

**LAPORAN PRAKTIKUM  
STRUKTUR DATA  
MODUL 9  
“ TREE ”**



**Disusun Oleh:**  
**Dhiya Ulhaq Ramadhan 2211104053**  
**Kelas :**  
**S1SE-07-02**  
**Dosen :**  
**Wahyu Andi Saputra, S.Pd., M.Eng.**

**PROGRAM STUDI S1 SOFTWARE ENGINEERING  
FAKULTAS INFORMATIKA  
TELKOM UNIVERSITY  
PURWOKERTO  
2024**

## 1. Tujuan

- Memahami konsep penggunaan fungsi rekursif.
- Mengimplementasikan bentuk-bentuk fungsi rekursif.
- Mengaplikasikan struktur data tree dalam sebuah kasus pemrograman.
- Mengimplementasikan struktur data tree, khususnya Binary Tree.

## 2. Landasan Teori

Struktur data tree merupakan struktur data non-linear yang terdiri dari node-node yang saling terhubung dalam bentuk hierarki. Konsep utama yang dibahas meliputi rekursif sebagai dasar operasi tree, di mana rekursif adalah proses yang dapat memanggil dirinya sendiri dengan kondisi tertentu. Tree memiliki beberapa karakteristik penting seperti adanya root (akar), parent-child relationship, dan berbagai jenis tree seperti Binary Tree, Binary Search Tree, AVL Tree, dan Heap Tree.

## 3. Guided 1

Source code :

```
1  #include <iostream>
2  using namespace std;
3
4  /// PROGRAM BINARY TREE
5
6  // Struktur data pohon biner untuk menyimpan data dan pointer ke anak kiri, kan
7  struct Pohon {
8      char data;                // Data yang disimpan di node (tipe char)
9      Pohon *left, *right;      // Pointer ke anak kiri dan anak kanan
10     Pohon *parent;            // Pointer ke node induk
11 };
12
13 // Variabel global untuk menyimpan root (akar) pohon dan node baru
14 Pohon *root, *baru;
15
16 // Inisialisasi pohon agar kosong
17 void init() {
18     root = NULL; // Mengatur root sebagai NULL (pohon kosong)
19 }
20
21 // Mengecek apakah pohon kosong
22 bool isEmpty() {
23     return root == NULL; // Mengembalikan true jika root adalah NULL
24 }
25
26 // Membuat node baru sebagai root pohon
27 void buatNode(char data) {
28     if (isEmpty()) { // Jika pohon kosong
29         root = new Pohon(data, NULL, NULL, NULL); // Membuat node baru sebagai
30         cout << "\nNode " << data << " berhasil dibuat menjadi root." << endl;
31     } else {
```

```

32     cout << "\nPohon sudah dibuat." << endl; // Root sudah ada, tidak membuat node baru
33 }
34 }
35
36 // Menambahkan node baru sebagai anak kiri dari node tertentu
37 Pohon* insertLeft(char data, Pohon *node) {
38     if (node->left != NULL) { // Jika anak kiri sudah ada
39         cout << "\nNode " << node->data << " sudah ada child kiri!" << endl;
40         return NULL; // Tidak menambahkan node baru
41     }
42     // Membuat node baru dan menghubungkannya ke node sebagai anak kiri
43     baru = new Pohon(data, NULL, NULL, node);
44     node->left = baru;
45     cout << "\nNode " << data << " berhasil ditambahkan ke child kiri " << node->data << endl;
46     return baru; // Mengembalikan pointer ke node baru
47 }
48
49 // Menambahkan node baru sebagai anak kanan dari node tertentu
50 Pohon* insertRight(char data, Pohon *node) {
51     if (node->right != NULL) { // Jika anak kanan sudah ada
52         cout << "\nNode " << node->data << " sudah ada child kanan!" << endl;
53         return NULL; // Tidak menambahkan node baru
54     }
55     // Membuat node baru dan menghubungkannya ke node sebagai anak kanan
56     baru = new Pohon(data, NULL, NULL, node);
57     node->right = baru;
58     cout << "\nNode " << data << " berhasil ditambahkan ke child kanan " << node->data << endl;
59     return baru; // Mengembalikan pointer ke node baru
60 }
61
62 // Mengubah data di dalam sebuah node
63 void update(char data, Pohon *node) {
64     if (!node) { // Jika node tidak ditemukan
65         cout << "\nNode yang ingin diubah tidak ditemukan!" << endl;
66         return;
67     }
68     char temp = node->data; // Menyimpan data lama
69     node->data = data; // Mengubah data dengan nilai baru
70     cout << "\nNode " << temp << " berhasil diubah menjadi " << data << endl;
71 }
72
73 // Mencari node dengan data tertentu
74 void find(char data, Pohon *node) {
75     if (!node) return; // Jika node tidak ada, hentikan pencarian
76
77     if (node->data == data) { // Jika data ditemukan
78         cout << "\nNode ditemukan: " << data << endl;
79         return;
80     }
81     // Melakukan pencarian secara rekursif ke anak kiri dan kanan
82     find(data, node->left);
83     find(data, node->right);
84 }
85
86 // Traversal Pre-order (Node -> Kiri -> Kanan)
87 void preOrder(Pohon *node) {
88     if (!node) return; // Jika node kosong, hentikan traversal
89     cout << node->data << " "; // Cetak data node saat ini
90     preOrder(node->left); // Traversal ke anak kiri
91     preOrder(node->right); // Traversal ke anak kanan
92 }

```

```

94 // Traversal In-order (Kiri -> Node -> Kanan)
95 void inOrder(Pohon *node) {
96     if (!node) return; // Jika node kosong, hentikan
97     inOrder(node->left); // Traversal ke anak kiri
98     cout << node->data << " "; // Cetak data node sa
99     inOrder(node->right); // Traversal ke anak kanan
100 }
101
102 // Traversal Post-order (Kiri -> Kanan -> Node)
103 void postOrder(Pohon *node) {
104     if (!node) return; // Jika node kosong, hentikan
105     postOrder(node->left); // Traversal ke anak kiri
106     postOrder(node->right); // Traversal ke anak kanan
107     cout << node->data << " "; // Cetak data node sa
108 }
109
110 // Menghapus node dengan data tertentu
111 Pohon* deleteNode(Pohon *node, char data) {
112     if (!node) return NULL; // Jika node kosong, hentikan
113
114     // Rekursif mencari node yang akan dihapus
115     if (data < node->data) {
116         node->left = deleteNode(node->left, data); //
117     } else if (data > node->data) {
118         node->right = deleteNode(node->right, data);
119     } else {
120         // Jika node ditemukan
121         if (!node->left) { // Jika tidak ada anak kiri
122             Pohon *temp = node->right; // Anak kanan
123             delete node;
124             return temp;
125
126         } else if (!node->right) { // Jika tidak ada anak kanan
127             Pohon *temp = node->left; // Anak kiri menggantikan
128             delete node;
129             return temp;
130
131         }
132
133         // Jika node memiliki dua anak, cari node pengganti (suksesor)
134         Pohon *successor = node->right;
135         while (successor->left) successor = successor->left; // Iterasi ke anak kiri
136         node->data = successor->data; // Gantikan data dengan suksesor
137         node->right = deleteNode(node->right, successor->data);
138     }
139     return node;
140 }
141
142 // Menemukan node paling kiri
143 Pohon* mostLeft(Pohon *node) {
144     if (!node) return NULL; // Jika node kosong, hentikan
145     while (node->left) node = node->left; // Iterasi ke anak kiri
146     return node;
147 }
148
149 // Menemukan node paling kanan
150 Pohon* mostRight(Pohon *node) {
151     if (!node) return NULL; // Jika node kosong, hentikan
152     while (node->right) node = node->right; // Iterasi ke anak kanan
153     return node;
154 }

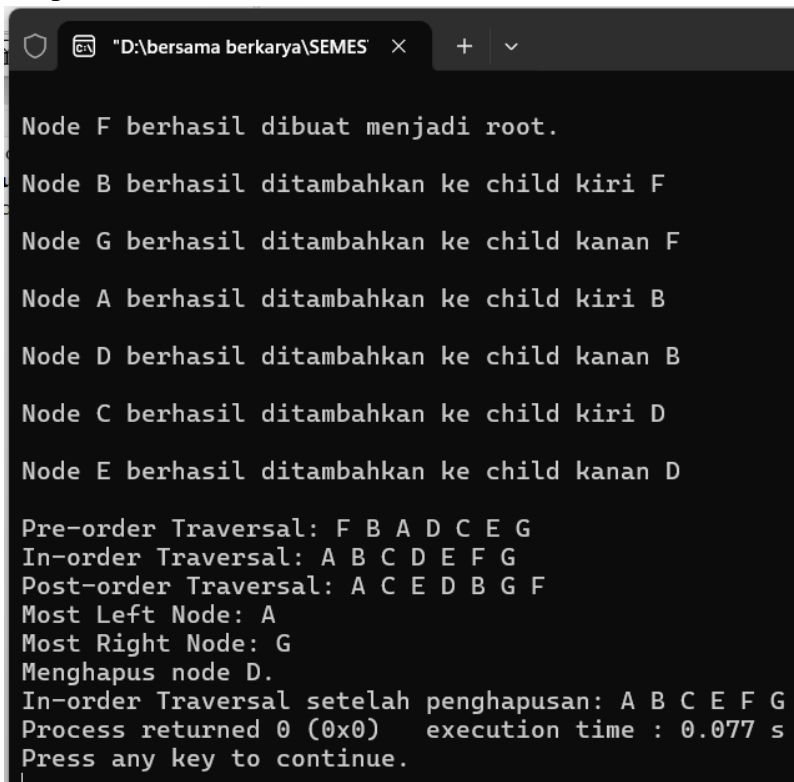
```

```

154 // Fungsi utama
155 int main() {
156     init(); // Inisialisasi pohon
157     buatNode('F'); // Membuat root dengan data 'F'
158     insertLeft('B', root); // Menambahkan 'B' ke anak kiri root
159     insertRight('G', root); // Menambahkan 'G' ke anak kanan root
160     insertLeft('A', root->left); // Menambahkan 'A' ke anak kiri dari 'B'
161     insertRight('D', root->left); // Menambahkan 'D' ke anak kanan dari 'B'
162     insertLeft('C', root->left->right); // Menambahkan 'C' ke anak kiri dari 'D'
163     insertRight('E', root->left->right); // Menambahkan 'E' ke anak kanan dari 'D'
164
165     // Traversal pohon
166     cout << "\nPre-order Traversal: ";
167     preOrder(root);
168     cout << "\nIn-order Traversal: ";
169     inOrder(root);
170     cout << "\nPost-order Traversal: ";
171     postOrder(root);
172
173     // Menampilkan node paling kiri dan kanan
174     cout << "\nMost Left Node: " << mostLeft(root)->data;
175     cout << "\nMost Right Node: " << mostRight(root)->data;
176
177     // Menghapus node
178     cout << "\nMenghapus node D.";
179     root = deleteNode(root, 'D');
180     cout << "\nIn-order Traversal setelah penghapusan: ";
181     inOrder(root);
182
183     return 0;
184 }

```

Output :



```

Node F berhasil dibuat menjadi root.
Node B berhasil ditambahkan ke child kiri F
Node G berhasil ditambahkan ke child kanan F
Node A berhasil ditambahkan ke child kiri B
Node D berhasil ditambahkan ke child kanan B
Node C berhasil ditambahkan ke child kiri D
Node E berhasil ditambahkan ke child kanan D

Pre-order Traversal: F B A D C E G
In-order Traversal: A B C D E F G
Post-order Traversal: A C E D B G F
Most Left Node: A
Most Right Node: G
Menghapus node D.
In-order Traversal setelah penghapusan: A B C E F G
Process returned 0 (0x0)   execution time : 0.077 s
Press any key to continue.

```

Deskripsi program

Pertama-tama dilakukan inisialisasi pohon kosong dengan fungsi init(). Kemudian program membuat root (akar) pohon dengan data 'F' menggunakan fungsi buatNode(). Setelah root terbentuk, program menambahkan beberapa node baru ke dalam pohon

menggunakan fungsi `insertLeft()` dan `insertRight()` sehingga membentuk struktur pohon yang lengkap. Node 'B' ditambahkan sebagai anak kiri root dan 'G' sebagai anak kanan. Selanjutnya node 'A' ditambahkan sebagai anak kiri dari 'B', dan 'D' sebagai anak kanan 'B'. Program kemudian menambahkan node 'C' sebagai anak kiri 'D' dan 'E' sebagai anak kanan 'D'.

Program melakukan tiga jenis traversal untuk menampilkan data dalam pohon. Pre-order traversal menelusuri pohon dengan urutan node saat ini, anak kiri, lalu anak kanan. In-order traversal menelusuri dengan urutan anak kiri, node saat ini, lalu anak kanan. Post-order traversal menelusuri dengan urutan anak kiri, anak kanan, lalu node saat ini.

Program juga menampilkan node yang berada paling kiri dan paling kanan dalam pohon menggunakan fungsi `mostLeft()` dan `mostRight()`. Terakhir, program mendemonstrasikan operasi penghapusan node dengan menghapus node 'D' menggunakan fungsi `deleteNode()`, lalu menampilkan hasil traversal in-order setelah penghapusan untuk memperlihatkan perubahan struktur pohon.

Output program akan menampilkan urutan node-node sesuai dengan ketiga jenis traversal tersebut, nilai node paling kiri dan kanan, serta urutan node setelah penghapusan node 'D'.

1. Modifikasi guided tree diatas dengan program menu menggunakan input data tree dari user dan berikan fungsi tambahan untuk menampilkan node child dan descendant dari node yang diinputkan!

Jawaban :

Source code

```

14 //Nomor 1: Implementasi Menu dan Fungsi Tambahan
15 void init() {
16     root = NULL;
17 }
18
19 bool isEmpty() {
20     return root == NULL;
21 }
22
23 //Membuat node baru dengan input user
24 void buatNode() {
25     int data;
26     cout << "Masukkan nilai node: ";
27     cin >> data;
28
29     if (isEmpty()) {
30         root = new Pohon(data, NULL, NULL, NULL);
31         cout << "\nNode " << data << " berhasil dibuat menjadi root." << endl;
32     } else {
33         cout << "\nPohon sudah memiliki root." << endl;
34     }
35 }
36
37 //Menampilkan child dari sebuah node
38 void tampilkanChild(Pohon* node) {
39     if (!node) {
40         cout << "Node tidak ditemukan!" << endl;
41         return;
42     }
43
44     cout << "Node " << node->data << ":" << endl;
45
46     if (node->left)
47         cout << "Child kiri: " << node->left->data << endl;
48     else
49         cout << "Tidak ada child kiri" << endl;
50
51     if (node->right)
52         cout << "Child kanan: " << node->right->data << endl;
53     else
54         cout << "Tidak ada child kanan" << endl;
55 }
56
57 //Menampilkan semua descendant dari sebuah node
58 void tampilkanDescendant(Pohon* node) {
59     if (!node) return;
60
61     if (node->left) {
62         cout << node->left->data << " ";
63         tampilkanDescendant(node->left);
64     }
65     if (node->right) {
66         cout << node->right->data << " ";
67         tampilkanDescendant(node->right);
68     }
69 }

```

Output :

```

=== MENU BINARY TREE ===
1. Buat Root
2. Tambah Node Kiri
3. Tambah Node Kanan
4. Tampilkan Child
5. Tampilkan Descendant
6. Cek Valid BST
7. Hitung Simpul Daun
8. Keluar
Pilihan: 1
Masukkan nilai node: 3

Node 3 berhasil dibuat menjadi root.

=== MENU BINARY TREE ===
1. Buat Root
2. Tambah Node Kiri
3. Tambah Node Kanan
4. Tampilkan Child
5. Tampilkan Descendant
6. Cek Valid BST
7. Hitung Simpul Daun
8. Keluar
Pilihan: 2
Masukkan nilai parent: 3
Masukkan nilai node baru: 5
Node berhasil ditambahkan sebagai child kiri

=== MENU BINARY TREE ===
1. Buat Root
2. Tambah Node Kiri
3. Tambah Node Kanan
4. Tampilkan Child
5. Tampilkan Descendant
6. Cek Valid BST
7. Hitung Simpul Daun
8. Keluar
Pilihan: 3
Masukkan nilai parent: 3
Masukkan nilai node baru: 7
Node berhasil ditambahkan sebagai child kanan

=== MENU BINARY TREE ===
1. Buat Root
2. Tambah Node Kiri
3. Tambah Node Kanan
4. Tampilkan Child
5. Tampilkan Descendant
6. Cek Valid BST
7. Hitung Simpul Daun
8. Keluar
Pilihan: 4
Masukkan nilai node: 3
Node 3:
Child kiri: 5
Child kanan: 7

=== MENU BINARY TREE ===
1. Buat Root
2. Tambah Node Kiri
3. Tambah Node Kanan
4. Tampilkan Child
5. Tampilkan Descendant
6. Cek Valid BST
7. Hitung Simpul Daun
8. Keluar
Pilihan: 5
Masukkan nilai node: 3
Descendant dari node 3: 5 7

```

2. Buatlah fungsi rekursif `is_valid_bst(node, min_val, max_val)` untuk memeriksa apakah suatu pohon memenuhi properti Binary Search Tree. Uji fungsi ini pada berbagai pohon, baik yang valid maupun tidak valid sebagai BST.

Source code :

```

70 //Nomor 2: Validasi BST
71 bool is_valid_bst(Pohon* node, int min_val, int max_val) {
72     if (!node) return true;
73
74     // Cek apakah nilai node berada dalam range yang valid
75     if (node->data <= min_val || node->data >= max_val)
76         return false;
77
78     // Rekursif cek subtree kiri dan kanan
79     return is_valid_bst(node->left, min_val, node->data) &&
80            is_valid_bst(node->right, node->data, max_val);
81 }

```



Output :

Lanjutan dari program pada no 1

```
=== MENU BINARY TREE ===
1. Buat Root
2. Tambah Node Kiri
3. Tambah Node Kanan
4. Tampilkan Child
5. Tampilkan Descendant
6. Cek Valid BST
7. Hitung Simpul Daun
8. Keluar
Pilihan: 6
Pohon bukan BST valid
```

Jika BST valid

```
=== MENU BINARY TREE ===
1. Buat Root
2. Tambah Node Kiri
3. Tambah Node Kanan
4. Tampilkan Child
5. Tampilkan Descendant
6. Cek Valid BST
7. Hitung Simpul Daun
8. Keluar
Pilihan: 1
Masukkan nilai node: 15

Node 15 berhasil dibuat menjadi root.

=== MENU BINARY TREE ===
1. Buat Root
2. Tambah Node Kiri
3. Tambah Node Kanan
4. Tampilkan Child
5. Tampilkan Descendant
6. Cek Valid BST
7. Hitung Simpul Daun
8. Keluar
Pilihan: 2
Masukkan nilai parent: 15
Masukkan nilai node baru: 10
Node berhasil ditambahkan sebagai child kiri

=== MENU BINARY TREE ===
1. Buat Root
2. Tambah Node Kiri
3. Tambah Node Kanan
4. Tampilkan Child
5. Tampilkan Descendant
6. Cek Valid BST
7. Hitung Simpul Daun
8. Keluar
Pilihan: 3
Masukkan nilai parent: 15
Masukkan nilai node baru: 20
Node berhasil ditambahkan sebagai child kanan

=== MENU BINARY TREE ===
1. Buat Root
2. Tambah Node Kiri
3. Tambah Node Kanan
4. Tampilkan Child
5. Tampilkan Descendant
6. Cek Valid BST
7. Hitung Simpul Daun
8. Keluar
Pilihan: 6
Pohon adalah BST valid
```

3. Buatlah fungsi rekursif cari\_simpul\_daun(node) untuk menghitung jumlah simpul daun dalam Binary Tree. Simpul daun adalah node yang tidak memiliki anak kiri maupun kanan.

Source code

```

83 //Nomor 3: Menghitung Simpul Daun
84 int cari_simpul_daun(Pohon* node) {
85     if (!node) return 0;
86
87     // Jika node adalah daun (tidak punya anak)
88     if (!node->left && !node->right)
89         return 1;
90
91     // Rekursif hitung daun di subtree kiri dan kanan
92     return cari_simpul_daun(node->left) + cari_simpul_daun(node->right);
93 }
94
95 // Fungsi helper untuk mencari node berdasarkan nilai
96 Pohon* cariNode(Pohon* node, int nilai) {
97     if (!node || node->data == nilai) return node;
98
99     Pohon* kiri = cariNode(node->left, nilai);
100    if (kiri) return kiri;
101
102    return cariNode(node->right, nilai);
103 }

```

Output

Lanjutan dari program no 2

```

=== MENU BINARY TREE ===
1. Buat Root
2. Tambah Node Kiri
3. Tambah Node Kanan
4. Tampilkan Child
5. Tampilkan Descendant
6. Cek Valid BST
7. Hitung Simpul Daun
8. Keluar
Pilihan: 7
Jumlah simpul daun: 2

```

**Full source code di Github**

### Kesimpulan

Dari implementasi dan pengujian program pada tugas Unguided kali ini, dapat disimpulkan bahwa:

1. Modifikasi program dengan menu interaktif memberikan fleksibilitas dalam membangun dan memanipulasi binary tree, memungkinkan saya dengan mudah menambahkan node dan memeriksa struktur pohon.
2. Validasi BST menunjukkan pentingnya mempertahankan properti terurut dalam binary search tree. Pohon yang dibangun dalam contoh tidak valid sebagai BST karena pelanggaran aturan penempatan nilai (node 5 di sebelah kiri node 3).
3. Penghitungan simpul daun (2 simpul) memperlihatkan bagaimana rekursi dapat



digunakan untuk menganalisis karakteristik struktural pohon. Jumlah simpul daun memberikan informasi tentang seberapa luas pohon tersebut berkembang.