

**LAPORAN PRAKTIKUM
STRUKTUR DATA**

**MODUL X
TREE**



Disusun Oleh :

NAMA : Taufik Hafit Zakaria

NIM : 103112430093

Dosen

WAHYU ANDI SAPUTRA

**PROGRAM STUDI STRUKTUR DATA
FAKULTAS INFORMATIKA
TELKOM UNIVERSITY PURWOKERTO
2025**

A. Dasar Teori

Tree merupakan struktur data non-linear yang menggambarkan hubungan hierarkis (*one-to-many*) antar elemen, dimulai dari elemen teratas (root) hingga elemen terbawah (leaf). Salah satu implementasi utamanya adalah **Binary Search Tree (BST)**, di mana setiap simpul maksimal memiliki dua anak dengan aturan pengurutan tertentu (anak kiri $<$ parent $<$ anak kanan). Karena struktur Tree bersifat *self-similar* (setiap cabang adalah tree yang lebih kecil), pengelolaannya sangat erat dikaitkan dengan **Rekursif**, yaitu teknik pemrograman di mana fungsi memanggil dirinya sendiri. Metode rekursif digunakan untuk menelusuri (*traversal*), mencari, atau memanipulasi data dalam Tree dengan memecah masalah besar menjadi sub-masalah yang lebih sederhana hingga mencapai kondisi berhenti (*base case*).

Karakteristik Utama

- **Struktur Hierarkis & Terurut:** Data disusun bertingkat (bukan linear), dan pada BST berlaku aturan ketat: nilai Left Child $<$ Parent $<$ Right Child.
- **Prinsip Rekursif:** Operasi pada Tree dilakukan dengan fungsi yang memanggil dirinya sendiri untuk menelusuri setiap *subtree* (cabang).
- **Kondisi Berhenti (*Base Case*):** Setiap operasi rekursif pada Tree (seperti pencarian) harus memiliki kondisi berhenti, misalnya saat menemukan data atau saat mencapai *node* kosong (NULL).

Keuntungan Penggunaan

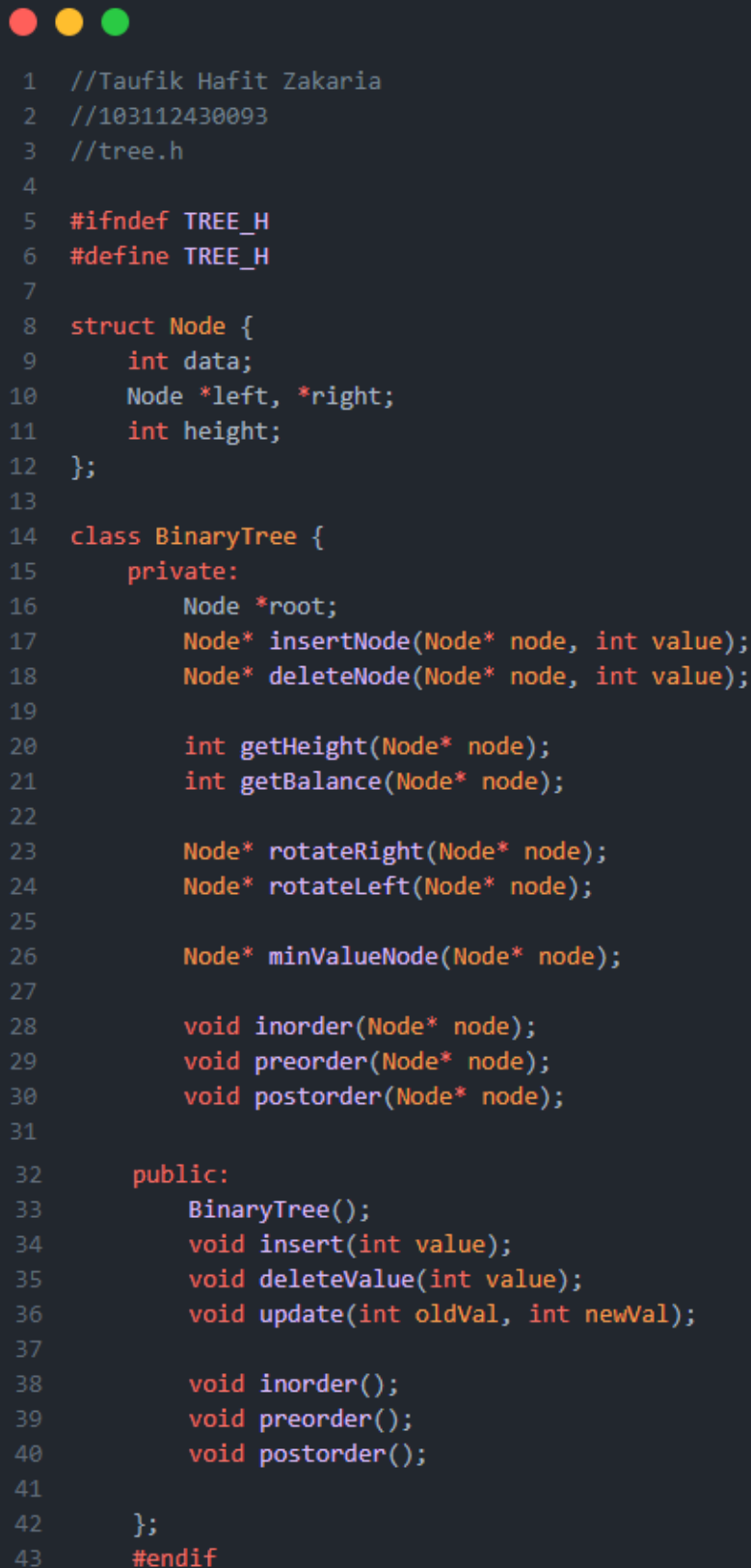
- **Efisiensi Pencarian Data:** Kombinasi BST dan rekursif memungkinkan pencarian data yang sangat cepat dengan prinsip *divide and conquer* (memangkas setengah area pencarian).
- **Penyederhanaan Kode Program:** Teknik rekursif membuat algoritma yang rumit (seperti *traversal* In-Order, Pre-Order, Post-Order) menjadi jauh lebih ringkas, bersih, dan mudah dibaca dibandingkan metode iteratif.
- **Fleksibilitas Struktur:** Sangat ideal untuk merepresentasikan dan mengelola data yang memiliki hubungan bertingkat secara dinamis, seperti struktur folder atau logika matematika.

Keterbatasan

- **Konsumsi Memori Ekstra:** Implementasi ini membutuhkan memori lebih besar, baik untuk menyimpan *pointer* pada Tree maupun untuk menyimpan tumpukan (*stack*) memori akibat pemanggilan fungsi rekursif yang berulang.
- **Risiko *Stack Overflow*:** Jika Tree terlalu dalam atau *base case* tidak terpenuhi, tumpukan memori dapat penuh dan menyebabkan program berhenti mendadak.
- **Masalah Keseimbangan (*Balancing*):** Jika data diinput secara berurutan, Tree dapat menjadi miring (*skewed*) menyerupai garis lurus, yang menyebabkan penurunan performa pencarian secara drastis.

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

Guided 1



```
1 //Taufik Hafit Zakaria
2 //103112430093
3 //tree.h
4
5 #ifndef TREE_H
6 #define TREE_H
7
8 struct Node {
9     int data;
10    Node *left, *right;
11    int height;
12 };
13
14 class BinaryTree {
15     private:
16         Node *root;
17         Node* insertNode(Node* node, int value);
18         Node* deleteNode(Node* node, int value);
19
20         int getHeight(Node* node);
21         int getBalance(Node* node);
22
23         Node* rotateRight(Node* node);
24         Node* rotateLeft(Node* node);
25
26         Node* minValueNode(Node* node);
27
28         void inorder(Node* node);
29         void preorder(Node* node);
30         void postorder(Node* node);
31
32     public:
33         BinaryTree();
34         void insert(int value);
35         void deleteValue(int value);
36         void update(int oldVal, int newVal);
37
38         void inorder();
39         void preorder();
40         void postorder();
41
42     };
43 #endif
```



```
1 //Taufik Hafit Zakaria
2 //103112430093
3 //tree.h
4
5 #include "tree.h"
6 #include <iostream>
7 using namespace std;
8
9 BinaryTree::BinaryTree() {
10     root = nullptr;
11 }
12
13 int BinaryTree::getHeight(Node* n) {
14     return (n == nullptr) ? 0 : n->height;
15 }
16
17 int BinaryTree::getBalance(Node* n) {
18     return (n == nullptr) ? 0 :
19         getHeight(n->left) - getHeight(n->right);
20 }
21
22 Node* BinaryTree::rotateRight(Node* y) {
23     Node* x = y->left;
24     Node* T2 = x->right;
25
26     x->right = y;
27     y->left = T2;
28
29     y->height = max(getHeight(y->left),
30         getHeight(y->right)) + 1;
31     x->height = max(getHeight(x->left),
32         getHeight(x->right)) + 1;
33
34     return x;
35 }
36
37 Node* BinaryTree::rotateLeft(Node* x) {
38     Node* y = x->right;
39     Node* T2 = y->left;
40
41     y->left = x;
42     x->right = T2;
43
44     x->height = max(getHeight(x->left),
45         getHeight(x->right)) + 1;
46     y->height = max(getHeight(y->left),
47         getHeight(y->right)) + 1;
48
49     return y;
50 }
51
```

```

52 Node* BinaryTree::insertNode(Node* node, int value) {
53     if (node == nullptr) {
54         Node* newNode = new Node{value, nullptr, nullptr, 1};
55         return newNode;
56     }
57
58     if (value < node->data)
59         node->left = insertNode(node->left, value);
60     else if (value > node->data)
61         node->right = insertNode(node->right, value);
62     else
63         return node;
64
65     node->height = 1 + max(getHeight(node->left),
66                          getHeight(node->right));
67
68     int balance = getBalance(node);
69
70     if (balance > 1 && value < node->left->data)
71         return rotateRight(node);
72
73     if (balance < -1 && value > node->right->data)
74         return rotateLeft(node);
75
76     if (balance > 1 && value > node->left->data) {
77         node->left = rotateLeft(node->left);
78         return rotateRight(node);
79     }
80
81     if (balance < -1 && value < node->right->data) {
82         node->right = rotateRight(node->right);
83         return rotateLeft(node);
84     }
85
86     return node;
87 }
88

```


```

89 void BinaryTree::insert(int value) {
90     root = insertNode(root, value);
91 }
92
93 Node* BinaryTree::minValueNode(Node* node) {
94     Node* current = node;
95     while (current->left != nullptr)
96         current = current->left;
97     return current;
98 }
99
100 Node* BinaryTree::deleteNode(Node* root, int key) {
101     if (root == nullptr)
102         return root;
103
104     if (key < root->data)
105         root->left = deleteNode(root->left, key);
106     else if (key > root->data)
107         root->right = deleteNode(root->right, key);
108     else {
109         if ((root->left == nullptr) || (root->right == nullptr)) {
110             Node* temp = root->left ? root->left : root->right;
111
112             if (temp == nullptr) {
113                 temp = root;
114                 root = nullptr;
115             } else {
116                 *root = *temp;
117             }
118             delete temp;
119         } else {
120             Node* temp = minValueNode(root->right);
121             root->data = temp->data;
122             root->right = deleteNode(root->right, temp->data);
123         }
124     }
125 }

```

```
126     if (root == nullptr)
127         return root;
128
129     root->height = 1 + max(getHeight(root->left), getHeight(root->right));
130
131     int balance = getBalance(root);
132
133     if (balance > 1 && getBalance(root->left) >= 0)
134         return rotateRight(root);
135
136     if (balance > 1 && getBalance(root->left) < 0) {
137         root->left = rotateLeft(root->left);
138         return rotateRight(root);
139     }
140
141     if (balance < -1 && getBalance(root->right) <= 0)
142         return rotateLeft(root);
143
144     if (balance < -1 && getBalance(root->right) > 0) {
145         root->right = rotateRight(root->right);
146         return rotateLeft(root);
147     }
148
149     return root;
150 }
151
152 void BinaryTree::deleteValue(int value) {
153     root = deleteNode(root, value);
154 }
155
156 void BinaryTree::update(int oldVal, int newVal) {
157     deleteValue(oldVal);
158     insert(newVal);
159 }
160
```

```
161 void BinaryTree::inorder(Node* node) {
162     if (node == nullptr) return;
163     inorder(node->left);
164     cout << node->data << " ";
165     inorder(node->right);
166 }
167
168 void BinaryTree::preorder(Node* node) {
169     if (node == nullptr) return;
170     cout << node->data << " ";
171     preorder(node->left);
172     preorder(node->right);
173 }
174
175 void BinaryTree::postorder(Node* node) {
176     if (node == nullptr) return;
177     postorder(node->left);
178     postorder(node->right);
179     cout << node->data << " ";
180 }
181
182 void BinaryTree::inorder() { inorder(root); cout << endl; }
183 void BinaryTree::preorder() { preorder(root); cout << endl; }
184 void BinaryTree::postorder() { postorder(root); cout << endl; }
```



```

1 //Taufik Hafit Zakaria
2 //103112430093
3 //mainTree.cpp
4
5 #include <iostream>
6 #include "tree.h"
7 #include "tree.cpp"
8 using namespace std;
9
10 int main() {
11     BinaryTree tree;
12
13     cout << "=== INSERT DATA ===" << endl;
14     tree.insert(10);
15     tree.insert(15);
16     tree.insert(20);
17     tree.insert(30);
18     tree.insert(35);
19     tree.insert(40);
20     tree.insert(50);
21
22     cout << "Data yang diinsert:10, 15, 20, 30, 35, 40, 50" << endl;
23
24     cout << "\nTrasversal setelah Insert:" << endl;
25     cout << "Inorder: "; tree.inorder();
26     cout << "Preorder: "; tree.preorder();
27     cout << "Postorder: "; tree.postorder();
28
29     cout << "\n=== UPDATE DATA ===" << endl;
30     cout << "Update (20 -> 25)" << endl;
31     cout << "inorder      : "; tree.inorder();
32
33     tree.update(20, 25);
34
35     cout << "Setelah Update (20 -> 25):" << endl;
36     cout << "inorder      : "; tree.inorder();
37
38     cout << "\n=== DELETE DATA ===" << endl;
39     cout << "Sebelum delete (hapus sebtrees dengan root 30):" << endl;
40     cout << "inorder      : "; tree.inorder();
41
42     tree.deleteValue(30);
43     cout << "Setelah delete (hapus sebtrees dengan root 30):" << endl;
44     cout << "inorder      : "; tree.inorder();
45
46     return 0;
47 }

```

Screenshots Output

```
Problems  Output  Debug Console  Terminal  Ports

PS C:\programing\Struktur Data\LAPRAK 10\guided> cd "c:\programing\Struktur
Data\LAPRAK 10\guided\" ; if ($?) { g++ mainTree.cpp -o mainTree } ; if ($?)
{ .\mainTree }
=== INSERT DATA ===
Data yang diinsert:10, 15, 20, 30, 35, 40, 50

Trasversal 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 ===
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 sebtrees dengan root 30):
inorder    : 10 15 25 30 35 40 50
Setelah delete (hapus sebtrees dengan root 30):
inorder    : 10 15 25 35 40 50
PS C:\programing\Struktur Data\LAPRAK 10\guided>
```

Deskripsi:

Program ini adalah implementasi struktur data *Binary Search Tree* (BST) menggunakan C++ dengan pendekatan representasi pointer (linked list dinamis), yang dilengkapi dengan mekanisme penyeimbangan otomatis (AVL Tree) melalui rotasi. Intinya, program ini menerapkan prinsip hierarki terurut, di mana setiap *node* menempatkan nilai yang lebih kecil di *subtree* kiri dan nilai yang lebih besar di *subtree* kanan, guna mempercepat proses pencarian dan manipulasi data. Kode program diorganisir secara modular ke dalam tiga file: *tree.h* sebagai header yang berisi definisi struktur Node dan prototipe kelas, *tree.cpp* yang berisi implementasi logika rotasi dan rekursif, serta *mainTree.cpp* yang berfungsi sebagai driver untuk mendemonstrasikan operasi tree.

Alur program di fungsi main secara langsung mendemonstrasikan operasi manipulasi data pada tree. Awalnya, sebuah objek tree dibuat dan diisi dengan tujuh nilai integer (10, 15, 20, 30, 35, 40, dan 50) menggunakan fungsi insert. Program kemudian menampilkan struktur data yang terbentuk melalui tiga metode traversal: *Inorder*, *Preorder*, dan *Postorder*. Setelah itu, dilakukan operasi update untuk mengganti nilai 20 menjadi 25, diikuti dengan operasi delete untuk menghapus nilai 30 dari struktur. Hasil akhir ditampilkan kembali menggunakan traversal *Inorder*, yang membuktikan bahwa tree mampu mempertahankan urutan data yang benar dan struktur tetap seimbang meskipun telah terjadi penyisipan, pembaruan, dan penghapusan elemen.

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

Unguided 1

```
1 //Taufik Hafit Zakaria
2 //103112430093
3 //bstree.h
4
5 #ifndef BSTREE_H
6 #define BSTREE_H
7
8 #define Nil nullptr
9
10 typedef int infotype;
11 typedef struct Node *address;
12
13 struct Node {
14     infotype info;
15     address left;
16     address right;
17 };
18
19 typedef address BinTree;
20
21 // Fungsi alokasi
22 address alokasi(infotype x);
23
24 // Fungsi insert
25 void insertNode(address &root, infotype x);
26
27 // Fungsi find
28 address findNode(infotype x, address root);
29
30 // Fungsi traversal
31 void printInorder(address root);
32 void printPreorder(address root);
33 void printPostorder(address root);
34
35 // Fungsi untuk Soal 2
36 int hitungNode(address root);
37 int hitungTotal(address root);
38 int hitungKedalaman(address root, int start);
39
40 #endif
```



```
1 //Taufik Hafit Zakaria
2 //103112430093
3 //bstree.cpp
4
5 #include "bstree.h"
6 #include <iostream>
7 using namespace std;
8
9 // Fungsi alokasi untuk membuat node baru
10 address alokasi(infotype x) {
11     address newNode = new Node;
12     if (newNode != Nil) {
13         newNode->info = x;
14         newNode->left = Nil;
15         newNode->right = Nil;
16     }
17     return newNode;
18 }
19
20 // Fungsi insert node ke BST secara rekursif
21 void insertNode(address &root, infotype x) {
22     if (root == Nil) {
23         root = alokasi(x);
24     } else {
25         if (x < root->info) {
26             insertNode(root->left, x);
27         } else if (x > root->info) {
28             insertNode(root->right, x);
29         }
30         // Jika x == root->info, tidak insert (menghindari duplikasi)
31     }
32 }
33
```

```

34 // Fungsi untuk mencari node dengan nilai tertentu
35 address findNode(inftype x, address root) {
36     if (root == Nil) {
37         return Nil;
38     }
39
40     if (root->info == x) {
41         return root;
42     } else if (x < root->info) {
43         return findNode(x, root->left);
44     } else {
45         return findNode(x, root->right);
46     }
47 }
48
49 void printInorderRecursive(address root, bool &isFirst) {
50     if (root != Nil) {
51         printInorderRecursive(root->left, isFirst);
52
53         if (isFirst) {
54             isFirst = false;
55         } else {
56             cout << " - ";
57         }
58         cout << root->info;
59
60         printInorderRecursive(root->right, isFirst);
61     }
62 }
63
64 // Fungsi traversal Inorder: Left - Root - Right
65 void printInorder(address root) {
66     bool isFirst = true;
67     printInorderRecursive(root, isFirst);
68 }
69

```

```

70 // Fungsi traversal Preorder: Root - Left - Right
71 void printPreorder(address root) {
72     if (root != Nil) {
73         cout << root->info << " ";
74         printPreorder(root->left);
75         printPreorder(root->right);
76     }
77 }
78
79 // Fungsi traversal Postorder: Left - Right - Root
80 void printPostorder(address root) {
81     if (root != Nil) {
82         printPostorder(root->left);
83         printPostorder(root->right);
84         cout << root->info << " ";
85     }
86 }
87
88 // Fungsi untuk menghitung jumlah node dalam BST
89 int hitungNode(address root) {
90     if (root == Nil) {
91         return 0;
92     }
93     return 1 + hitungNode(root->left) + hitungNode(root->right);
94 }
95
96 // Fungsi untuk menghitung total nilai semua node
97 int hitungTotal(address root) {
98     if (root == Nil) {
99         return 0;
100     }
101     return root->info + hitungTotal(root->left) + hitungTotal(root->right);
102 }
103
104 // Fungsi rekursif untuk menghitung kedalaman maksimal tree
105 int hitungKedalaman(address root, int start) {
106     if (root == Nil) {
107         return start;
108     }
109
110     int kiri = hitungKedalaman(root->left, start + 1);
111     int kanan = hitungKedalaman(root->right, start + 1);
112
113     return (kiri > kanan) ? kiri : kanan;
114 }

```

```

1 //Taufik Hafit Zakaria
2 //103112430093
3 //mainBstree.cpp
4
5 #include <iostream>
6 #include "bstree.h"
7 #include "bstree.cpp"
8 using namespace std;
9
10 int main() {
11     cout << "Hello World" << endl;
12     address root = Nil;
13     // ===== SOAL 1 =====
14     cout << "===== " << endl;
15     cout << "SOAL 1: Implementasi ADT Binary Search Tree" << endl;
16     cout << "-----" << endl;
17     cout << "Memasukkan data: 1, 2, 6, 4, 5, 3, 6, 7" << endl;
18
19     insertNode(root, 1);
20     insertNode(root, 2);
21     insertNode(root, 6);
22     insertNode(root, 4);
23     insertNode(root, 5);
24     insertNode(root, 3);
25     insertNode(root, 6); // Duplikasi, tidak akan dimasukkan
26     insertNode(root, 7);
27
28     cout << "\nHasil Inorder traversal: ";
29     printInorder(root);
30     cout << endl;
31
32     // ===== SOAL 2 =====
33     cout << "===== " << endl;
34     cout << "SOAL 2: Fungsi Perhitungan" << endl;
35     cout << "-----" << endl;
36     cout << "Kedalaman      : " << hitungKedalaman(root, 0) << endl;
37     cout << "Jumlah Node      : " << hitungNode(root) << endl;
38     cout << "Total nilai Node: " << hitungTotal(root) << endl;
39     cout << endl;
40
41     // ===== SOAL 3 =====
42     cout << "===== " << endl;
43     cout << "SOAL 3: Print Tree (Pre-order & Post-order)" << endl;
44     cout << "-----" << endl;
45     cout << "Preorder  : ";
46     printPreorder(root);
47     cout << endl;
48
49     cout << "Postorder : ";
50     printPostorder(root);
51     cout << endl;
52
53     return 0;
54 }

```

Screenshots Output

```
PS C:\programing\Struktur Data\LAPRAK 10\unguided> cd "c:\programing\Struktur Data\LAPRAK 10\unguided\" ; if ($?) { g++ mainBstree.cpp -o mainBstree } ; if ($?) { .\mainBstree }
Hello World
=====
SOAL 1: Implementasi ADT Binary Search Tree
-----
Memasukkan data: 1, 2, 6, 4, 5, 3, 6, 7

Hasil Inorder traversal: 1 - 2 - 3 - 4 - 5 - 6 - 7
=====
SOAL 2: Fungsi Perhitungan
-----
Kedalaman      : 5
Jumlah Node     : 7
Total nilai Node: 28

=====
SOAL 3: Print Tree (Pre-order & Post-order)
-----
Preorder  : 1 2 6 4 3 5 7
Postorder : 3 5 4 7 6 2 1
PS C:\programing\Struktur Data\LAPRAK 10\unguided> |
```

Deskripsi:

Program ini adalah implementasi struktur data *Binary Search Tree* (BST) menggunakan C++ dengan pendekatan representasi pointer (linked list dinamis). Intinya, program ini menerapkan prinsip hierarki terurut, di mana setiap *node* menempatkan nilai yang lebih kecil di *subtree* kiri dan nilai yang lebih besar di *subtree* kanan, serta mencegah adanya duplikasi data. Kode program diorganisir secara modular ke dalam tiga file: *bstree.h* sebagai header yang berisi deklarasi struktur dan prototipe fungsi, *bstree.cpp* yang berisi implementasi detail logika rekursif, dan *mainBstree.cpp* yang berfungsi untuk mendemonstrasikan operasi pada tree.

Alur program di fungsi *main* secara langsung mendemonstrasikan cara kerjanya. Awalnya, sebuah tree kosong diinisialisasi, lalu program memasukkan serangkaian nilai (1, 2, 6, 4, 5, 3, 6, dan 7) secara berurutan. Karena BST ini menangani duplikasi, nilai 6 yang dimasukkan kedua kalinya secara otomatis diabaikan. Program kemudian membuktikan keterurutan data dengan menampilkan hasil traversal *Inorder* (1 - 2 - 3 ...), dilanjutkan dengan perhitungan statistik tree yang menghasilkan kedalaman 5, jumlah node 7, dan total nilai elemen sebesar 28. Terakhir, struktur hierarki tree divalidasi ulang dengan menampilkan urutan traversal *Preorder* dan *Postorder*, yang menunjukkan bahwa node-node telah tersusun dengan benar sesuai logika BST.

D. Kesimpulan

Dari hasil praktikum yang telah dilakukan, dapat disimpulkan bahwa struktur data Tree, khususnya *Binary Search Tree* (BST), memungkinkan pengelolaan data hierarkis secara efisien dengan menerapkan aturan pengurutan spesifik ($\text{Left Child} < \text{Root} < \text{Right Child}$). Melalui implementasi program latihan ini, konsep dasar operasi tree seperti penyisipan (*insert*), pencarian (*search*), dan perhitungan statistik node, serta berbagai metode penelusuran (*traversal*) dapat dipahami dan diterapkan dengan baik menggunakan teknik pemrograman rekursif. Praktikum ini secara khusus mendemonstrasikan bagaimana traversal *Inorder* pada BST selalu menghasilkan urutan data yang terurut secara menaik (*ascending*), serta bagaimana pointer digunakan secara dinamis untuk membentuk struktur percabangan tanpa batasan ukuran yang kaku seperti pada array statis.

Selain itu, program ini membuktikan bahwa penggunaan *Binary Search Tree* sangat berguna dalam kasus-kasus yang memerlukan pencarian data yang cepat dan manajemen data yang memiliki hubungan bertingkat, seperti pada struktur direktori file, hierarki organisasi, atau pengindeksan dalam basis data. Dengan memahami prinsip dasar BST dan logika rekursif yang menyertainya, mahasiswa dapat secara efektif memecahkan masalah algoritmik yang melibatkan struktur non-linear. Pemahaman ini juga menjadi fondasi penting untuk mempelajari struktur data pohon yang lebih kompleks seperti AVL Tree (untuk penyeimbangan otomatis), Heap Tree, maupun konsep algoritma lanjut yang berbasis graf.

E. Referensi

Malik, D. S. (2010). *Data Structures Using C++* (2nd ed.). Cengage Learning.

GeeksforGeeks. (2024). *Binary Search Tree Data Structure*. Diakses pada tahun 2025 dari <https://www.geeksforgeeks.org/binary-search-tree-data-structure/>

Programiz. (2024). *Binary Search Tree (BST)*. Diakses pada tahun 2025 dari <https://www.programiz.com/dsa/binary-search-tree>