LAPORAN PRAKTIKUM
STRUKTUR DATA

MODUL X
TREE

Disusun oleh :

Junadil Muqorobin (103112400281)

Dosen

Fahrudin Mukti Wibowo S.Kom., M.Eng

PROGRAM STUDI TEKNIK INFORMATIKA
FAKULTAS INFORMATIKA
TELKOM UNIVERSITY PURWOKERTO
2025

A. Dasar Teori

Tree merupakan salah satu struktur data non-linear yang tersusun secara hierarkis dan terdiri atas simpul (node) yang saling terhubung melalui relasi parent–child. Berbeda dengan struktur data linear seperti array, stack, dan queue, tree memungkinkan representasi hubungan bertingkat yang lebih kompleks sehingga banyak digunakan pada sistem berorientasi hierarki, seperti struktur direktori file, sistem organisasi data, dan algoritma pencarian. Dalam konteks ilmu komputer, sebuah tree terdiri dari satu node akar (root), node internal, dan node daun (leaf), serta memiliki karakteristik tidak mengandung sirkuit (acyclic) dan hanya terdapat satu jalur unik dari root ke setiap node. Struktur ini menjadikannya efisien untuk operasi pencarian, penyisipan, dan penghapusan data (Wulandari & Prasetyo, 2020).

Binary Tree merupakan bentuk tree yang paling umum, di mana setiap node maksimal memiliki dua anak, yaitu left child dan right child. Variasi dari binary tree yang banyak digunakan adalah Binary Search Tree (BST), yaitu tree terurut dengan aturan bahwa nilai subtree kiri selalu lebih kecil daripada parent dan nilai subtree kanan lebih besar. Sifat terurut ini memungkinkan proses pencarian dilakukan dengan kompleksitas rata-rata O(log n), menjadikannya lebih efisien dibanding pencarian linear. Berbagai penelitian nasional juga menyebutkan bahwa BST efektif digunakan dalam pembangunan sistem penyimpanan data terstruktur, sistem indeks, serta algoritma pemrosesan informasi berbasis hierarki (Ramadhan & Hakim, 2021).

Namun, BST memiliki kelemahan ketika data yang dimasukkan tidak terdistribusi merata, sehingga tree dapat menjadi tidak seimbang (skewed). Kondisi ini menyebabkan efisiensi pencarian turun menjadi O(n). Untuk mengatasi hal tersebut, dikembangkan struktur AVL Tree, yaitu bentuk BST yang selalu menjaga keseimbangan tinggi antara subtree kiri dan kanan dengan selisih maksimal satu. AVL Tree menggunakan rotasi (rotate left, rotate right, left-right, dan right-left) sebagai mekanisme penyeimbang setiap kali dilakukan operasi insert atau delete. Penelitian terbaru menunjukkan bahwa AVL Tree mampu menjaga kestabilan performa pencarian dan pengelolaan data dalam berbagai aplikasi seperti database ringan, sistem navigasi, dan manajemen indeks pencarian (Siregar & Ardiansyah, 2022). Selain itu, Tree juga bergantung kuat pada konsep rekursi karena struktur hierarkisnya memungkinkan pemanggilan fungsi secara berulang melalui subtree. Traversal seperti preorder, inorder, dan postorder merupakan contoh implementasi rekursif yang penting, dimana tree dijelajahi berdasarkan urutan peninjauan node. Teknik traversal ini mendukung berbagai operasi seperti pengurutan data, pencetakan struktur tree, hingga serialisasi objek. Menurut studi ilmiah dalam pengembangan algoritma pendidikan, penggunaan rekursi pada tree dapat meningkatkan pemahaman logika algoritmik mahasiswa karena struktur dan alurnya teratur serta mudah dipetakan (Yuliani & Sembiring, 2023).

Dalam implementasi komputasi modern, tree berperan penting sebagai dasar pembentukan berbagai struktur tingkat lanjut seperti heap, trie, B-tree, dan struktur indeks database. Penelitian terbaru juga menunjukkan bahwa tree memberikan fleksibilitas tinggi dalam memetakan data besar dan kompleks,

sekaligus mempertahankan efisiensi memori dan waktu eksekusi jika diimbangi dengan mekanisme penyeimbang seperti pada AVL Tree (Nurjaman & Fitriansyah, 2024). Oleh karena itu, pemahaman konsep tree menjadi fundamental dalam pengembangan perangkat lunak, sistem operasi, basis data, dan kecerdasan buatan.

B. Guided

1. Implementasi Tree menggunakan ADT

Source code 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* noded);
    void preorder(Node* noded);
    void postorder(Node* noded);

public:
    BinaryTree();
    void insert(int value);
```

```
    void deleteValue(int value);
    void update(int oldVal, int newVal);

    void inorder();
    void preorder();
    void postorder();
};
#endif
```

Source code 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; }
```

Source code 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 << "\nTransversal 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 = 30):"
<< endl;
    cout << "Inorder    : "; tree.inorder();

    tree.deleteValue(30);

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

    return 0;
}
```

Screenshoot Output:

Deskripsi:

Program ini mengimplementasikan AVL Tree menggunakan konsep ADT (Abstract Data Type). AVL Tree adalah jenis Binary Search Tree (BST) yang selalu menjaga keseimbangan tinggi (height) di setiap node melalui operasi rotasi kiri, rotasi kanan, dan rotasi ganda. Program berjalan dengan langkah kerja sebagai berikut:

- Source code tree.h

  File ini berisi deklarasi tipe data abstrak untuk AVL Tree dan seluruh fungsi yang dibutuhkan.
  - Data yaitu nilai integer yang disimpan dalam node.
  - Left dan right sebagai pointer ke anak kiri dan kanan.
  - Height sebagai tinggi node yang digunakan dalam perhitungan balance factor.
  - Class BinaryTree berisi beberapa fungsi seperti insertNode() dan deleteNode() untuk proses insert dan delete secara rekursif. getHeight() dan getBalance() untuk menghitung tinggi dan keseimbangan node. rotateRight() dan rotateLeft() sebagai operasi utama penyeimbangan tree. Dan minValueNode() untuk mencari node terkecil pada subTree kanan ketika delete.

    Selain itu juga ada fungsi public seperti konstruktor untuk menginisialisasi tree kosong, insert() untuk memasukkan nilai, deleteValue() untuk menghapus nolai, update() untuk mengganti nilai lama dengan nilai baru, dan ada fungsi traversal inorder(), preorder(), dan postorder().
- Source code tree.cpp

  File ini berisi implementasi seluruh fungsi yang dideklarasikan pada ADT AVL Tree, sebagai berikut:

- BinaryTree() , sebagai konstruktor yang menginisialisasi tree dalam keadaan kosong dengan root = nullptr.
- getHeight(Node n)* digunakan untuk mengembalikan tinggi sebubah node dan nilai 0 nila node kosong.
- getBalance(Node n)* untuk me-ngembalikan *balance factor* dari node, yaitu perbedaan tinggi subtree kiri dan kanan.
- rotateRight(Node y)* dan rotateLeft(Node x)*, digunakan untuk menyeimbangkan tree saat terjadi ketidakseimbangan. Rotasi kanan menangani kasus LL, sedangkan rotasi kiri menangani kasus RR. Pada kasus LR dan RL, digunakan kombinasi kedua rotasi.
- insertNode(Node node, int value)* bekerja dengan cara memperbarui tinggi node, balance factor dihitung dan jika tree tidak seimbang maka akan dirotasi sesuai kasus.
- deleteNode(Node root, int key)* , jika node tanpa anak maka akan langasung dihapus, jika node satu anak akan digantikan anaknya jika node dua anak akan menggunakan nilai terkecil dari subtree kanan.
- Update(int oldVal, int newVal) bertujuan mengganti nilai lama dengan nilai baru melalui dua Langkah yaitu deleteValue(oldVal) dan insert(newVal)

- Traversal (inorder, preorder, postorder) .

- Source code main.cpp

  File ini berisi alur utama profram untuk menguji seluruh operasi pada tree. Berikut langkah kerja programnya:

  - Membuat objek BinaryTree dan memulai tree dalam keadaan kosong.
  - Melakukan insert terhadap nilai: 10, 15, 20, 30, 35, 40, 50
  - Menampilkan hasil traversal baik secara inroder, preorder, dan postorder, setelah semua data diinsert
  - Melakukan update nilai dengan mengubah nilai 20 menjadi 25 dan menampilkan traversal inorder sebelum dan sesudah update.
  - Kemudian melakukan delete dengan menghapus node dengan nilai 30, kemudian menampilkan traversal inorder setelah delete
  - Program selesai.

C. Unguided

1. Membuat ADT Binary Search Tree

Source code bstree.h

```
#ifndef BSTREE_H
#define BSTREE_H

#include <iostream>
#define Nil NULL

using namespace std;

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);

// Tambahan nomor 2
int hitungNode(address root);
int hitungTotal(address root);
int hitungKedalaman(address root);

// tambahan nomor 3
void PreOrder(address root);
void PostOrder(address root);


#endif
```

Source code 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);
    }
    else {
    }
}

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);
    }
}

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) {
    if (root == Nil) return 0;
    int kiri = hitungKedalaman(root->left);
    int kanan = hitungKedalaman(root->right);
    return 1 + max(kiri, kanan);
}

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

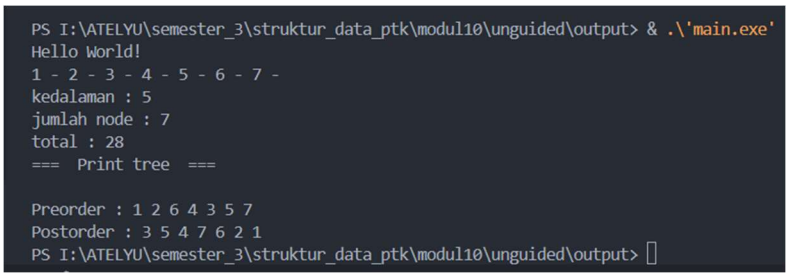
```
    }
}

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

Source code main.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,6);
    insertNode(root,7);

    InOrder(root);

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

    cout << "===  Print tree  ===" << endl;
    cout << "\nPreorder : ";
    PreOrder(root);

    cout << "\nPostorder : ";
```

```
    PostOrder(root);

    return 0;
}
```

Screenshot Output:



Deskripsi:

Program ini mengimplementasikan **ADT Binary Search Tree (BST)** menggunakan struktur data **linked list**. Setiap node pada BST memiliki tiga komponen utama, yaitu info yang menyimpan nilai, serta pointer left dan right sebagai penunjuk ke subtree kiri dan kanan. Tree yang terbentuk mengikuti aturan BST, yaitu nilai lebih kecil ditempatkan di kiri, dan nilai lebih besar ditempatkan di kanan. Program berjalan dengan alur kerja sebagai berikut:

- Source code bstree.h

  File ini berisi deklarasi struktur data dan fungsi-fungsi yang ada dalam BST, diantaranya yaitu:

  - Type node , berisi field info bertipe integer, serta pointer left dan right untuk membentuk struktur pohon biner.
  - Type address untuk node agar lebih mudah digunakan dalam proses manipulasi tree.

- alokasi(x) untuk mengalokasikan node baru.
- insertNode(root, x) untuk menambahkan node sesuai aturan BST.
- findNode(x, root) untuk mencari nilai tertentu dalam tree.
- InOrder(root) sebagai proses traversal inorder.
- hitungNode(root) menghitung jumlah node dalam BST.
- hitungTotal(root) menjumlahkan seluruh nilai info.
- hitungKedalaman(root) menentukan kedalaman maksimal tree.
- PreOrder(root) dan postOrder(root) merupakan dua fungsi traversal tambahan sesuai permintaan soal.

- Source code bstree.cpp

  File ini berisi implementasi fungsi-fungsi pada ADT BST. Penjabaran fungsi utama sebagai berikut:
  - Alokasi(x) untuk membuat node baru, mengisi info dengan nilai x, dan mengubah pointer kiri dan kanan menjadi Nil.
  - insertNode(root, x) digunakan untuk menyisipkan nilai x secara rekursif, jika root kosong maka node baru menjadi root, jika x lebih kecil maka masuk ke subtree kiri, jika x lebih besar maka masuk ke subtree kanan dan jika nilai duplikat maka tidak ditambahkan.

- findNode(x, root) digunakan untuk melakukan pencarian dengan menelusuri subtree kiri atau kanan sesuai aturan BST.
- InOrder(root) digunakan untuk menampilkan tree secara inorder yaitu kiri , root , kanan.
- hitungNode(root) digunakan untuk menghitung jumlah node menggunakan rekursi 1 + kiri + kanan
- hitungTotal(root) digunakan utnk mengembalikan total penjumlahan nilai dari setiap node pada tree.
- HitungKedalaman(root) digunakan untuk mengembalikan tinggi maksimal tree yaitu 1 + max(kedalaman kiri, kedalaman kanan)
- PreOrder(root) digunakan untuk menampilkan tree dengan urutan: root – kiri – kanan.
- PostOrder(root) digunakan untuk menampilkan tree dengan urutan: kiri – kanan – root.

- Source code main.cpp

  File main bertugas memanggil seluruh fungsi BST, kemudian memulai program dengan membuat tree kosong, lalu menambahkan beberapa nilai menggunakan insertNode. Berikut Langkah kerja programnya:
  - Membuat root kosong, address root = Nill;
  - Melakukan insert, urutan nilai yang dimasukkan yaitu 1,2,3,4,5,6,7
  - Kemudian menampilkan traversal inorder

- Mencetak hasil perhitungan tree yaitu kedalaman tree, jumlah node, dan total penjumlahan info.
- Menampilkan traversal tambahan yaitu secara preorder dan postorder

D. Kesimpulan

Pada Pada praktikum ini, saya dapat memahami dan mengimplementasikan struktur data Tree, khususnya Binary Search Tree (BST) yang dibangun menggunakan linked list. Melalui proses pembuatan ADT, penyisipan node, pencarian, hingga traversal, saya mempelajari bagaimana tree bekerja sebagai struktur data non-linear yang mampu merepresentasikan hubungan hierarkis secara efisien. Seluruh operasi utama pada BST, seperti *insert*, *find*, *inorder*, *preorder*, dan *postorder* untuk menunjukkan bahwa tree sangat bergantung pada mekanisme rekursif, karena setiap operasi dilakukan dengan menelusuri subtree secara berulang.

Dari hasil praktikum, saya juga dapat menghitung berbagai karakteristik penting dari tree, seperti jumlah node, total nilai dalam tree, serta kedalaman maksimal. Perhitungan ini memperkuat pemahaman bahwa struktur dan bentuk tree sangat dipengaruhi oleh urutan input, sehingga pola penyisipan elemen dapat menghasilkan bentuk tree yang berbeda dan berdampak langsung pada hasil traversal.

Secara keseluruhan, praktikum ini membantu memperdalam pemahaman mengenai cara kerja Binary Search Tree, penerapan rekursi, serta pengaruh struktur tree terhadap efisiensi operasi pencarian dan pengelolaan data.

E. Referensi

Nurjaman, D., & Fitriansyah, A. (2024). *Analisis Struktur Data Pohon dalam Optimasi Penyimpanan Berbasis Hierarki*. Jurnal Teknologi dan Sistem Informasi, 12(1), 33–42.

Ramadhan, F., & Hakim, L. (2021). *Implementasi Binary Search Tree pada Sistem Manajemen Data Terstruktur*. Jurnal Ilmu Komputer dan Informatika, 7(2), 55–63.

Siregar, R., & Ardiansyah, T. (2022). *Penerapan AVL Tree untuk Optimalisasi Pencarian pada Sistem Informasi*. Jurnal Teknologi Informasi Indonesia, 10(3), 145–154.

Wulandari, S., & Prasetyo, R. (2020). *Kajian Struktur Data Tree dalam Penyelesaian Permasalahan Hierarki Sistem*. Jurnal Sains dan Informatika, 6(1), 12–20.

Yuliani, M., & Sembiring, A. (2023). *Pembelajaran Rekursi Berbasis Tree untuk Meningkatkan Pemahaman Algoritmik Mahasiswa*. Jurnal Pendidikan Teknologi Informasi, 9(2), 101–110.