LAPORAN PRAKTIKUM
STRUKTUR DATA

MODUL 9
TREE

Disusun Oleh :
NAMA : Muhammad Omar Nadiv
NIM : 103112430063

Dosen
Fahrudin Mukti Wibowo, S.Kom., M.Eng.

PROGRAM STUDI STRUKTUR DATA
FAKULTAS INFORMATIKA
TELKOM UNIVERSITY PURWOKERTO
2025

A. Dasar Teori

Tree atau pohon merupakan struktur data non-linier yang tersusun secara hierarkis. Tree terdiri dari sekumpulan node yang saling terhubung melalui relasi parent–child. Node paling atas disebut root, sementara node yang tidak memiliki turunan disebut leaf. Tree digunakan untuk merepresentasikan hubungan bertingkat, seperti struktur folder, silsilah, dan organisasi.

Salah satu bentuk tree yang paling umum adalah Binary Tree, yaitu tree dimana setiap node memiliki maksimal dua child: left dan right. Dari binary tree, berkembang struktur yang lebih spesifik seperti Binary Search Tree (BST) yang menerapkan aturan terurut: semua nilai di subtree kiri harus lebih kecil daripada parent, sedangkan nilai di subtree kanan harus lebih besar. Karena sifat terurut ini, operasi pencarian, penambahan, dan penghapusan data pada BST dapat dilakukan dengan kompleksitas rata-rata O(log n).

Tree juga memiliki beberapa teknik traversal untuk mengunjungi setiap node, seperti pre-order, in-order, dan post-order. Struktur ini banyak digunakan dalam berbagai aplikasi, termasuk penyimpanan data, kompresi (Huffman Tree), perancangan compiler, artificial intelligence (decision tree), hingga sistem file. Dengan kemampuan merepresentasikan data secara hierarkis, tree menjadi salah satu struktur data yang sangat penting dalam pemrograman.

B. Guided (berisi screenshot source code & output program disertai penjelasannya)

Guided 1 (file tree.h)

```
#ifndef TREE_H
#define TREE_H

struct Node {
   int data;
   Node *left, *right;
   int height;
};
```

```
class BinaryTree {
private:
    Node* root;

    Node* insertNode(Node* node, int value);
    Node* deleteNode(Node* node, int value);

    int getHeight(Node* node);
    int getBalance(Node* node);

    Node* rotateRight(Node* y);
    Node* rotateLeft(Node* x);

    Node* minValueNode (Node* node);

    void inorder(Node* node);
    void preorder(Node* node);
    void postorder(Node* node);

public:
    BinaryTree();
    void insert(int value);
    void deleteValue(int value);
    void update(int oldVal, int newVal);

    void inorder();
    void preorder();
    void postorder();
};

#endif
```

Guided 1 (file tree.cpp)

```
#include "tree.h"
#include <iostream>
using namespace std;

BinaryTree::BinaryTree() {
    root = nullptr;
```

```
}

int BinaryTree::getHeight(Node* n) {
   return (n == nullptr) ? 0 : n->height;
}

int BinaryTree::getBalance(Node* n) {
   return (n == nullptr) ? 0 :
   getHeight(n->left) - getHeight(n->right);
}

Node* BinaryTree::rotateRight(Node* y) {
   Node* x = y->left;
   Node* T2 = x->right;

   x->right = y;
   y->left = T2;

   y->height = max(getHeight(y->left),
    getHeight(y->right)) + 1;
   x->height = max(getHeight(x->left),
    getHeight(x->right)) + 1;

   return x;
}

Node* BinaryTree::rotateLeft(Node* x) {
   Node* y = x->right;
   Node* T2 = y->left;

   y->left = x;
   x->right = T2;

   x->height = max(getHeight(x->left),
    getHeight(x->right)) + 1;
   y->height = max(getHeight(y->left),
    getHeight(y->right)) + 1;

   return y;
}
```

```
Node* BinaryTree::insertNode(Node* node, int value) {
    if (node == nullptr) {
        Node* newNode = new Node{value, nullptr, nullptr, 1};
        return newNode;
    }

    if (value < node->data)
        node->left = insertNode(node->left, value);
    else if (value > node->data)
        node->right = insertNode(node->right, value);
    else
        return node;

    node->height = 1 + max(getHeight(node->left),
    getHeight(node->right));

    int balance = getBalance(node);

    if (balance > 1 && value < node->left->data)
        return rotateRight(node);

    if (balance < -1 && value > node->right->data)
        return rotateLeft(node);

    if (balance > 1 && value > node->left->data) {
        node->left = rotateLeft(node->left);
        return rotateRight(node);
    }

    if (balance < -1 && value < node->right->data) {
        node->right = rotateRight(node->right);
        return rotateLeft(node);
    }

    return node;
}

void BinaryTree::insert(int value) {
    root = insertNode(root, value);
}
```

```
Node* BinaryTree::minValueNode(Node* node) {
    Node* current = node;
    while (current->left != nullptr)
        current = current->left;
    return current;
}

Node* BinaryTree::deleteNode(Node* root, int key) {
    if (root == nullptr)
        return root;

    if (key < root->data)
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
        root->right = deleteNode(root->right, key);
    else {
        if ((root->left == nullptr) || (root->right == nullptr)) {
            Node* temp = root->left ? root->left : root->right;

            if (temp == nullptr) {
                temp = root;
                root = nullptr;
            } else {
                *root = *temp;
            }
            delete temp;
        } else {
            Node* temp = minValueNode(root->right);
            root->data = temp->data;
            root->right = deleteNode(root->right, temp->data);
        }
    }

    if (root == nullptr)
        return root;

    root->height = 1 + max(getHeight(root->left), getHeight(root->right));

    int balance = getBalance(root);

    if (balance > 1 && getBalance(root->left) >= 0)
```

```
        return rotateRight(root);

    if (balance > 1 && getBalance(root->left) < 0) {
        root->left = rotateLeft(root->left);
        return rotateRight(root);
    }

    if (balance < -1 && getBalance(root->right) <= 0)
        return rotateLeft(root);

    if (balance < -1 && getBalance(root->right) > 0) {
        root->right = rotateRight(root->right);
        return rotateLeft(root);
    }

    return root;
}

void BinaryTree::deleteValue(int value) {
    root = deleteNode(root, value);
}

void BinaryTree::update(int oldVal, int newVal) {
    deleteValue(oldVal);
    insert(newVal);
}

void BinaryTree::inorder(Node* node) {
    if (node == nullptr) return;
    inorder(node->left);
    cout << node->data << " ";
    inorder(node->right);
}

void BinaryTree::preorder(Node* node) {
    if (node == nullptr) return;
    cout << node->data << " ";
    preorder(node->left);
    preorder(node->right);
}
```

```
void BinaryTree::postorder(Node* node) {
   if (node == nullptr) return;
   postorder(node->left);
   postorder(node->right);
   cout << node->data << " ";
}

void BinaryTree::inorder() { inorder(root); cout << endl; }
void BinaryTree::preorder() { preorder(root); cout << endl; }
void BinaryTree::postorder() { postorder(root); cout << endl; }
```

Guided 1 (file main.cpp)

```
#include <iostream>
#include "tree.h"

using namespace std;

int main() {
   BinaryTree tree;

   cout << "=== INSERT DATA ===" << endl;
   tree.insert(10);
   tree.insert(15);
   tree.insert(20);
   tree.insert(30);
   tree.insert(35);
   tree.insert(40);
   tree.insert(50);

   cout << "Data yang diinsert: 10, 15, 20, 30, 35, 40, 50" << endl;

   cout << "\nTraversal setelah insert:" << endl;
   cout << "Inorder    : "; tree.inorder();
   cout << "Preorder   : "; tree.preorder();
   cout << "Postorder  : "; tree.postorder();

   cout << "\n=== Update Data ===" << endl;
   cout << "Sebelum update (20 -> 25): " << endl;
   cout << "Inorder   : "; tree.inorder();
```

```
    tree.update(20, 25);

    cout << "Setelah update (20 -> 25): " << endl;
    cout << "Inorder   : "; tree.inorder();

    cout << "\n=== Delete Data ===" << endl;
    cout << "Sebelum delete (hapus subtree dengan root = 30): " << endl;
    cout << "Inorder   : "; tree.inorder();

    tree.deleteValue(30);

    cout << "Setelah delete (subtree root = 30 dihapus): " << endl;
    cout << "Inorder   : "; tree.inorder();

    return 0;
}
```

Screenshots Output

```
nadiv@omarnadip Guided % ./run
=== INSERT DATA ===
Data yang diinsert: 10, 15, 20, 30, 35, 40, 50

Traversal setelah insert:
Inorder    : 10 15 20 30 35 40 50
Preorder   : 30 15 10 20 40 35 50
Postorder  : 10 20 15 35 50 40 30

=== Update Data ===
Sebelum update (20 -> 25):
Inorder   : 10 15 20 30 35 40 50
Setelah update (20 -> 25):
Inorder   : 10 15 25 30 35 40 50

=== Delete Data ===
Sebelum delete (hapus subtree dengan root = 30):
Inorder   : 10 15 25 30 35 40 50
Setelah delete (subtree root = 30 dihapus):
Inorder   : 10 15 25 35 40 50
nadiv@omarnadip Guided %
```

Deskripsi:

Program ini digunakan untuk mengelola data dalam struktur data Tree menggunakan pendekatan AVL Tree di bahasa C++. AVL Tree adalah bentuk Binary Search Tree yang menjaga keseimbangan tinggi setiap nodenya secara otomatis. Program menyediakan operasi dasar seperti insert, update, dan delete, di mana setiap perubahan pada tree akan diperiksa kembali nilai balance factor-nya. Jika terjadi ketidakseimbangan, tree akan diperbaiki melalui rotasi kiri, kanan, atau kombinasi keduanya agar struktur tetap optimal.

Selain operasi modifikasi data, program ini juga menyediakan fungsi traversal yaitu inorder, preorder, dan postorder untuk menampilkan isi tree sesuai aturan penelusuran masing-masing. Program ini memperlihatkan proses pembentukan tree, memperbarui nilai, menghapus subtree tertentu, serta menelusuri data secara menyeluruh, sehingga pengguna dapat memahami bagaimana AVL Tree bekerja secara dinamis namun tetap efisien.

C. Unguided/Tugas (berisi screenshot source code & output program disertai penjelasannya)

Unguided 1 (file bstree.h)

```
#ifndef BSTREE_H
#define BSTREE_H

typedef int infotype;
typedef struct Node* address;

struct Node {
   infotype info;
   address left;
   address right;
};

address alokasi(infotype x);
void insertNode(address &root, infotype x);
address findNode(infotype x, address root);
void InOrder(address root);
int hitungJumlahNode(address root);
int hitungTotalInfo(address root);
```

```
int hitungKedalaman(address root, int start);

#endif
```

(file bstree.cpp)

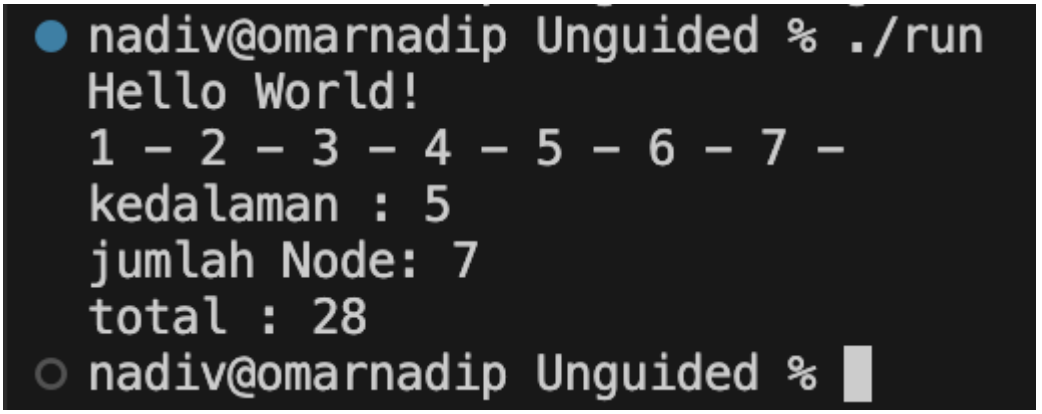
```
#include <iostream>
#include "bstree.h"
using namespace std;

address alokasi(infotype x)
{
   address p = new Node;
   p->info = x;
   p->left = nullptr;
   p->right = nullptr;
   return p;
}

void insertNode(address &root, infotype x)
{
   if (root == nullptr)
   {
       root = alokasi(x);
   }
   else if (x < root->info)
   {
       insertNode(root->left, x);
   }
   else if (x > root->info)
   {
       insertNode(root->right, x);
   }
}

address findNode(infotype x, address root)
{
   if (root == nullptr)
       return nullptr;
   if (root->info == x)
```

```
        return root;
    if (x < root->info)
        return findNode(x, root->left);
    return findNode(x, root->right);
}

void InOrder(address root)
{
    if (root != nullptr)
    {
        InOrder(root->left);
        cout << root->info << " - ";
        InOrder(root->right);
    }
}

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

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

int hitungKedalaman(address root, int start)
{
    if (root == nullptr)
        return start;
    int left = hitungKedalaman(root->left, start + 1);
    int right = hitungKedalaman(root->right, start + 1);
    return (left > right) ? left : right;
}
```

(file main.cpp)

```cpp
#include <iostream>
#include "bstree.h"
using namespace std;

int main() {
    cout << "Hello World!" << endl;

    address root = nullptr;

    insertNode (root, 1);
    insertNode (root, 2);
    insertNode (root, 6);
    insertNode (root, 4);
    insertNode (root, 5);
    insertNode (root, 3);
    insertNode (root, 7);

    InOrder(root);
    cout << endl;

    cout << "kedalaman : " << hitungKedalaman(root, 0) << endl;
    cout << "jumlah Node: " << hitungJumlahNode(root) << endl;
    cout << "total : " << hitungTotalInfo(root) << endl;

    return 0;
}
```

Screenshots Output



Deskripsi:

Program ini mengimplementasikan struktur data Binary Search Tree (BST) untuk menyimpan dan mengolah data secara hierarkis. Setiap node dalam tree terdiri dari nilai info serta pointer left dan right untuk menunjuk ke anak kiri dan anak kanan. Program menyediakan fungsi alokasi untuk membuat node baru, insertNode untuk menempatkan data ke posisi yang sesuai berdasarkan aturan BST, findNode untuk mencari elemen tertentu, serta InOrder sebagai metode traversal yang menampilkan nilai-nilai tree secara terurut. Seluruh fungsi dirancang agar tree dapat dibangun dan ditelusuri dengan mudah.

Selain operasi dasar, program ini juga menyertakan fungsi tambahan untuk kebutuhan analisis struktur tree, yaitu hitungJumlahNode untuk mengetahui berapa banyak node yang ada, hitungTotalInfo untuk menjumlahkan seluruh nilai, dan hitungKedalaman untuk menghitung kedalaman maksimum tree dari root. Melalui kombinasi fungsi tersebut, program menunjukkan bagaimana BST bekerja dalam penyimpanan data terurut, pencarian, penelusuran, serta penghitungan karakteristik struktural dari tree yang terbentuk.

D. Kesimpulan

Pada modul ini, saya mempelajari penerapan struktur data Binary Search Tree (BST) dalam bahasa C++. Saya memahami cara kerja BST yang menyimpan data secara terurut dengan aturan bahwa nilai lebih kecil ditempatkan di kiri dan nilai lebih besar di kanan. Saya juga mempraktekkan operasi dasar seperti pembuatan node, penyisipan data, pencarian, serta penelusuran menggunakan metode inorder.

Selain itu, saya mempelajari cara menghitung jumlah node, total nilai, dan kedalaman tree menggunakan pendekatan rekursif. Melalui modul ini, saya mendapatkan pemahaman mengenai bagaimana BST mengorganisir data secara efisien dan bagaimana struktur ini digunakan dalam pemrosesan data yang membutuhkan pencarian dan penelusuran terstruktur.

E. Referensi

https://www.trivusi.web.id/2022/07/struktur-data-tree.html

https://fikti.umsu.ac.id/struktur-data-tree/

https://bds.telkomuniversity.ac.id/struktur-data-tree-konsep-jenis-dan-aplikasinya/