LAPORAN PRAKTIKUM
STRUKTUR DATA

MODUL X
TREE

Disusun Oleh :

Nama: Dina Nadhyfa

NIM : 103112430052

Dosen

Fahrudin Mukti Wibowo

PROGRAM STUDI TEKNIK INFORMATIKA

FAKULTAS INFORMATIKA

TELKOM UNIVERSITY PURWOKERTO

2025

A. Dasar Teori

Queue adalah struktur data linear yang mengikuti prinsip FIFO (First In, First Out), di mana elemen yang pertama kali dimasukkan akan menjadi elemen pertama yang dikeluarkan. Dalam queue, elemen baru selalu ditambahkan di bagian belakang (rear/back) dan elemen dihapus dari bagian depan (front). Struktur ini banyak digunakan untuk menangani antrian data dalam berbagai aplikasi, seperti buffer perangkat keras untuk mengelola perbedaan kecepatan antara CPU dan perangkat input/output. Implementasi queue dapat dilakukan dengan menggunakan array atau linked list, namun dalam bahasa pemrograman C++ telah tersedia implementasi queue yang efisien melalui Standard Template Library (STL) menggunakan kelas std::queue<T>, di mana T adalah tipe elemen yang disimpan.

Operasi dasar pada queue di C++ meliputi penambahan elemen dengan fungsi push() (enqueue), penghapusan elemen dengan fungsi pop() (dequeue), serta akses ke elemen depan dan belakang dengan fungsi front() dan back(). Selain itu, terdapat fungsi untuk memeriksa apakah queue kosong (empty()) dan ukuran queue (size()), di mana semua operasi ini memiliki kompleksitas waktu O(1). Queue sering digunakan dalam berbagai algoritma dan aplikasi, seperti pemrograman level-order traversal pohon, Breadth-First Search (BFS) pada graf, penjadwalan CPU, dan manajemen memori. Dalam konteks praktikum C++, queue sangat berguna untuk mensimulasikan antrian proses atau buffer data secara real-time. Semua informasi ini mendukung pemahaman dan implementasi praktis queue dalam pengembangan program berbasis C++.

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

Guided

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_H
```

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) {
```

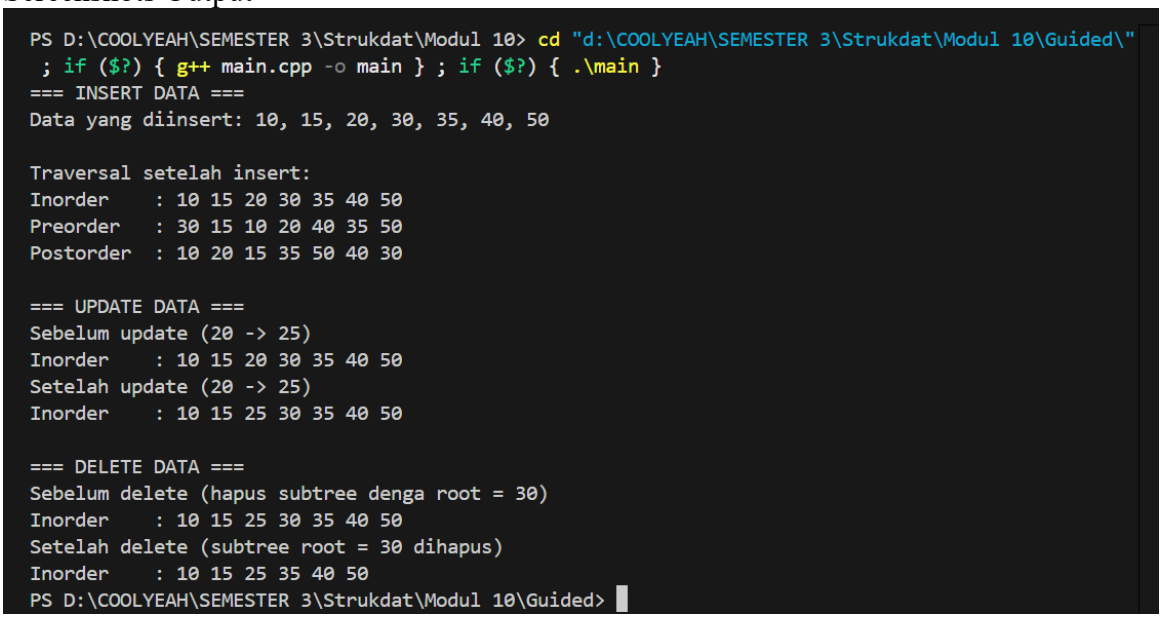
```
    if (node == nullptr) return;
    postOrder(node->left);
    postOrder(node->right);
    cout << node->data << " ";
}

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

main.cpp

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

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 denga 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 di atas mendemonstrasikan implementasi struktur data Binary Search Tree (BST) dengan tiga jenis traversal yang berbeda: inorder, preorder, dan postorder. Program dimulai dengan memasukkan tujuh nilai data yaitu 10, 15, 20, 30, 35, 40, dan 50 ke dalam struktur pohon biner. Setelah data diinsert, program menampilkan hasil penelusuran menggunakan ketiga metode traversal tersebut, di mana inorder menghasilkan urutan 10 15 20 30 35 40 50 (urutan terurut), preorder menghasilkan 30 15 10 20 40 35 50 (root-left-right), dan postorder menghasilkan 10 20 15 35 50 40 30 (left-right-root).

Selanjutnya, program menunjukkan operasi update data di mana nilai 20 diubah menjadi 25. Sebelum update, traversal inorder menampilkan urutan yang sama, kemudian setelah update dilakukan, angka 20 berhasil diganti dengan 25 sehingga hasil inorder traversal menjadi 10 15 25 30 35 40 50. Terakhir, program melakukan operasi delete data dengan menghapus node yang memiliki nilai 30, yang merupakan root dari pohon tersebut. Proses penghapusan ini dilakukan dengan mencari subtree yang sesuai untuk menggantikan posisi node yang dihapus agar struktur BST tetap terjaga. Setelah node 30 dihapus, hasil inorder traversal menunjukkan urutan 10 15 25 35 40 50, membuktikan bahwa operasi penghapusan berhasil dilakukan dan pohon tetap mempertahankan properti BST-nya.

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

1. Soal 1

a) Bstree.h

```h
#ifndef BSTREE_H
#define BSTREE_H

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

#endif
```

b) Bstree.cpp

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

using namespace std;

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

/* insert node ke BST */
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);
    }
    /* jika sama, tidak dimasukkan */
}

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

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

c) Main.cpp

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

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,6);
    insertNode(root,7);

    InOrder(root);
```

```
    return 0;
}
```

Screenshot Output:

```
PS D:\COOLYEAH\SEMESTER 3\Strukdat\Modul 10\Unguided> g++ main.cpp bstree.cpp -o main
PS D:\COOLYEAH\SEMESTER 3\Strukdat\Modul 10\Unguided> .\main.exe
Hello World
1 - 2 - 3 - 4 - 5 - 6 - 7 -
PS D:\COOLYEAH\SEMESTER 3\Strukdat\Modul 10\Unguided>
```

Deskripsi:

Program ini merupakan implementasi Binary Search Tree (BST) menggunakan struktur data linked list dalam bahasa C++. Program mendefinisikan sebuah ADT BST yang terdiri dari struktur node dengan tiga komponen, yaitu info untuk menyimpan data bertipe integer, serta pointer left dan right untuk menunjuk ke anak kiri dan kanan. Operasi utama yang diimplementasikan meliputi alokasi node, penyisipan data ke dalam BST (insertNode) berdasarkan aturan BST (nilai lebih kecil ke kiri dan lebih besar ke kanan), pencarian data (findNode), serta penelusuran tree secara InOrder. Pada fungsi main, beberapa nilai integer dimasukkan ke dalam BST, kemudian isi tree ditampilkan menggunakan traversal InOrder sehingga menghasilkan data yang terurut menaik. Program ini menunjukkan cara kerja dasar Binary Search Tree dalam mengelola dan menampilkan data secara terstruktur.

2. Soal 2

a) Bstree.h

```
/* fungsi tambahan latihan no 2 */
int hitungNode(address root);
int hitungTotal(address root);
int hitungKedalaman(address root, int start);
```

b) Bstree.cpp

```
/* menghitung jumlah node */
int hitungNode(address root) {
    if (root == Nil)
        return 0;
    else
        return 1 + hitungNode(root->left) + hitungNode(root-
>right);
}

/* menghitung total seluruh info node */
int hitungTotal(address root) {
    if (root == Nil)
        return 0;
    else
        return root->info
             + hitungTotal(root->left)
```

```
            + hitungTotal(root->right);
}

/* menghitung kedalaman maksimum tree */
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 (kiri > kanan) ? kiri : kanan;
}
```

c) Main.cpp

```
 /*fungsi tambahan latihan no 2*/
    cout << endl;

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

    return 0;
```

Screenshot output:

```
PS D:\COOLYEAH\SEMESTER 3\Strukdat\Modul 10\Unguided> g++ main.cpp bstree.cpp -o mai
PS D:\COOLYEAH\SEMESTER 3\Strukdat\Modul 10\Unguided> .\main.exe
Hello World
1 - 2 - 3 - 4 - 5 - 6 - 7 -
kedalaman : 5
jumlah Node : 7
total : 28
PS D:\COOLYEAH\SEMESTER 3\Strukdat\Modul 10\Unguided>
```

Deskripsi:

Program pada nomor 2 ini menambahkan beberapa fungsi rekursif pada Binary Search Tree untuk melakukan perhitungan terhadap struktur tree. Fungsi hitungNode digunakan untuk menghitung jumlah seluruh node yang ada di dalam BST dengan cara menjumlahkan setiap node yang dikunjungi. Fungsi hitungTotal berfungsi menghitung total nilai data (info) dari semua node dalam tree. Sementara itu, fungsi hitungKedalaman digunakan untuk menentukan kedalaman maksimum dari Binary Search Tree dengan membandingkan kedalaman subtree kiri dan kanan secara rekursif. Ketiga fungsi tersebut dipanggil pada program utama setelah data dimasukkan ke dalam BST, sehingga program dapat menampilkan isi tree sekaligus informasi jumlah node, total nilai, dan kedalaman tree.

3. Soal 3

a) Bstee.h

```
/*SOAL 3*/
void PreOrder(address root);
void PostOrder(address root);
```

b) Bstree.cpp

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

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

c) Main.cpp

```
/*fungsi tambahan latihan no 3*/
    cout << "PreOrder  : ";
    PreOrder(root);
    cout << endl;

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

    return 0;
```

Screenshot output:

```
PS D:\COOLYEAH\SEMESTER 3\Strukdat\Modul 10\Unguided> g++ main.cpp bstree.cpp -o main
PS D:\COOLYEAH\SEMESTER 3\Strukdat\Modul 10\Unguided> .\main.exe
Hello World
1 - 2 - 3 - 4 - 5 - 6 - 7 -
PreOrder  : 1 - 2 - 6 - 4 - 3 - 5 - 7 -
PostOrder : 3 - 5 - 4 - 7 - 6 - 2 - 1 -
PS D:\COOLYEAH\SEMESTER 3\Strukdat\Modul 10\Unguided>
```

Deskripsi:

Program pada nomor 3 menambahkan fungsi PreOrder dan PostOrder untuk

menampilkan isi Binary Search Tree menggunakan metode traversal yang berbeda. Pre-order menampilkan data dengan urutan root, kiri, lalu kanan, sedangkan post-order menampilkan data dengan urutan kiri, kanan, lalu root.