LAPORAN PRAKTIKUM
STRUKTUR DATA

MODUL X
TREE

Disusun Oleh :
NAMA : Afief Amar Purnomo
NIM : 103112430067

Dosen
FAHRUDIN MUKTI WIBOWO

PROGRAM STUDI STRUKTUR DATA
FAKULTAS INFORMATIKA
TELKOM UNIVERSITY PURWOKERTO
2025

A. Dasar Teori

Struktur data pohon (tree) merupakan struktur data non-linear yang digunakan untuk merepresentasikan hubungan hierarkis antar elemen data, dengan satu simpul utama bernama akar (root) dan simpul-simpul lain yang disebut node atau vertex yang saling terhubung melalui garis penghubung (edge). Setiap simpul dapat memiliki nol atau lebih simpul anak (child), di mana simpul tanpa anak disebut daun (leaf), dan hubungan ini membentuk subtree yang bersifat rekursif tanpa siklus atau loop.

Pohon bersifat hierarkis, berbeda dengan struktur linear seperti array atau linked list, sehingga efisien untuk operasi pencarian, penyisipan, dan pengurutan data, terutama pada binary tree di mana setiap simpul maksimal memiliki dua anak (kiri dan kanan), serta binary search tree (BST) yang memenuhi aturan child kiri < parent < child kanan. Implementasi pada C++ menggunakan struktur Node dengan pointer left dan right, mendukung traversal rekursif seperti pre-order (root-left-right), in-order (left-root-right), dan post-order (left-right-root) untuk mengakses data secara sistematis.

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

Guided 1

(file tree.h)

tree.h > ...

```
#ifndef TREE_H
#define TREE_H

struct Node {
    int data;
    Node *left, *right;
    int height;
};

class BinaryTree {
private:
    Node* root;

    Tabnine | Edit | Test | Explain | Document
    Node* insertNode(Node* node, int value);
    Tabnine | Edit | Test | Explain | Document
    Node* deleteNode(Node* node, int value);

    Tabnine | Edit | Test | Explain | Document
    int getHeight(Node* node);
    Tabnine | Edit | Test | Explain | Document
    int getBalance(Node* node);

    Tabnine | Edit | Test | Explain | Document
    Node* rotateRight(Node* y);
    Tabnine | Edit | Test | Explain | Document
    Node* rotateLeft(Node* x);

    Tabnine | Edit | Test | Explain | Document
    Node* minValueNode (Node* node);

    Tabnine | Edit | Test | Explain | Document
    void inorder(Node* node);
```

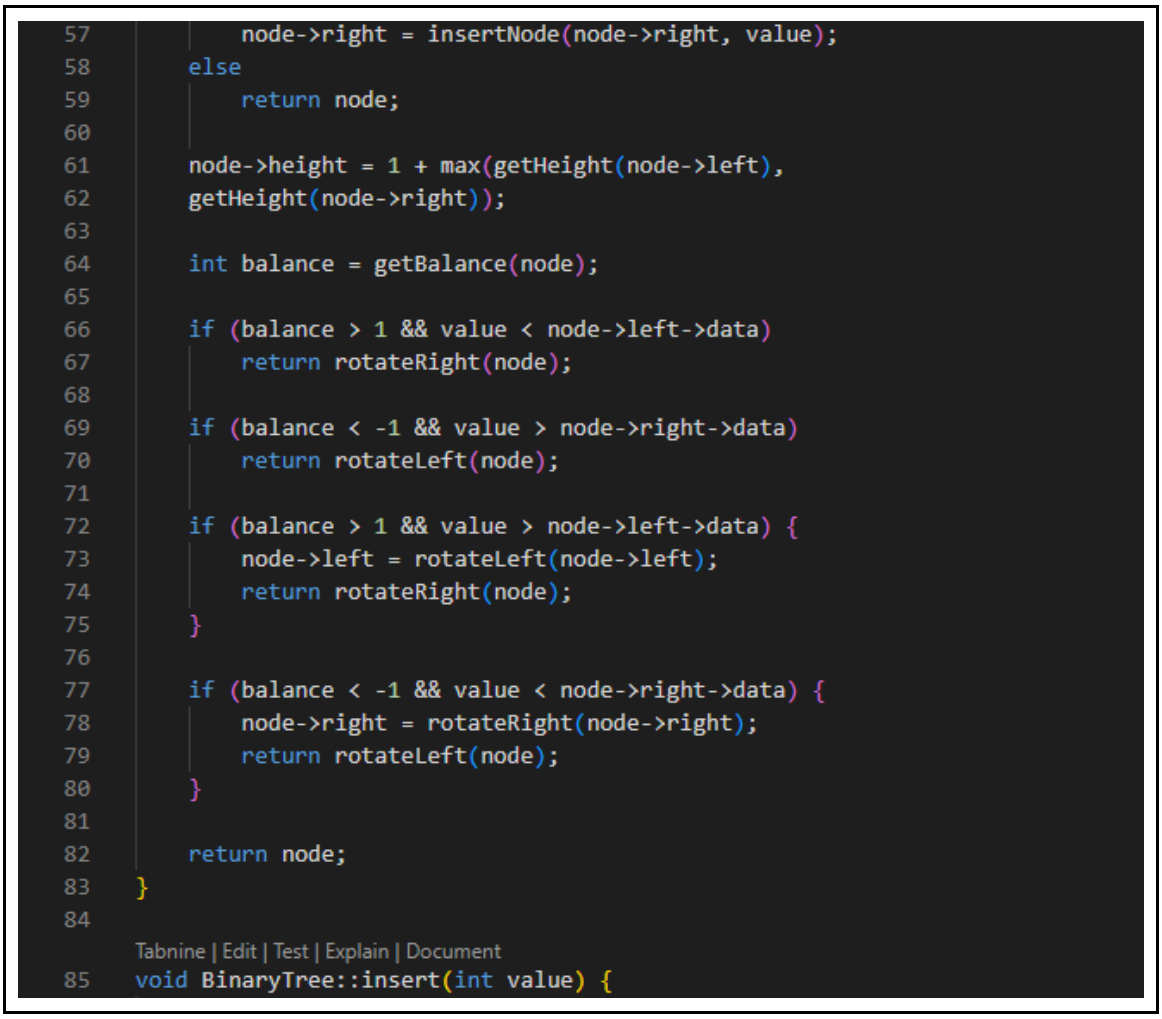
```
        Tabnine | Edit | Test | Explain | Document
26      void preorder(Node* node);
        Tabnine | Edit | Test | Explain | Document
27      void postorder(Node* node);
28
29  public:
        Tabnine | Edit | Test | Explain | Document
30      BinaryTree();
        Tabnine | Edit | Test | Explain | Document
31      void insert(int value);
        Tabnine | Edit | Test | Explain | Document
32      void deleteValue(int value);
        Tabnine | Edit | Test | Explain | Document
33      void update(int oldVal, int newVal);
34
        Tabnine | Edit | Test | Explain | Document
35      void inorder();
        Tabnine | Edit | Test | Explain | Document
36      void preorder();
        Tabnine | Edit | Test | Explain | Document
37      void postorder();
38  };
39
40  #endif
```

(file tree.cpp)

tree.cpp > getBalance(Node *)

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

Tabnine | Edit | Test | Explain | Document
BinaryTree::BinaryTree() {
    root = nullptr;
}

Tabnine | Edit | Test | Explain | Document
int BinaryTree::getHeight(Node* n) {
    return (n == nullptr) ? 0 : n->height;
}

Tabnine | Edit | Test | Explain | Document
int BinaryTree::getBalance(Node* n) {
    return (n == nullptr) ? 0 :
    getHeight(n->left) - getHeight(n->right);
}

Tabnine | Edit | Test | Explain | Document
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;
```

```
29
30      return x;
31  }
32
    Tabnine | Edit | Test | Explain | Document
33  Node* BinaryTree::rotateLeft(Node* x) {
34      Node* y = x->right;
35      Node* T2 = y->left;
36
37      y->left = x;
38      x->right = T2;
39
40      x->height = max(getHeight(x->left),
41       getHeight(x->right)) + 1;
42      y->height = max(getHeight(y->left),
43       getHeight(y->right)) + 1;
44
45      return y;
46  }
47
    Tabnine | Edit | Test | Explain | Document
48  Node* BinaryTree::insertNode(Node* node, int value) {
49      if (node == nullptr) {
50          Node* newNode = new Node{value, nullptr, nullptr, 1};
51          return newNode;
52      }
53
54      if (value < node->data)
55          node->left = insertNode(node->left, value);
56      else if (value > node->data)
```



```
57          node->right = insertNode(node->right, value);
58      else
59          return node;
60
61      node->height = 1 + max(getHeight(node->left),
62      getHeight(node->right));
63
64      int balance = getBalance(node);
65
66      if (balance > 1 && value < node->left->data)
67          return rotateRight(node);
68
69      if (balance < -1 && value > node->right->data)
70          return rotateLeft(node);
71
72      if (balance > 1 && value > node->left->data) {
73          node->left = rotateLeft(node->left);
74          return rotateRight(node);
75      }
76
77      if (balance < -1 && value < node->right->data) {
78          node->right = rotateRight(node->right);
79          return rotateLeft(node);
80      }
81
82      return node;
83  }
84
    Tabnine | Edit | Test | Explain | Document
85  void BinaryTree::insert(int value) {
```

```
86      root = insertNode(root, value);
87  }
88
    Tabnine | Edit | Test | Explain | Document
89  Node* BinaryTree::minValueNode(Node* node) {
90      Node* current = node;
91      while (current->left != nullptr)
92          current = current->left;
93      return current;
94  }
95
    Tabnine | Edit | Test | Explain | Document
96  Node* BinaryTree::deleteNode(Node* root, int key) {
97      if (root == nullptr)
98          return root;
99
100     if (key < root->data)
101         root->left = deleteNode(root->left, key);
102     else if (key > root->data)
103         root->right = deleteNode(root->right, key);
104     else {
105         if ((root->left == nullptr) || (root->right == nullptr)) {
106             Node* temp = root->left ? root->left : root->right;
107
108             if (temp == nullptr) {
109                 temp = root;
110                 root = nullptr;
111             } else {
112                 *root = *temp;
113             }
114             delete temp;
115         } else {
116             Node* temp = minValueNode(root->right);
117             root->data = temp->data;
118             root->right = deleteNode(root->right, temp->data);
119         }
120     }
121
122     if (root == nullptr)
123         return root;
124
125     root->height = 1 + max(getHeight(root->left), getHeight(root->right));
126
127     int balance = getBalance(root);
128
129     if (balance > 1 && getBalance(root->left) >= 0)
130         return rotateRight(root);
131
132     if (balance > 1 && getBalance(root->left) < 0) {
133         root->left = rotateLeft(root->left);
134         return rotateRight(root);
135     }
136
137     if (balance < -1 && getBalance(root->right) <= 0)
138         return rotateLeft(root);
139
140     if (balance < -1 && getBalance(root->right) > 0) {
141         root->right = rotateRight(root->right);
```

```
142             return rotateLeft(root);
143         }
144
145         return root;
146     }
147
    Tabnine | Edit | Test | Explain | Document
148     void BinaryTree::deleteValue(int value) {
149         root = deleteNode(root, value);
150     }
151
    Tabnine | Edit | Test | Explain | Document
152     void BinaryTree::update(int oldVal, int newVal) {
153         deleteValue(oldVal);
154         insert(newVal);
155     }
156
    Tabnine | Edit | Test | Explain | Document
157     void BinaryTree::inorder(Node* node) {
158         if (node == nullptr) return;
159         inorder(node->left);
160         cout << node->data << " ";
161         inorder(node->right);
162     }
163
    Tabnine | Edit | Test | Explain | Document
164     void BinaryTree::preorder(Node* node) {
165         if (node == nullptr) return;
166         cout << node->data << " ";
167         preorder(node->left);
168         preorder(node->right);
169     }
170
    Tabnine | Edit | Test | Explain | Document
171     void BinaryTree::postorder(Node* node) {
172         if (node == nullptr) return;
173         postorder(node->left);
174         postorder(node->right);
175         cout << node->data << " ";
176     }
177
    Tabnine | Edit | Test | Explain | Document
178     void BinaryTree::inorder() { inorder(root); cout << endl; }
    Tabnine | Edit | Test | Explain | Document
179     void BinaryTree::preorder() { preorder(root); cout << endl; }
    Tabnine | Edit | Test | Explain | Document
180     void BinaryTree::postorder() { postorder(root); cout << endl; }
```

(file main.cpp)

main.cpp > main()

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

using namespace std;

Tabnine | Edit | Test | Explain | Document
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 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

```
PS D:\Kuliah\Semester 3\MATKUL\Praktikum Struktur Data\Praktikum\Modul_10\Guide> g++ main.cpp tree.cpp
PS D:\Kuliah\Semester 3\MATKUL\Praktikum Struktur Data\Praktikum\Modul_10\Guide> ./a.exe
=== INSERT DATA ===
Data yang diinsert: 10, 15, 20, 30, 35, 40, 50

Traversal setelah insert:
Inorder    : 10 15 20 30 35 40 50
Preorder   : 30 15 10 20 40 35 50
Postorder  : 10 20 15 35 50 40 30

=== Update Data ===
Sebelum update (20 -> 25):
Inorder    : 10 15 20 30 35 40 50
Setelah update (20 -> 25):
Inorder    : 10 15 25 30 35 40 50

=== Delete Data ===
Sebelum delete (hapus subtree dengan root = 30):
Inorder    : 10 15 25 30 35 40 50
Setelah delete (subtree root = 30 dihapus):
Inorder    : 10 15 25 35 40 50
```

Deskripsi:

Program di atas merupakan implementasi AVL Tree, yaitu struktur data pohon biner pencarian (Binary Search Tree) yang selalu menjaga keseimbangan tinggi (height-balanced). Setiap node memiliki tiga komponen utama: data, pointer left & right, serta height yang digunakan untuk menghitung faktor keseimbangan (balance factor). Kelas BinaryTree menyediakan berbagai operasi penting seperti insert untuk menambahkan node baru, deleteValue untuk menghapus node, update untuk mengganti nilai lama dengan nilai baru, serta tiga metode traversal yaitu inorder, preorder, dan postorder. Untuk menjaga pohon tetap seimbang, program menggunakan empat jenis rotasi: rotateLeft, rotateRight, Left-Right Rotation, dan Right-Left Rotation. Rotasi ini secara otomatis dijalankan apabila balance factor suatu node tidak berada pada rentang -1 hingga 1. Dengan demikian, setiap operasi insert dan delete terjamin berjalan efisien dengan kompleksitas O(log n). Pada fungsi main, program pertama-tama memasukkan data secara berurutan: 10, 15, 20, 30, 35, 40, dan 50. Karena data dimasukkan dalam urutan naik, pohon awalnya cenderung miring ke kanan, tetapi AVL Tree menyeimbangkannya lewat rotasi otomatis. Setelah semua data masuk, program menampilkan hasil traversal inorder, preorder, dan postorder untuk menunjukkan struktur pohon. Selanjutnya, program melakukan operasi update dengan mengganti nilai 20 menjadi 25. Proses ini dilakukan dengan menghapus nilai lama dan menambahkan nilai baru, lalu AVL Tree kembali menyeimbangkan dirinya. Hasil traversal inorder ditampilkan kembali untuk menunjukkan perubahan. Terakhir, program menjalankan operasi delete untuk menghapus node bernilai 30, yang juga menghapus subtree di bawahnya. AVL Tree kembali melakukan rotasi jika diperlukan untuk memastikan pohon tetap seimbang. Traversal inorder ditampilkan lagi untuk menunjukkan struktur pohon setelah penghapusan.

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

Unguided 1 (soal 1)

(file bstree.h)

bstree.h > InOrder(address)

```h
#ifndef BSTREE_H
#define BSTREE_H

typedef int infotype;

typedef struct Node *address;

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

Tabnine | Edit | Test | Explain | Document
address alokasi(infotype x);
Tabnine | Edit | Test | Explain | Document
void insertNode(address &root, infotype x);
Tabnine | Edit | Test | Explain | Document
address findNode(infotype x, address root);
Tabnine | Edit | Test | Explain | Document
void InOrder(address root);
Tabnine | Edit | Test | Explain | Document
int hitungJumlahNode(address root);
Tabnine | Edit | Test | Explain | Document
int hitungTotalInfo(address root);
Tabnine | Edit | Test | Explain | Document
int hitungKedalaman(address root, int start);

#endif
```

(file bstree.cpp)

bstree.cpp > ...

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

Tabnine | Edit | Test | Explain | Document
address alokasi(infotype x) {
    address p = new Node;
    p->info = x;
    p->left = nullptr;
    p->right = nullptr;
    return p;
}

Tabnine | Edit | Test | Explain | Document
void insertNode(address &root, infotype x) {
    if (root == nullptr) {
        root = alokasi(x);
    } else if (x < root->info) {
        insertNode(root->left, x);
    } else {
        insertNode(root->right, x);
    }
}

Tabnine | Edit | Test | Explain | Document
address findNode(infotype x, address root) {
    if (root == nullptr || root->info == x)
        return root;

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

```

```
30      return findNode(x, root->right);
31  }
32
    Tabnine | Edit | Test | Explain | Document
33  void InOrder(address root) {
34      if (root != nullptr) {
35          InOrder(root->left);
36          cout << root->info << " - ";
37          InOrder(root->right);
38      }
39  }
40
    Tabnine | Edit | Test | Explain | Document
41  int hitungJumlahNode(address root) {
42      if (root == nullptr)
43          return 0;
44      return 1 + hitungJumlahNode(root->left) + hitungJumlahNode(root->right);
45  }
46
    Tabnine | Edit | Test | Explain | Document
47  int hitungTotalInfo(address root) {
48      if (root == nullptr)
49          return 0;
50      return root->info + hitungTotalInfo(root->left) + hitungTotalInfo(root->right);
51  }
52
    Tabnine | Edit | Test | Explain | Document
53  int hitungKedalaman(address root, int start) {
54      if (root == nullptr)
55          return start;
56
57      int leftDepth = hitungKedalaman(root->left, start + 1);
58      int rightDepth = hitungKedalaman(root->right, start + 1);
59
60      return (leftDepth > rightDepth) ? leftDepth : rightDepth;
61  }
62
```

(file main.cpp)

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

using namespace std;

Tabnine | Edit | Test | Explain | Document
int main() {
    cout << "Hello World!" << endl;

    address root = nullptr;

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

    InOrder(root);

    cout << "\n";

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

    return 0;
}
```

Screenshots Output

```
PS D:\Kuliah\Semester 3\MATKUL\Praktikum Struktur Data\Praktikum\Modul_10\Unguide> g++ main.cpp bstree.cpp
PS D:\Kuliah\Semester 3\MATKUL\Praktikum Struktur Data\Praktikum\Modul_10\Unguide> ./a.exe
Hello World!
1 - 2 - 3 - 4 - 5 - 6 - 7 -
kedalaman : 5
jumlah Node : 7
total : 28
```

Deskripsi:

Program di atas merupakan implementasi struktur data **Binary Search Tree (BST)** menggunakan bahasa C++ yang menyimpan data secara terurut dengan aturan bahwa nilai yang lebih kecil ditempatkan di subtree kiri dan nilai yang lebih besar ditempatkan di subtree kanan. Setiap node memiliki nilai info serta pointer ke anak kiri dan kanan. Program menyediakan fungsi insertNode untuk menambahkan data ke dalam BST, findNode untuk mencari nilai tertentu, serta InOrder untuk menampilkan data secara terurut. Selain itu, terdapat fungsi hitungJumlahNode untuk menghitung total node, hitungTotalInfo untuk menjumlahkan seluruh nilai info, dan hitungKedalaman untuk

menentukan kedalaman maksimum pohon. Pada fungsi main, program membangun BST dengan memasukkan data 1, 2, 6, 4, 5, 3, dan 7, kemudian menampilkan traversal inorder yang menghasilkan urutan 1 – 2 – 3 – 4 – 5 – 6 – 7. Setelah itu, program menghitung dan menampilkan kedalaman pohon, jumlah node, serta total seluruh nilai info, sehingga menunjukkan bagaimana BST bekerja dalam menyimpan data terurut dan menyediakan informasi mengenai struktur pohonnya.

D. Kesimpulan

Berdasarkan hasil praktikum pada Modul X tentang Struktur Data Tree, dapat disimpulkan bahwa tree merupakan struktur data non-linear yang digunakan untuk merepresentasikan hubungan hierarkis antar elemen, dengan satu simpul utama sebagai root dan node-node lain yang saling terhubung melalui subtree. Melalui praktikum ini, mahasiswa memahami konsep dasar tree, khususnya Binary Tree, Binary Search Tree (BST), serta AVL Tree yang mampu menjaga keseimbangan tinggi pohon agar operasi tetap efisien. Implementasi meliputi pembuatan node, operasi insert, search, update, delete, hingga traversal seperti preorder, inorder, dan postorder. Pada AVL Tree, mahasiswa mempraktikkan rotasi kiri, kanan, left-right, dan right-left untuk menjaga keseimbangan pohon. Dalam latihan BST, mahasiswa mengimplementasikan perhitungan jumlah node, total nilai info, dan kedalaman pohon. Sebagai contoh, ketika data dimasukkan secara berurutan seperti 10, 15, 20, 30, 35, 40, dan 50, struktur pohon awalnya miring ke kanan, namun AVL Tree secara otomatis melakukan rotasi sehingga pohon kembali seimbang. Secara keseluruhan, praktikum ini memberikan pemahaman mendalam mengenai cara kerja tree dalam penyimpanan data terstruktur, pencarian yang lebih efisien, serta penerapannya pada berbagai kasus pemrograman seperti hierarki data, indexing, dan sistem berorientasi pencarian.

E. Referensi

ADT tree: Rekursi binary tree C++ Universitas Indonesia. (2022). Fasilkom UI - IKI20100. http://aren.cs.ui.ac.id/sda/resources/sda2010/10_tree.pdf

Implementasi insert binary search tree single dan double pointer C++. (2023). School of Computer Science BINUS. https://socs.binus.ac.id/2017/05/10/implementasi-insert-pada-binary-search-tree-dengan-single-dan-double-pointer/

Pelatihan pemrograman C++ struktur data tree dan graph. (2024). Jurnal Reswara Dharmawangsa, 5(2). https://jurnal.dharmawangsa.ac.id/index.php/reswara/article/download/5101/pdf