

LAPORAN PRAKTIKUM
Modul 09
“TREE(BAGIAN KEDUA)”



Disusun Oleh:
Faishal Arif Setiawan -2311104066
Kelas:
SE 07 02

Dosen :
WAHYU ANDI SAPUTRA.S.PD.,M.ENG

PROGRAM STUDI S1 SOFTWARE ENGINEERING
FAKULTAS INFORMATIKA
TELKOM UNIVERSITY
PURWOKERTO
2024

1. TUJUAN

1. Mengimplementasikan struktur data *tree*, khususnya *Binary Tree*

2. Landasan Teori

Tree adalah struktur data hierarkis yang terdiri dari node-node yang saling terhubung, sering digunakan untuk merepresentasikan hubungan hierarkis seperti silsilah keluarga atau struktur file. Operasi penting pada tree meliputi traversal (pre-order, in-order, post-order), penambahan node, dan penghapusan node. Dalam Binary Search Tree (BST), aturan utamanya adalah semua node di subtree kiri memiliki nilai lebih kecil, dan semua node di subtree kanan memiliki nilai lebih besar dari node induknya. Node yang paling kiri (most-left) dalam BST mewakili nilai terkecil, sementara node yang paling kanan (most-right) adalah nilai terbesar.

Simpul daun (leaf node) adalah node tanpa anak kiri maupun kanan, dan sering digunakan untuk menghitung elemen terminal dalam tree. Penghapusan node pada tree bergantung pada jumlah anak node yang dihapus: simpul daun dihapus langsung, node dengan satu anak digantikan oleh anaknya, dan node dengan dua anak digantikan oleh nilai pengganti dari subtree (most-right dari kiri atau most-left dari kanan).

3. Guided

```
1 struct tnode {
2     int data;
3     tnode *left;
4     tnode *right;
5     tnode *parent;
6 }
7
8 // Fungsi untuk membuat node baru
9 tnode *new_node(int data) {
10     tnode *new_node = (tnode *) malloc(sizeof(tnode));
11     new_node->data = data;
12     new_node->left = NULL;
13     new_node->right = NULL;
14     new_node->parent = NULL;
15     return new_node;
16 }
17
18 // Fungsi untuk mencari node yang sesuai
19 tnode *find_node(tnode *root, int data) {
20     if (root == NULL) return NULL;
21     if (root->data == data) return root;
22     tnode *left = find_node(root->left, data);
23     if (left != NULL) return left;
24     tnode *right = find_node(root->right, data);
25     if (right != NULL) return right;
26     return NULL;
27 }
28
29 // Fungsi untuk menambahkan node ke pohon
30 void insert_node(tnode *root, int data) {
31     if (root == NULL) {
32         root = new_node(data);
33         return;
34     }
35     tnode *parent = NULL;
36     tnode *current = root;
37     while (current != NULL) {
38         parent = current;
39         if (current->data < data) {
40             current = current->right;
41         } else {
42             current = current->left;
43         }
44     }
45     current->parent = parent;
46     if (parent->data < data) {
47         parent->right = new_node(data);
48     } else {
49         parent->left = new_node(data);
50     }
51 }
52
53 // Fungsi untuk menghapus node dari pohon
54 void delete_node(tnode *root, int data) {
55     if (root == NULL) return;
56     tnode *parent = NULL;
57     tnode *current = root;
58     while (current != NULL) {
59         parent = current;
60         if (current->data == data) {
61             if (current->left == NULL) {
62                 parent->right = current->right;
63             } else if (current->right == NULL) {
64                 parent->left = current->left;
65             } else {
66                 tnode *successor = find_node(current->right, current->left->data);
67                 successor->parent = parent;
68                 parent->right = successor;
69                 current->right = successor->right;
70             }
71             return;
72         }
73         if (current->data < data) {
74             current = current->right;
75         } else {
76             current = current->left;
77         }
78     }
79 }
80
81 // Fungsi untuk mencari node yang sesuai
82 tnode *find_node(tnode *root, int data) {
83     if (root == NULL) return NULL;
84     if (root->data == data) return root;
85     tnode *left = find_node(root->left, data);
86     if (left != NULL) return left;
87     tnode *right = find_node(root->right, data);
88     if (right != NULL) return right;
89     return NULL;
90 }
91
92 // Fungsi untuk menambahkan node ke pohon
93 void insert_node(tnode *root, int data) {
94     if (root == NULL) {
95         root = new_node(data);
96         return;
97     }
98     tnode *parent = NULL;
99     tnode *current = root;
100     while (current != NULL) {
101         parent = current;
102         if (current->data < data) {
103             current = current->right;
104         } else {
105             current = current->left;
106         }
107     }
108     current->parent = parent;
109     if (parent->data < data) {
110         parent->right = new_node(data);
111     } else {
112         parent->left = new_node(data);
113     }
114 }
115
116 // Fungsi untuk menghapus node dari pohon
117 void delete_node(tnode *root, int data) {
118     if (root == NULL) return;
119     tnode *parent = NULL;
120     tnode *current = root;
121     while (current != NULL) {
122         parent = current;
123         if (current->data == data) {
124             if (current->left == NULL) {
125                 parent->right = current->right;
126             } else if (current->right == NULL) {
127                 parent->left = current->left;
128             } else {
129                 tnode *successor = find_node(current->right, current->left->data);
130                 successor->parent = parent;
131                 parent->right = successor;
132                 current->right = successor->right;
133             }
134             return;
135         }
136         if (current->data < data) {
137             current = current->right;
138         } else {
139             current = current->left;
140         }
141     }
142 }
143
144 // Fungsi untuk mencari node yang sesuai
145 tnode *find_node(tnode *root, int data) {
146     if (root == NULL) return NULL;
147     if (root->data == data) return root;
148     tnode *left = find_node(root->left, data);
149     if (left != NULL) return left;
150     tnode *right = find_node(root->right, data);
151     if (right != NULL) return right;
152     return NULL;
153 }
154
155 // Fungsi untuk menambahkan node ke pohon
156 void insert_node(tnode *root, int data) {
157     if (root == NULL) {
158         root = new_node(data);
159         return;
160     }
161     tnode *parent = NULL;
162     tnode *current = root;
163     while (current != NULL) {
164         parent = current;
165         if (current->data < data) {
166             current = current->right;
167         } else {
168             current = current->left;
169         }
170     }
171     current->parent = parent;
172     if (parent->data < data) {
173         parent->right = new_node(data);
174     } else {
175         parent->left = new_node(data);
176     }
177 }
178
179 // Fungsi untuk menghapus node dari pohon
180 void delete_node(tnode *root, int data) {
181     if (root == NULL) return;
182     tnode *parent = NULL;
183     tnode *current = root;
184     while (current != NULL) {
185         parent = current;
186         if (current->data == data) {
187             if (current->left == NULL) {
188                 parent->right = current->right;
189             } else if (current->right == NULL) {
190                 parent->left = current->left;
191             } else {
192                 tnode *successor = find_node(current->right, current->left->data);
193                 successor->parent = parent;
194                 parent->right = successor;
195                 current->right = successor->right;
196             }
197             return;
198         }
199         if (current->data < data) {
200             current = current->right;
201         } else {
202             current = current->left;
203         }
204     }
205 }
```

-Struct pohon

Struktur ini di gunakan untuk mendefinisikan node dari pohon biner.

Data:menyimpan data berupa karakter

Left:pointer ke anak kiri

Right:pointer ke anak kanan

Parent:pointer ke induk node

-Variabel global

Root:menyimpan akar pohon(node pertama)

Baru:Variabel sementara untuk membuat node baru.

Init():

Inisialisasi pohon dengan mengosongkan root

Iseempty():

Mengecek apakah pohon kosong

buatNode():

membuat node baru dengan data tertentu.

insertLeft() dan insertRight():

menambahkan node baru sebagai anak kiri atau kanan dari node tertentu

mengecek apakah anak kiri atau kanan sudah ada sebelum menambahkan

updateChar():

mengubah data dalam node tertentu.

-Traversal:

preOrder():

traversal dalam urutan Root-> Left -> Right.

inOrder():

traversal dalam urutan Left->Root->Right.

postOrder()

traversal dalam urutan Left->Right->Root.

-deleteNode()

Menghapus Node berdasarkan aturan.

-mostLeft() dan mostRight():

Mencari node paling kiri(nilai terkecil)atau paling kanan(nilai terbesar) dalam pohon.

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

PS D:\struktur data pemograman\GUIDED9\output> █

4. Unguided

```

1  #include <iostream>
2  #include <iomanip>
3  using namespace std;
4
5  struct Node {
6      char* data;
7      Node* left, *right, *parent;
8  };
9
10 Node* root;
11
12 void init() {
13     root = NULL;
14 }
15
16 Node* insertNode(char* data, Node* parent = NULL) {
17     return new Node(data, NULL, NULL, parent);
18 }
19
20 Node* insertLeft(char* data, Node* node) {
21     if (node->left) {
22         cout << "Node " << node->left->data << " sudah ada child kiri!" << endl;
23         return NULL;
24     }
25     node->left = insertNode(data, node);
26     return node->left;
27 }
28
29 Node* insertRight(char* data, Node* node) {
30     if (node->right) {
31         cout << "Node " << node->right->data << " sudah ada child kanan!" << endl;
32         return NULL;
33     }
34     node->right = insertNode(data, node);
35     return node->right;
36 }
37
38 void swapLeftRight(Node* node) {
39     if (node) {
40         swap << "Node swap left-right!" << endl;
41         Node* left = node->left;
42         Node* right = node->right;
43         cout << "Left child << node->left->data << ",
44             if (node->left) {
45                 if (node->left->left) cout << " " << left->left->data << " (Kiri)";
46                 if (node->left->right) cout << " " << node->left->right->data << " (kanan)";
47             }
48             if (node->right) cout << " " << right->data << " (kanan)";
49             cout << endl;
50     }
51     node->left = right;
52     node->right = left;
53 }
54
55 void tampilInorder(Node* node) {
56     if (node) tampil();
57     if (node->left) tampilInorder(node->left);
58     if (node->right) tampilInorder(node->right);
59 }
60
61 void tampilPreorder(Node* node) {
62     if (node) tampil();
63     if (node->left) tampilPreorder(node->left);
64     if (node->right) tampilPreorder(node->right);
65 }
66
67 void tampilPostorder(Node* node) {
68     if (node) tampil();
69     if (node->left) tampilPostorder(node->left);
70     if (node->right) tampilPostorder(node->right);
71 }
72
73 void tampilLevel(Node* node, int level) {
74     if (node) tampilLevel(node->left, level + 1);
75     if (node) tampilLevel(node->right, level + 1);
76     if (node) tampilLevel(node, level);
77 }
78
79 void tampilLevel(Node* node, int level) {
80     if (node) tampilLevel(node->left, level + 1);
81     if (node) tampilLevel(node->right, level + 1);
82     if (node) tampilLevel(node, level);
83 }
84
85 void tampilLevel(Node* node, int level) {
86     if (node) tampilLevel(node->left, level + 1);
87     if (node) tampilLevel(node->right, level + 1);
88     if (node) tampilLevel(node, level);
89 }
90
91 void tampilLevel(Node* node, int level) {
92     if (node) tampilLevel(node->left, level + 1);
93     if (node) tampilLevel(node->right, level + 1);
94     if (node) tampilLevel(node, level);
95 }
96
97 void tampilLevel(Node* node, int level) {
98     if (node) tampilLevel(node->left, level + 1);
99     if (node) tampilLevel(node->right, level + 1);
100    if (node) tampilLevel(node, level);
101 }
102
103 void tampilLevel(Node* node, int level) {
104     if (node) tampilLevel(node->left, level + 1);
105     if (node) tampilLevel(node->right, level + 1);
106     if (node) tampilLevel(node, level);
107 }
108
109 void tampilLevel(Node* node, int level) {
110     if (node) tampilLevel(node->left, level + 1);
111     if (node) tampilLevel(node->right, level + 1);
112     if (node) tampilLevel(node, level);
113 }
114
115 void tampilLevel(Node* node, int level) {
116     if (node) tampilLevel(node->left, level + 1);
117     if (node) tampilLevel(node->right, level + 1);
118     if (node) tampilLevel(node, level);
119 }
120
121 void tampilLevel(Node* node, int level) {
122     if (node) tampilLevel(node->left, level + 1);
123     if (node) tampilLevel(node->right, level + 1);
124     if (node) tampilLevel(node, level);
125 }
126
127 void tampilLevel(Node* node, int level) {
128     if (node) tampilLevel(node->left, level + 1);
129     if (node) tampilLevel(node->right, level + 1);
130     if (node) tampilLevel(node, level);
131 }
132
133 void tampilLevel(Node* node, int level) {
134     if (node) tampilLevel(node->left, level + 1);
135     if (node) tampilLevel(node->right, level + 1);
136     if (node) tampilLevel(node, level);
137 }
138
139 void tampilLevel(Node* node, int level) {
140     if (node) tampilLevel(node->left, level + 1);
141     if (node) tampilLevel(node->right, level + 1);
142     if (node) tampilLevel(node, level);
143 }
144
145 void tampilLevel(Node* node, int level) {
146     if (node) tampilLevel(node->left, level + 1);
147     if (node) tampilLevel(node->right, level + 1);
148     if (node) tampilLevel(node, level);
149 }
150
151 void tampilLevel(Node* node, int level) {
152     if (node) tampilLevel(node->left, level + 1);
153     if (node) tampilLevel(node->right, level + 1);
154     if (node) tampilLevel(node, level);
155 }
156
157 void tampilLevel(Node* node, int level) {
158     if (node) tampilLevel(node->left, level + 1);
159     if (node) tampilLevel(node->right, level + 1);
160     if (node) tampilLevel(node, level);
161 }
162
163 void tampilLevel(Node* node, int level) {
164     if (node) tampilLevel(node->left, level + 1);
165     if (node) tampilLevel(node->right, level + 1);
166     if (node) tampilLevel(node, level);
167 }
168
169 void tampilLevel(Node* node, int level) {
170     if (node) tampilLevel(node->left, level + 1);
171     if (node) tampilLevel(node->right, level + 1);
172     if (node) tampilLevel(node, level);
173 }
174
175 void tampilLevel(Node* node, int level) {
176     if (node) tampilLevel(node->left, level + 1);
177     if (node) tampilLevel(node->right, level + 1);
178     if (node) tampilLevel(node, level);
179 }
180
181 void tampilLevel(Node* node, int level) {
182     if (node) tampilLevel(node->left, level + 1);
183     if (node) tampilLevel(node->right, level + 1);
184     if (node) tampilLevel(node, level);
185 }
186
187 void tampilLevel(Node* node, int level) {
188     if (node) tampilLevel(node->left, level + 1);
189     if (node) tampilLevel(node->right, level + 1);
190     if (node) tampilLevel(node, level);
191 }
192
193 void tampilLevel(Node* node, int level) {
194     if (node) tampilLevel(node->left, level + 1);
195     if (node) tampilLevel(node->right, level + 1);
196     if (node) tampilLevel(node, level);
197 }
198
199 void tampilLevel(Node* node, int level) {
200     if (node) tampilLevel(node->left, level + 1);
201     if (node) tampilLevel(node->right, level + 1);
202     if (node) tampilLevel(node, level);
203 }
204
205 void tampilLevel(Node* node, int level) {
206     if (node) tampilLevel(node->left, level + 1);
207     if (node) tampilLevel(node->right, level + 1);
208     if (node) tampilLevel(node, level);
209 }
210
211 void tampilLevel(Node* node, int level) {
212     if (node) tampilLevel(node->left, level + 1);
213     if (node) tampilLevel(node->right, level + 1);
214     if (node) tampilLevel(node, level);
215 }
216
217 void tampilLevel(Node* node, int level) {
218     if (node) tampilLevel(node->left, level + 1);
219     if (node) tampilLevel(node->right, level + 1);
220     if (node) tampilLevel(node, level);
221 }
222
223 void tampilLevel(Node* node, int level) {
224     if (node) tampilLevel(node->left, level + 1);
225     if (node) tampilLevel(node->right, level + 1);
226     if (node) tampilLevel(node, level);
227 }
228
229 void tampilLevel(Node* node, int level) {
230     if (node) tampilLevel(node->left, level + 1);
231     if (node) tampilLevel(node->right, level + 1);
232     if (node) tampilLevel(node, level);
233 }
234
235 void tampilLevel(Node* node, int level) {
236     if (node) tampilLevel(node->left, level + 1);
237     if (node) tampilLevel(node->right, level + 1);
238     if (node) tampilLevel(node, level);
239 }
240
241 void tampilLevel(Node* node, int level) {
242     if (node) tampilLevel(node->left, level + 1);
243     if (node) tampilLevel(node->right, level + 1);
244     if (node) tampilLevel(node, level);
245 }
246
247 void tampilLevel(Node* node, int level) {
248     if (node) tampilLevel(node->left, level + 1);
249     if (node) tampilLevel(node->right, level + 1);
250     if (node) tampilLevel(node, level);
251 }
252
253 void tampilLevel(Node* node, int level) {
254     if (node) tampilLevel(node->left, level + 1);
255     if (node) tampilLevel(node->right, level + 1);
256     if (node) tampilLevel(node, level);
257 }
258
259 void tampilLevel(Node* node, int level) {
260     if (node) tampilLevel(node->left, level + 1);
261     if (node) tampilLevel(node->right, level + 1);
262     if (node) tampilLevel(node, level);
263 }
264
265 void tampilLevel(Node* node, int level) {
266     if (node) tampilLevel(node->left, level + 1);
267     if (node) tampilLevel(node->right, level + 1);
268     if (node) tampilLevel(node, level);
269 }
270
271 void tampilLevel(Node* node, int level) {
272     if (node) tampilLevel(node->left, level + 1);
273     if (node) tampilLevel(node->right, level + 1);
274     if (node) tampilLevel(node, level);
275 }
276
277 void tampilLevel(Node* node, int level) {
278     if (node) tampilLevel(node->left, level + 1);
279     if (node) tampilLevel(node->right, level + 1);
280     if (node) tampilLevel(node, level);
281 }
282
283 void tampilLevel(Node* node, int level) {
284     if (node) tampilLevel(node->left, level + 1);
285     if (node) tampilLevel(node->right, level + 1);
286     if (node) tampilLevel(node, level);
287 }
288
289 void tampilLevel(Node* node, int level) {
290     if (node) tampilLevel(node->left, level + 1);
291     if (node) tampilLevel(node->right, level + 1);
292     if (node) tampilLevel(node, level);
293 }
294
295 void tampilLevel(Node* node, int level) {
296     if (node) tampilLevel(node->left, level + 1);
297     if (node) tampilLevel(node->right, level + 1);
298     if (node) tampilLevel(node, level);
299 }
300
301 void tampilLevel(Node* node, int level) {
302     if (node) tampilLevel(node->left, level + 1);
303     if (node) tampilLevel(node->right, level + 1);
304     if (node) tampilLevel(node, level);
305 }
306
307 void tampilLevel(Node* node, int level) {
308     if (node) tampilLevel(node->left, level + 1);
309
```

Fungsi tampilChild dan tampilDescendant menampilkan anak langsung dan keturunan dari node tertentu.

is_valid_bst memeriksa apakah pohon memenuhi sifat BST.

cari_simpul_daun menghitung jumlah simpul daun menggunakan rekursi.

Output:

```
MENU TREE:
1. Buat root
2. Tambah child kiri
3. Tambah child kanan
4. Tampilkan child
5. Tampilkan descendant
6. Cek apakah BST
7. Hitung jumlah simpul daun
0. Keluar
Pilih: 1
Masukkan data root: F
Root dengan data F berhasil dibuat.
```

```
MENU TREE:
1. Buat root
2. Tambah child kiri
3. Tambah child kanan
4. Tampilkan child
5. Tampilkan descendant
6. Cek apakah BST
7. Hitung jumlah simpul daun
0. Keluar
Pilih: 2
Masukkan data parent: F
Masukkan data anak kiri: B
```

```
MENU TREE:
1. Buat root
2. Tambah child kiri
3. Tambah child kanan
4. Tampilkan child
5. Tampilkan descendant
6. Cek apakah BST
7. Hitung jumlah simpul daun
0. Keluar
Pilih: 3
Masukkan data parent: F
Masukkan data anak kanan: G
```

```
MENU TREE:
1. Buat root
2. Tambah child kiri
3. Tambah child kanan
4. Tampilkan child
5. Tampilkan descendant
6. Cek apakah BST
7. Hitung jumlah simpul daun
0. Keluar
Pilih: 4
Masukkan data node: F
Child dari F: B (kiri) G (kanan)
```

```
MENU TREE:
1. Buat root
2. Tambah child kiri
3. Tambah child kanan
4. Tampilkan child
5. Tampilkan descendant
6. Cek apakah BST
7. Hitung jumlah simpul daun
0. Keluar
Pilih: 5
Masukkan data node: F
Descendant dari F: B G
```

```
MENU TREE:
1. Buat root
2. Tambah child kiri
3. Tambah child kanan
4. Tampilkan child
5. Tampilkan descendant
6. Cek apakah BST
7. Hitung jumlah simpul daun
0. Keluar
Pilih: 6
Pohon adalah Binary Search Tree (BST).
```

```
MENU TREE:
1. Buat root
2. Tambah child kiri
3. Tambah child kanan
4. Tampilkan child
5. Tampilkan descendant
6. Cek apakah BST
7. Hitung jumlah simpul daun
0. Keluar
Pilih: 7
Jumlah simpul daun: 2
```


5. Kesimpulan

Pada praktikum ini, berhasil mengimplementasikan berbagai operasi dasar pada struktur data pohon, khususnya pada Binary Search Tree (BST). Dalam implementasi ini, kami mempelajari cara menambahkan, menghapus, serta melakukan traversal pada pohon dengan metode pre-order, in-order, dan post-order. Selain itu, kami juga mengimplementasikan fungsi untuk mencari node paling kiri dan paling kanan dalam pohon, yang mewakili nilai terkecil dan terbesar pada BST. Dengan memahami operasi dasar seperti ini, kami dapat menerapkan konsep tree pada berbagai masalah pemrograman yang memerlukan penyimpanan data secara hierarkis. Secara keseluruhan, penggunaan struktur data tree, terutama BST, sangat berguna untuk masalah yang melibatkan pencarian data yang cepat dan efisien.