LAPORAN PRAKTIKUM
STRUKTUR DATA

MODUL 10
TREE

Disusun Oleh :
NAMA : FARIS WALID AWWAL AIDI
NIM : 103112430133

Dosen
FAHRUDIN MUKTI WIBOWO

PROGRAM STUDI STRUKTUR DATA
FAKULTAS INFORMATIKA
TELKOM UNIVERSITY PURWOKERTO
2025

A. Dasar Teori

Struktur data *tree* merupakan jenis struktur data non-linear yang digunakan untuk merepresentasikan data yang memiliki hubungan hierarkis, di mana setiap elemen, yang disebut *node* atau simpul, dapat terhubung dengan beberapa simpul anak. Berbeda dengan struktur data linear seperti *list*, *stack*, dan *queue*, *tree* memungkinkan akses dan penyimpanan data yang lebih efisien untuk kasus-kasus tertentu, seperti sistem berkas, basis data, atau ekspresi matematika. Dalam sebuah *tree*, terdapat satu simpul khusus yang disebut *root* atau akar, yang merupakan simpul paling atas dan tidak memiliki simpul pendahulu (*parent*). Setiap simpul selain *root* hanya memiliki satu *parent* dan dapat memiliki nol atau lebih simpul anak (*child*). Simpul yang tidak memiliki anak disebut *leaf* (daun), sedangkan simpul yang memiliki anak disebut *internal node*. Konsep dasar ini juga mencakup terminologi penting lain seperti *edge* (sisi/garis penghubung), *path* (lintasan), *depth* (kedalaman), dan *height* (tinggi), yang menentukan posisi dan jarak simpul dalam struktur.

Salah satu implementasi *tree* yang paling umum adalah *Binary Tree*, di mana setiap simpul dibatasi hanya boleh memiliki maksimum dua simpul anak: anak kiri (*left child*) dan anak kanan (*right child*). Lebih lanjut, terdapat spesialisasi dari *Binary Tree*, yaitu *Binary Search Tree* (BST), yang menambahkan aturan pengurutan ketat untuk mempermudah operasi pencarian, penyisipan (*insert*), dan penghapusan (*delete*). Dalam BST, semua nilai pada *left subtree* (pohon bagian kiri) harus lebih kecil dari nilai simpul *parent*, dan semua nilai pada *right subtree* (pohon bagian kanan) harus lebih besar dari nilai simpul *parent*. Aturan ini memastikan bahwa pencarian elemen dapat dilakukan dengan sangat cepat rata-rata dalam waktu O(log n)mirip dengan cara kerja algoritma *binary search* pada array yang terurut. Pemahaman mendalam mengenai jenis-jenis *tree* dan cara kerja operasinya adalah fundamental dalam praktikum struktur data.

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

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
```

tree.cpp

```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;
```

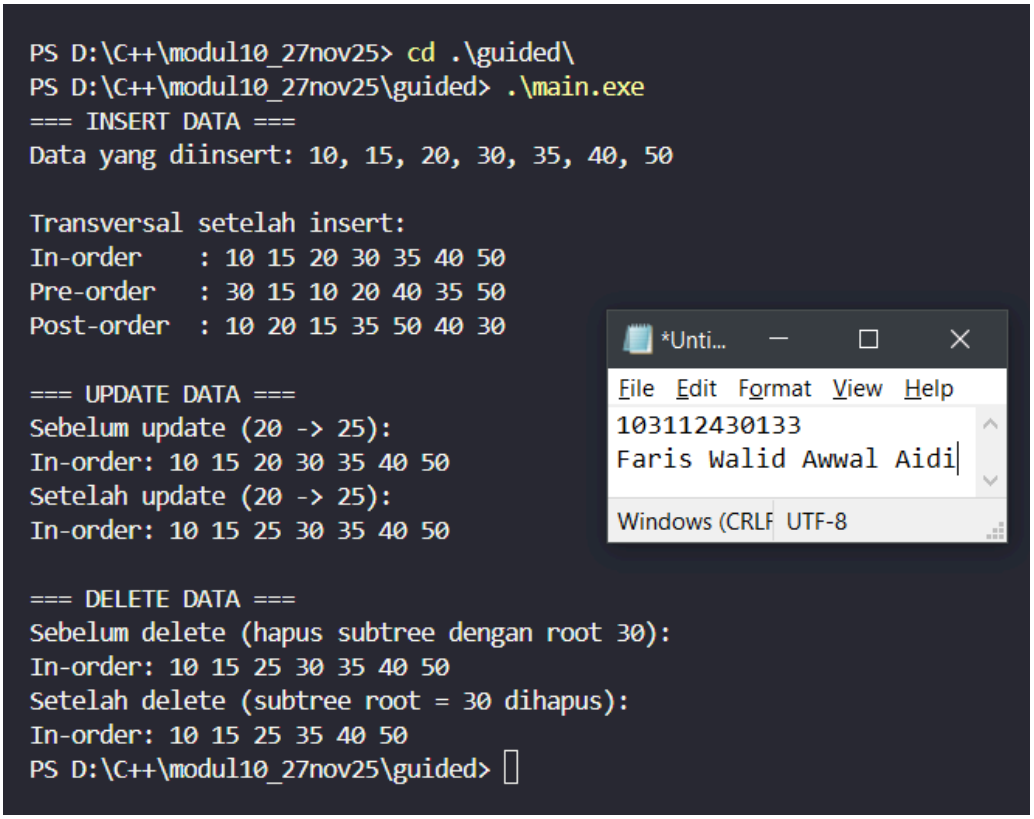
```
    postOrder(node->left);
    postOrder(node->right);
    cout << node->data << " ";
}

void BinaryTree::inOrder() { inOrder(root); cout << endl; }
void BinaryTree::preOrder() { preOrder(root); cout << endl; }
void BinaryTree::postOrder() { postOrder(root); cout << endl; }
```

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);

    // TRANSVERSAL
    cout << "Data yang diinsert: 10, 15, 20, 30, 35, 40, 50" << endl;
    cout << "\nTransversal setelah insert:" << endl;
    cout << "In-order    : "; tree.inOrder();
    cout << "Pre-order   : "; tree.preOrder();
    cout << "Post-order  : "; tree.postOrder();

    // UPDATE DATA
    cout << "\n=== UPDATE DATA ===" << endl;
    cout << "Sebelum update (20 -> 25):" << endl;
    cout << "In-order: "; tree.inOrder();
```

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

    cout << "Setelah update (20 -> 25):" << endl;
    cout << "In-order: "; tree.inOrder();

    // DELETE DATA
    cout << "\n=== DELETE DATA ===" << endl;
    cout << "Sebelum delete (hapus subtree dengan root 30):" << endl;
    cout << "In-order: "; tree.inOrder();

    tree.deleteValue(30);

    cout << "Setelah delete (subtree root = 30 dihapus):" << endl;
    cout << "In-order: "; tree.inOrder();

    return 0;
}
```

Screenshots Output



Deskripsi:

Program ini merupakan implementasi dari struktur data tree seimbang otomatis yang dikenal sebagai AVL Tree (Adelson-Velsky and Landis Tree), menggunakan bahasa pemrograman C++. Tujuan utama dari AVL Tree adalah memastikan bahwa struktur pohon selalu seimbang yaitu, perbedaan tinggi (keseimbangan) antara subtree kiri dan kanan pada setiap node tidak pernah melebihi satu. Hal ini dicapai dengan secara otomatis melakukan operasi rotasi (rotateRight dan rotateLeft) segera setelah penyisipan (insert) atau penghapusan (delete) yang mungkin menyebabkan ketidakseimbangan. Kelas BinaryTree mengelola root dari pohon dan menyediakan fungsi publik untuk berinteraksi, seperti insert(value), deleteValue(value), dan update(oldVal, newVal). Selain itu, program ini juga menyertakan tiga metode traversal standar inOrder, preOrder, dan postOrder untuk menampilkan isi pohon secara terurut dan sistematis, membuktikan bahwa operasi manipulasi data (penyisipan, pembaruan, dan penghapusan) telah dilakukan dengan benar sambil tetap mempertahankan sifat keseimbangan pohon.

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

bstree.h

```
#ifndef BSTREE_H
#define BSTREE_H

typedef int infotype;

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

typedef Node* address;
#define Nil NULL

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);

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

#endif
```

bstree.cpp

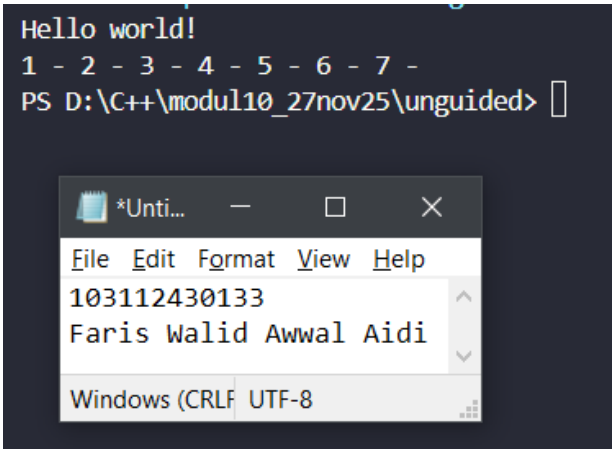
```cpp
#include <iostream>
#include "bstree.h"

using namespace std;

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

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

int hitungJumlahNode(address root) {
    if (root == Nil) {
        return 0;
```

```
    } else {
        return 1 + hitungJumlahNode(root->left) + hitungJumlahNode(root->right);
    }
}

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

int hitungKedalaman(address root) {
    if (root == Nil) {
        return -1;
    } else {

        int leftDepth = hitungKedalaman(root->left);
        int rightDepth = hitungKedalaman(root->right);

        return 1 + max(leftDepth, rightDepth);
    }
}

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 << " - ";
    }
}
```

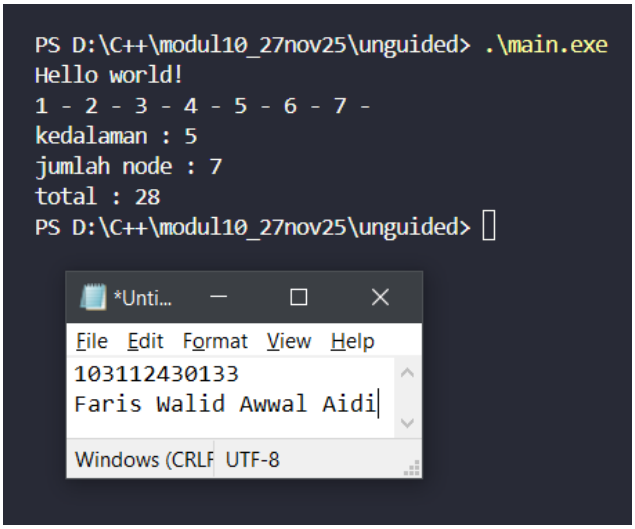
main.cpp

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

using namespace std;

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

    address root = Nil;

    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;

    int kedalaman = hitungKedalaman(root) + 1;
    cout << "kedalaman : " << kedalaman << endl;

    int jumlahNode = hitungJumlahNode(root);
    cout << "jumlah node : " << jumlahNode << endl;

    int totalInfo = hitungTotalInfo(root);
    cout << "total : " << totalInfo << endl;

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

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

    return 0;
}
```

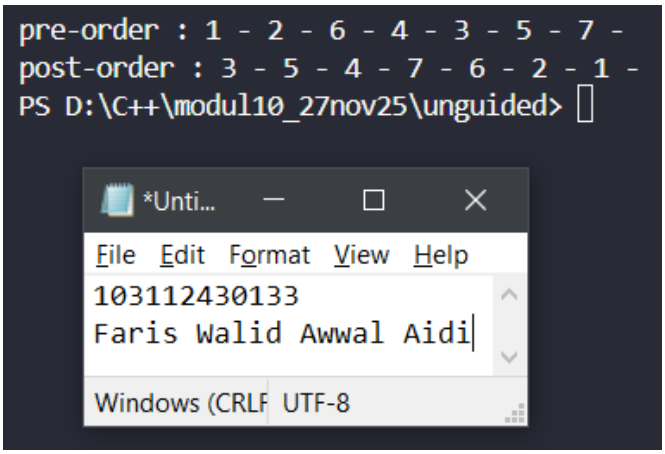
Screenshot Output 1



Screenshot Output 2



Screenshot Output 3



Deskripsi:

Program utama ini dirancang untuk menguji fungsionalitas dasar dari struktur data Binary Search Tree (BST) yang diimplementasikan melalui Abstract Data Type (ADT). Program ini pertama-tama membangun BST dengan menyisipkan nilai-nilai 1, 2, 6, 4, 5, 3, dan 7 secara berurutan, memastikan properti BST (nilai kiri lebih kecil, nilai kanan lebih besar) tetap terjaga. Setelah struktur pohon terbentuk, program kemudian menampilkan isi pohon menggunakan tiga metode traversal utama: In-order, yang menghasilkan urutan nilai yang terurut; Pre-order, di mana root dikunjungi terlebih dahulu; dan Post-order, di mana root dikunjungi terakhir, sesuai dengan perintah Latihan 1 dan 3. Selanjutnya, program mengimplementasikan dan menguji fungsi rekursif yang diminta pada Latihan 2, yaitu hitungJumlahNode untuk mengetahui total simpul, hitungTotalInfo untuk menjumlahkan semua nilai data dalam simpul, dan hitungKedalaman untuk menentukan tinggi atau kedalaman maksimum dari BST yang telah dibuat. Keseluruhan program berfungsi sebagai validasi fungsionalitas BST dan penggunaan konsep rekursif dalam mengelola dan menganalisis struktur data non-linear.

D. Kesimpulan

Berdasarkan implementasi program ini, dapat disimpulkan bahwa Binary Search Tree (BST) adalah struktur data yang sangat efisien untuk operasi pencarian, penyisipan, dan penghapusan data terurut. Keberhasilan dalam memproses data dan menghitung metrik (jumlah node, total info, dan kedalaman) sepenuhnya bergantung pada penggunaan fungsi rekursif yang merupakan pendekatan alami dan elegan untuk memproses struktur hierarkis seperti tree. Melalui implementasi traversal (In-order, Pre-order, dan Post-order), program membuktikan pemahaman yang mendalam tentang cara mengakses setiap simpul secara sistematis. Secara khusus, traversal In-order memvalidasi properti BST dengan menghasilkan data dalam urutan yang meningkat, sementara fungsi rekursif metrik menunjukkan bagaimana masalah yang kompleks (misalnya, menghitung tinggi pohon) dapat dipecah menjadi masalah yang lebih kecil, yang merupakan inti dari pemrograman rekursif dalam konteks struktur data tree.

E. Referensi

Raharjo, Budi. 2025. *Buku Pemrograman C++ Mudah dan Cepat Menjadi Master C.*

Wikipedia contributors. (2024, 8 Mei). C++. Wikipedia, Ensiklopedia Bebas. Diakses pada 2 Desember 2025, dari https://id.wikipedia.org/wiki/C%2B%2B