Telkom University Purwokerto

## Aturan Praktikum Struktur Data

1. **Akun GitHub:** Setiap praktikan wajib memiliki akun GitHubyang aktif dan digunakan selama praktikum berlangsung.
2. **Invite Collaborator:** Setiap praktikan diwajibkan untuk menambahkan collaborator di setiap repository
    a. Asisten Praktikum: AndiniNH
    b. AsistenPraktikum:4ldiputra
3. **Repository Praktikum:** Setiap praktikan diwajibkan untuk membuat satu repository di GitHub yang akan digunakan untuk seluruh tugas dan laporan praktikum. Repository ini harus diatur dengan rapi dan sesuai dengan instruksi yang akan diberikan di lampiran.
4. **Penamaan Folder:** Penamaan folderdalam repositoryakan dibahas secara rinci di lampiran. Praktikan wajib mengikuti aturan penamaan yang telah ditentukan.

| Nomor | Pertemuan | Penamaan |
|---|---|---|
| 1 | Penganalan Bahasa C++ Bagian Pertama | 01_Pengenalan_CPP_Bagian_1 |
| 2 | Pengenalan Bahasa C++ Bagian Kedua | 02_Pengenalan_CPP_Bagian_2 |
| 3 | Abstract Data Type | 03_Abstract_Data_Type |
| 4 | Single Linked List Bagian Pertama | 04_Single_Linked_List_Bagian_1 |
| 5 | Single Linked List Bagian Kedua | 05_Single_Linked_List_Bagian_2 |
| 6 | Double Linked List Bagian Pertama | 06_Double_Linked_List_Bagian_1 |
| 7 | Stack | 07_Stack |
| 8 | Queue | 08_Queue |
| 9 | Assessment Bagian Pertama | 09_Assessment_Bagian_1 |
| 10 | Tree Bagian Pertama | 10_Tree_Bagian_1 |
| 11 | Tree Bagian Kedua | 11_Tree_Bagian_2 |
| 12 | Asistensi Tugas Besar | 12_Asistensi_Tugas_Besar |

| | | |
|---|---|---|
| 13 | Multi Linked List | 13_Multi_Linked_List |
| 14 | Graph | 14_Graph |
| 15 | Assessment Bagian Kedua | 15_Assessment_Bagian_2 |
| 16 | Tugas Besar | 16_Tugas_Besar |

5. **Jam Praktikum:**
    - Jam masuk praktikum adalah **1 jam lebih lambat** dari jadwal yang tercantum. Sebagai contoh, jika jadwalpraktikumadalah pukul 06.30 - 09.30, maka aturan praktikum akan diatur sebagai berikut:
        - **06.30 - 07.30**: Waktuini digunakan untuk **Tugas Praktikum dan Laporan Praktikum** yang dilakukan di luar laboratorium.
        - **07.30 - 0G.30**: Sesi ini mencakup **tutorial, diskusi, dan kasus problem- solving**. Kegiatan ini berlangsung di dalam laboratorium dengan alokasi waktu sebagai berikut:
            - **60 menit pertama**: Tugas terbimbing.
            - **60 menit kedua**: Tugas mandiri.
6. **Pengumpulan Tugasn Pendahuluan:** Tugas Pendahuluan (TP) wajib dikumpulkan melalui GitHub sesuai dengan format berikut:

    **nama_repo/nama_pertemuan/TP_Pertemuan_Ke.md**

    Sebagai contoh:

    **STD_Yudha_Islalmi_Sulistya_XXXXXXXX/01_Running_Modul/TP_01.md**

7. **Pengecekan Tugas Pendahuluan:** Pengumpulan laporan praktikum akan diperiksa **1 hari sebelum praktikum selanjutnya** dimulai. Pastikan tugas telah diunggah tepat waktu untuk menghindari sanksi.

# LAPORAN PRAKTIKUM

# MODUL 10 & 11

# TREE 1 & 2


**Disusun Oleh :**

**Izzaty Zahara Br Barus – 2311104052**

**Kelas :**

**SE-07-02**

**Dosen :**

**Wahyu Andi Saputra, S.pd,M.Eng**

**PROGRAM STUDI SOFTWARE ENGINEERING**

**FAKULTAS INFORMATIKA**

**TELKOM UNIVERSITY**

**PURWOKERTO**


**2024**

# I. TUJUAN

1. Memahami konsep penggunaan fungsi rekursif.
2. Mengimplementasikan bentuk-bentuk fungsi rekursif.
3. Mengaplikasikan struktur data tree dalam sebuah kasus pemrograman.
4. Mengimplementasikan struktur data tree, khususnya Binary Tree.
5. Mengimplementasikan struktur data tree, khususnya Binary Tree.

# II. LANDASAN TEORI

1. **Pengertian Struktur Data Tree**

   Tree adalah salah satu struktur data non-linear yang merepresentasikan hubungan hierarkis antara elemen data. Dalam tree, terdapat node-nodes yang saling terhubung, di mana salah satu node disebut *root* (akar), dan node lainnya merupakan *child* (anak) dari root atau node lainnya. Karakteristik utama tree meliputi keberadaan root, anak, orang tua, daun (*leaf*), dan tingkat kedalaman (*depth*).

2. Binary Tree dan Jenisnya

   Binary Tree adalah struktur data tree di mana setiap node hanya dapat memiliki maksimum dua anak, yaitu *left child* dan *right child*. Beberapa jenis Binary Tree:

   a. Binary Search Tree (BST): Tree dengan aturan bahwa *left child* memiliki nilai lebih kecil dari *parent* dan *right child* lebih besar dari *parent*.
   b. Complete Binary Tree: Semua tingkat terisi penuh kecuali tingkat terakhir, yang diisi dari kiri ke kanan.
   c. AVL Tree: Binary Tree yang memastikan perbedaan ketinggian (*height*) antara subtree kiri dan kanan tidak lebih dari 1.
   d. Heap Tree: Tree yang mengikuti aturan *heap*, yaitu setiap *parent* lebih besar (*max-heap*) atau lebih kecil (*min-heap*) dari kedua anaknya.

3. Operasi dalam Binary Search Tree

   a. Insert: Menambahkan node ke tree berdasarkan aturan BST. Jika nilai lebih kecil dari node saat ini, masuk ke subtree kiri; jika lebih besar, masuk ke subtree kanan.
   b. Delete: Menghapus node dengan 3 kondisi:
      i. Node tanpa anak (leaf): langsung dihapus.
      ii. Node dengan 1 anak: anak menggantikan posisi node.
      iii. Node dengan 2 anak: menggunakan pengganti dari *left subtree* (nilai terbesar) atau *right subtree* (nilai terkecil).
   c. Search: Mencari elemen berdasarkan algoritma rekursif, mengikuti aturan BST.

4. Traversal pada Binary Tree

   d. Pre-order: Kunjungi *root*, lalu subtree kiri, diikuti subtree kanan.
   e. In-order: Kunjungi subtree kiri, lalu *root*, diikuti subtree kanan.
   f. Post-order: Kunjungi subtree kiri, diikuti subtree kanan, lalu *root*.

5. Kelebihan dan Kekurangan Tree

   g. Kelebihan: Mempermudah representasi data hierarkis dan mendukung pencarian serta pengelolaan data secara efisien.
   h. Kekurangan: Operasi tertentu seperti rekursif bisa memakan memori lebih banyak karena penggunaan *stack activation record*.

Landasan teori ini mencakup konsep dasar dari tree, jenis-jenisnya, serta operasi utama yang mendukung implementasi Binary Search Tree.

## III. GUIDE

1. Guide

a. Syntax

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

/// PROGRAM BINARY TREE

// Struktur data pohon biner untuk menyimpan data dan pointer ke anak kiri, kanan, dan induk
struct Pohon {
   char data;              // Data yang disimpan di node (tipe char)
   Pohon *left, *right;    // Pointer ke anak kiri dan anak kanan
   Pohon *parent;          // Pointer ke node induk
};

// Variabel global untuk menyimpan root (akar) pohon dan node baru
Pohon *root, *baru;

// Inisialisasi pohon agar kosong
void init() {
   root = NULL; // Mengatur root sebagai NULL (pohon kosong)
}

// Mengecek apakah pohon kosong
bool isEmpty() {
   return root == NULL; // Mengembalikan true jika root adalah NULL
}

// Membuat node baru sebagai root pohon
void buatNode(char data) {
```

```
    if (isEmpty()) { // Jika pohon kosong
        root = new Pohon{data, NULL, NULL, NULL}; // 
Membuat node baru sebagai root
        cout << "\nNode " << data << " berhasil dibuat menjadi 
root." << endl;
    } else {
        cout << "\nPohon sudah dibuat." << endl; // Root sudah 
ada, tidak membuat node baru
    }
}

// Menambahkan node baru sebagai anak kiri dari node tertentu
Pohon* insertLeft(char data, Pohon *node) {
    if (node->left != NULL) { // Jika anak kiri sudah ada
        cout << "\nNode " << node->data << " sudah ada child 
kiri!" << endl;
        return NULL; // Tidak menambahkan node baru
    }
    // Membuat node baru dan menghubungkannya ke node 
sebagai anak kiri
    baru = new Pohon{data, NULL, NULL, node};
    node->left = baru;
    cout << "\nNode " << data << " berhasil ditambahkan ke child 
kiri " << node->data << endl;
    return baru; // Mengembalikan pointer ke node baru
}

// Menambahkan node baru sebagai anak kanan dari node 
tertentu
Pohon* insertRight(char data, Pohon *node) {
    if (node->right != NULL) { // Jika anak kanan sudah ada
        cout << "\nNode " << node->data << " sudah ada child 
kanan!" << endl;
```

```
        return NULL; // Tidak menambahkan node baru
    }
    // Membuat node baru dan menghubungkannya ke node sebagai anak kanan
    baru = new Pohon{data, NULL, NULL, node};
    node->right = baru;
    cout << "\nNode " << data << " berhasil ditambahkan ke child kanan " << node->data << endl;
    return baru; // Mengembalikan pointer ke node baru
}

// Mengubah data di dalam sebuah node
void update(char data, Pohon *node) {
    if (!node) { // Jika node tidak ditemukan
        cout << "\nNode yang ingin diubah tidak ditemukan!" << endl;
        return;
    }
    char temp = node->data; // Menyimpan data lama
    node->data = data;      // Mengubah data dengan nilai baru
    cout << "\nNode " << temp << " berhasil diubah menjadi " << data << endl;
}

// Mencari node dengan data tertentu
void find(char data, Pohon *node) {
    if (!node) return; // Jika node tidak ada, hentikan pencarian

    if (node->data == data) { // Jika data ditemukan
        cout << "\nNode ditemukan: " << data << endl;
        return;
    }
    // Melakukan pencarian secara rekursif ke anak kiri dan
```

```
kanan
    find(data, node->left);
    find(data, node->right);
}

// Traversal Pre-order (Node -> Kiri -> Kanan)
void preOrder(Pohon *node) {
    if (!node) return; // Jika node kosong, hentikan traversal
    cout << node->data << " "; // Cetak data node saat ini
    preOrder(node->left);      // Traversal ke anak kiri
    preOrder(node->right);     // Traversal ke anak kanan
}

// Traversal In-order (Kiri -> Node -> Kanan)
void inOrder(Pohon *node) {
    if (!node) return; // Jika node kosong, hentikan traversal
    inOrder(node->left);  // Traversal ke anak kiri
    cout << node->data << " "; // Cetak data node saat ini
    inOrder(node->right); // Traversal ke anak kanan
}

// Traversal Post-order (Kiri -> Kanan -> Node)
void postOrder(Pohon *node) {
    if (!node) return; // Jika node kosong, hentikan traversal
    postOrder(node->left);  // Traversal ke anak kiri
    postOrder(node->right); // Traversal ke anak kanan
    cout << node->data << " "; // Cetak data node saat ini
}

// Menghapus node dengan data tertentu
Pohon* deleteNode(Pohon *node, char data) {
    if (!node) return NULL; // Jika node kosong, hentikan
```

```
    // Rekursif mencari node yang akan dihapus
    if (data < node->data) {
        node->left = deleteNode(node->left, data); // Cari di anak kiri
    } else if (data > node->data) {
        node->right = deleteNode(node->right, data); // Cari di anak kanan
    } else {
        // Jika node ditemukan
        if (!node->left) { // Jika tidak ada anak kiri
            Pohon *temp = node->right; // Anak kanan menggantikan posisi node
            delete node;
            return temp;
        } else if (!node->right) { // Jika tidak ada anak kanan
            Pohon *temp = node->left; // Anak kiri menggantikan posisi node
            delete node;
            return temp;
        }

        // Jika node memiliki dua anak, cari node pengganti (successor)
        Pohon *successor = node->right;
        while (successor->left) successor = successor->left; // Cari node terkecil di anak kanan
        node->data = successor->data; // Gantikan data dengan successor
        node->right = deleteNode(node->right, successor->data); // Hapus successor
    }
    return node;
}
```

```
// Menemukan node paling kiri
Pohon* mostLeft(Pohon *node) {
    if (!node) return NULL; // Jika node kosong, hentikan
    while (node->left) node = node->left; // Iterasi ke anak kiri hingga mentok
    return node;
}

// Menemukan node paling kanan
Pohon* mostRight(Pohon *node) {
    if (!node) return NULL; // Jika node kosong, hentikan
    while (node->right) node = node->right; // Iterasi ke anak kanan hingga mentok
    return node;
}

// Fungsi utama
int main() {
    init(); // Inisialisasi pohon
    buatNode('F'); // Membuat root dengan data 'F'
    insertLeft('B', root); // Menambahkan 'B' ke anak kiri root
    insertRight('G', root); // Menambahkan 'G' ke anak kanan root
    insertLeft('A', root->left); // Menambahkan 'A' ke anak kiri dari 'B'
    insertRight('D', root->left); // Menambahkan 'D' ke anak kanan dari 'B'
    insertLeft('C', root->left->right); // Menambahkan 'C' ke anak kiri dari 'D'
    insertRight('E', root->left->right); // Menambahkan 'E' ke anak kanan dari 'D'

    // Traversal pohon
```

```
    cout << "\nPre-order Traversal: ";
    preOrder(root);
    cout << "\nIn-order Traversal: ";
    inOrder(root);
    cout << "\nPost-order Traversal: ";
    postOrder(root);

    // Menampilkan node paling kiri dan kanan
    cout << "\nMost Left Node: " << mostLeft(root)->data;
    cout << "\nMost Right Node: " << mostRight(root)->data;

    // Menghapus node
    cout << "\nMenghapus node D.";
    root = deleteNode(root, 'D');
    cout << "\nIn-order Traversal setelah penghapusan: ";
    inOrder(root);

    return 0;
}
```

b. Penjelasan syntax

Program ini mengimplementasikan struktur data pohon biner untuk menyimpan dan mengelola data secara hierarkis. Setiap node dalam pohon terdiri dari data yang disimpan (dalam bentuk karakter), serta pointer ke anak kiri, anak kanan, dan induk. Program dimulai dengan menginisialisasi pohon yang kosong, di mana root pohon diatur menjadi NULL. Jika pohon kosong, fungsi buatNode(data) akan membuat node pertama sebagai root. Selanjutnya, fungsi insertLeft(data, node) dan insertRight(data, node) digunakan untuk menambahkan node baru sebagai anak kiri atau kanan dari node yang sudah ada, dengan pengecekan untuk memastikan tidak ada anak yang duplikat pada posisi tersebut.

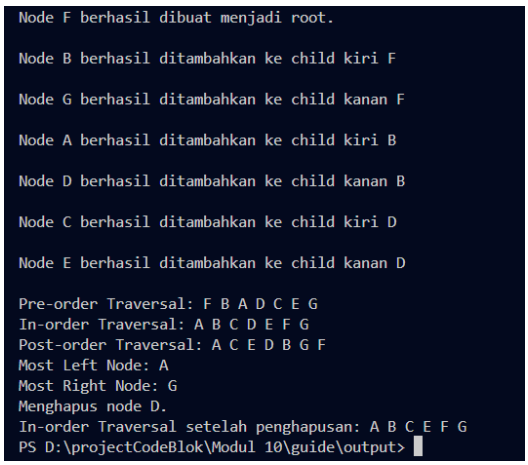
Selain itu, program menyediakan fungsi untuk mengubah data dalam sebuah node melalui fungsi update(data, node) dan melakukan pencarian node dengan data tertentu menggunakan fungsi find(data, node). Pencarian dilakukan secara rekursif melalui anak kiri dan kanan hingga menemukan node yang dicari. Untuk menampilkan isi pohon, terdapat tiga metode traversal: pre-order (Node -> Kiri -> Kanan), in-order (Kiri -> Node ->

Kanan), dan post-order (Kiri -> Kanan -> Node), yang memungkinkan pengguna melihat data dalam urutan yang berbeda.

Program ini juga memungkinkan penghapusan node melalui fungsi deleteNode(node, data), yang dapat menangani berbagai kasus, seperti node dengan dua anak yang digantikan oleh node successor (node dengan nilai terkecil di subtree kanan), atau node dengan satu anak yang digantikan oleh anak tersebut. Fungsi mostLeft(node) dan mostRight(node) digunakan untuk menemukan node paling kiri dan paling kanan dalam pohon, yang biasanya merujuk pada nilai terkecil dan terbesar dalam pohon. Fungsi utama program menginisialisasi pohon, membuat node root, menambahkan anak-anak ke node root, dan kemudian melakukan traversal untuk mencetak hasilnya. Setelah itu, program menghapus node tertentu (misalnya node D) dan menampilkan hasil traversal pohon setelah penghapusan tersebut.

Dengan demikian, program ini menggambarkan cara kerja pohon biner untuk menyimpan, mengubah, dan mengelola data, serta menunjukkan bagaimana operasi dasar seperti penyisipan, pencarian, penghapusan, dan traversal dapat dilakukan dengan efektif.

c. Output




## IV. UNGUIDED

1. Unguided 1,2,3

a. Syntax :

```
#ifndef BSTREE_H
#define BSTREE_H
```

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

typedef int infotype;
struct Node {
   infotype info;
   Node* left;
   Node* right;
};

Node* createNode(infotype value);
void insertNode(Node*& root, infotype value);
Node* findNode(Node* root, infotype value);
void printInOrder(Node* root);
void printPreOrder(Node* root);
void printPostOrder(Node* root);
int hitungJumlahNode(Node* root);
int hitungTotalInfo(Node* root);
int hitungKedalaman(Node* root);

#endif
```

```
#include "bstree.h"

Node* createNode(infotype value) {
   Node* newNode = new Node;
   newNode->info = value;
   newNode->left = nullptr;
   newNode->right = nullptr;
   return newNode;
}

void insertNode(Node*& root, infotype value) {
   if (root == nullptr) {
```

```
        root = createNode(value);
    } else if (value < root->info) {
        insertNode(root->left, value);
    } else if (value > root->info) {
        insertNode(root->right, value);
    }
}

Node* findNode(Node* root, infotype value) {
    if (root == nullptr || root->info == value) {
        return root;
    }
    if (value < root->info) {
        return findNode(root->left, value);
    } else {
        return findNode(root->right, value);
    }
}

void printInOrder(Node* root) {
    if (root == nullptr) return;
    printInOrder(root->left);
    cout << root->info << " ";
    printInOrder(root->right);
}

void printPreOrder(Node* root) {
    if (root == nullptr) return;
    cout << root->info << " ";
    printPreOrder(root->left);
    printPreOrder(root->right);
}
```

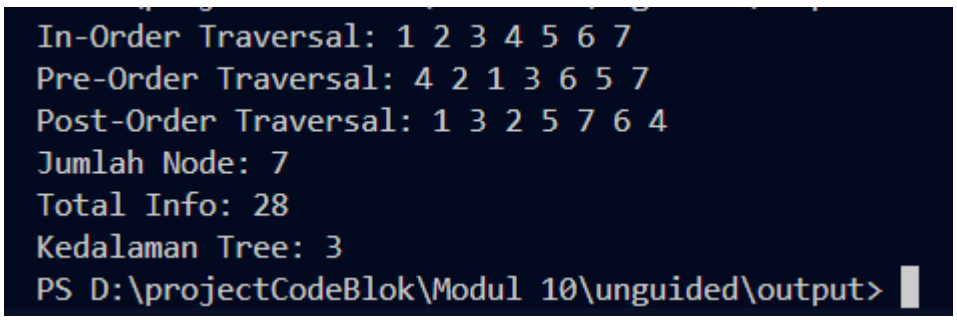
```
void printPostOrder(Node* root) {
    if (root == nullptr) return;
    printPostOrder(root->left);
    printPostOrder(root->right);
    cout << root->info << " ";
}

int hitungJumlahNode(Node* root) {
    if (root == nullptr) return 0;
     return 1 + hitungJumlahNode(root->left) + hitungJumlahNode(root->right);
}

int hitungTotalInfo(Node* root) {
    if (root == nullptr) return 0;
   return root->info + hitungTotalInfo(root->left) + hitungTotalInfo(root->right);
}

int hitungKedalaman(Node* root) {
    if (root == nullptr) return 0;
    int leftDepth = hitungKedalaman(root->left);
    int rightDepth = hitungKedalaman(root->right);
    return 1 + max(leftDepth, rightDepth);
}
```

```
#include "bstree.h"
#include "bstree.cpp"
int main() {
    Node* root = nullptr;

    insertNode(root, 4);
    insertNode(root, 2);
    insertNode(root, 6);
```

```
    insertNode(root, 1);
    insertNode(root, 3);
    insertNode(root, 5);
    insertNode(root, 7);

    cout << "In-Order Traversal: ";
    printInOrder(root);
    cout << endl;

    cout << "Pre-Order Traversal: ";
    printPreOrder(root);
    cout << endl;

    cout << "Post-Order Traversal: ";
    printPostOrder(root);
    cout << endl;

    cout << "Jumlah Node: " << hitungJumlahNode(root) << endl;
    cout << "Total Info: " << hitungTotalInfo(root) << endl;
    cout << "Kedalaman Tree: " << hitungKedalaman(root) << endl;

    return 0;
}
```

b. Penjelasan Syntax

c. Output



## V. KESIMPULAN

Kesimpulannya, program ini berhasil mengimplementasikan konsep pohon biner, khususnya Binary Search Tree (BST), untuk menyimpan, mengelola, dan memanipulasi data secara hierarkis. Program dapat melakukan operasi dasar seperti penyisipan node, pencarian data, penghapusan node, dan traversal dalam berbagai urutan (pre-order, in-order, post-order). Selain itu, program juga mampu menghitung jumlah node, total nilai dalam pohon, serta kedalaman maksimal pohon. Melalui implementasi ini, konsep dasar struktur data pohon biner dapat dipahami dengan baik, termasuk kelebihannya dalam efisiensi pengelolaan data hierarkis serta tantangannya dalam penggunaan memori pada operasi rekursif.