LAPORAN PRAKTIKUM
STRUKTUR DATA

MODUL X
TREE (BAGIAN PERTAMA)

Disusun Oleh :
NAMA : Azzahra Farelika Esti Ning Tyas
NIM : 103112430023

Dosen
FAHRUDIN MUKTI WIBOWO

PROGRAM STUDI STRUKTUR DATA
FAKULTAS INFORMATIKA
TELKOM UNIVERSITY PURWOKERTO
2025

A. Dasar Teori

Rekursif adalah teknik dalam pemrograman di mana sebuah fungsi memanggil dirinya sendiri untuk menyelesaikan suatu permasalahan yang dapat dipecah menjadi bagian-bagian kecil dengan syarat berhenti (base case). Teknik ini umum digunakan dalam algoritma yang memiliki struktur bertingkat atau repetitif, seperti perhitungan matematis dan penelusuran pohon (tree) karena mempermudah penulisan logika program meskipun memerlukan penggunaan memori yang lebih besar. Tree adalah struktur data non-linear yang tersusun secara hierarkis dengan elemen bernama node yang saling terhubung; node teratas disebut root, sedangkan node bawah disebut child. Struktur ini menggambarkan hubungan parent-child yang cocok untuk merepresentasikan data bertingkat seperti sistem file dan indeks. Salah satu bentuk populer dari tree adalah Binary Tree, di mana setiap node hanya memiliki paling banyak dua child.

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

Guided 1

tree.h

Modul10 > GUIDED > tree.h > ...

```h
#ifndef TREE_H
#define TREE_H

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

class BinaryTree {
    private:
        Node* root;

        Tabnine | Edit | Test | Explain | Document
        Node* insertNode(Node* node, int value);
        Tabnine | Edit | Test | Explain | Document
        Node* deleteNode(Node* node, int value);

        Tabnine | Edit | Test | Explain | Document
        int getHeight(Node* node);
        Tabnine | Edit | Test | Explain | Document
        int getBalance(Node* node);

        Tabnine | Edit | Test | Explain | Document
        Node* rotateRight(Node* y);
        Tabnine | Edit | Test | Explain | Document
        Node* rotateLeft(Node* x);

        Tabnine | Edit | Test | Explain | Document
        Node* minValueNode(Node* node);

        Tabnine | Edit | Test | Explain | Document
        void inorder(Node* node);
        Tabnine | Edit | Test | Explain | Document
        void preorder(Node* node);
        Tabnine | Edit | Test | Explain | Document
        void postorder(Node* node);

    public:
        Tabnine | Edit | Test | Explain | Document
        BinaryTree();

        Tabnine | Edit | Test | Explain | Document
        void insert(int value);
        Tabnine | Edit | Test | Explain | Document
        void deleteValue(int value);
        Tabnine | Edit | Test | Explain | Document
        void update(int oldVal, int newVal);

        Tabnine | Edit | Test | Explain | Document
        void inorder();
        Tabnine | Edit | Test | Explain | Document
        void preorder();
        Tabnine | Edit | Test | Explain | Document
        void postorder();
};
#endif
```

tree.cpp

Modul10 > GUIDED > tree.cpp > postorder()

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

Tabnine | Edit | Test | Explain | Document
BinaryTree::BinaryTree() {
    root = nullptr;
}

Tabnine | Edit | Test | Explain | Document
int BinaryTree::getHeight(Node* n) {
    return (n == nullptr) ? 0 : n->height;
}

Tabnine | Edit | Test | Explain | Document
int BinaryTree::getBalance(Node* n) {
    return (n == nullptr) ? 0 :
    getHeight(n->left) - getHeight(n->right);
}

Tabnine | Edit | Test | Explain | Document
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;
}

Tabnine | Edit | Test | Explain | Document
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;
}

Tabnine | Edit | Test | Explain | Document
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;
}
```

tree.cpp

Modul10 > GUIDED > tree.cpp > postorder()

```cpp
Node* BinaryTree::insertNode(Node* node, int value) {

Tabnine | Edit | Test | Explain | Document
void BinaryTree::insert(int value) {
    root = insertNode(root, value);
}

Tabnine | Edit | Test | Explain | Document
Node* BinaryTree::minValueNode(Node* node) {
    Node* current = node;
    while (current->left != nullptr)
        current = current->left;
    return current;
}

Tabnine | Edit | Test | Explain | Document
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;
}

Tabnine | Edit | Test | Explain | Document
void BinaryTree::deleteValue(int value) {
    root = deleteNode(root, value);
}

Tabnine | Edit | Test | Explain | Document
void BinaryTree::update(int oldVal, int newVal) {
    deleteValue(oldVal);
    insert(newVal);
}

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

Tabnine | Edit | Test | Explain | Document
void BinaryTree::preorder(Node* node) {
    if (node == nullptr) return;
    cout << node->data << " ";
    preorder(node->left);
    preorder(node->right);
}

Tabnine | Edit | Test | Explain | Document
void BinaryTree::postorder(Node* node) {
    if (node == nullptr) return;
    postorder(node->left);
    postorder(node->right);
    cout << node->data << " ";
}

Tabnine | Edit | Test | Explain | Document
void BinaryTree::inorder() { inorder(root); cout << endl; }
Tabnine | Edit | Test | Explain | Document
void BinaryTree::preorder() { preorder(root); cout << endl; }
Tabnine | Edit | Test | Explain | Document
void BinaryTree::postorder() { postorder(root); cout << endl; }
```

main.cpp

Modul10 > GUIDED > main.cpp > main()

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

using namespace std;

Tabnine | Edit | Test | Explain | Document
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 menjadi 25): " << endl;
    cout << "Inorder: ";
    tree.inorder();

    tree.update(20, 25);

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

    cout << "\n=== DELETE DATA ===" << endl;
    cout << "Sebelum delete (menghapus subtree dengan root 30): " << endl;
    cout << "Inorder: ";
    tree.inorder();

    tree.deleteValue(30);

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

    return 0;
}
```

Screenshots Output

```
PS D:\StrukturData> cd "d:\StrukturData\Modul10\GUIDED\" ; if ($?) { g++ main.cpp -o main } ; if ($?) { .\main }
=== 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 menjadi 25):
Inorder: 10 15 20 30 35 40 50
Setelah update (20 menjadi 25):
Inorder: 10 15 25 30 35 40 50

=== DELETE DATA ===
Sebelum delete (menghapus subtree dengan root 30):
Inorder: 10 15 25 30 35 40 50
Setelah delete (menghapus subtree dengan root 30):
Inorder: 10 15 25 35 40 50
```

Deskripsi:

Program ini merupakan implementasi struktur data AVL Tree menggunakan bahasa C++. Struktur Node menyimpan nilai data, pointer ke anak kiri dan kanan, serta tinggi node yang digunakan untuk menjaga keseimbangan pohon. Kelas BinaryTree menyediakan operasi utama seperti insert, delete, dan update, di mana setiap perubahan pada tree akan diikuti dengan perhitungan balance factor dan penerapan rotasi kiri atau kanan agar tinggi pohon tetap seimbang. Selain itu, program juga menyediakan traversal inorder, preorder, dan postorder untuk menampilkan isi tree. Pada main.cpp, program digunakan untuk mendemonstrasikan proses penyisipan data, pembaruan nilai node, serta penghapusan node tertentu, kemudian menampilkan hasil traversal untuk menunjukkan perubahan struktur tree setelah setiap operasi.

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

Unguided 1

bstree.h

Modul10 > UNGUIDED > bstree.h > hitungKedalaman(address)

```h
#ifndef BSTREE_H_INCLUDED
#define BSTREE_H_INCLUDED

#include <iostream>
using namespace std;

#define Nil NULL

typedef int infotype;
typedef struct Node *address;

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

Tabnine | Edit | Test | Explain | Document
address alokasi(infotype x);
Tabnine | Edit | Test | Explain | Document
void insertNode(address &root, infotype x);
Tabnine | Edit | Test | Explain | Document
address findNode(address root, infotype x);
Tabnine | Edit | Test | Explain | Document
void printInOrder(address root);
Tabnine | Edit | Test | Explain | Document
int hitungJumlahNode(address root);
Tabnine | Edit | Test | Explain | Document
int hitungTotalInfo(address root);
Tabnine | Edit | Test | Explain | Document
int hitungKedalaman(address root);
Tabnine | Edit | Test | Explain | Document
void printPreOrder(address root);
Tabnine | Edit | Test | Explain | Document
void printPostOrder(address root);

#endif
```

bstree.cpp

Modul10 > UNGUIDED > bstree.cpp > printPostOrder(address)

```cpp
#include "bstree.h"

Tabnine | Edit | Test | Explain | Document
address alokasi(infotype x) {
    address P = new Node;
    P->info = x;
    P->left = Nil;
    P->right = Nil;
    return P;
}

Tabnine | Edit | Test | Explain | Document
void insertNode(address &root, infotype x) {
    if (root == Nil) {
        root = alokasi(x);
    } else {
        if (x < root->info)
            insertNode(root->left, x);
        else if (x > root->info)
            insertNode(root->right, x);
    }
}

Tabnine | Edit | Test | Explain | Document
address findNode(address root, infotype x) {
    if (root == Nil) return Nil;
    if (x == root->info) return root;
    if (x < root->info) return findNode(root->left, x);
    return findNode(root->right, x);
}

Tabnine | Edit | Test | Explain | Document
void printInOrder(address root) {
    if (root != Nil) {
        printInOrder(root->left);
        cout << root->info << " ";
        printInOrder(root->right);
    }
}

Tabnine | Edit | Test | Explain | Document
int hitungJumlahNode(address root) {
    if (root == Nil) return 0;
    return 1 + hitungJumlahNode(root->left) + hitungJumlahNode(root->right);
}

Tabnine | Edit | Test | Explain | Document
int hitungTotalInfo(address root) {
    if (root == Nil) return 0;
    return root->info + hitungTotalInfo(root->left) + hitungTotalInfo(root->right);
}

Tabnine | Edit | Test | Explain | Document
int hitungKedalaman(address root) {
    if (root == Nil) return 0;
    int L = hitungKedalaman(root->left);
    int R = hitungKedalaman(root->right);
    return 1 + max(L, R);
}

Tabnine | Edit | Test | Explain | Document
void printPreOrder(address root) {
    if (root != Nil) {
        cout << root->info << " ";
        printPreOrder(root->left);
        printPreOrder(root->right);
    }
}

Tabnine | Edit | Test | Explain | Document
void printPostOrder(address root) {
    if (root != Nil) {
        printPostOrder(root->left);
        printPostOrder(root->right);
        cout << root->info << " ";
    }
}
```

```cpp
main.cpp
Modul10 > UNGUIDED > main.cpp > main()
Tabnine | Edit | Test | Explain | Document
int main() {
    address root = Nil;

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

    cout << "Inorder        : ";
    printInOrder(root);
    cout << endl;

    cout << "Preorder       : ";
    printPreOrder(root);
    cout << endl;

    cout << "Postorder      : ";
    printPostOrder(root);
    cout << endl;

    cout << "\nKedalaman      : " << hitungKedalaman(root) << endl;
    cout << "Jumlah Node    : " << hitungJumlahNode(root) << endl;
    cout << "Total          : " << hitungTotalInfo(root) << endl;
            (int)0
    return 0;
}
```

Screenshots Output

```
PS D:\StrukturData> cd "d:\StrukturData\Modul10\UNGUIDED\" ; if ($?) { g++ main.cpp -o main } ; if ($?) { .\main }
Inorder        : 1 2 3 4 5 6 7
Preorder       : 1 2 6 4 3 5 7
Postorder      : 3 5 4 7 6 2 1

Kedalaman      : 5
Jumlah Node    : 7
Total          : 28
```

Deskripsi:

Program ini merupakan implementasi struktur data Binary Search Tree (BST) menggunakan bahasa C++. Setiap node menyimpan sebuah nilai data serta pointer ke anak kiri dan kanan yang disusun berdasarkan aturan BST, yaitu nilai yang lebih kecil berada di subtree kiri dan nilai yang lebih besar berada di subtree kanan. Program menyediakan fungsi untuk melakukan alokasi node, penyisipan data, pencarian node, serta traversal inorder, preorder, dan postorder. Selain itu, terdapat fungsi untuk menghitung jumlah node, total nilai seluruh node, dan kedalaman (tinggi) pohon. Pada bagian main, program mendemonstrasikan proses memasukkan beberapa data ke dalam

BST dan menampilkan hasil traversal serta informasi statistik pohon untuk menggambarkan struktur dan isi tree yang terbentuk.

D. Kesimpulan

Berdasarkan praktikum yang telah dilakukan, dapat disimpulkan bahwa penggunaan konsep rekursif sangat efektif dalam implementasi struktur data tree, khususnya pada Binary Tree, Binary Search Tree (BST), dan AVL Tree. Rekursi mempermudah proses penelusuran, penyisipan, dan penghapusan node karena struktur tree yang bersifat hierarkis. Implementasi AVL Tree menunjukkan bagaimana mekanisme penyeimbangan otomatis melalui rotasi dapat menjaga tinggi pohon agar tetap optimal, sehingga operasi berjalan lebih efisien. Sementara itu, implementasi BST memberikan pemahaman dasar mengenai pengelolaan data terurut serta perhitungan karakteristik pohon seperti jumlah node, total nilai, dan kedalaman. Dengan demikian, praktikum ini membantu memahami penerapan rekursi dan struktur data tree dalam pemrograman serta perannya dalam membangun program yang terstruktur dan efisien.

E. Referensi

https://idcsharp.com/rekursif-recursion-pada-bahasa-pemrograman-c/

https://classroom.itats.ac.id/storage/file_tugas_dosen/1689313601_MDAxMTE4_Week%2015%20-%20TREE.pdf

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