LAPORAN PRAKTIKUM
STRUKTUR DATA

MODUL 10
TREE

Disusun Oleh :
NAMA : AGUNG RAMADHAN
NIM : 103112430060

Dosen:
FAHRUDIN MUKTI WIBOWO

PROGRAM STUDI STRUKTUR DATA
FAKULTAS INFORMATIKA
TELKOM UNIVERSITY PURWOKERTO

2025

A. Dasar Teori

Queue adalah salah satu struktur data linear dalam informatika yang menerapkan prinsip First In First Out (FIFO), yaitu elemen yang pertama kali masuk akan menjadi elemen pertama yang keluar. Struktur data ini memiliki dua operasi utama, yaitu *enqueue* untuk menambahkan data ke bagian belakang (rear) dan *dequeue* untuk menghapus data dari bagian depan (front). Queue banyak digunakan dalam berbagai sistem komputer, seperti penjadwalan proses pada sistem operasi, antrian printer, serta manajemen permintaan pada server, karena mampu merepresentasikan proses antrian secara teratur dan efisien.

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

Guided 1

Kode 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
```

Kode 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;
}
```

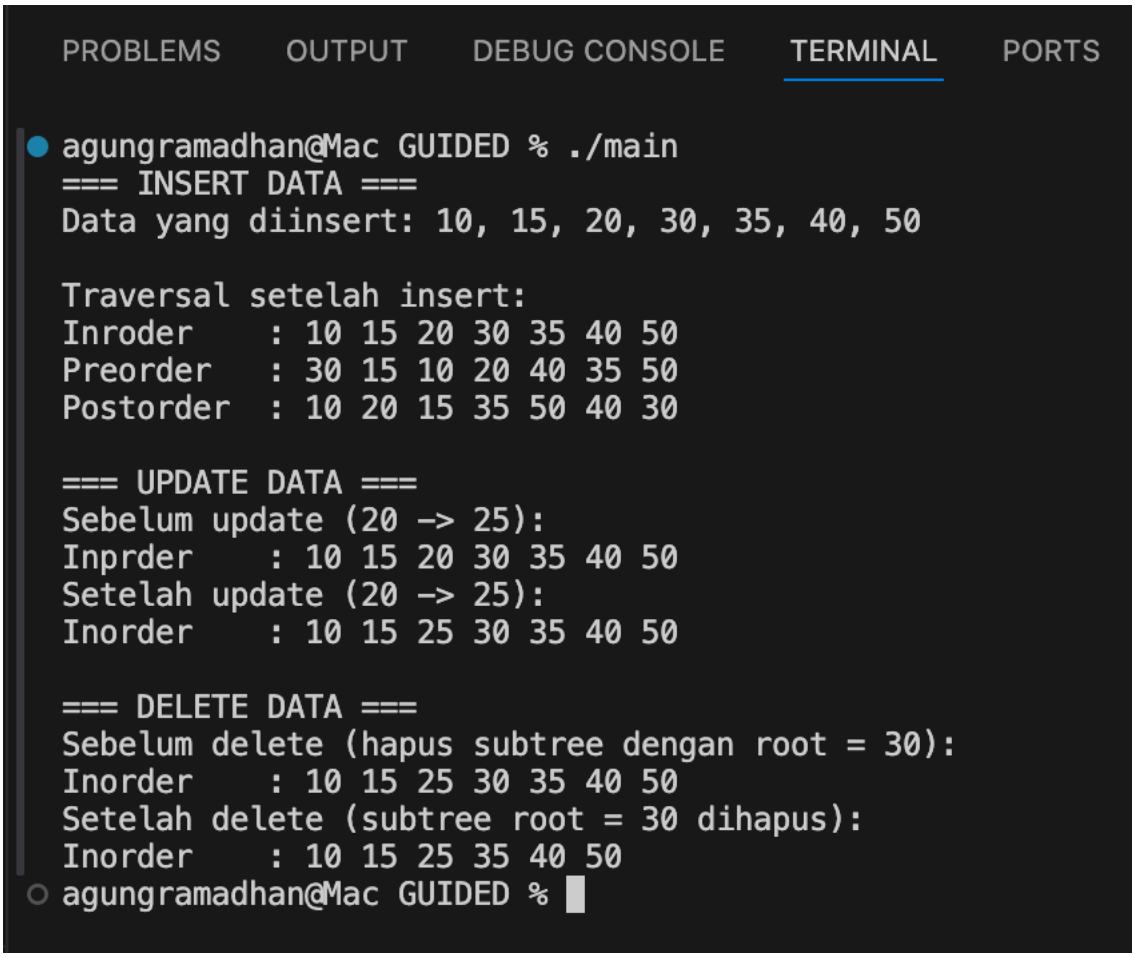
```
void BinaryTree::deleteValue(int value) {
    root = deleteNode(root, value);
}

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; }
```

Kode 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 << "Inroder    : "; tree.inorder();
    cout << "Preorder   : "; tree.preorder();
    cout << "Postorder  : "; tree.postorder();

    cout << "\n=== UPDATE DATA ===" << endl;
    cout << "Sebelum update (20 -> 25):" << endl;
    cout << "Inprder    : "; 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;
}
```

Screenshot:



Deskripsi:

Kode tersebut merupakan implementasi binary search tree yang diseimbangkan menggunakan konsep AVL Tree dalam bahasa C++. Program terdiri dari tiga file utama, yaitu tree.h yang mendefinisikan struktur node dan kelas BinaryTree, tree.cpp yang mengimplementasikan operasi inti seperti insert, delete, update, rotasi kiri dan kanan untuk menjaga keseimbangan pohon, serta traversal inorder, preorder, dan postorder, dan main.cpp yang berfungsi sebagai pengujian program. Pohon secara otomatis menjaga keseimbangan tinggi melalui perhitungan *balance factor* dan rotasi, sehingga operasi pencarian, penyisipan, dan penghapusan data tetap efisien. Program ini juga menunjukkan penggunaan traversal untuk menampilkan isi tree sebelum dan sesudah operasi insert, update, dan delete.

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

Unguided 1

Kode bstree.h

```h
#ifndef BSTREE_H
#define BSTREE_H

#include <iostream>
using namespace std;

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);
void preOrder(address root);
void postOrder(address root);

int hitungNode(address root);
int hitungTotal(address root);
int hitungKedalaman(address root, int start);

#endif
```

Kode bstree.cpp

```cpp
#include "bstree.h"

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

void insertNode(address &root, infotype x)
{
    if (root == NULL)
```

```
    {
        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 == NULL || root->info == x)
    {
        return root;
    }
    else if (x < root->info)
    {
        return findNode(x, root->left);
    }
    else
    {
        return findNode(x, root->right);
    }
}

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

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

void postOrder(address root)
{
```

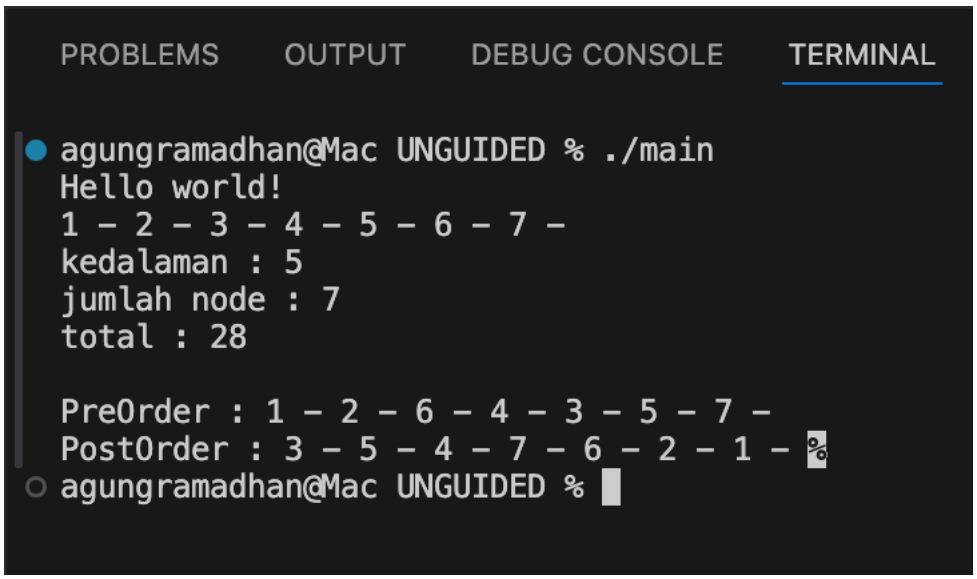
```
    if (root != NULL)
    {
        postOrder(root->left);
        postOrder(root->right);
        cout << root->info << " - ";
    }
}

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

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

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

Kode main.cpp

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

using namespace std;

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

    address root = NULL;
```

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

    inOrder(root);
    cout << endl;

    cout << "kedalaman : " << hitungKedalaman(root, 0) << endl;
    cout << "jumlah node : " << hitungNode(root) << endl;
    cout << "total : " << hitungTotal(root) << endl;

    cout << "\nPreOrder : ";
    preOrder(root);

    cout << "\nPostOrder : ";
    postOrder(root);

    return 0;
}
```

Screenshots Output



Deskripsi:
Kode ADT Binary Search Tree (BST) / AVL Tree tersebut mengimplementasikan struktur data pohon biner terurut yang menjaga keseimbangan tinggi pohon menggunakan metode AVL. ADT ini menyediakan operasi dasar seperti *insert*, *delete*, dan *update* data, serta tiga jenis traversal yaitu *inorder*, *preorder*, dan *postorder*. Setiap node menyimpan data, pointer

ke anak kiri dan kanan, serta nilai tinggi (*height*) yang digunakan untuk menghitung *balance factor*. Mekanisme rotasi kiri dan kanan diterapkan secara otomatis setelah proses penyisipan dan penghapusan untuk memastikan pohon tetap seimbang, sehingga performa operasi tetap optimal. ADT ini memisahkan definisi struktur dan operasi dalam file header dan implementasi, sehingga kode bersifat modular, terstruktur, dan mudah digunakan kembali.

D. Kesimpulan

Kesimpulan dari kode ini yaitu program tersebut berhasil mengimplementasikan struktur data pohon biner terurut yang efisien dan seimbang. Operasi dasar seperti *insert*, *delete*, dan *update* data dapat dilakukan dengan baik, sementara mekanisme perhitungan tinggi dan *balance factor* serta rotasi kiri dan kanan menjaga agar pohon tidak menjadi timpang. Selain itu, penyediaan traversal *inorder*, *preorder*, dan *postorder* memudahkan proses penelusuran dan penampilan data. Secara keseluruhan, kode ini menunjukkan penerapan konsep ADT BST/AVL yang terstruktur, modular, dan efektif untuk pengelolaan data secara hierarkis dengan kompleksitas operasi yang tetap optimal.

E. Referensi

Suwanty & Pribadi, O. (2015). *Metode AVL Tree untuk Penyeimbangan Tinggi Binary Tree*. Jurnal TIMES, Vol. IV No. 2, 61–65.

Jabde, M., Upasani, M., Jadhav, V. A., Bachhav, L. P. (2016). *AVL Tree Implementation*. International Journal on Recent and Innovation Trends in Computing and Communication, 4(1), 40–47.

Monika (2015). *Comparison Based Analysis of Advanced Tree Structures*. IJRDO – Journal of Computer Science Engineering, 1(3), 70–78.

Štrbac-Savić, S., Tomašević, M., Minchev, Z., & Maček, N. (2023). *Comparative Performance Evaluation of Suboptimal Binary Search Trees*. Journal of Computer and Forensic Sciences.