LAPORAN PRAKTIKUM
STRUKTUR DATA

MODUL X
REKURSIF

Disusun Oleh :

NAMA : Damanik, Yohanes Geovan Ondova

NIM : 103112400022

Dosen

FAHRUDIN MUKTI WIBOWO

PROGRAM STUDI STRUKTUR DATA

FAKULTAS INFORMATIKA

TELKOM UNIVERSITY PURWOKERTO

2025

A. Dasar Teori

Rekursif:

- Pengertian: Suatu proses pengulangan sesuatu dengan cara kesamaan-diri atau suatu proses yang memanggil dirinya sendiri.
- Sub Program Rekrusif: Sub program yang memanggil dirinya sendiri selama kondisi pemanggilan dipenuhi.
- Kriteria:
  - Kondisi yang menyebabkan pemanggilan dirinya berhenti (kondisi khusus atau special edition)
  - Pemanggilan diri sub program (bila kondisi khusus tidak dipenuhi)
- Kekurangan:
  - Memerlukan memori lebih banyak untuk menyimpan *activation record* dan variabel lokal.
  - Memerlukan waktu lebih banyak untuk menangani *activation record.*

Tree

- Pengertian: Struktur data non-linear (*non-linear data structure*). Digambarkan sebagai suatu *graph* tak berarah terhubung dan tidak mengandung sirkuit.
- Karakteristik:
  - Hanya terdapat satu *node* tanpa pendahulu, disebut akar (root).
  - Semua *node* lainnya hanya mempunyai satu *node* pendahulu.

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

Guided 1

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; }
```

tree.h

```
#ifndef THREE_H
#define THREE_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
```

main.cpp

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

    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 = 3-):"
    << endl;
    cout << "Inorder    :"; tree.inorder();

    tree.deleteValue(30);

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

    return 0;

}
```

Output

```
PS C:\Users\Lenovo\Documents\STURKTUR DATA MODUL 10> cd "c:\Users\Lenovo\Documents\STURKTUR DAT
$?) { g++ main.cpp -o main } ; if ($?) { .\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 -> 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 = 3-):
Inorder    :10 15 25 30 35 40 50
Setelah delete (subtree root = 30 dihapus):
Inorder    :10 15 25 35 40 50
PS C:\Users\Lenovo\Documents\STURKTUR DATA MODUL 10>
```

Deskripsi:

Program ini mengimplementasikan struktur data pohon biner pencarian seimbang (AVL Tree) dengan tiga file utama: tree.h untuk deklarasi struktur Node dan kelas BinaryTree, tree.cpp untuk definisi fungsi-fungsi metode seperti rotasi (rotateRight, rotateLeft), penyisipan (insertNode), penghapusan (deleteNode), pembaruan (update), dan traversal (inorder, preorder, postorder), serta main.cpp sebagai pengujian dengan menyisipkan beberapa nilai, menampilkan traversal, memperbarui nilai, dan menghapus node sambil menjaga keseimbangan pohon melalui pengecekan faktor keseimbangan dan rotasi yang sesuai.

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

Unguided 1

bstree.cpp

```
#include "bstree.h"

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

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);
    }
}

address findNode(infotype x, address root) {
    if (root == Nil) return Nil;
    if (x == root->info) return root;
    if (x < root->info) return findNode(x, root->left);
    return findNode(x, root->right);
}

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

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

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

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

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

int hitungKedalaman(address root) {
```

```
    if (root == Nil) return 0;

    int kiri  = hitungKedalaman(root->left);
    int kanan = hitungKedalaman(root->right);

    return 1 + max(kiri, kanan);
}
```

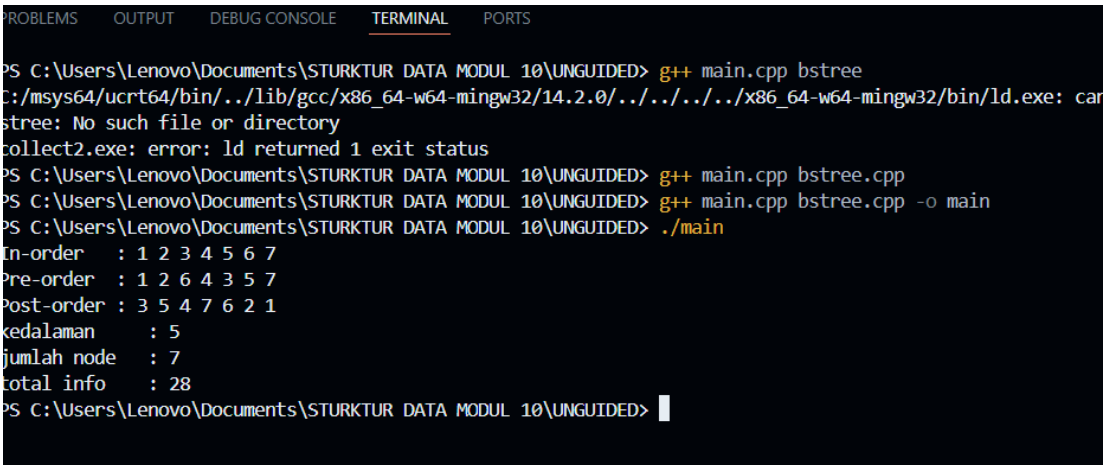
bstree.h

```
#ifndef BSTREE_H
#define BSTREE_H

#include <iostream>
using namespace std;

#define Nil NULL

typedef int infotype;

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

typedef Node* address;
address alokasi(infotype x);
void insertNode(address &root, infotype x);
address findNode(infotype x, address root);

void InOrder(address root);
void PreOrder(address root);
void PostOrder(address root);

int hitungJumlahNode(address root);
int hitungTotalInfo(address root);
int hitungKedalaman(address root);

#endif
```

Main.cpp

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

using namespace std;

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 << "In-order   : ";
    InOrder(root);
    cout << endl;

    cout << "Pre-order  : ";
    PreOrder(root);
    cout << endl;

    cout << "Post-order : ";
    PostOrder(root);
    cout << endl;

    cout << "kedalaman     : " << hitungKedalaman(root) << endl;
    cout << "jumlah node   : " << hitungJumlahNode(root) << endl;
    cout << "total info    : " << hitungTotalInfo(root) << endl;

    return 0;
}
```

Output



Deskripsi:

Program ini adalah implementasi **Binary Search Tree (BST)** dalam C++ yang terdiri dari tiga file: main.cpp, bstree.cpp, dan bstree.h. Program ini menyediakan fungsi dasar untuk membuat dan mengelola BST, termasuk penyisipan node (insertNode), pencarian (findNode), dan tiga jenis traversal (In-Order, Pre-Order, Post-Order). Selain itu, program dapat menghitung jumlah node, total nilai informasi (info) dari semua node, serta kedalaman (tinggi) pohon. Pada main.cpp, beberapa nilai disisipkan ke dalam pohon, kemudian pohon ditelusuri dan beberapa statistik pohon ditampilkan, seperti kedalaman, jumlah node, dan total informasi. Program ini mengikuti prinsip BST di mana nilai yang lebih kecil dari root ditempatkan di subpohon kiri dan nilai yang lebih besar di subpohon kanan, dengan penanganan duplikat yang diabaikan.

C. Kesimpulan

Laporan praktikum Struktur Data Modul 10 ini berfokus pada konsep rekursif dan penerapannya dalam struktur data pohon (tree). Laporan terdiri dari bagian teoritis yang menjelaskan rekursif dan karakteristik pohon, dilanjutkan dengan dua bagian implementasi: Guided yang mengembangkan AVL Tree dengan mekanisme penyeimbangan otomatis menggunakan rotasi, dan Unguided yang mengimplementasikan BST dasar beserta fungsi-fungsi utilitas seperti penghitungan node, total nilai, dan kedalaman. Kedua program menggunakan rekursi untuk operasi traversal dan modifikasi pohon, menunjukkan bagaimana rekursif mempermudah penanganan struktur hierarkis. Output dari setiap program ditampilkan dan dijelaskan untuk memverifikasi kebenaran implementasi.

D. Referensi

https://www.geeksforgeeks.org/cpp/cpp-recursion/

https://www.programiz.com/cpp-programming/recursion