

LAPORAN PRAKTIKUM
STRUKTUR DATA 9
"TREE"



Oleh:

NAMA: Ammar Dzaki Nandana

NIM: 2311104071

KELAS: SE 07 02

DOSEN: Wahyu Andi Saputra

PRODI S1 REKAYASA PERANGKAT LUNAK

FAKULTAS INFORMATIKA
INSTITUT TEKNOLOGI TELKOM PURWOKERTO

2023/2024

I. TUJUAN

Tujuan laporan praktikum tentang *tree* dalam C++ adalah untuk memberikan dokumentasi dan pemahaman mengenai praktik implementasi struktur data *tree*. Berikut adalah poin-poin tujuan yang dapat dijadikan panduan:

- Memahami Konsep Dasar Tree

Menjelaskan teori dasar tentang *tree*, jenis-jenisnya (seperti *binary tree*, *binary search tree*, *AVL tree*), dan bagaimana struktur data ini digunakan.

- Mengetahui Implementasi Tree di C++

Memahami bagaimana *tree* dapat diimplementasikan dalam C++, baik secara manual menggunakan pointer maupun dengan bantuan pustaka STL.

- Mengetahui Operasi Dasar pada Tree

Melakukan operasi seperti:

1. Penambahan (*insertion*).
2. Penghapusan (*deletion*).
3. Pencarian (*searching*).
4. Traversal (*in-order*, *pre-order*, *post-order*, dan *level-order*).

- Memahami Aplikasi Tree

Mengerti aplikasi dari *tree* dalam kehidupan nyata, seperti sistem file, struktur database, atau algoritma pencarian.

- Menguji Kinerja Implementasi

Menganalisis efisiensi implementasi dari segi waktu (*time complexity*) dan ruang (*space complexity*).

- Mengembangkan Kemampuan Pemecahan Masalah

Meningkatkan kemampuan pemrograman dengan memecahkan masalah yang membutuhkan penggunaan struktur data *tree*.

- Dokumentasi dan Penyampaian Hasil Praktikum

Mendokumentasikan hasil eksperimen, termasuk kode, *output*, dan kesimpulan dari praktikum

II. DASAR TEORI

Pengertian Dasar Tree

Sebuah *tree* adalah struktur data hierarkis yang terdiri dari simpul-simpul (*nodes*), dengan satu simpul utama disebut **root**. Setiap simpul dapat memiliki nol atau lebih anak (*children*), dan hubungan antar simpul menciptakan struktur seperti pohon.

Rekursi pada Tree

Rekursi adalah teknik pemrograman di mana suatu fungsi memanggil dirinya sendiri untuk memecahkan submasalah yang lebih kecil dari masalah utama. Rekursi sangat cocok digunakan pada *tree* karena struktur data ini memiliki sifat rekursif:

- Setiap simpul pada *tree* dapat dianggap sebagai **subtree** yang memiliki karakteristik yang sama dengan *tree* utamanya.
- Root memiliki beberapa *child nodes*, dan setiap *child node* dapat dilihat sebagai *root* dari *subtree*.

Implementasi Rekursif pada Tree

Rekursi sering digunakan untuk melakukan operasi pada *tree*, seperti:

Traversal

- *In-order traversal*: Kunjungi simpul kiri, root, lalu simpul kanan.
- *Pre-order traversal*: Kunjungi root, lalu simpul kiri, dan simpul kanan.
- *Post-order traversal*: Kunjungi simpul kiri, simpul kanan, lalu root.

Contoh kode rekursif untuk *in-order traversal*:

```
void inOrderTraversal(Node* root) {
    if (root != nullptr) {
        inOrderTraversal(root->left); // Kunjungi subtree kiri
        cout << root->data << " ";   // Cetak data root
        inOrderTraversal(root->right); // Kunjungi subtree kanan
    }
}
```

Pencarian (Search) Rekursi digunakan untuk mencari elemen tertentu, terutama pada *binary search tree* (BST):

```
bool search(Node* root, int value) {
    if (root == nullptr) return false; // Basis rekursi: simpul kosong
    if (root->data == value) return true;
    if (value < root->data)
        return search(root->left, value); // Rekursi ke subtree kiri
    else
        return search(root->right, value); // Rekursi ke subtree kanan
}
```

Penambahan (Insertion) Penambahan elemen baru dalam *tree* sering dilakukan secara rekursif:

```

Node* insert(Node* root, int value) {
    if (root == nullptr) { // Basis rekursi: simpul kosong
        root = new Node(value);
        return root;
    }
    if (value < root->data)
        root->left = insert(root->left, value); // Rekursi ke subtree kiri
    else
        root->right = insert(root->right, value); // Rekursi ke subtree kanan
    return root;
}

```

Penghapusan (Deletion) Penghapusan simpul dari *tree* juga melibatkan logika rekursif untuk menemukan simpul yang akan dihapus dan mengatur kembali struktur *tree*.


III. GUIDED

```

1 #include <iostream>
2 using namespace std;
3
4 struct Pohon {
5     char data;
6     Pohon *left, *right, *parent;
7 };
8
9 Pohon *root, *baru;
10
11 void init() { root = NULL; }
12 bool isEmpty() { return root == NULL; }
13
14 void buatNode(char data) {
15     if (isEmpty()) {
16         root = new Pohon(data, NULL, NULL, NULL);
17         cout << "\nNode " << data << " berhasil dibuat menjadi root." << endl;
18     } else {
19         cout << "\nPohon sudah dibuat." << endl;
20     }
21 }
22
23 Pohon* insertLeft(char data, Pohon *node) {
24     if (node->left != NULL) {
25         cout << "\nNode " << node->data << " sudah ada child kiri!" << endl;
26         return NULL;
27     }
28     baru = new Pohon(data, NULL, NULL, node);
29     node->left = baru;
30     cout << "\nNode " << data << " berhasil ditambahkan ke child kiri " << node->data << endl;
31     return baru;
32 }
33
34 Pohon* insertRight(char data, Pohon *node) {
35     if (node->right != NULL) {
36         cout << "\nNode " << node->data << " sudah ada child kanan!" << endl;
37         return NULL;
38     }
39     baru = new Pohon(data, NULL, NULL, node);
40     node->right = baru;
41     cout << "\nNode " << data << " berhasil ditambahkan ke child kanan " << node->data << endl;
42     return baru;
43 }
44
45 void update(char data, Pohon *node) {
46     if (!node) {
47         cout << "\nNode yang ingin diubah tidak ditemukan!" << endl;
48         return;
49     }
50     char temp = node->data;
51     node->data = data;
52     cout << "\nNode " << temp << " berhasil diubah menjadi " << data << endl;
53 }
54
55 void find(char data, Pohon *node) {
56     if (!node) return;
57     if (node->data == data) {
58         cout << "\nNode ditemukan: " << data << endl;
59         return;
60     }
61     find(data, node->left);
62     find(data, node->right);
63 }
64
65 void preOrder(Pohon *node) {
66     if (!node) return;
67     cout << node->data << " ";
68     preOrder(node->left);
69     preOrder(node->right);
70 }
71
72 void inOrder(Pohon *node) {
73     if (!node) return;
74     inOrder(node->left);
75     cout << node->data << " ";
76     inOrder(node->right);
77 }
78
79 void postOrder(Pohon *node) {
80     if (!node) return;
81     postOrder(node->left);
82     postOrder(node->right);
83     cout << node->data << " ";
84 }
85
86 Pohon* deleteNode(Pohon *node, char data) {
87     if (!node) return NULL;
88     if (data < node->data) {
89         node->left = deleteNode(node->left, data);
90     } else if (data > node->data) {
91         node->right = deleteNode(node->right, data);
92     } else {
93         if (!node->left) {
94             Pohon *temp = node->right;
95             delete node;
96             return temp;
97         } else if (!node->right) {
98             Pohon *temp = node->left;
99             delete node;
100            return temp;
101        }
102        Pohon *successor = node->right;
103        while (successor->left) successor = successor->left;
104        node->data = successor->data;
105        node->right = deleteNode(node->right, successor->data);
106    }
107    return node;
108 }
109
110 Pohon* mostLeft(Pohon *node) {
111     if (!node) return NULL;
112     while (node->left) node = node->left;
113     return node;
114 }
115
116 Pohon* mostRight(Pohon *node) {
117     if (!node) return NULL;
118     while (node->right) node = node->right;
119     return node;
120 }
121
122 int main() {
123     init();
124     buatNode('F');
125     insertLeft('B', root);
126     insertRight('G', root);
127     insertLeft('A', root->left);
128     insertRight('D', root->left);
129     insertLeft('C', root->left->right);
130     insertRight('E', root->left->right);
131
132     cout << "\nPre-order Traversal: ";
133     preOrder(root);
134     cout << "\nin-order Traversal: ";
135     inOrder(root);
136     cout << "\nPost-order Traversal: ";
137     postOrder(root);
138
139     cout << "\nMost Left Node: " << mostLeft(root)->data;
140     cout << "\nMost Right Node: " << mostRight(root)->data;
141
142     cout << "\nMenghapus node D.";
143     root = deleteNode(root, 'D');
144     cout << "\nin-order Traversal setelah penghapusan: ";
145     inOrder(root);
146
147     return 0;
148 }
149

```

IV. UNGUIDED



```
1  #ifndef BSTREE_H
2  #define BSTREE_H
3
4  #include <iostream>
5  using namespace std;
6
7  // Definisi tipe data
8  typedef int infotype;
9  typedef struct Node* address;
10
11 struct Node {
12     infotype info;
13     address left;
14     address right;
15 };
16
17 // Fungsi dan prosedur
18 address alokasi(infotype x);
19 void insertNode(address& root, infotype x);
20 address findNode(infotype x, address root);
21 void printInorder(address root);
22 void printPreorder(address root);
23 void printPostorder(address root);
24 int hitungJumlahNode(address root);
25 int hitungTotalInfo(address root);
26 int hitungKedalaman(address root);
27
28 #endif
29
```

```

1  #include "BSTREE_H"
2
3  // Fungsi alokasi node baru
4  address alokasi(infotype x) {
5      address newNode = new Node;
6      newNode->info = x;
7      newNode->left = nullptr;
8      newNode->right = nullptr;
9      return newNode;
10 }
11
12 // Fungsi untuk menyisipkan node ke dalam BST
13 void insertNode(address& root, infotype x) {
14     if (root == nullptr) {
15         root = alokasi(x);
16     } else if (x < root->info) {
17         insertNode(root->left, x);
18     } else if (x > root->info) {
19         insertNode(root->right, x);
20     }
21 }
22
23 // Fungsi untuk mencari node berdasarkan nilai
24 address findNode(infotype x, address root) {
25     if (root == nullptr || root->info == x) {
26         return root;
27     } else if (x < root->info) {
28         return findNode(x, root->left);
29     } else {
30         return findNode(x, root->right);
31     }
32 }
33
34 // Traversal in-order
35 void printInorder(address root) {
36     if (root != nullptr) {
37         printInorder(root->left);
38         cout << root->info << " ";
39         printInorder(root->right);
40     }
41 }
42
43 // Traversal pre-order
44 void printPreorder(address root) {
45     if (root != nullptr) {
46         cout << root->info << " ";
47         printPreorder(root->left);
48         printPreorder(root->right);
49     }
50 }
51
52 // Traversal post-order
53 void printPostorder(address root) {
54     if (root != nullptr) {
55         printPostorder(root->left);
56         printPostorder(root->right);
57         cout << root->info << " ";
58     }
59 }
60
61 // Fungsi untuk menghitung jumlah node
62 int hitungJumlahNode(address root) {
63     if (root == nullptr) {
64         return 0;
65     } else {
66         return 1 + hitungJumlahNode(root->left) + hitungJumlahNode(root->right);
67     }
68 }
69
70 // Fungsi untuk menghitung total nilai info pada node
71 int hitungTotalInfo(address root) {
72     if (root == nullptr) {
73         return 0;
74     } else {
75         return root->info + hitungTotalInfo(root->left) + hitungTotalInfo(root->right);
76     }
77 }
78
79 // Fungsi untuk menghitung kedalaman maksimal BST
80 int hitungKedalaman(address root) {
81     if (root == nullptr) {
82         return 0;
83     } else {
84         int leftDepth = hitungKedalaman(root->left);
85         int rightDepth = hitungKedalaman(root->right);
86         return 1 + max(leftDepth, rightDepth);
87     }
88 }
89

```

```

1  #include "BSTREE_H"
2
3  int main() {
4      address root = nullptr;
5
6      // Menambahkan node ke dalam BST
7      insertNode(root, 1);
8      insertNode(root, 2);
9      insertNode(root, 6);
10     insertNode(root, 4);
11     insertNode(root, 5);
12     insertNode(root, 3);
13     insertNode(root, 7);
14
15     // Traversal
16     cout << "In-order Traversal: ";
17     printInorder(root);
18     cout << "\nPre-order Traversal: ";
19     printPreorder(root);
20     cout << "\nPost-order Traversal: ";
21     printPostorder(root);
22
23     // Informasi tambahan
24     cout << "\nJumlah Node: " << hitungJumlahNode(root);
25     cout << "\nTotal Info: " << hitungTotalInfo(root);
26     cout << "\nKedalaman Maksimal: " << hitungKedalaman(root);
27
28     return 0;
29 }
30

```

V. KESIMPULAN

Rekursi merupakan konsep penting dalam struktur data *tree* karena sifat alami *tree* yang bersifat hierarkis dan rekursif. Dalam *tree*, setiap simpul dapat dianggap sebagai *subtree* yang memiliki karakteristik serupa dengan *tree* utamanya, sehingga banyak operasi seperti *traversal*, pencarian, penambahan, dan penghapusan dapat diimplementasikan secara rekursif. Rekursi memberikan cara yang sederhana dan intuitif untuk memproses *tree*, terutama untuk menelusuri semua elemen atau memanipulasi struktur data ini. Meskipun rekursi menawarkan keunggulan dalam hal kemudahan implementasi, penggunaan rekursi pada *tree* juga memiliki keterbatasan, seperti potensi penggunaan memori yang besar dan risiko *stack overflow* pada *tree* yang sangat dalam. Oleh karena itu, pemahaman yang baik tentang prinsip rekursi dan implementasinya sangat penting untuk memanfaatkan struktur data *tree* secara efisien.

