada array seperti pengaksesan elemen, mengganti elemen, dan penambahan elemen.
- Mampu memecahkan masalah sederhana dengan menggunakan array.

2. DASAR TEORI

A. Array: Konsep Dasar

1. **Definisi:** Array adalah struktur data yang mengandung kumpulan elemen dengan tipe data yang sama. Setiap elemen dalam array dapat diakses menggunakan indeks numerik.
2. **Elemen dan Indeks:** Setiap elemen dalam array memiliki indeks yang mengidentifikasikan posisinya dalam array. Indeks dimulai dari 0 (untuk elemen pertama) dan berlanjut hingga n-1, di mana n adalah jumlah total elemen dalam array.
3. **Tipe Data Seragam:** Semua elemen dalam array harus memiliki tipe data yang sama, misalnya, semua elemen integer, float, atau karakter.

B. Operasi pada Array

1. **Inisialisasi:** Array dapat diinisialisasi dengan elemen-elemen awal saat deklarasi, atau elemen-elemen dapat diisi secara terpisah.
2. **Pengaksesan:** Anda dapat mengakses elemen array menggunakan indeks. Misalnya, arr[0] mengakses elemen pertama dalam array arr.
3. **Pengisian Elemen:** Anda dapat mengisi atau mengganti elemen dalam array dengan nilai baru, misalnya, arr[1] = 42 mengganti elemen kedua dengan nilai 42.
4. **Panjang Array:** Anda dapat mendapatkan panjang atau jumlah elemen dalam array dengan menggunakan fungsi atau properti khusus, misalnya, arr.length() atau arr.size().
5. **Iterasi:** Anda dapat melakukan iterasi atau perulangan melalui elemen-elemen dalam array menggunakan loop, seperti for atau while, atau menggunakan iterasi dengan foreach dalam beberapa bahasa.

C. Array Multidimensi

1. **Array Dua Dimensi:** Array dua dimensi (matriks) adalah array yang memiliki dua indeks untuk mengakses elemennya. Ini digunakan untuk merepresentasikan tabel atau matriks.
2. **Array Tiga Dimensi dan Lebih:** Anda juga dapat memiliki array tiga dimensi (atau lebih) yang digunakan untuk merepresentasikan data dengan lebih banyak dimensi, seperti data spasial dalam grafika komputer atau data berkelanjutan dalam ilmu fisika.

D. Operasi Pencarian dan Pengurutan pada Array

1. **Pencarian:** Pencarian adalah proses mencari elemen tertentu dalam array. Beberapa algoritma pencarian umum meliputi pencarian linear (sequential) dan pencarian biner (binary search). Pencarian linear memeriksa setiap elemen secara berurutan, sementara pencarian biner bekerja pada array yang sudah terurut.
2. **Pengurutan:** Pengurutan adalah proses mengurutkan elemen-elemen array dalam urutan tertentu. Beberapa algoritma pengurutan yang umum digunakan meliputi Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, dan Quick Sort.

E. Array dalam Bahasa Pemrograman

1. **Deklarasi:** Setiap bahasa pemrograman memiliki sintaksis khusus untuk mendeklarasikan dan menginisialisasi array. Misalnya, dalam C++, Anda dapat mendeklarasikan array dengan int arr[5] untuk array berukuran lima.
2. **Akses Elemen:** Cara mengakses elemen array juga bervariasi antara bahasa pemrograman. Misalnya, dalam C++, Anda dapat menggunakan arr[i] untuk mengakses elemen ke-i dalam array.
3. **Panjang Array:** Cara mendapatkan panjang array juga berbeda-beda. Beberapa bahasa memiliki properti seperti length, sementara yang lain memerlukan perhitungan manual.
4. **Array Dinamis:** Beberapa bahasa pemrograman memungkinkan penggunaan array dinamis yang dapat diperluas atau dikurangi ukurannya saat diperlukan.
5. **Operasi Built-in:** Beberapa bahasa pemrograman menyediakan operasi built-in untuk operasi umum pada array, seperti pencarian dan pengurutan.

3. KEGIATAN PRAKTIKUM

A. Membuat dan Mengakses Array:

1. Deklarasikan sebuah array dengan tipe data tertentu.
2. Inisialisasikan array tersebut dengan beberapa nilai awal.
3. Coba akses elemen-elemen dalam array dengan menggunakan indeks.

```
#include <iostream>

int main() {
  // 1. Deklarasikan sebuah array dengan tipe data tertentu.
  int myArray[5];

  // 2. Inisialisasikan array tersebut dengan beberapa nilai awal.
  myArray[0] = 10;
  myArray[1] = 20;
  myArray[2] = 30;
  myArray[3] = 40;
  myArray[4] = 50;

  // 3. Coba akses elemen-elemen dalam array dengan menggunakan indeks.
  std::cout << "Isi array:" << std::endl;
```

```
    for (int i = 0; i < 5; i++) {
        std::cout << "myArray[" << i << "] = " << myArray[i] << std::endl;
    }

    return 0;
}
```

B. Mengubah dan Menambahkan Elemen dalam Array:

1. Ubah nilai elemen-elemen dalam array.
2. Tambahkan beberapa elemen baru ke array.

```
#include <iostream>

int main() {
    int myArray[5] = {10, 20, 30, 40, 50};

    // 1. Ubah nilai elemen-elemen dalam array.
    myArray[2] = 35;

    // 2. Tambahkan beberapa elemen baru ke array.
    int newSize = 7;
    int newArray[newSize] = {60, 70, 80, 90, 100, 110, 120};
    for (int i = 0; i < newSize; i++) {
        myArray[i + 5] = newArray[i];
    }

    // Tampilkan isi array setelah perubahan dan penambahan.
    std::cout << "Isi array setelah perubahan dan penambahan:" << std::endl;
    for (int i = 0; i < newSize; i++) {
        std::cout << "myArray[" << i << "] = " << myArray[i] << std::endl;
    }

    return 0;
}
```

C. Menghapus Elemen dari Array:

1. Hapus elemen dari array

```
#include <iostream>

int main() {
    int myArray[5] = {10, 20, 30, 40, 50};

    // Hapus elemen ke-2 (indeks 1)
    for (int i = 1; i < 5; i++) {
        myArray[i - 1] = myArray[i];
    }

    // Kurangi ukuran array
```

```
    int newSize = 4;

    // Tampilkan isi array setelah penghapusan.
    std::cout << "Isi array setelah penghapusan:" << std::endl;
    for (int i = 0; i < newSize; i++) {
        std::cout << "myArray[" << i << "] = " << myArray[i] << std::endl;
    }

    return 0;
}
```

D. Operasi Matematika pada Array:

1. Buatlah dua daftar dengan elemen-elemen numerik.
2. Buat fungsi yang dapat menjumlahkan elemen-elemen dari kedua array.
3. Buat fungsi yang dapat mengalikan elemen-elemen daftar dengan skalar tertentu.

```
#include <iostream>

// Fungsi untuk menjumlahkan elemen-elemen dari dua array
void tambahArray(int arr1[], int arr2[], int hasil[], int ukuran) {
    for (int i = 0; i < ukuran; i++) {
        hasil[i] = arr1[i] + arr2[i];
    }
}

// Fungsi untuk mengalikan elemen-elemen array dengan skalar tertentu
void kaliDenganSkalar(int arr[], int skalar, int hasil[], int ukuran) {
    for (int i = 0; i < ukuran; i++) {
        hasil[i] = arr[i] * skalar;
    }
}

int main() {
    int arr1[] = {1, 2, 3, 4, 5};
    int arr2[] = {5, 4, 3, 2, 1};
    int ukuran = 5;
    int hasilTambah[ukuran];
    int hasilKali[ukuran];

    // Memanggil fungsi untuk menjumlahkan elemen-elemen array
    tambahArray(arr1, arr2, hasilTambah, ukuran);

    // Memanggil fungsi untuk mengalikan elemen-elemen array dengan skalar
    int skalar = 2;
    kaliDenganSkalar(arr1, skalar, hasilKali, ukuran);

    // Menampilkan hasil
    std::cout << "Hasil Penjumlahan:" << std::endl;
```

```
    for (int i = 0; i < ukuran; i++) {
        std::cout << hasilTambah[i] << " ";
    }
    std::cout << std::endl;

    std::cout << "Hasil Perkalian dengan Skalar:" << std::endl;
    for (int i = 0; i < ukuran; i++) {
        std::cout << hasilKali[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

4. TUGAS

Buatlah sebuah program Python yang mencakup fungsi-fungsi berikut:

1. Menggabungkan Dua Array: Buat fungsi yang menerima dua daftar sebagai argumen dan menggabungkannya menjadi satu array baru.
2. Menjumlahkan Elemen-elemen Array: Buat fungsi yang menerima sebuah daftar sebagai argumen dan mengembalikan hasil penjumlahan semua elemen dalam daftar tersebut.
3. Menemukan Nilai Maksimum dalam Array: Buat fungsi yang menerima sebuah daftar sebagai argumen dan mengembalikan nilai maksimum dalam daftar tersebut.
4. Mengganti Elemen dalam Array: Buat fungsi yang menerima sebuah daftar, indeks elemen yang akan diganti, dan nilai baru yang akan digunakan untuk menggantikan elemen tersebut.

MODUL II LINKED LIST

1. TUJUAN

- Memahami konsep dasar Linked List dalam bahasa pemrograman C++.
- Mampu membuat dan mengelola Linked List.
- Mampu melakukan operasi dasar pada Linked List seperti penyisipan, penghapusan, dan pencarian elemen.

2. DASAR TEORI

A. Linked List: Konsep Dasar

1. **Definisi:** Linked List adalah struktur data linear yang digunakan untuk menyimpan sekumpulan elemen yang dikenal sebagai "node". Setiap node memiliki dua komponen: data dan referensi (link) ke node berikutnya dalam urutan.
2. **Node:** Node adalah unit dasar dalam linked list. Setiap node memiliki dua bagian: data, yang menyimpan nilai atau informasi, dan referensi (atau pointer/link) yang mengarah ke node berikutnya dalam linked list.
3. **Head:** Head adalah referensi pertama dalam linked list yang mengarah ke node pertama. Ini digunakan untuk mengakses linked list secara keseluruhan.
4. **Tail:** Tail adalah node terakhir dalam linked list. Biasanya, tail memiliki referensi yang menunjuk ke nullptr atau null untuk menunjukkan akhir dari linked list.
5. **Singly Linked List:** Dalam singly linked list, setiap node hanya memiliki satu referensi yang mengarah ke node berikutnya dalam urutan.
6. **Doubly Linked List:** Dalam doubly linked list, setiap node memiliki dua referensi: satu mengarah ke node berikutnya dan satu mengarah ke node sebelumnya dalam urutan. Hal ini memungkinkan perjalanan maju dan mundur dalam linked list.
7. **Circular Linked List:** Dalam circular linked list, tail memiliki referensi yang mengarah kembali ke node pertama, membuatnya berbentuk lingkaran. Ini memungkinkan untuk iterasi tak terbatas melalui linked list.

B. Operasi pada Linked List

1. **Penambahan Elemen:** Anda dapat menambahkan elemen baru ke linked list di awal (prepend), di akhir (append), atau di antara dua node tertentu.
2. **Penghapusan Elemen:** Anda dapat menghapus elemen dari linked list berdasarkan nilai atau posisi tertentu.
3. **Pencarian:** Anda dapat mencari elemen berdasarkan nilai tertentu dalam linked list.
4. **Iterasi:** Anda dapat melakukan iterasi melalui seluruh linked list untuk mengakses atau memproses elemen-elemen di dalamnya.

C. Kelebihan Linked List

1. **Fleksibilitas dalam Penambahan dan Penghapusan:** Linked list memungkinkan penambahan dan penghapusan elemen dengan lebih efisien daripada array yang harus menggeser elemen-elemen lain.
2. **Dynamic Size:** Linked list dapat diperpanjang atau dikurangi ukurannya sesuai kebutuhan, sehingga cocok untuk situasi di mana ukuran data berubah-ubah.
3. **Memory Allocation:** Linked list dapat mengalokasikan memori secara dinamis untuk setiap node, memungkinkan penggunaan memori yang efisien.

D. Kekurangan Linked List

1. **Random Access Terbatas:** Akses elemen secara acak dalam linked list lebih lambat daripada dalam array, karena Anda harus melakukan iterasi dari awal untuk mencapai elemen yang diinginkan.
2. **Penggunaan Memori Tambahan:** Setiap node dalam linked list memerlukan alokasi memori tambahan untuk menyimpan referensi ke node berikutnya, yang dapat mengakibatkan penggunaan memori yang lebih besar daripada array.
3. **Kompleksitas Kode:** Operasi pada linked list memerlukan kode yang lebih kompleks daripada array, seperti alokasi dan dealokasi memori, penanganan referensi, dan sebagainya.

3. KEGIATAN PRAKTIKUM:

A. Membuat Linked List:

1. Mendefinisikan struktur (struct) untuk simpul Linked List.
2. Membuat Linked List kosong.
3. Menambahkan beberapa simpul ke dalam Linked List.

```
#include <iostream>
// 1. Mendefinisikan struktur (struct) untuk simpul Linked List.
struct Node {
   int data;
   Node* next;
};

int main() {
   // 2. Membuat Linked List kosong.
   Node* head = nullptr;

   // 3. Menambahkan beberapa simpul ke dalam Linked List.
   Node* newNode1 = new Node;
   newNode1->data = 10;
   newNode1->next = nullptr;
   head = newNode1;

   Node* newNode2 = new Node;
```

```
    newNode2->data = 20;
    newNode2->next = nullptr;
    newNode1->next = newNode2;

    // Cetak elemen-elemen Linked List.
    Node* current = head;
    while (current != nullptr) {
        std::cout << current->data << " ";
        current = current->next;
    }

    return 0;
}
```

B. Penyisipan dan Penghapusan Elemen:

1. Menyisipkan simpul baru di awal Linked List.
2. Menyisipkan simpul baru di akhir Linked List.
3. Menghapus simpul di awal Linked List.
4. Menghapus simpul di akhir Linked List.

```
#include <iostream>
struct Node {
    int data;
    Node* next;
};

// 1. Menyisipkan simpul baru di awal Linked List.
Node* sisipkanAwal(Node* head, int nilai) {
    Node* newNode = new Node;
    newNode->data = nilai;
    newNode->next = head;
    return newNode;
}

// 2. Menyisipkan simpul baru di akhir Linked List.
Node* sisipkanAkhir(Node* head, int nilai) {
    Node* newNode = new Node;
    newNode->data = nilai;
    newNode->next = nullptr;

    if (head == nullptr) {
        return newNode;
    }

    Node* current = head;
    while (current->next != nullptr) {
        current = current->next;
    }
```

```
    current->next = newNode;
    return head;
}

// 3. Menghapus simpul di awal Linked List.
Node* hapusAwal(Node* head) {
    if (head == nullptr) {
        return nullptr;
    }

    Node* temp = head;
    head = head->next;
    delete temp;
    return head;
}

// 4. Menghapus simpul di akhir Linked List.
Node* hapusAkhir(Node* head) {
    if (head == nullptr) {
        return nullptr;
    }

    if (head->next == nullptr) {
        delete head;
        return nullptr;
    }

    Node* current = head;
    while (current->next->next != nullptr) {
        current = current->next;
    }

    delete current->next;
    current->next = nullptr;
    return head;
}

int main() {
    Node* head = nullptr;

    // Menambahkan simpul di awal Linked List.
    head = sisipkanAwal(head, 30);

    // Menambahkan simpul di akhir Linked List.
    head = sisipkanAkhir(head, 40);

    // Menghapus simpul di awal Linked List.
    head = hapusAwal(head);
```

```
    // Menghapus simpul di akhir Linked List.
    head = hapusAkhir(head);

    // Cetak elemen-elemen Linked List.
    Node* current = head;
    while (current != nullptr) {
        std::cout << current->data << " ";
        current = current->next;
    }

    return 0;
}
```

C. Pencarian Elemen:

1. Mencari simpul dengan nilai tertentu dalam Linked List.

```
#include <iostream>

struct Node {
    int data;
    Node* next;
};

// Mencari simpul dengan nilai tertentu dalam Linked List.
bool cari(Node* head, int nilai) {
    Node* current = head;
    while (current != nullptr) {
        if (current->data == nilai) {
            return true;
        }
        current = current->next;
    }
    return false;
}

int main() {
    Node* head = nullptr;

    // Menambahkan beberapa simpul ke dalam Linked List.
    head = sisipkanAwal(head, 30);
    head = sisipkanAkhir(head, 40);
    head = sisipkanAkhir(head, 50);

    int nilaiCari = 40;

    // Mencari simpul dengan nilai tertentu dalam Linked List.
    bool ditemukan = cari(head, nilaiCari);

    if (ditemukan) {
```

```
        std::cout << "Nilai " << nilaiCari << " ditemukan dalam Linked List." << std::endl;
    } else {
        std::cout << "Nilai " << nilaiCari << " tidak ditemukan dalam Linked List." << std::endl;
    }

    return 0;
}
```

D. Traversing Linked List:

1. Melintasi seluruh Linked List dan mencetak elemen-elemen.

```
#include <iostream>

struct Node {
    int data;
    Node* next;
};

void cetakLinkedList(Node* head) {
    Node* current = head;
    while (current != nullptr) {
        std::cout << current->data << " ";
        current = current->next;
    }
    std::cout << std::endl;
}

int main() {
    Node* head = nullptr;

    // Menambahkan beberapa simpul ke dalam Linked List.
    head = sisipkanAkhir(head, 10);
    head = sisipkanAkhir(head, 20);
    head = sisipkanAkhir(head, 30);

    // Melintasi seluruh Linked List dan mencetak elemen-elemen.
    cetakLinkedList(head);

    return 0;
}
```

4. TUGAS:

Buatlah sebuah program C++ yang mencakup implementasi Linked List dan fungsi-fungsi berikut:

1. **Penyisipan Elemen di Awal Linked List:** Buat fungsi yang menerima data sebagai argumen dan menyisipkan elemen baru di awal Linked List.
2. **Penyisipan Elemen di Akhir Linked List:** Buat fungsi yang menerima data sebagai argumen dan menyisipkan elemen baru di akhir Linked List.

3. **Penghapusan Elemen di Awal Linked List:** Buat fungsi yang menghapus elemen di awal Linked List.
4. **Penghapusan Elemen di Akhir Linked List:** Buat fungsi yang menghapus elemen di akhir Linked List.
5. **Pencarian Elemen dalam Linked List:** Buat fungsi yang menerima nilai sebagai argumen dan mencari elemen dengan nilai tersebut dalam Linked List.
6. **Traversing Linked List:** Buat fungsi yang melintasi seluruh Linked List dan mencetak elemen-elemen.

MODUL III STACK

1. TUJUAN:

- Memahami konsep dasar dari stack dalam bahasa pemrograman C++.
- Mampu membuat dan mengelola stack.
- Mampu melakukan operasi dasar pada stack seperti push, pop, dan pencarian elemen.

2. DASAR TEORI:

A. Stack: Konsep Dasar

1. **Definisi**: Stack adalah struktur data linear yang mengikuti prinsip "*Last In, First Out*" (LIFO), yang berarti elemen yang terakhir dimasukkan ke dalam stack akan menjadi yang pertama dikeluarkan.
2. **Elemen**: Setiap elemen dalam stack disebut sebagai "node" atau "item". Elemen teratas dari stack sering disebut "top".
3. **Operasi Utama**: Stack mendukung dua operasi utama:
   - **Push**: Menambahkan elemen ke atas stack.
   - **Pop**: Menghapus elemen teratas dari stack.
4. **Top**: Top adalah elemen teratas dalam stack yang saat ini aktif dan dapat diakses. Ini adalah satu-satunya elemen yang dapat dihapus atau dilihat tanpa mengubah elemen-elemen lain dalam stack.

B. Operasi pada Stack

1. **Push**: Operasi push digunakan untuk menambahkan elemen ke atas stack. Ini dilakukan dengan meletakkan elemen baru di atas elemen teratas yang ada. Push menggeser elemen-elemen sebelumnya ke bawah.
2. **Pop**: Operasi pop digunakan untuk menghapus elemen teratas dari stack. Ini mengakibatkan elemen di bawahnya menjadi elemen teratas yang baru. Pop mengembalikan nilai dari elemen yang dihapus.
3. **Peek (Top)**: Peek adalah operasi untuk melihat elemen teratas dari stack tanpa menghapusnya. Ini berguna untuk melihat nilai teratas tanpa mengganggu struktur stack.
4. **Size**: Operasi size digunakan untuk mengukur jumlah elemen dalam stack.
5. **isEmpty**: Operasi isEmpty digunakan untuk memeriksa apakah stack kosong atau tidak. Jika stack kosong, operasi pop tidak dapat dilakukan.

C. Penggunaan Stack

1. **Eksekusi Program**: Stack digunakan dalam eksekusi program untuk menyimpan informasi tentang pemanggilan fungsi (fungsi call stack), termasuk alamat pengembalian dan variabel lokal.

2. **Pemecahan Masalah**: Stack digunakan dalam pemecahan masalah seperti evaluasi ekspresi matematika dalam notasi postfix, pengecekan keseimbangan tanda kurung dalam ekspresi, dan pencarian jalur dalam algoritma DFS (Depth-First Search).
3. **Undo/Redo**: Stack sering digunakan dalam aplikasi yang mendukung operasi undo (membatalkan) dan redo (mengulangi) untuk mengingat perubahan terakhir.
4. **Backtracking**: Stack digunakan dalam algoritma backtracking untuk mencatat jejak perjalanan dan pencarian solusi dalam masalah seperti Sudoku atau N-Queens.

D. Kelebihan *Stack*

1. **Struktur Sederhana**: Stack adalah struktur data yang sederhana dan mudah dimengerti.
2. **Operasi Cepat**: Operasi push dan pop pada stack memiliki kompleksitas waktu konstan (O(1)).
3. **Memory Management**: Stack memiliki alokasi memori statis yang efisien.

E. Kekurangan Stack

1. **Kapasitas Terbatas**: Ukuran stack biasanya tetap dan tidak dapat diperbesar dinamis, yang dapat menyebabkan overflow jika tidak diatur dengan baik.
2. **Random Access Terbatas**: Akses elemen di tengah stack atau pada posisi tertentu tidak efisien, karena Anda harus mengeluarkan elemen-elemen di atasnya terlebih dahulu.

3. KEGIATAN PRAKTIKUM:

A. Membuat dan Mengelola Stack:

1. Mendefinisikan struktur (struct) untuk stack.
2. Membuat stack kosong.
3. Menambahkan beberapa elemen ke dalam stack (push).

B. Operasi Dasar pada Stack:

1. Melakukan operasi push untuk menambahkan elemen ke dalam stack.
2. Melakukan operasi pop untuk menghapus elemen dari atas stack.
3. Mengakses elemen teratas (top) stack tanpa menghapusnya.
4. Memeriksa apakah stack kosong atau tidak.
5. Menghitung jumlah elemen dalam stack (ukuran stack).

C. Pencarian Elemen dalam Stack:

1. Mencari elemen tertentu dalam stack.

```
#include <iostream>

// 1. Mendefinisikan struktur (struct) untuk stack.
struct StackNode {
   int data;
   StackNode* next;
};
```

```
// Struktur Stack yang menyimpan tumpukan node.
struct Stack {
    StackNode* top;
};

// Inisialisasi stack kosong.
void initStack(Stack& stack) {
    stack.top = nullptr;
}

// 2. Melakukan operasi push untuk menambahkan elemen ke dalam stack.
void push(Stack& stack, int value) {
    StackNode* newNode = new StackNode;
    newNode->data = value;
    newNode->next = stack.top;
    stack.top = newNode;
}
```

2. Operasi Dasar pada Stack:

```
// 3. Melakukan operasi pop untuk menghapus elemen dari atas stack.
bool pop(Stack& stack) {
    if (stack.top == nullptr) {
        return false; // Stack kosong, tidak ada yang bisa di-pop.
    }

    StackNode* temp = stack.top;
    stack.top = stack.top->next;
    delete temp;
    return true;
}

// 4. Mengakses elemen teratas (top) stack tanpa menghapusnya.
int peek(const Stack& stack) {
    if (stack.top != nullptr) {
        return stack.top->data;
    } else {
        // Handle jika stack kosong. Anda dapat mengembalikan nilai default atau memunculkan pesan
kesalahan.
        return -1; // Misalnya, -1 sebagai penanda stack kosong.
    }
}

// 5. Memeriksa apakah stack kosong atau tidak.
bool isEmpty(const Stack& stack) {
    return stack.top == nullptr;
}

// 6. Menghitung jumlah elemen dalam stack (ukuran stack).
```

```
int size(const Stack& stack) {
    int count = 0;
    StackNode* current = stack.top;
    while (current != nullptr) {
        count++;
        current = current->next;
    }
    return count;
}
```

3. Pencarian Elemen dalam Stack:

```
// 7. Mencari elemen tertentu dalam stack.
bool search(const Stack& stack, int target) {
    StackNode* current = stack.top;
    while (current != nullptr) {
        if (current->data == target) {
            return true;
        }
        current = current->next;
    }
    return false;
}

int main() {
    Stack myStack;
    initStack(myStack);

    // Menambahkan beberapa elemen ke dalam stack.
    push(myStack, 10);
    push(myStack, 20);
    push(myStack, 30);

    // Melakukan operasi pop untuk menghapus elemen dari atas stack.
    pop(myStack);

    // Mengakses elemen teratas (top) stack tanpa menghapusnya.
    int topElement = peek(myStack);
    if (topElement != -1) {
        std::cout << "Elemen teratas stack: " << topElement << std::endl;
    } else {
        std::cout << "Stack kosong." << std::endl;
    }

    // Memeriksa apakah stack kosong atau tidak.
    bool isEmptyStack = isEmpty(myStack);
    if (isEmptyStack) {
        std::cout << "Stack kosong." << std::endl;
    } else {
```

```
    std::cout << "Stack tidak kosong." << std::endl;
  }

  // Menghitung jumlah elemen dalam stack (ukuran stack).
  int stackSize = size(myStack);
  std::cout << "Ukuran stack: " << stackSize << std::endl;

  // Mencari elemen tertentu dalam stack.
  int target = 20;
  bool found = search(myStack, target);
  if (found) {
    std::cout << "Elemen " << target << " ditemukan dalam stack." << std::endl;
  } else {
    std::cout << "Elemen " << target << " tidak ditemukan dalam stack." << std::endl;
  }

  return 0;
}
```

4. TUGAS:

Buatlah sebuah program C++ yang mencakup implementasi stack dan fungsi-fungsi berikut:

1. **Push:** Buat fungsi untuk menambahkan elemen ke dalam stack.
2. **Pop:** Buat fungsi untuk menghapus elemen dari atas stack.
3. **Top:** Buat fungsi untuk mengakses elemen teratas stack tanpa menghapusnya.
4. **Cek Kosong:** Buat fungsi untuk memeriksa apakah stack kosong atau tidak.
5. **Cek Ukuran:** Buat fungsi untuk menghitung jumlah elemen dalam stack (ukuran stack).
6. **Pencarian Elemen dalam Stack:** Buat fungsi yang menerima nilai sebagai argumen dan mencari elemen dengan nilai tersebut dalam stack.

MODUL IV QUEUE

1. TUJUAN:

- Memahami konsep dasar dari queue dalam bahasa pemrograman C++.
- Mampu membuat dan mengelola queue.
- Mampu melakukan operasi dasar pada queue seperti enqueue, dequeue, dan pencarian elemen.

2. DASAR TEORI:

A. Queue: Konsep Dasar

1. **Definisi**: Queue adalah struktur data linear yang mengikuti prinsip "First In, First Out" (FIFO), yang berarti elemen yang pertama dimasukkan ke dalam queue akan menjadi yang pertama dikeluarkan.
2. **Elemen**: Setiap elemen dalam queue disebut sebagai "node" atau "item". Elemen yang pertama dimasukkan ke dalam queue sering disebut sebagai "front", sementara elemen yang terakhir dimasukkan disebut sebagai "rear" atau "back".
3. **Operasi Utama**: Queue mendukung dua operasi utama:
    - **Enqueue**: Menambahkan elemen ke dalam belakang queue (rear).
    - **Dequeue**: Menghapus elemen dari depan queue (front).
4. **Front dan Rear**: Front adalah elemen pertama dalam queue yang dapat diakses. Rear adalah elemen terakhir dalam queue.

B. Operasi pada Queue

1. **Enqueue**: Operasi enqueue digunakan untuk menambahkan elemen ke dalam queue. Ini dilakukan dengan meletakkan elemen baru di belakang elemen yang ada. Enqueue tidak mengubah elemen-elemen sebelumnya dalam queue.
2. **Dequeue**: Operasi dequeue digunakan untuk menghapus elemen dari depan queue. Ini mengakibatkan elemen yang berada di belakangnya menjadi elemen depan yang baru. Dequeue mengembalikan nilai dari elemen yang dihapus.
3. **Peek (Front dan Rear)**: Peek adalah operasi untuk melihat elemen depan dan belakang queue tanpa menghapusnya. Ini berguna untuk melihat nilai depan dan belakang tanpa mengganggu struktur queue.
4. **Size**: Operasi size digunakan untuk mengukur jumlah elemen dalam queue.
5. **isEmpty**: Operasi isEmpty digunakan untuk memeriksa apakah queue kosong atau tidak. Jika queue kosong, operasi dequeue tidak dapat dilakukan.

C. Penggunaan Queue

1. **Antrian Pemanggilan**: Queue digunakan dalam simulasi antrian pemanggilan, seperti antrian di pusat panggilan atau antrean pelanggan di toko.
2. **Pemrosesan Data Berurutan**: Queue digunakan dalam pemrosesan data yang harus diurutkan dan diproses dalam urutan tertentu, seperti antrian tugas pemrosesan data.

3. **Algoritma BFS**: Queue digunakan dalam algoritma Breadth-First Search (BFS) untuk menjelajah atau mencari solusi dalam struktur data berbentuk graf.
4. **Buffering Data**: Queue digunakan dalam buffering data, seperti ketika data yang diproduksi dengan laju tertentu harus diambil dan diproses dengan laju yang berbeda.

D. Keuntungan Queue

1. **Prinsip FIFO**: Queue mengikuti prinsip FIFO, yang sesuai dengan banyak situasi dalam dunia nyata.
2. **Operasi Cepat**: Operasi enqueue dan dequeue pada queue memiliki kompleksitas waktu konstan (O(1)).

E. Kerugian Queue

1. **Kapasitas Terbatas**: Kapasitas queue biasanya tetap dan tidak dapat diperbesar dinamis, yang dapat menyebabkan overflow jika tidak diatur dengan baik.
2. **Random Access Terbatas**: Akses elemen di tengah queue atau pada posisi tertentu tidak efisien, karena Anda harus mengeluarkan elemen-elemen di depannya terlebih dahulu.

3. KEGIATAN PRAKTIKUM:

A. Membuat dan Mengelola Queue:

1. Mendefinisikan struktur (struct) untuk queue.
2. Membuat queue kosong.
3. Menambahkan beberapa elemen ke dalam queue (enqueue).

B. Operasi Dasar pada Queue:

1. Melakukan operasi enqueue untuk menambahkan elemen ke dalam queue.
2. Melakukan operasi dequeue untuk menghapus elemen dari depan queue.
3. Mengakses elemen di depan queue (front) tanpa menghapusnya.
4. Memeriksa apakah queue kosong atau tidak.
5. Menghitung jumlah elemen dalam queue (ukuran queue).

C. Pencarian Elemen dalam Queue:

1. Mencari elemen tertentu dalam queue.

```
#include <iostream>

//  Mendefinisikan struktur (struct) untuk queue.
struct QueueNode {
   int data;
   QueueNode* next;
};

// Struktur Queue yang menyimpan depan dan belakang queue.
struct Queue {
   QueueNode* front;
```

```
    QueueNode* rear;
};

// Inisialisasi queue kosong.
void initQueue(Queue& queue) {
    queue.front = nullptr;
    queue.rear = nullptr;
}

//  Melakukan operasi enqueue untuk menambahkan elemen ke dalam queue.
void enqueue(Queue& queue, int value) {
    QueueNode* newNode = new QueueNode;
    newNode->data = value;
    newNode->next = nullptr;

    if (queue.rear == nullptr) {
        queue.front = newNode;
        queue.rear = newNode;
    } else {
        queue.rear->next = newNode;
        queue.rear = newNode;
    }
}

//  Melakukan operasi dequeue untuk menghapus elemen dari depan queue.
bool dequeue(Queue& queue) {
    if (queue.front == nullptr) {
        return false; // Queue kosong, tidak ada yang bisa di-dequeue.
    }

    QueueNode* temp = queue.front;
    queue.front = queue.front->next;
    delete temp;
    if (queue.front == nullptr) {
        queue.rear = nullptr; // Jika queue kosong setelah dequeue, reset rear juga.
    }
    return true;
}

//  Mengakses elemen di depan queue (front) tanpa menghapusnya.
int front(const Queue& queue) {
    if (queue.front != nullptr) {
        return queue.front->data;
    } else {
        // Handle jika queue kosong. Anda dapat mengembalikan nilai default atau memunculkan pesan
kesalahan.
        return -1; // Misalnya, -1 sebagai penanda queue kosong.
    }
}
```

```
// Memeriksa apakah queue kosong atau tidak.
bool isEmpty(const Queue& queue) {
    return queue.front == nullptr;
}

// Menghitung jumlah elemen dalam queue (ukuran queue).
int size(const Queue& queue) {
    int count = 0;
    QueueNode* current = queue.front;
    while (current != nullptr) {
        count++;
        current = current->next;
    }
    return count;
}

// Mencari elemen tertentu dalam queue.
bool search(const Queue& queue, int target) {
    QueueNode* current = queue.front;
    while (current != nullptr) {
        if (current->data == target) {
            return true;
        }
        current = current->next;
    }
    return false;
}

int main() {
    Queue myQueue;
    initQueue(myQueue);

    // Menambahkan beberapa elemen ke dalam queue.
    enqueue(myQueue, 10);
    enqueue(myQueue, 20);
    enqueue(myQueue, 30);

    // Melakukan operasi dequeue untuk menghapus elemen dari depan queue.
    dequeue(myQueue);

    // Mengakses elemen di depan queue tanpa menghapusnya.
    int frontElement = front(myQueue);
    if (frontElement != -1) {
        std::cout << "Elemen di depan queue: " << frontElement << std::endl;
    } else {
        std::cout << "Queue kosong." << std::endl;
    }
```

```
    // Memeriksa apakah queue kosong atau tidak.
    bool isEmptyQueue = isEmpty(myQueue);
    if (isEmptyQueue) {
        std::cout << "Queue kosong." << std::endl;
    } else {
        std::cout << "Queue tidak kosong." << std::endl;
    }

    // Menghitung jumlah elemen dalam queue (ukuran queue).
    int queueSize = size(myQueue);
    std::cout << "Ukuran queue: " << queueSize << std::endl;

    // Mencari elemen tertentu dalam queue.
    int target = 20;
    bool found = search(myQueue, target);
    if (found) {
        std::cout << "Elemen " << target << " ditemukan dalam queue." << std::endl;
    } else {
        std::cout << "Elemen " << target << " tidak ditemukan dalam queue." << std::endl;
    }

    return 0;
}
```

4. TUGAS:

Buatlah sebuah program C++ yang mencakup implementasi queue dan fungsi-fungsi berikut:

1. **Enqueue:** Buat fungsi untuk menambahkan elemen ke dalam queue.
2. **Dequeue:** Buat fungsi untuk menghapus elemen dari depan queue.
3. **Front:** Buat fungsi untuk mengakses elemen di depan queue tanpa menghapusnya.
4. **Cek Kosong:** Buat fungsi untuk memeriksa apakah queue kosong atau tidak.
5. **Cek Ukuran:** Buat fungsi untuk menghitung jumlah elemen dalam queue (ukuran queue).
6. **Pencarian Elemen dalam Queue:** Buat fungsi yang menerima nilai sebagai argumen dan mencari elemen dengan nilai tersebut dalam queue.

MODUL V TREES

1. TUJUAN:

- Memahami konsep dasar dari struktur data Trees dalam bahasa pemrograman C++.
- Mampu membuat dan mengelola Trees.
- Mampu melakukan operasi dasar pada Trees seperti penambahan simpul, pencarian simpul, dan pencarian jalur tertentu.

2. DASAR TEORI:

A. Trees: Konsep Dasar

1. **Definisi**: Tree adalah struktur data hirarkis yang terdiri dari node-node yang saling terhubung. Setiap node dalam tree memiliki satu node yang disebut sebagai "parent" (induk) dan nol atau lebih node yang disebut sebagai "children" (anak).
2. **Node**: Node adalah elemen dasar dalam tree. Setiap node memiliki dua komponen utama: data dan referensi ke node-node anaknya.
3. **Root**: Root adalah satu-satunya node dalam tree yang tidak memiliki parent. Ini adalah node paling atas dalam hierarki tree.
4. **Parent dan Children**: Node yang berada di bawah node lain disebut sebagai children, sementara node yang di atasnya disebut sebagai parent. Sebuah node dapat memiliki banyak children.
5. **Ancestor dan Descendant**: Node yang berada di jalur dari root ke suatu node tertentu adalah ancestor (leluhur) dari node tersebut, sementara node tersebut adalah descendant (keturunan) dari leluhurnya.

B. Tipe-tipe Trees

1. **Binary Tree**: Binary tree adalah tree di mana setiap node memiliki paling banyak dua children, yaitu satu child kiri dan satu child kanan.
2. **Binary Search Tree (BST)**: BST adalah tipe khusus dari binary tree di mana setiap node memiliki nilai yang lebih kecil dari semua nilai pada anak-anaknya di subtree kiri dan nilai yang lebih besar dari semua nilai pada anak-anaknya di subtree kanan.
3. **Balanced Tree**: Balanced tree adalah tree di mana tinggi dari subtree kanan dan tinggi dari subtree kiri dari setiap node tidak berbeda lebih dari satu.
4. **Full Tree**: Full tree (atau perfect tree) adalah binary tree di mana setiap node memiliki nol atau dua children. Dalam full tree, semua leaf (node yang tidak memiliki children) berada pada tingkat yang sama.
5. **Complete Tree**: Complete tree adalah binary tree di mana semua tingkat, kecuali mungkin tingkat terakhir, diisi sepenuhnya. Pada tingkat terakhir, semua node terletak sejajar di sebelah kiri.
6. **Binary Heap**: Binary heap adalah tipe khusus dari binary tree yang digunakan dalam implementasi struktur data prioritas.

C. Operasi pada Trees

1. **Traversal**: Traversal adalah proses mengunjungi semua node dalam tree sesuai dengan urutan tertentu. Beberapa metode traversal umum meliputi in-order, pre-order, dan post-order traversal.
2. **Penambahan Node**: Menambahkan node baru ke dalam tree pada lokasi yang sesuai berdasarkan nilai-nilai node.
3. **Penghapusan Node**: Menghapus node tertentu dari tree dengan mempertahankan struktur tree yang valid.
4. **Pencarian**: Mencari node dengan nilai tertentu dalam tree.
5. **Minimum dan Maximum**: Menemukan node dengan nilai minimum dan maksimum dalam tree.

D. Penggunaan Trees

1. **Struktur Data**: Trees digunakan dalam implementasi berbagai struktur data seperti binary search tree (BST), heap, dan struktur data prioritas.
2. **Struktur Hirarkis**: Trees digunakan untuk merepresentasikan struktur hirarkis dalam berbagai aplikasi, termasuk sistem file komputer, pengurai (parser) bahasa pemrograman, dan struktur organisasi perusahaan.
3. **Algoritma Pencarian dan Penelusuran**: Trees digunakan dalam berbagai algoritma pencarian, seperti pencarian dalam kedalaman (depth-first search) dan pencarian dalam lebar (breadth-first search).
4. **Kompresi Data**: Trees digunakan dalam teknik kompresi data seperti Huffman coding untuk mengurangi ukuran data

3. KEGIATAN PRAKTIKUM:

A. Membuat dan Mengelola Trees:

1. Mendefinisikan struktur (struct) untuk simpul Trees.
2. Membuat Trees dengan simpul akar.
3. Menambahkan beberapa simpul ke dalam Trees.

```
#include <iostream>

// 1. Mendefinisikan struktur (struct) untuk simpul Trees.
struct TreeNode {
   int data;
   TreeNode* left;
   TreeNode* right;
};

// 2. Membuat Trees dengan simpul akar.
TreeNode* createTree(int rootData) {
   TreeNode* root = new TreeNode;
   root->data = rootData;
```

```
    root->left = nullptr;
    root->right = nullptr;
    return root;
}

// 3. Menambahkan beberapa simpul ke dalam Trees.
void addNode(TreeNode* parent, int data, bool isLeft) {
    TreeNode* newNode = new TreeNode;
    newNode->data = data;
    newNode->left = nullptr;
    newNode->right = nullptr;
    if (isLeft) {
        parent->left = newNode;
    } else {
        parent->right = newNode;
    }
}
```

B. Operasi Dasar pada Trees:

1. Melakukan operasi penambahan node ke dalam Trees.
2. Melakukan operasi pencarian node dalam Trees.

```
// 1. Melakukan operasi penambahan node ke dalam Trees.
void insertNode(TreeNode* root, int data) {
    if (data < root->data) {
        if (root->left == nullptr) {
            root->left = new TreeNode;
            root->left->data = data;
            root->left->left = nullptr;
            root->left->right = nullptr;
        } else {
            insertNode(root->left, data);
        }
    } else {
        if (root->right == nullptr) {
            root->right = new TreeNode;
            root->right->data = data;
            root->right->left = nullptr;
            root->right->right = nullptr;
        } else {
            insertNode(root->right, data);
        }
    }
}

// 2. Melakukan operasi pencarian node dalam Trees.
bool searchNode(TreeNode* root, int target) {
    if (root == nullptr) {
```

```
        return false;
    }
    if (root->data == target) {
        return true;
    }
    if (target < root->data) {
        return searchNode(root->left, target);
    } else {
        return searchNode(root->right, target);
    }
}
```

C. Traversal Trees:

1. Melakukan traversal pre-order Trees (mengunjungi simpul, kemudian cabang kiri, kemudian cabang kanan).
2. Melakukan traversal in-order Trees (mengunjungi cabang kiri, kemudian simpul, kemudian cabang kanan).
3. Melakukan traversal post-order Trees (mengunjungi cabang kiri, kemudian cabang kanan, kemudian simpul).

```
// 1. Melakukan traversal pre-order Trees.
void preOrderTraversal(TreeNode* root) {
    if (root != nullptr) {
        std::cout << root->data << " ";
        preOrderTraversal(root->left);
        preOrderTraversal(root->right);
    }
}

// 2. Melakukan traversal in-order Trees.
void inOrderTraversal(TreeNode* root) {
    if (root != nullptr) {
        inOrderTraversal(root->left);
        std::cout << root->data << " ";
        inOrderTraversal(root->right);
    }
}

// 3. Melakukan traversal post-order Trees.
void postOrderTraversal(TreeNode* root) {
    if (root != nullptr) {
        postOrderTraversal(root->left);
        postOrderTraversal(root->right);
        std::cout << root->data << " ";
    }
}

int main() {
```

```
    // Membuat Trees dengan simpul akar.
    TreeNode* root = createTree(50);

    // Menambahkan beberapa simpul ke dalam Trees.
    addNode(root, 30, true);
    addNode(root, 70, false);
    addNode(root->left, 20, true);
    addNode(root->left, 40, false);
    addNode(root->right, 60, true);
    addNode(root->right, 80, false);

    // Operasi penambahan node ke dalam Trees.
    insertNode(root, 45);
    insertNode(root, 55);

    // Operasi pencarian node dalam Trees.
    int target = 40;
    bool found = searchNode(root, target);
    if (found) {
        std::cout << "Node dengan nilai " << target << " ditemukan dalam Trees." << std::endl;
    } else {
        std::cout << "Node dengan nilai " << target << " tidak ditemukan dalam Trees." << std::endl;
    }

    // Traversal Trees: pre-order, in-order, post-order.
    std::cout << "Traversal Pre-order: ";
    preOrderTraversal(root);
    std::cout << std::endl;

    std::cout << "Traversal In-order: ";
    inOrderTraversal(root);
    std::cout << std::endl;

    std::cout << "Traversal Post-order: ";
    postOrderTraversal(root);
    std::cout << std::endl;

    return 0;
}
```

4. TUGAS:

Buatlah sebuah program C++ yang mencakup implementasi Trees dan fungsi-fungsi berikut:

1. **Penambahan Node ke Trees:** Buat fungsi untuk menambahkan node ke dalam Trees.
2. **Pencarian Node dalam Trees:** Buat fungsi yang menerima nilai sebagai argumen dan mencari node dengan nilai tersebut dalam Trees.
3. **Traversal Trees:** Buat fungsi-fungsi yang melintasi Trees dengan metode pre-order, in-order, dan post-order.

MODUL VI GRAPHS

1. TUJUAN:

- Memahami konsep dasar dari struktur data Graphs dalam bahasa pemrograman C++.
- Mampu membuat dan mengelola Graphs.
- Mampu melakukan operasi dasar pada Graphs seperti penambahan edge, pencarian vertex, dan traversal.

2. DASAR TEORI:

A. Graphs: Konsep Dasar

1. **Definisi**: Graf adalah struktur data yang terdiri dari sejumlah node (vertices) yang saling terhubung oleh sejumlah edge (edges). Graph digunakan untuk merepresentasikan hubungan atau ketergantungan antar objek.
2. **Node (Vertex)**: Node adalah elemen dasar dalam graph. Setiap node dalam graph dapat memiliki atribut atau informasi tambahan yang terkait dengan itu.
3. **Edge**: Edge adalah koneksi yang menghubungkan dua node dalam graph. Edge dapat memiliki atribut berupa bobot (weight) atau arah (directed).
4. **Directed vs. Undirected**: Dalam graph directed, edge memiliki arah, yang berarti edge menghubungkan node dari satu arah ke node lainnya. Dalam graph undirected, edge tidak memiliki arah, yang berarti hubungan antara node bersifat simetris.
5. **Weighted vs. Unweighted**: Dalam graph weighted, setiap edge memiliki bobot atau nilai yang terkait dengannya. Dalam graph unweighted, edge tidak memiliki bobot.

B. Tipe-tipe Graphs

1. **Directed Graph (Digraph)**: Graph yang memiliki edge berarah, di mana setiap edge memiliki arah dari satu node ke node lainnya.
2. **Undirected Graph**: Graph yang memiliki edge tanpa arah, sehingga hubungan antara dua node bersifat simetris.
3. **Weighted Graph**: Graph di mana setiap edge memiliki bobot atau nilai yang terkait dengannya.
4. **Unweighted Graph**: Graph di mana edge tidak memiliki bobot.
5. **Cyclic vs. Acyclic**: Graph yang memiliki siklus (cycle) disebut sebagai cyclic graph, sedangkan graph yang tidak memiliki siklus disebut sebagai acyclic graph.
6. **Connected vs. Disconnected**: Graph yang memiliki setidaknya satu jalur yang menghubungkan semua node disebut sebagai connected graph, sedangkan graph yang tidak memiliki jalur tersebut disebut sebagai disconnected graph.
7. **Tree**: Graph acyclic yang terhubung dan memiliki tepat satu node yang berperan sebagai root disebut sebagai tree.

C. Operasi pada Graphs

1. **Traversal**: Traversal adalah proses mengunjungi node-node dalam graph sesuai dengan urutan tertentu. Dua metode traversal umum adalah Depth-First Search (DFS) dan Breadth-First Search (BFS).
2. **Pencarian**: Pencarian adalah proses mencari jalur atau koneksi antara dua node tertentu dalam graph.
3. **Minimum Spanning Tree (MST)**: Minimum Spanning Tree adalah subgraph dari graph yang menghubungkan semua node dengan total bobot yang minimal.
4. **Shortest Path**: Shortest Path adalah mencari jalur terpendek antara dua node dalam weighted graph. Algoritma Dijkstra dan Bellman-Ford digunakan untuk mencari jalur terpendek.

D. Penggunaan Graphs

1. **Sistem Jaringan**: Graphs digunakan untuk merepresentasikan berbagai sistem jaringan, seperti jaringan sosial, jaringan komputer, dan jaringan transportasi.
2. **Pemecahan Masalah**: Graphs digunakan dalam pemecahan masalah yang melibatkan ketergantungan antar objek, seperti perencanaan proyek, rute terpendek, dan alokasi sumber daya.
3. **Rekomendasi**: Graphs digunakan dalam pengembangan sistem rekomendasi, seperti rekomendasi film, produk, atau teman.
4. **Grafik dan Permainan**: Graphs digunakan dalam pengembangan permainan video, grafik komputer, dan simulasi fisika.
5. **Analisis Data**: Graphs digunakan dalam analisis data dan pemahaman hubungan antar data.

3. KEGIATAN PRAKTIKUM:

A. Membuat dan Mengelola Graphs:

1. Mendefinisikan struktur (struct) untuk vertex dan edge dalam Graphs.
2. Membuat Graphs dengan sejumlah vertex dan edge.
3. Menambahkan vertex dan edge ke dalam Graphs.

```
#include <iostream>
#include <vector>
#include <queue>

// 1. Mendefinisikan struktur (struct) untuk vertex dalam Graphs.
struct Vertex {
    int data;
    std::vector<int> edges;
};

// 2. Mendefinisikan struktur (struct) untuk edge dalam Graphs.
struct Edge {
    int from;
    int to;
};
```

```
// Struktur Graph yang menyimpan daftar vertex dan edge.
struct Graph {
   std::vector<Vertex> vertices;
   std::vector<Edge> edges;
};

// 3. Membuat Graphs dengan sejumlah vertex.
Graph createGraph(int numVertices) {
   Graph graph;
   graph.vertices.resize(numVertices);
   return graph;
}

// Menambahkan edge ke dalam Graphs.
void addEdge(Graph& graph, int from, int to) {
   Edge edge;
   edge.from = from;
   edge.to = to;
   graph.edges.push_back(edge);

   // Tambahkan edge ke daftar edge pada vertex 'from'.
   graph.vertices[from].edges.push_back(graph.edges.size() - 1);
}
```

B. Operasi Dasar pada Graphs:

1. Melakukan operasi penambahan edge dalam Graphs.
2. Melakukan operasi pencarian vertex dalam Graphs.

```
// 1. Melakukan operasi penambahan edge dalam Graphs.
void insertEdge(Graph& graph, int from, int to) {
   addEdge(graph, from, to);
}

// 2. Melakukan operasi pencarian vertex dalam Graphs.
bool searchVertex(const Graph& graph, int target) {
   for (const Vertex& vertex : graph.vertices) {
      if (vertex.data == target) {
         return true;
      }
   }
   return false;
}
```

C. Traversal Graphs:

1. Melakukan traversal Breadth-First Search (BFS) pada Graphs.
2. Melakukan traversal Depth-First Search (DFS) pada Graphs.

```
// 1. Melakukan traversal Breadth-First Search (BFS) pada Graphs.
void BFS(const Graph& graph, int startVertex) {
    std::vector<bool> visited(graph.vertices.size(), false);
    std::queue<int> queue;

    visited[startVertex] = true;
    queue.push(startVertex);

    while (!queue.empty()) {
        int currentVertex = queue.front();
        queue.pop();
        std::cout << graph.vertices[currentVertex].data << " ";

        for (int edgeIndex : graph.vertices[currentVertex].edges) {
            int adjacentVertex = graph.edges[edgeIndex].to;
            if (!visited[adjacentVertex]) {
                visited[adjacentVertex] = true;
                queue.push(adjacentVertex);
            }
        }
    }
}

// 2. Melakukan traversal Depth-First Search (DFS) pada Graphs.
void DFS(const Graph& graph, int startVertex, std::vector<bool>& visited) {
    if (visited[startVertex]) {
        return;
    }

    visited[startVertex] = true;
    std::cout << graph.vertices[startVertex].data << " ";

    for (int edgeIndex : graph.vertices[startVertex].edges) {
        int adjacentVertex = graph.edges[edgeIndex].to;
        if (!visited[adjacentVertex]) {
            DFS(graph, adjacentVertex, visited);
        }
    }
}

int main() {
    // Membuat Graphs dengan sejumlah vertex.
    Graph myGraph = createGraph(6);

    // Menambahkan edge ke dalam Graphs.
    addEdge(myGraph, 0, 1);
    addEdge(myGraph, 0, 2);
    addEdge(myGraph, 1, 3);
```

```
    addEdge(myGraph, 1, 4);
    addEdge(myGraph, 2, 5);

    // Operasi penambahan edge dalam Graphs.
    insertEdge(myGraph, 3, 5);

    // Operasi pencarian vertex dalam Graphs.
    int targetVertex = 4;
    bool foundVertex = searchVertex(myGraph, targetVertex);
    if (foundVertex) {
        std::cout << "Vertex " << targetVertex << " ditemukan dalam Graph." << std::endl;
    } else {
        std::cout << "Vertex " << targetVertex << " tidak ditemukan dalam Graph." << std::endl;
    }

    // Traversal BFS pada Graphs.
    std::cout << "Traversal BFS: ";
    BFS(myGraph, 0);
    std::cout << std::endl;

    // Traversal DFS pada Graphs.
    std::cout << "Traversal DFS: ";
    std::vector<bool> visited(myGraph.vertices.size(), false);
    DFS(myGraph, 0, visited);
    std::cout << std::endl;

    return 0;
}
```

4. TUGAS:

Buatlah sebuah program C++ yang mencakup implementasi Graphs dan fungsi-fungsi berikut:

1. **Penambahan Edge:** Buat fungsi untuk menambahkan edge antara dua vertex dalam Graphs.
2. **Pencarian Vertex:** Buat fungsi yang menerima nama vertex sebagai argumen dan mencari vertex dalam Graphs.
3. **Traversal BFS:** Buat fungsi untuk melakukan Breadth-First Search (BFS) traversal pada Graphs.
4. **Traversal DFS:** Buat fungsi untuk melakukan Depth-First Search (DFS) traversal pada Graphs.

MODUL VII SORTING

1. TUJUAN:

- Memahami konsep dasar algoritma pengurutan dalam bahasa pemrograman C++.
- Mampu mengimplementasikan dan memahami algoritma pengurutan seperti Bubble Sort, Selection Sort, dan Merge Sort.
- Mampu memilih algoritma pengurutan yang sesuai untuk situasi tertentu.

2. DASAR TEORI:

A. Sorting: Konsep Dasar

1. **Definisi**: Sorting adalah proses pengurutan elemen dalam suatu koleksi atau struktur data dalam urutan tertentu, seperti urutan numerik atau alfabetis.
2. **Tujuan**: Tujuan utama dari sorting adalah untuk mengatur data agar lebih mudah diakses, dicari, dan dianalisis. Ini membantu meningkatkan efisiensi dalam pencarian data.
3. **Kriteria Pengurutan**: Elemen-elemen dalam data dapat diurutkan berdasarkan berbagai kriteria, termasuk nilai numerik (ascending atau descending), urutan alfabetis (ascending atau descending), tanggal, dan banyak lagi.

B. Algoritma Sorting Umum

1. **Bubble Sort**: Bubble Sort adalah algoritma pengurutan sederhana yang membandingkan dan menukar pasangan elemen yang berdekatan sampai data terurut.
2. **Selection Sort**: Selection Sort memilih elemen terkecil dari data dan menukarnya dengan elemen pertama. Kemudian, elemen terkecil berikutnya dipilih dari data yang tersisa, dan begitu seterusnya.
3. **Insertion Sort**: Insertion Sort mengambil setiap elemen dari data satu per satu dan memasukkannya ke dalam posisi yang benar di antara elemen-elemen yang telah diurutkan.
4. **Merge Sort**: Merge Sort adalah algoritma pengurutan yang menggunakan pendekatan rekursif. Ini membagi data menjadi dua bagian, mengurutkan masing-masing bagian, dan menggabungkannya kembali dalam urutan yang benar.
5. **Quick Sort**: Quick Sort adalah algoritma pengurutan yang juga menggunakan pendekatan rekursif. Ini memilih elemen tertentu sebagai pivot, mempartisi data menjadi dua bagian (lebih kecil dan lebih besar dari pivot), dan mengurutkan masing-masing bagian.
6. **Heap Sort**: Heap Sort menggunakan struktur data heap untuk mengurutkan data. Data diubah menjadi max-heap (atau min-heap), dan elemen terbesar (atau terkecil) diambil secara berulang hingga data terurut.

C. Kompleksitas Waktu Pengurutan

1. **Average Case vs. Worst Case**: Algoritma sorting memiliki kompleksitas waktu yang berbeda dalam kasus rata-rata dan kasus terburuk. Beberapa algoritma memiliki kinerja yang baik dalam kasus tertentu tetapi buruk dalam kasus lain.
2. **Notasi Big O**: Kompleksitas waktu algoritma sorting sering diukur dengan notasi Big O (misalnya, O(n^2) untuk Bubble Sort atau O(n log n) untuk Merge Sort).
3. **Stabilitas**: Beberapa algoritma pengurutan adalah stabil, yang berarti elemen-elemen yang sama akan tetap dalam urutan relatif mereka jika mereka memiliki nilai yang sama.

D. Penggunaan Sorting

1. **Basis untuk Algoritma Lain**: Pengurutan adalah langkah awal dalam banyak algoritma dan tugas lainnya. Sebagai contoh, pengurutan sering digunakan dalam pencarian biner.
2. **Database dan Sistem Penyimpanan**: Sorting digunakan dalam sistem database dan penyimpanan data untuk mengoptimalkan pencarian data.
3. **Visualisasi Data**: Pengurutan juga digunakan dalam visualisasi data, seperti grafik batang yang diurutkan dalam urutan tertentu.
4. **Pengolahan Data**: Sorting adalah bagian penting dari pengolahan data dalam berbagai aplikasi, termasuk analisis statistik dan ilmu data.

3. KEGIATAN PRAKTIKUM:

A. Implementasi Algoritma Pengurutan:

1. Implementasikan algoritma Bubble Sort dalam bahasa pemrograman C++.

```
// Implementasi algoritma Bubble Sort.
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Tukar elemen jika elemen saat ini lebih besar dari elemen berikutnya.
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

2. Implementasikan algoritma Selection Sort dalam bahasa pemrograman C++.

```
// Implementasi algoritma Selection Sort.
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
```

```
            minIndex = j;
        }
      }
      // Tukar elemen terkecil dengan elemen pertama dari iterasi saat ini.
      int temp = arr[i];
      arr[i] = arr[minIndex];
      arr[minIndex] = temp;
    }
}
```

3. Implementasikan algoritma Merge Sort dalam bahasa pemrograman C++.

```
// Fungsi untuk menggabungkan dua subarray terurut.
void merge(int arr[], int left, int middle, int right) {
    int n1 = middle - left + 1;
    int n2 = right - middle;

    // Buat array temp untuk subarray kiri dan kanan.
    int leftArr[n1];
    int rightArr[n2];

    // Salin data ke dalam array temp leftArr[] dan rightArr[].
    for (int i = 0; i < n1; i++) {
        leftArr[i] = arr[left + i];
    }
    for (int i = 0; i < n2; i++) {
        rightArr[i] = arr[middle + 1 + i];
    }

    // Gabungkan dua subarray menjadi satu array terurut.
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k] = leftArr[i];
            i++;
        } else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }

    // Salin elemen-elemen yang tersisa dari leftArr[], jika ada.
    while (i < n1) {
        arr[k] = leftArr[i];
        i++;
        k++;
    }

    // Salin elemen-elemen yang tersisa dari rightArr[], jika ada.
```

```
    while (j < n2) {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}

// Implementasi algoritma Merge Sort.
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int middle = left + (right - left) / 2;

        // Panggil rekursi untuk dua subarray yang lebih kecil.
        mergeSort(arr, left, middle);
        mergeSort(arr, middle + 1, right);

        // Gabungkan dua subarray.
        merge(arr, left, middle, right);
    }
}
```

B. Pengujian Algoritma Pengurutan:

1. Buatlah array dengan elemen-elemen acak.
2. Terapkan algoritma pengurutan yang telah Anda implementasikan pada array tersebut.
3. Bandingkan hasil pengurutan dengan array yang sudah terurut secara benar.

```
#include <iostream>
#include <ctime>
#include <cstdlib>
#include <algorithm>

int main() {
    const int n = 10; // Ukuran array

    // Buat array dengan elemen-elemen acak.
    int arr[n];
    std::srand(static_cast<unsigned>(std::time(nullptr)));
    for (int i = 0; i < n; i++) {
        arr[i] = std::rand() % 100; // Angka acak antara 0 dan 99.
    }

    std::cout << "Array sebelum pengurutan: ";
    for (int i = 0; i < n; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
```

```
    // Salin array ke array yang akan diurutkan untuk perbandingan.
    int sortedArr[n];
    std::copy(arr, arr + n, sortedArr);

    // Terapkan algoritma pengurutan yang telah diimplementasikan.
    // Contoh untuk Bubble Sort:
    // bubbleSort(arr, n);

    // Contoh untuk Selection Sort:
    // selectionSort(arr, n);

    // Contoh untuk Merge Sort:
    mergeSort(arr, 0, n - 1);

    // Bandingkan hasil pengurutan dengan array yang sudah terurut secara benar.
    std::sort(sortedArr, sortedArr + n);

    bool isSorted = std::equal(arr, arr + n, sortedArr);

    if (isSorted) {
        std::cout << "Array setelah pengurutan: ";
        for (int i = 0; i < n; i++) {
            std::cout << arr[i] << " ";
        }
        std::cout << std::endl;
    } else {
        std::cout << "Algoritma pengurutan tidak berhasil." << std::endl;
    }

    return 0;
}
```

4. TUGAS:

Buatlah sebuah program C++ yang mencakup implementasi algoritma pengurutan dan fungsi-fungsi berikut:

1. **Bubble Sort:** Buat fungsi untuk mengurutkan array menggunakan algoritma Bubble Sort.
2. **Selection Sort:** Buat fungsi untuk mengurutkan array menggunakan algoritma Selection Sort.
3. **Merge Sort:** Buat fungsi untuk mengurutkan array menggunakan algoritma Merge Sort.
4. **Pengujian:** Buat fungsi untuk menghasilkan array acak dan menguji ketiga algoritma pengurutan yang telah Anda implementasikan. Tampilkan hasil pengurutan dan waktu yang dibutuhkan oleh masing-masing algoritma.

MODUL VIII SEARCHING

1. TUJUAN:

- Memahami konsep dasar algoritma pencarian dalam bahasa pemrograman C++.
- Mampu mengimplementasikan dan memahami algoritma pencarian seperti Linear Search, Binary Search, dan Hashing.
- Mampu memilih algoritma pencarian yang sesuai untuk situasi tertentu.

2. DASAR TEORI:

A. Searching: Konsep Dasar

1. **Definisi**: Searching adalah proses mencari elemen tertentu dalam suatu koleksi data atau struktur data untuk menemukan apakah elemen tersebut ada atau tidak, serta menentukan lokasinya jika ada.
2. **Tujuan**: Tujuan utama dari searching adalah menemukan informasi yang diperlukan dengan efisien tanpa harus memeriksa setiap elemen dalam koleksi data.
3. **Kriteria Pencarian**: Pencarian dapat dilakukan berdasarkan berbagai kriteria, termasuk nilai numerik, teks, atau atribut lainnya.

B. Metode Searching Umum

1. **Linear Search**: Linear Search adalah metode searching yang sederhana. Ini memeriksa elemen-elemen satu per satu secara berurutan sampai elemen yang dicari ditemukan atau sampai seluruh koleksi data diperiksa.
2. **Binary Search**: Binary Search adalah metode searching yang lebih efisien, tetapi hanya berlaku untuk data yang sudah terurut. Ini membandingkan elemen tengah dengan elemen yang dicari dan secara berulang membagi data menjadi dua bagian sampai elemen ditemukan atau data habis.
3. **Hashing**: Hashing adalah metode yang menggunakan fungsi hash untuk mengindeks dan mencari data dalam struktur data seperti tabel hash. Ini memiliki kompleksitas waktu rata-rata O(1) dalam pencarian.
4. **Interpolation Search**: Interpolation Search adalah variasi dari binary search yang digunakan ketika data dalam koleksi berada dalam urutan linier. Ini menggunakan interpolasi untuk memperkirakan posisi elemen yang dicari.

C. Kompleksitas Waktu Pencarian

1. **Average Case vs. Worst Case**: Kompleksitas waktu dalam searching sering dibedakan antara kasus rata-rata dan kasus terburuk. Binary Search memiliki kompleksitas waktu rata-rata O(log n), sementara Linear Search memiliki kompleksitas waktu terburuk O(n).
2. **Notasi Big O**: Kompleksitas waktu searching sering diukur dengan notasi Big O untuk memberikan estimasi atas jumlah operasi yang diperlukan dalam pencarian terburuk.

D. Penggunaan Searching

1. **Basis untuk Algoritma Lain**: Searching adalah langkah awal dalam banyak algoritma dan tugas lainnya. Sebagai contoh, searching sering digunakan dalam algoritma pengurutan.
2. **Pengolahan Data**: Searching adalah bagian penting dari pengolahan data dalam berbagai aplikasi, termasuk pencarian data dalam database, pengambilan data dari file, dan analisis data.
3. **Penemuan Informasi**: Searching digunakan dalam mesin pencari web untuk menemukan halaman web yang sesuai dengan kueri pengguna.
4. **Penyaringan Data**: Searching digunakan dalam filter data dalam berbagai aplikasi, seperti filter pencarian email atau filter konten web.

3. KEGIATAN PRAKTIKUM:

A. Implementasi Algoritma Pencarian:

1. Implementasikan algoritma Linear Search dalam bahasa pemrograman C++.

```
#include <iostream>

// Implementasi algoritma Linear Search.
int linearSearch(int arr[], int n, int target) {
   for (int i = 0; i < n; i++) {
      if (arr[i] == target) {
         return i; // Mengembalikan indeks elemen jika ditemukan.
      }
   }
   return -1; // Mengembalikan -1 jika elemen tidak ditemukan.
}
```

2. Implementasikan algoritma Binary Search dalam bahasa pemrograman C++.

```
#include <iostream>

// Implementasi algoritma Binary Search (array harus terurut).
int binarySearch(int arr[], int n, int target) {
   int left = 0;
   int right = n - 1;

   while (left <= right) {
      int middle = left + (right - left) / 2;

      if (arr[middle] == target) {
         return middle; // Mengembalikan indeks elemen jika ditemukan.
      }

      if (arr[middle] < target) {
         left = middle + 1;
      } else {
         right = middle - 1;
      }
```

```
    }

    return -1; // Mengembalikan -1 jika elemen tidak ditemukan.
}
```

3. Implementasikan algoritma Hashing untuk pencarian sederhana dalam bahasa pemrograman C++.

```
#include <iostream>
#include <unordered_map>

// Implementasi algoritma Hashing untuk pencarian sederhana.
int hashingSearch(std::unordered_map<int, int>& map, int target) {
    auto it = map.find(target);
    if (it != map.end()) {
        return it->second; // Mengembalikan nilai jika ditemukan.
    } else {
        return -1; // Mengembalikan -1 jika elemen tidak ditemukan.
    }
}
```

B. Pengujian Algoritma Pencarian:

1. Buatlah array atau struktur data yang sesuai untuk setiap algoritma yang Anda implementasikan.
2. Lakukan pencarian elemen tertentu menggunakan masing-masing algoritma pencarian.
3. Bandingkan hasil pencarian dengan hasil yang diharapkan.

```
#include <iostream>
#include <unordered_map>

int main() {
    const int n = 10; // Ukuran array
    int arr[n] = {4, 2, 8, 1, 5, 9, 7, 3, 6, 0};
    int target = 7;

    // 1. Pengujian Linear Search:
    int linearResult = linearSearch(arr, n, target);
    if (linearResult != -1) {
        std::cout << "Linear Search: Elemen " << target << " ditemukan di indeks " << linearResult <<
std::endl;
    } else {
        std::cout << "Linear Search: Elemen " << target << " tidak ditemukan." << std::endl;
    }

    // 2. Pengujian Binary Search (array harus terurut):
    std::sort(arr, arr + n);
    int binaryResult = binarySearch(arr, n, target);
```

```
    if (binaryResult != -1) {
        std::cout << "Binary Search: Elemen " << target << " ditemukan di indeks " << binaryResult <<
std::endl;
    } else {
        std::cout << "Binary Search: Elemen " << target << " tidak ditemukan." << std::endl;
    }

    // 3. Pengujian Hashing (pencarian sederhana):
    std::unordered_map<int, int> hashMap;
    hashMap[4] = 40;
    hashMap[7] = 70;
    hashMap[2] = 20;
    hashMap[1] = 10;

    int hashingResult = hashingSearch(hashMap, target);
    if (hashingResult != -1) {
        std::cout << "Hashing Search: Nilai " << target << " ditemukan: " << hashingResult << std::endl;
    } else {
        std::cout << "Hashing Search: Nilai " << target << " tidak ditemukan." << std::endl;
    }

    return 0;
}
```

4. **TUGAS:**

   Buatlah sebuah program C++ yang mencakup implementasi algoritma pencarian dan fungsi-fungsi berikut:

   1. **Linear Search:** Buat fungsi untuk melakukan pencarian elemen menggunakan algoritma Linear Search.
   2. **Binary Search:** Buat fungsi untuk melakukan pencarian elemen menggunakan algoritma Binary Search. Pastikan data yang digunakan sudah terurut.
   3. **Hashing:** Buat struktur data dan fungsi-fungsi untuk melakukan pencarian elemen menggunakan algoritma Hashing. Buat juga fungsi hash yang sesuai untuk struktur data Anda.
   4. **Pengujian:** Buat beberapa contoh data dan lakukan pencarian menggunakan ketiga algoritma yang telah Anda implementasikan. Tampilkan hasil pencarian.

MODUL IX DYNAMIC PROGRAMMING

1. TUJUAN:

- Memahami konsep dasar dari pemrograman dinamis dalam bahasa pemrograman C++.
- Mampu mengimplementasikan dan memahami pendekatan pemrograman dinamis untuk menyelesaikan masalah optimisasi dan perhitungan kombinatorial.
- Mampu mengenali masalah yang dapat diselesaikan dengan pendekatan pemrograman dinamis.

2. DASAR TEORI:

A. Dynamic Programming: Konsep Dasar

1. **Definisi**: Dynamic Programming adalah teknik dalam ilmu komputer yang digunakan untuk menyelesaikan masalah yang dapat dipecahkan menjadi submasalah yang lebih kecil. Solusi untuk setiap submasalah disimpan dan digunakan kembali untuk menghindari perhitungan berulang.
2. **Tujuan**: Tujuan dari dynamic programming adalah mengoptimalkan waktu komputasi dengan cara menghindari perhitungan berulang, terutama dalam masalah yang memiliki struktur submasalah yang tumpang tindih.
3. **Ciri-ciri**: Masalah yang dapat dipecahkan dengan teknik dynamic programming umumnya memenuhi dua ciri-ciri utama:
   - **Optimal Substructure**: Solusi optimal dari masalah utuh dapat dibangun dari solusi optimal submasalah yang lebih kecil.
   - **Overlapping Subproblems**: Masalah utuh sering kali memiliki submasalah yang sama yang perlu dihitung kembali.

B. Elemen-elemen Dynamic Programming

1. **Memoization**: Memoization adalah teknik penyimpanan hasil perhitungan submasalah untuk mencegah perhitungan ulang. Biasanya, hasil submasalah disimpan dalam tabel atau array.
2. **Bottom-Up Approach**: Bottom-up approach adalah pendekatan dalam dynamic programming di mana solusi untuk submasalah yang lebih kecil dihitung terlebih dahulu, dan hasilnya digunakan untuk menghitung solusi masalah yang lebih besar.
3. **Top-Down Approach**: Top-down approach adalah pendekatan dalam dynamic programming di mana masalah utuh dipecahkan menjadi submasalah yang lebih kecil secara rekursif, dan solusi submasalah disimpan untuk mencegah perhitungan ulang.

C. Contoh Kasus Pemrograman Dinamis

1. **Fibonacci Sequence**: Fibonacci sequence adalah contoh klasik penggunaan dynamic programming. Dalam algoritma rekursif sederhana, perhitungan untuk nilai yang sama dihitung ulang berkali-kali. Dengan memoization, perhitungan yang sudah dilakukan hanya dilakukan sekali.

2. **Longest Common Subsequence**: Dalam masalah ini, kita mencari urutan terpanjang yang sama antara dua urutan. Dengan dynamic programming, kita dapat menghitung panjang dari subsequence terpanjang secara efisien.
3. **Knapsack Problem**: Knapsack problem melibatkan pemilihan elemen dari satu set elemen dengan batasan kapasitas tertentu untuk mencapai nilai maksimum. Dynamic programming dapat digunakan untuk menemukan solusi optimal.
4. **Shortest Path**: Dalam masalah mencari jalur terpendek antara dua titik dalam graf, algoritma seperti Dijkstra dan Bellman-Ford menggunakan dynamic programming untuk menghitung jalur terpendek.

D. Keuntungan dan Keterbatasan Dynamic Programming

Keuntungan:

- Memungkinkan pengoptimalan solusi masalah dengan efisiensi tinggi.
- Mencegah perhitungan berulang yang dapat memakan waktu dalam masalah dengan submasalah yang tumpang tindih.

Keterbatasan:

- Memerlukan ruang penyimpanan tambahan untuk menyimpan hasil perhitungan submasalah.
- Tidak selalu efektif dalam semua masalah; tidak semua masalah dapat dipecahkan dengan teknik dynamic programming.

3. KEGIATAN PRAKTIKUM:

A. Implementasi Pemrograman Dinamis:

1. Implementasikan pendekatan pemrograman dinamis untuk menyelesaikan masalah klasik seperti Fibonacci dengan rekursi sederhana dan rekursi dengan memoisasi (penyimpanan hasil perhitungan).

```
#include <iostream>
#include <vector>

// Implementasi Fibonacci dengan rekursi sederhana (tidak efisien).
int fibonacciRecursive(int n) {
    if (n <= 1) {
        return n;
    }
    return fibonacciRecursive(n - 1) + fibonacciRecursive(n - 2);
}

// Implementasi Fibonacci dengan rekursi dan memoisasi.
std::vector<int> memoizationTable;

int fibonacciMemoization(int n) {
    if (n <= 1) {
        return n;
    }
```

```
    if (memoizationTable.size() <= n) {
        memoizationTable.resize(n + 1, -1);
    }

    if (memoizationTable[n] == -1) {
        memoizationTable[n] = fibonacciMemoization(n - 1) + fibonacciMemoization(n - 2);
    }
    return memoizationTable[n];
}
```

2. Implementasikan pendekatan pemrograman dinamis untuk menyelesaikan masalah pencarian deret Fibonacci dengan pendekatan bottom-up (tanpa rekursi).

```
#include <iostream>
#include <vector>

// Implementasi Fibonacci dengan pendekatan bottom-up (tanpa rekursi).
int fibonacciBottomUp(int n) {
    std::vector<int> fib(n + 1);
    fib[0] = 0;
    fib[1] = 1;

    for (int i = 2; i <= n; i++) {
        fib[i] = fib[i - 1] + fib[i - 2];
    }
    return fib[n];
}
```

B. Pengujian dan Analisis Kinerja:

1. Ukur waktu eksekusi untuk mencari deret Fibonacci dengan pendekatan rekursi sederhana dan pemrograman dinamis dengan memoisasi.
2. Bandingkan hasil dan waktu eksekusi dari kedua pendekatan tersebut.

```
#include <iostream>
#include <chrono>

int main() {
    const int n = 40; // Nomer deret Fibonacci yang ingin dihitung

    // 1. Mengukur waktu eksekusi dengan rekursi sederhana dan memoisasi.
    auto start = std::chrono::high_resolution_clock::now();
    int result1 = fibonacciRecursive(n);
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration1 = end - start;

    std::cout << "Hasil Fibonacci dengan rekursi sederhana: " << result1 << std::endl;
```

```
    std::cout << "Waktu eksekusi: " << duration1.count() << " detik" << std::endl;

    memoizationTable.clear(); // Mengosongkan memoisasi tabel.

    start = std::chrono::high_resolution_clock::now();
    int result2 = fibonacciMemoization(n);
    end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration2 = end - start;

    std::cout << "Hasil Fibonacci dengan rekursi dan memoisasi: " << result2 << std::endl;
    std::cout << "Waktu eksekusi: " << duration2.count() << " detik" << std::endl;

    // 2. Mengukur waktu eksekusi dengan pendekatan bottom-up.
    start = std::chrono::high_resolution_clock::now();
    int result3 = fibonacciBottomUp(n);
    end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration3 = end - start;

    std::cout << "Hasil Fibonacci dengan pendekatan bottom-up: " << result3 << std::endl;
    std::cout << "Waktu eksekusi: " << duration3.count() << " detik" << std::endl;

    return 0;
}
```

4. **TUGAS:**

   Buatlah sebuah program C++ yang mencakup implementasi pendekatan pemrograman dinamis dan fungsi-fungsi berikut:

   1. **Fibonacci dengan Rekursi Sederhana:** Buat fungsi rekursi sederhana untuk mencari deret Fibonacci.
   2. **Fibonacci dengan Memoisasi:** Buat fungsi rekursi dengan memoisasi (penyimpanan hasil perhitungan) untuk mencari deret Fibonacci.
   3. **Fibonacci dengan Pendekatan Bottom-Up:** Buat fungsi dengan pendekatan bottom-up untuk mencari deret Fibonacci tanpa rekursi.
   4. **Pengujian:** Buat program utama yang mengukur waktu eksekusi dan membandingkan hasil antara tiga pendekatan di atas.

PROYEK AKHIR

A. Proyek 1: Manajemen Tugas dengan To-Do List

Deskripsi Singkat: Buat aplikasi manajemen tugas yang memungkinkan pengguna membuat, mengedit, dan menghapus tugas. Anda akan mengimplementasikan struktur data seperti array, linked list, dan hash table untuk menyimpan dan mengelola tugas.

1. Tujuan:

- Memahami konsep struktur data seperti array dan linked list.
- Mengimplementasikan operasi CRUD (Create, Read, Update, Delete) untuk tugas.
- Memahami penggunaan hash table untuk pencarian dan pengindeksan.

2. Dasar Teori:

- Struktur data: Array, Linked List, Hash Table
- Operasi CRUD (Create, Read, Update, Delete)

3. Komponen Utama:

- Membuat tugas baru
- Menampilkan daftar tugas
- Mengedit tugas
- Menghapus tugas
- Pencarian tugas menggunakan hash table

B. Proyek 2: Sistem Pencarian Film

Deskripsi Singkat: Buat sistem pencarian film yang memungkinkan pengguna mencari film berdasarkan judul, aktor, atau genre. Anda akan mengimplementasikan struktur data seperti linked list, trees, dan graphs untuk mengelola data film.

1. Tujuan:

- Memahami konsep struktur data seperti linked list, trees, dan graphs.
- Mengimplementasikan pencarian dan pemutakhiran data.
- Membangun hubungan antara film, aktor, dan genre menggunakan graphs.

2. Dasar Teori:

- Struktur data: Linked List, Trees, Graphs
- Pencarian data
- Pemutakhiran data
- Representasi grafik hubungan antara film, aktor, dan genre

3. Komponen Utama:

- Menambahkan film baru
- Menampilkan daftar film

- Mencari film berdasarkan judul, aktor, atau genre
- Menambahkan dan menghubungkan aktor ke film
- Membangun grafik hubungan antara film, aktor, dan genre

C. Proyek 3: Pemantauan Stok Barang

Deskripsi Singkat: Buat sistem pemantauan stok barang untuk toko. Anda akan mengimplementasikan algoritma pengurutan untuk melihat barang yang hampir habis stoknya dan algoritma pencarian untuk mencari barang dalam stok.

1. Tujuan:

- Memahami konsep algoritma pengurutan dan algoritma pencarian.
- Mengimplementasikan algoritma pengurutan untuk barang stok.
- Mengimplementasikan algoritma pencarian untuk mencari barang dalam stok.

2. Dasar Teori:

- Algoritma Pengurutan (misalnya: Quick Sort atau Merge Sort)
- Algoritma Pencarian (misalnya: Binary Search)
- Manajemen stok barang

3. Komponen Utama:

- Menambahkan barang ke stok
- Melihat daftar barang dalam stok
- Mengurutkan barang berdasarkan jumlah stok
- Mencari barang berdasarkan nama menggunakan algoritma pencarian

D. Proyek 4: Sistem Pengelolaan Karyawan

Deskripsi Singkat: Buat sistem pengelolaan karyawan untuk perusahaan. Anda akan mengimplementasikan pemrograman dinamis untuk menghitung bonus karyawan dan membuat daftar karyawan dengan hierarki.

1. Tujuan:

- Memahami konsep pemrograman dinamis dalam perhitungan bonus.
- Mengimplementasikan pemrograman dinamis untuk menghitung bonus.
- Membangun hierarki karyawan menggunakan trees.

2. Dasar Teori:

- Pemrograman Dinamis
- Hierarki Karyawan (Tree)

3. Komponen Utama:

- Menambahkan karyawan baru
- Melihat daftar karyawan
- Menghitung bonus karyawan menggunakan pemrograman dinamis

- Membangun hierarki karyawan dengan atasan dan bawahan

E. Proyek 5: Sistem Pendataan Perpustakaan

Deskripsi Singkat: Buat sistem pendataan perpustakaan yang lengkap, yang mencakup manajemen buku, peminjaman, pengembalian, dan pencarian buku. Anda akan menggabungkan konsep dan teknik dari semua bab yang telah dipelajari.

1. Tujuan:

- Menggabungkan konsep dan teknik dari semua bab yang telah dipelajari.
- Mengimplementasikan struktur data, algoritma pengurutan, algoritma pencarian, dan pemrograman dinamis dalam satu proyek.

2. Dasar Teori:

- Struktur Data (Array, Linked List, Trees, Graphs)
- Algoritma Pengurutan dan Pencarian
- Pemrograman Dinamis

3. Komponen Utama:

- Manajemen buku (tambah, edit, hapus)
- Peminjaman dan pengembalian buku
- Pencarian buku berdasarkan judul, penulis, atau kategori
- Perhitungan denda keterlambatan pengembalian

SOURCE CODE TUGAS

1. ARRAY

```
#include <iostream>
#include <vector>

using namespace std;

// Fungsi untuk menggabungkan dua array menjadi satu array baru
vector<int> gabungArray(vector<int>& arr1, vector<int>& arr2) {
   vector<int> gabungan;
   gabungan.insert(gabungan.end(), arr1.begin(), arr1.end());
   gabungan.insert(gabungan.end(), arr2.begin(), arr2.end());
   return gabungan;
}

// Fungsi untuk menjumlahkan elemen-elemen dalam array
int jumlahArray(vector<int>& arr) {
   int total = 0;
   for (int i = 0; i < arr.size(); i++) {
      total += arr[i];
   }
   return total;
}

// Fungsi untuk mencari nilai maksimum dalam array
int maksimumArray(vector<int>& arr) {
   int maksimum = arr[0];
   for (int i = 1; i < arr.size(); i++) {
      if (arr[i] > maksimum) {
         maksimum = arr[i];
      }
   }
   return maksimum;
}

// Fungsi untuk mengganti elemen dalam array
void gantiElemenArray(vector<int>& arr, int indeks, int nilaiBaru) {
   if (indeks >= 0 && indeks < arr.size()) {
      arr[indeks] = nilaiBaru;
   }
}

int main() {
   // Contoh array
   vector<int> array1 = {1, 2, 3};
   vector<int> array2 = {4, 5, 6};

   // Menggabungkan array
```

```
    vector<int> gabungan = gabungArray(array1, array2);
    cout << "Hasil penggabungan array: ";
    for (int i = 0; i < gabungan.size(); i++) {
        cout << gabungan[i] << " ";
    }
    cout << endl;

    // Menjumlahkan elemen dalam array
    int total = jumlahArray(array1);
    cout << "Total elemen array1: " << total << endl;

    // Mencari nilai maksimum dalam array
    int maks = maksimumArray(array2);
    cout << "Nilai maksimum dalam array2: " << maks << endl;

    // Mengganti elemen dalam array
    gantiElemenArray(array1, 1, 10);
    cout << "Array1 setelah mengganti elemen ke-1: ";
    for (int i = 0; i < array1.size(); i++) {
        cout << array1[i] << " ";
    }
    cout << endl;

    return 0;
}
```

2. LINKED LIST

```
#include <iostream>

using namespace std;

// Struktur simpul Linked List
struct Node {
    int data;
    Node* next;

    Node(int val) : data(val), next(nullptr) {}
};

// Kelas untuk Linked List
class LinkedList {
private:
    Node* head;

public:
    // Konstruktor untuk Linked List
    LinkedList() : head(nullptr) {}

    // Fungsi untuk menyisipkan elemen di awal Linked List
```

```
void sisipDiAwal(int val) {
   Node* newNode = new Node(val);
   newNode->next = head;
   head = newNode;
}

// Fungsi untuk menyisipkan elemen di akhir Linked List
void sisipDiAkhir(int val) {
   Node* newNode = new Node(val);

   if (!head) {
      head = newNode;
      return;
   }

   Node* curr = head;
   while (curr->next) {
      curr = curr->next;
   }

   curr->next = newNode;
}

// Fungsi untuk menghapus elemen di awal Linked List
void hapusDiAwal() {
   if (!head) {
      return;
   }

   Node* temp = head;
   head = head->next;
   delete temp;
}

// Fungsi untuk menghapus elemen di akhir Linked List
void hapusDiAkhir() {
   if (!head) {
      return;
   }

   if (!head->next) {
      delete head;
      head = nullptr;
      return;
   }

   Node* curr = head;
   while (curr->next->next) {
      curr = curr->next;
```

```
        }

        delete curr->next;
        curr->next = nullptr;
    }

    // Fungsi untuk mencari elemen dalam Linked List
    bool cariElemen(int val) {
        Node* curr = head;
        while (curr) {
            if (curr->data == val) {
                return true;
            }
            curr = curr->next;
        }
        return false;
    }

    // Fungsi untuk melintasi dan mencetak seluruh Linked List
    void cetakLinkedList() {
        Node* curr = head;
        while (curr) {
            cout << curr->data << " ";
            curr = curr->next;
        }
        cout << endl;
    }

    // Destruktor untuk menghapus Linked List
    ~LinkedList() {
        while (head) {
            Node* temp = head;
            head = head->next;
            delete temp;
        }
    }
};

int main() {
    LinkedList linkedList;

    linkedList.sisipDiAwal(3);
    linkedList.sisipDiAwal(2);
    linkedList.sisipDiAwal(1);

    linkedList.sisipDiAkhir(4);
    linkedList.sisipDiAkhir(5);

    cout << "Linked List setelah sisip di awal dan akhir: ";
```

```
    linkedList.cetakLinkedList();

    linkedList.hapusDiAwal();
    linkedList.hapusDiAkhir();

    cout << "Linked List setelah hapus di awal dan akhir: ";
    linkedList.cetakLinkedList();

    bool ditemukan = linkedList.cariElemen(3);
    cout << "Elemen 3 ditemukan dalam Linked List: " << (ditemukan ? "Ya" : "Tidak") << endl;

    return 0;
}
```

3. STACK

```
#include <iostream>
#include <stack>

using namespace std;

int main() {
    stack<int> myStack; // Mendeklarasikan stack dengan tipe int

    // Menambahkan elemen ke dalam stack (push)
    myStack.push(1);
    myStack.push(2);
    myStack.push(3);

    cout << "Elemen-elemen dalam stack:" << endl;

    // Mengakses elemen teratas (top) stack
    while (!myStack.empty()) {
        cout << myStack.top() << " ";
        myStack.pop(); // Menghapus elemen teratas (pop)
    }

    cout << endl;

    return 0;
}
```

4. QUEUE

```
#include <iostream>
#include <queue>

using namespace std;

int main() {
```

```
    queue<int> myQueue; // Mendeklarasikan queue dengan tipe int

    // Menambahkan elemen ke dalam queue (enqueue)
    myQueue.push(1);
    myQueue.push(2);
    myQueue.push(3);

    cout << "Elemen-elemen dalam queue:" << endl;

    // Mengakses elemen di depan queue (front) dan menghapusnya (dequeue)
    while (!myQueue.empty()) {
        cout << myQueue.front() << " ";
        myQueue.pop(); // Menghapus elemen dari depan queue
    }

    cout << endl;

    return 0;
}
```

5. TREES

```
#include <iostream>

using namespace std;

// Struktur simpul Binary Search Tree (BST)
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

// Kelas untuk Binary Search Tree (BST)
class BinarySearchTree {
private:
    Node* root;

public:
    BinarySearchTree() : root(nullptr) {}

    // Fungsi untuk menambahkan node ke dalam BST
    Node* tambahNode(Node* curr, int val) {
        if (!curr) {
            return new Node(val);
        }
```

```
        if (val < curr->data) {
            curr->left = tambahNode(curr->left, val);
        } else if (val > curr->data) {
            curr->right = tambahNode(curr->right, val);
        }

        return curr;
    }

    // Fungsi untuk mencari node dalam BST
    bool cariNode(Node* curr, int val) {
        if (!curr) {
            return false;
        }

        if (val == curr->data) {
            return true;
        } else if (val < curr->data) {
            return cariNode(curr->left, val);
        } else {
            return cariNode(curr->right, val);
        }
    }

    // Fungsi untuk traversal in-order BST
    void inOrderTraversal(Node* curr) {
        if (curr) {
            inOrderTraversal(curr->left);
            cout << curr->data << " ";
            inOrderTraversal(curr->right);
        }
    }

    // Fungsi untuk memulai traversal in-order BST
    void inOrder() {
        inOrderTraversal(root);
        cout << endl;
    }

    // Fungsi untuk menambahkan node baru ke dalam BST
    void tambah(int val) {
        root = tambahNode(root, val);
    }

    // Fungsi untuk mencari node dalam BST
    bool cari(int val) {
        return cariNode(root, val);
    }
};
```

```
int main() {
    BinarySearchTree bst;

    bst.tambah(50);
    bst.tambah(30);
    bst.tambah(70);
    bst.tambah(20);
    bst.tambah(40);
    bst.tambah(60);
    bst.tambah(80);

    cout << "Traversal in-order BST:" << endl;
    bst.inOrder();

    int nilaiCari = 40;
    bool ditemukan = bst.cari(nilaiCari);
    cout << "Nilai " << nilaiCari << " " << (ditemukan ? "ditemukan" : "tidak ditemukan") << " dalam BST." <<
endl;

    return 0;
}
```

6. GRAPHS

```
#include <iostream>
#include <vector>
#include <queue>
#include <stack>

using namespace std;

// Struktur vertex dalam Directed Graph
struct Vertex {
    string nama;
    vector<Vertex*> tetangga;

    Vertex(string nama) : nama(nama) {}
};

// Kelas untuk Directed Graph
class DirectedGraph {
private:
    vector<Vertex*> vertices;

    // Fungsi rekursif untuk traversal Depth-First Search (DFS)
    void DFSUtil(Vertex* v, vector<bool>& kunjungi) {
        cout << v->nama << " ";
        kunjungi[v - &vertices[0]] = true;
```

```
        for (Vertex* tetangga : v->tetangga) {
            if (!kunjungi[tetangga - &vertices[0]]) {
                DFSUtil(tetangga, kunjungi);
            }
        }
    }

public:
    // Fungsi untuk menambahkan vertex ke dalam Directed Graph
    void tambahVertex(string nama) {
        vertices.push_back(new Vertex(nama));
    }

    // Fungsi untuk menambahkan edge (sisi) dari vertex1 ke vertex2
    void tambahEdge(string namaVertex1, string namaVertex2) {
        Vertex* v1 = nullptr;
        Vertex* v2 = nullptr;

        // Cari vertex1 dan vertex2
        for (Vertex* v : vertices) {
            if (v->nama == namaVertex1) {
                v1 = v;
            }
            if (v->nama == namaVertex2) {
                v2 = v;
            }
        }

        // Tambahkan edge jika vertex ditemukan
        if (v1 && v2) {
            v1->tetangga.push_back(v2);
        }
    }

    // Fungsi untuk melakukan traversal Breadth-First Search (BFS)
    void BFS(string namaAwal) {
        vector<bool> kunjungi(vertices.size(), false);
        queue<Vertex*> antrian;

        for (Vertex* v : vertices) {
            if (v->nama == namaAwal) {
                kunjungi[v - &vertices[0]] = true;
                antrian.push(v);
                break;
            }
        }

        while (!antrian.empty()) {
            Vertex* v = antrian.front();
```

```
            antrian.pop();
            cout << v->nama << " ";

            for (Vertex* tetangga : v->tetangga) {
                if (!kunjungi[tetangga - &vertices[0]]) {
                    kunjungi[tetangga - &vertices[0]] = true;
                    antrian.push(tetangga);
                }
            }
        }

        cout << endl;
    }

    // Fungsi untuk melakukan traversal Depth-First Search (DFS)
    void DFS(string namaAwal) {
        vector<bool> kunjungi(vertices.size(), false);

        for (Vertex* v : vertices) {
            if (v->nama == namaAwal && !kunjungi[v - &vertices[0]]) {
                DFSUtil(v, kunjungi);
            }
        }

        cout << endl;
    }
};

int main() {
    DirectedGraph dg;

    dg.tambahVertex("A");
    dg.tambahVertex("B");
    dg.tambahVertex("C");
    dg.tambahVertex("D");
    dg.tambahVertex("E");

    dg.tambahEdge("A", "B");
    dg.tambahEdge("A", "C");
    dg.tambahEdge("B", "D");
    dg.tambahEdge("C", "D");
    dg.tambahEdge("D", "E");

    cout << "Traversal Breadth-First Search (BFS): ";
    dg.BFS("A");

    cout << "Traversal Depth-First Search (DFS): ";
    dg.DFS("A");
```

```
    return 0;
}
```

7. SORTING

```
#include <iostream>
#include <vector>
#include <ctime>

using namespace std;

// Fungsi untuk menukar dua elemen dalam array
void tukar(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

// Algoritma Bubble Sort
void bubbleSort(vector<int> &arr) {
    int n = arr.size();
    bool adaPenukaran;

    for (int i = 0; i < n - 1; i++) {
        adaPenukaran = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                tukar(arr[j], arr[j + 1]);
                adaPenukaran = true;
            }
        }
        if (!adaPenukaran) break; // Jika tidak ada penukaran, array sudah terurut
    }
}

// Algoritma Selection Sort
void selectionSort(vector<int> &arr) {
    int n = arr.size();

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

```
}

// Algoritma Merge Sort
void merge(vector<int> &arr, int left, int middle, int right) {
    int n1 = middle - left + 1;
    int n2 = right - middle;

    vector<int> L(n1);
    vector<int> R(n2);

    for (int i = 0; i < n1; i++) {
        L[i] = arr[left + i];
    }
    for (int i = 0; i < n2; i++) {
        R[i] = arr[middle + 1 + i];
    }

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(vector<int> &arr, int left, int right) {
    if (left < right) {
        int middle = left + (right - left) / 2;

        mergeSort(arr, left, middle);
        mergeSort(arr, middle + 1, right);
```

```
        merge(arr, left, middle, right);
    }
}

int main() {
    srand(time(NULL));
    vector<int> data;

    cout << "Array sebelum pengurutan:" << endl;
    for (int i = 0; i < 10; i++) {
        int nilaiAcak = rand() % 100;
        data.push_back(nilaiAcak);
        cout << nilaiAcak << " ";
    }
    cout << endl;

    // Pilih algoritma pengurutan yang ingin digunakan (Bubble Sort, Selection Sort, atau Merge Sort)
    // bubbleSort(data);
    // selectionSort(data);
    mergeSort(data, 0, data.size() - 1);

    cout << "Array setelah pengurutan:" << endl;
    for (int num : data) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

8. SEARCHING

- **Linear Search**

```
#include <iostream>
#include <vector>

using namespace std;

int linearSearch(vector<int> &arr, int target) {
    for (int i = 0; i < arr.size(); i++) {
        if (arr[i] == target) {
            return i; // Elemen ditemukan, mengembalikan indeks
        }
    }
    return -1; // Elemen tidak ditemukan
}

int main() {
```

```
    vector<int> data = {10, 20, 30, 40, 50, 60, 70};
    int target = 40;

    int hasil = linearSearch(data, target);

    if (hasil != -1) {
        cout << "Elemen " << target << " ditemukan pada indeks " << hasil << endl;
    } else {
        cout << "Elemen " << target << " tidak ditemukan." << endl;
    }

    return 0;
}
```

- **Binary Search**

```
#include <iostream>
#include <vector>

using namespace std;

int binarySearch(vector<int> &arr, int target) {
    int kiri = 0;
    int kanan = arr.size() - 1;

    while (kiri <= kanan) {
        int tengah = kiri + (kanan - kiri) / 2;

        if (arr[tengah] == target) {
            return tengah; // Elemen ditemukan, mengembalikan indeks
        }

        if (arr[tengah] < target) {
            kiri = tengah + 1;
        } else {
            kanan = tengah - 1;
        }
    }
    return -1; // Elemen tidak ditemukan
}

int main() {
    vector<int> data = {10, 20, 30, 40, 50, 60, 70};
    int target = 40;

    int hasil = binarySearch(data, target);

    if (hasil != -1) {
        cout << "Elemen " << target << " ditemukan pada indeks " << hasil << endl;
    } else {
```

```
        cout << "Elemen " << target << " tidak ditemukan." << endl;
    }

    return 0;
}
```

- **Hashing**

```
#include <iostream>
#include <vector>

using namespace std;

const int TABLE_SIZE = 128; // Ukuran tabel hash

struct HashNode {
    int key;
    int value;
    HashNode(int k, int v) : key(k), value(v) {}
};

class HashMap {
private:
    vector<HashNode*> table;

public:
    HashMap() {
        table.resize(TABLE_SIZE, nullptr);
    }

    void insert(int key, int value) {
        int index = key % TABLE_SIZE;
        while (table[index] != nullptr && table[index]->key != key) {
            index = (index + 1) % TABLE_SIZE;
        }
        if (table[index] != nullptr) {
            delete table[index];
        }
        table[index] = new HashNode(key, value);
    }

    int search(int key) {
        int index = key % TABLE_SIZE;
        while (table[index] != nullptr && table[index]->key != key) {
            index = (index + 1) % TABLE_SIZE;
        }
        if (table[index] != nullptr) {
            return table[index]->value; // Elemen ditemukan
        }
        return -1; // Elemen tidak ditemukan
```

```
    }
};

int main() {
    HashMap map;
    map.insert(1, 10);
    map.insert(2, 20);
    map.insert(3, 30);
    int target = 2;

    int hasil = map.search(target);

    if (hasil != -1) {
        cout << "Elemen dengan kunci " << target << " memiliki nilai " << hasil << endl;
    } else {
        cout << "Elemen dengan kunci " << target << " tidak ditemukan." << endl;
    }
    return 0;
}
```

9. DYNAMIC PROGRAMMING

- **Fibonacci dengan Rekursi sederhana**

```
#include <iostream>

using namespace std;

int fibonacci(int n) {
    if (n <= 1) {
        return n;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}

int main() {
    int n = 10; // Ubah nilai n sesuai dengan angka Fibonacci yang ingin dicari
    cout << "Deret Fibonacci ke-" << n << " adalah " << fibonacci(n) << endl;
    return 0;
}
```

- **Fibonacci dengan Memoisasi (Top-Down)**

```
#include <iostream>
#include <vector>

using namespace std;

vector<long long> memo;
```

```
long long fibonacciMemo(int n) {
   if (n <= 1) {
      return n;
   } else if (memo[n] != -1) {
      return memo[n];
   } else {
      memo[n] = fibonacciMemo(n - 1) + fibonacciMemo(n - 2);
      return memo[n];
   }
}

int main() {
   int n = 10; // Ubah nilai n sesuai dengan angka Fibonacci yang ingin dicari
   memo.resize(n + 1, -1);
   cout << "Deret Fibonacci ke-" << n << " adalah " << fibonacciMemo(n) << endl;
   return 0;
}
```

- **Fibonacci dengan Tabulasi (Bottom-Up)**

```
#include <iostream>
#include <vector>

using namespace std;

long long fibonacciBottomUp(int n) {
   vector<long long> dp(n + 1);
   dp[0] = 0;
   dp[1] = 1;

   for (int i = 2; i <= n; i++) {
      dp[i] = dp[i - 1] + dp[i - 2];
   }

   return dp[n];
}

int main() {
   int n = 10; // Ubah nilai n sesuai dengan angka Fibonacci yang ingin dicari
   cout << "Deret Fibonacci ke-" << n << " adalah " << fibonacciBottomUp(n) << endl;
   return 0;
}
```