LAPORAN PRAKTIKUM
STRUKTUR DATA

MODUL VII

STACK

Disusun Oleh :
Muhammad Fathammubina
NIM : 103112430188

Dosen
FAHRUDIN MUKTI WIBOWO

PROGRAM STUDI STRUKTUR DATA
FAKULTAS INFORMATIKA
TELKOM UNIVERSITY PURWOKERTO
2025

A. Dasar Teori

Rekursif adalah teknik pemrograman di mana sebuah fungsi memanggil dirinya sendiri untuk menyelesaikan masalah yang lebih kecil hingga mencapai kondisi berhenti (base case). Rekursif digunakan ketika solusi suatu masalah dapat dipecah menjadi pola berulang, seperti perhitungan pangkat, faktorial, dan traversing tree. Walaupun mempermudah penulisan kode, rekursif membutuhkan lebih banyak memori karena setiap pemanggilan fungsi menyimpan activation record di stack.

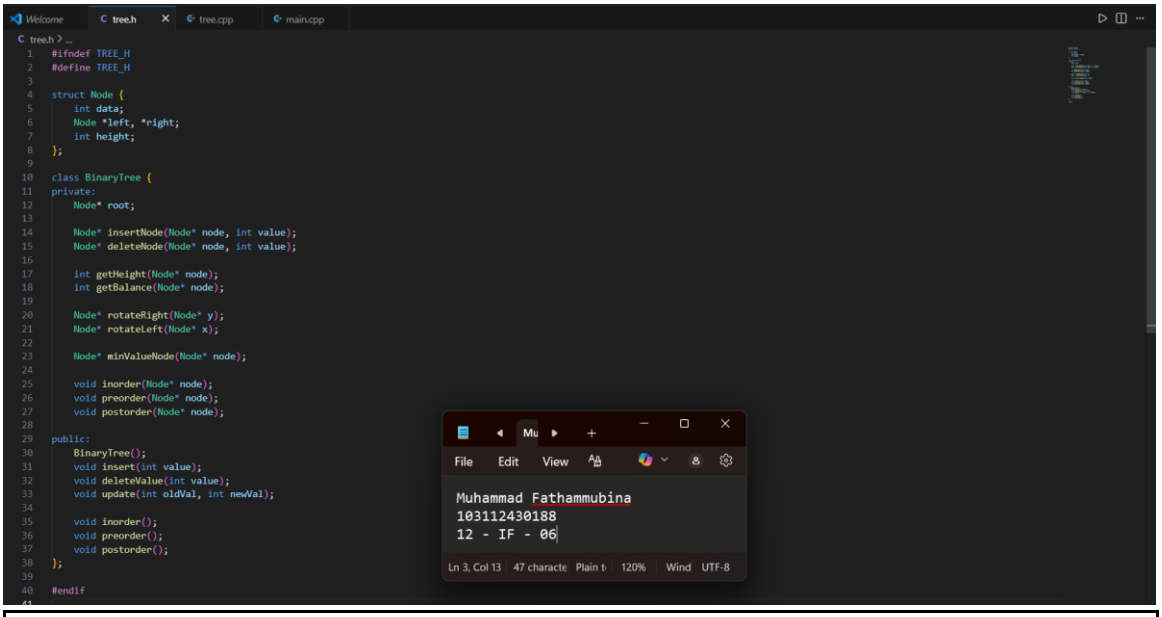
Tree adalah struktur data non-linear yang terdiri dari kumpulan node yang saling terhubung secara hierarkis. Tree memiliki sebuah akar (root), node anak (child), node orangtua (parent), node daun (leaf), serta node internal. Salah satu bentuk tree yang paling umum adalah Binary Tree, yaitu tree di mana setiap node memiliki maksimal dua anak: kiri (left child) dan kanan (right child). Bentuk khusus dari binary tree adalah Binary Search Tree (BST), yaitu tree terurut dengan ketentuan: nilai di left child lebih kecil daripada parent, nilai di right child lebih besar daripada parent.

BST memudahkan operasi insert, search, dan traversal seperti in-order, pre-order, dan post-order. In-order traversal menghasilkan data dalam urutan terurut naik. Rekursif biasanya digunakan untuk traversal BST karena bentuk strukturnya yang bercabang dan cocok diproses secara berulang.

B. Guided

Guided 1

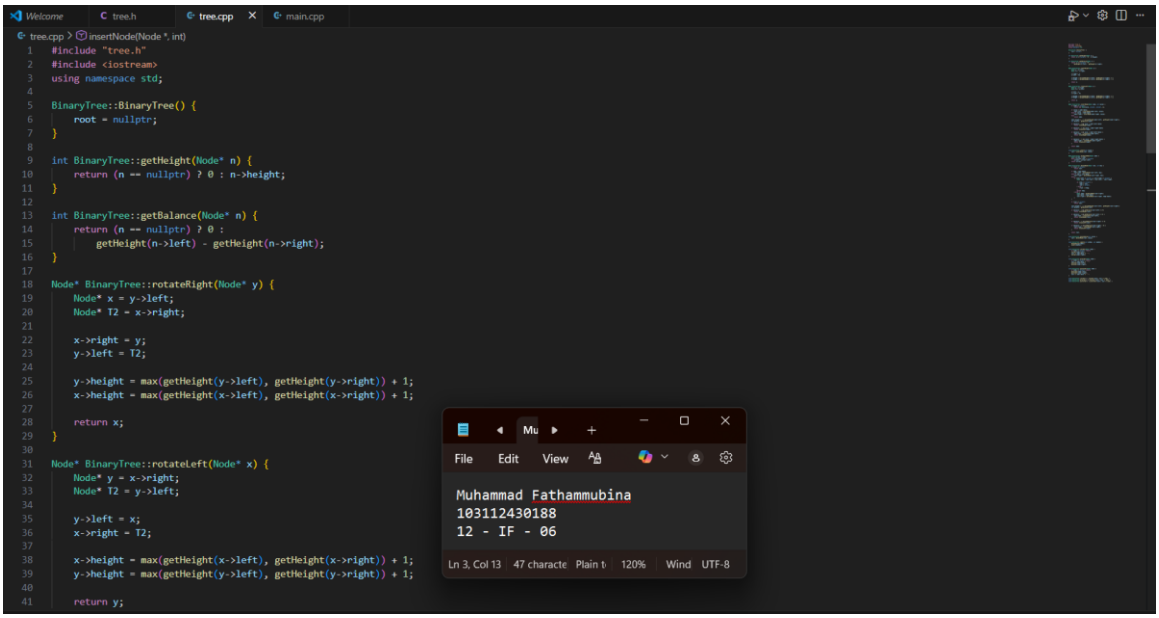
tree.h




```
#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



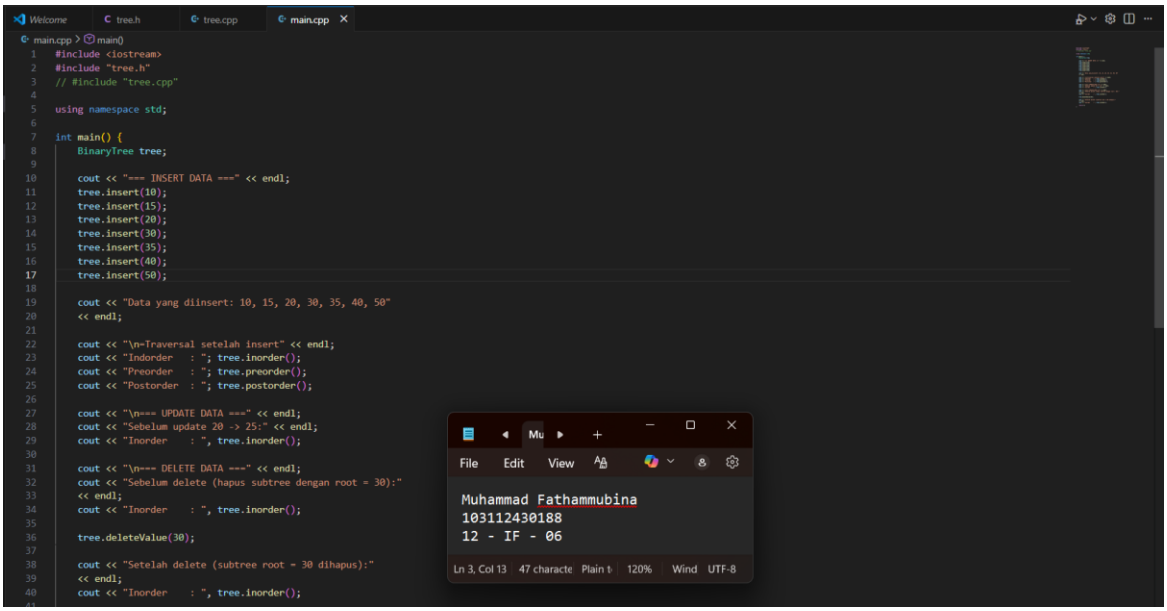

```cpp
#include "tree.h"
#include <iostream>
using namespace std;

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)
        return new Node{value, nullptr, nullptr, 1};

    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; }
```

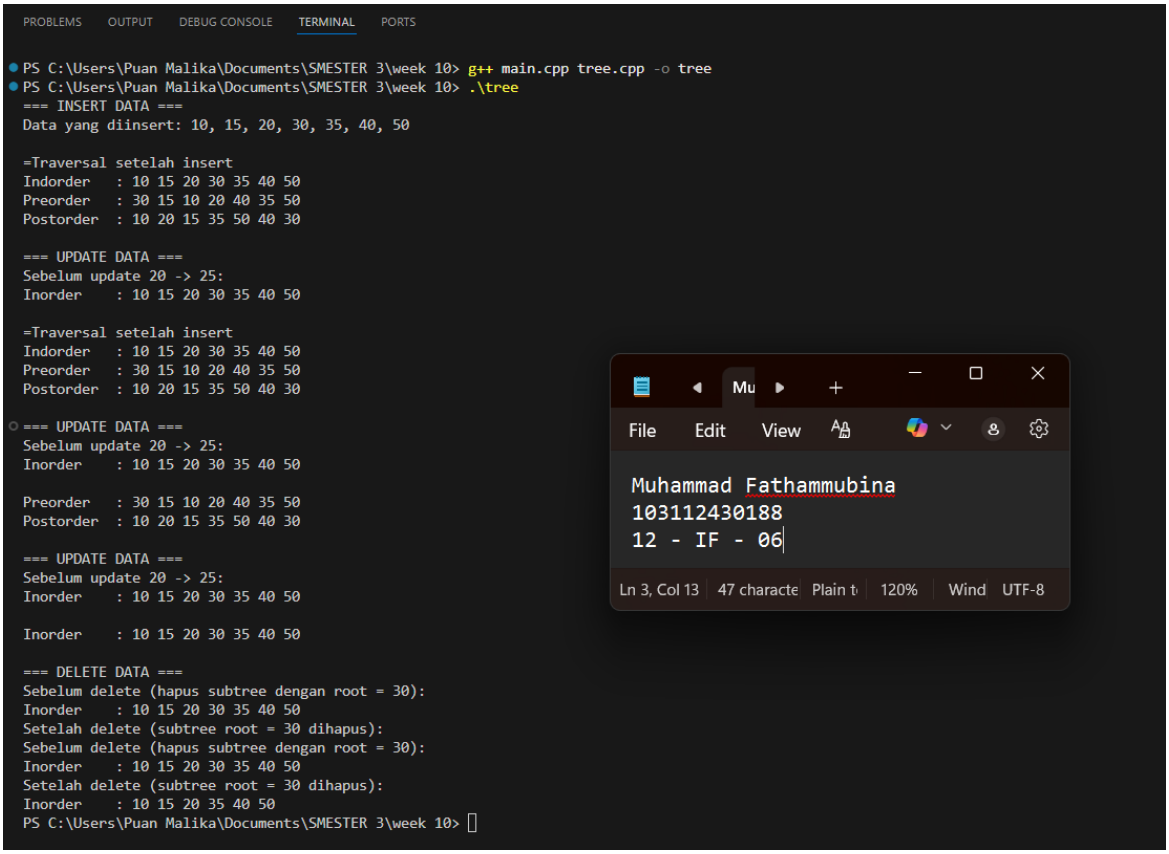
main.cpp




```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 << "\n=Traversal setelah insert" << endl;
    cout << "Indorder   : "; 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();
```

```
    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



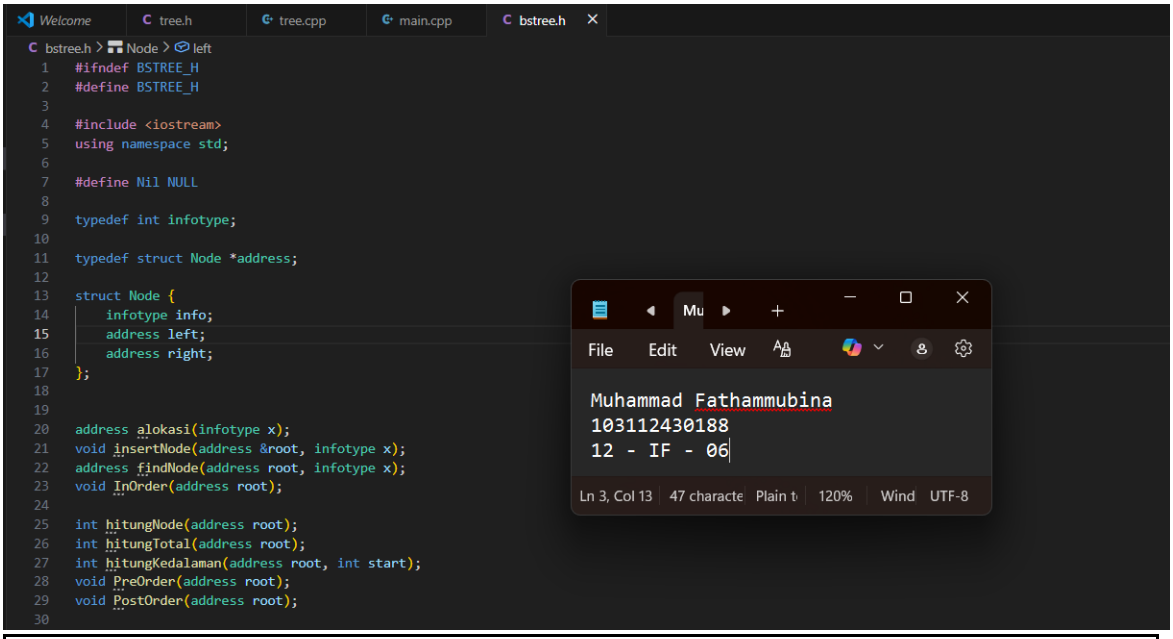
Deskripsi:

Program diatas mengimplementasikan Binary Tree berbasis AVL Tree, yaitu struktur data pohon biner terurut yang selalu menjaga keseimbangan tinggi subtree melalui proses rotasi. Program menyediakan operasi insert, delete, dan update data, di mana setiap penambahan atau penghapusan node akan otomatis menyeimbangkan kembali tree menggunakan rotasi left, right, maupun perpaduannya. Setiap node menyimpan nilai data,

pointer anak kiri-kanan, serta informasi tinggi node untuk perhitungan balance factor. Program juga menyediakan tiga jenis traversal yaitu inorder, preorder, dan postorder untuk menampilkan isi tree. Pada fungsi utama, program melakukan proses insert beberapa data, menampilkan hasil traversal, kemudian melakukan operasi update serta delete yang menunjukkan perubahan struktur tree setelah penyeimbangan otomatis dilakukan. Program ini menunjukkan bagaimana AVL Tree bekerja menjaga kestabilan struktur agar operasi pencarian dan manipulasi data tetap efisien.

Unguided 1

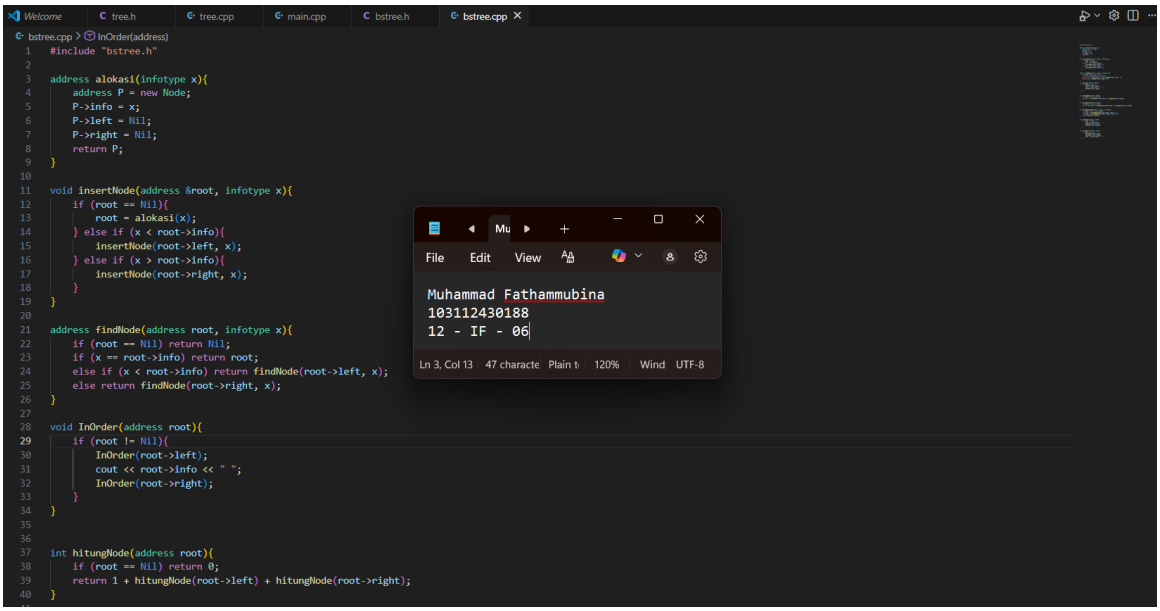
bsstree.h




```
#ifndef BSTREE_H
#define BSTREE_H

#include <iostream>
using namespace std;

#define Nil NULL

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(address root, infotype x);
void InOrder(address root);

int hitungNode(address root);
int hitungTotal(address root);
int hitungKedalaman(address root, int start);
void PreOrder(address root);
void PostOrder(address root);

#endif
```

bstree.cpp

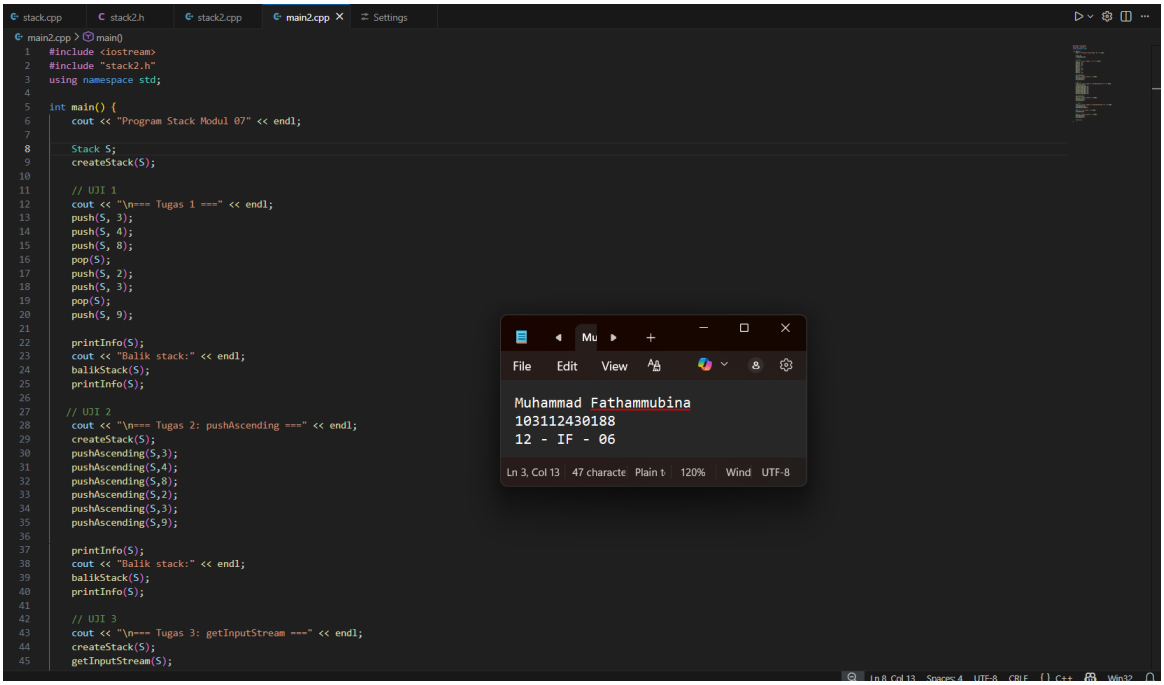



```
#include "bstree.h"

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

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


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

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

int hitungKedalaman(address root, int start){
    if (root == Nil) return start;
    int kiri = hitungKedalaman(root->left, start + 1);
    int kanan = hitungKedalaman(root->right, start + 1);
    return max(kiri, kanan);
}

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 << " ";
    }
}
```

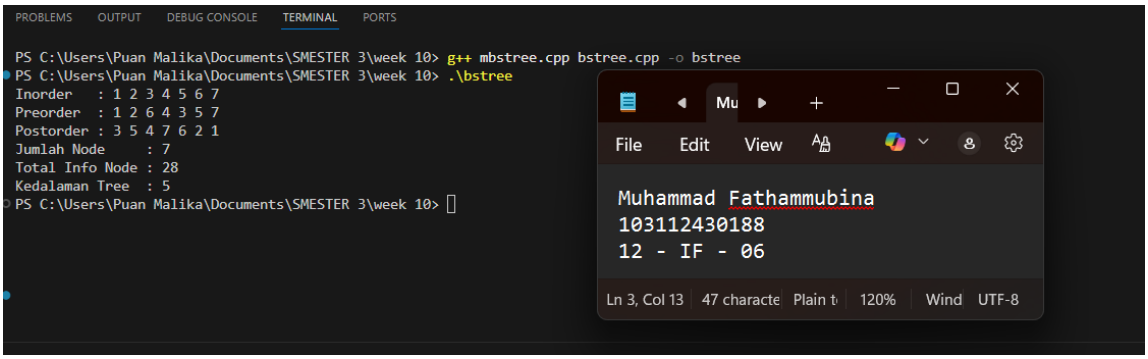
Main2.cpp




```
#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 << "Inorder    : ";
    InOrder(root);
```

```
    cout << endl;

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

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

    cout << "Jumlah Node     : " << hitungNode(root) << endl;
    cout << "Total Info Node : " << hitungTotal(root) << endl;
    cout << "Kedalaman Tree  : " << hitungKedalaman(root, 0) << endl;

    return 0;
}
```

Screenshots Output



Deskripsi:

Program ini mengimplementasikan Binary Search Tree (BST) menggunakan pointer, dengan operasi utama seperti insert untuk menambah data sesuai aturan BST, find untuk mencari nilai, serta traversal inorder, preorder, dan postorder untuk menampilkan isi tree. Program juga memiliki fungsi rekursif untuk menghitung jumlah node, menjumlahkan seluruh nilai node, dan mengukur kedalaman maksimal tree. Pada bagian utama, beberapa data dimasukkan ke BST kemudian program menampilkan hasil traversal serta informasi jumlah node, total nilai, dan kedalaman tree. Program ini menunjukkan cara kerja BST dalam penyimpanan dan pengolahan data secara terurut.

C. Kesimpulan

Dari praktikum ini dapat disimpulkan bahwa Binary Search Tree (BST) merupakan struktur data yang efisien untuk menyimpan dan mengolah data secara terurut menggunakan konsep rekursif. Melalui implementasi operasi insert, find, dan traversal, BST mampu menampilkan data dalam berbagai urutan dan mempermudah proses pencarian. Fungsi rekursif tambahan seperti perhitungan jumlah node, total nilai, dan kedalaman tree menunjukkan bahwa BST sangat fleksibel untuk berbagai kebutuhan

pengolahan data. Secara keseluruhan, program berhasil menggambarkan cara kerja BST dalam membangun, menelusuri, serta menganalisis struktur pohon secara efektif.

D. Referensi

Muliono, R. (2017). Abstract Data Type (ADT). Universitas Multimedia Nusantara.

Ulfah, J. (n.d.). Modul 9 – Fungsi Rekursi dan Tree [PDF]. Scribd