LAPORAN PRAKTIKUM

Modul 9

Tree

Disusun Oleh:

Yogi Hafidh Maulana - 2211104061

SE06-02

Dosen :

Wahyu Andi

PROGRAM STUDI S1 SOFTWARE ENGINEERING

FAKULTAS INFORMATIKA

TELKOM UNIVERSITY

PURWOKERTO

2024

A. Tujuan

- Memahami definisi pohon (tree) sebagai struktur data hierarkis yang terdiri dari simpul (node) yang saling terhubung melalui cabang (edge).
- Memahami komponen dasar dalam tree, seperti *root*, *parent*, *child*, *leaf*, dan *subtree*.
- Mengimplementasikan dan memahami operasi dasar pada tree.
- Menjelaskan dan menerapkan penggunaan tree dalam berbagai aplikasi praktis.

B. Landasan Teori

Tree (Pohon) merupakan salah satu struktur data non-linear yang digunakan untuk menyimpan data secara hierarkis. Sebuah tree terdiri dari simpul (node) yang terhubung satu sama lain melalui cabang (edge). Struktur ini biasanya digunakan untuk merepresentasikan hubungan yang terstruktur, seperti organisasi, file system, atau pohon keputusan. Dalam pohon, terdapat satu simpul yang disebut sebagai *root* (akar), yang menjadi titik awal untuk menjelajahi seluruh tree. Setiap node dapat memiliki satu atau lebih simpul anak, yang dikenal dengan istilah *child*. Node yang tidak memiliki anak disebut *leaf* atau simpul daun. Selain itu, terdapat juga hubungan antar node yang disebut *edge*, yang menghubungkan satu node dengan node lainnya.

Terdapat beberapa jenis pohon dalam struktur data, di antaranya adalah Binary Tree (Pohon Biner), yang setiap node-nya memiliki maksimal dua anak, yaitu anak kiri dan anak kanan. Binary Search Tree (BST) adalah turunan dari Binary Tree yang memiliki aturan khusus di mana nilai pada node kiri selalu lebih kecil dari node induk, dan nilai pada node kanan lebih besar. Hal ini memungkinkan pencarian data yang lebih efisien. AVL Tree adalah jenis pohon biner terurut yang seimbang, di mana perbedaan kedalaman antara subpohon kiri dan kanan dibatasi untuk meningkatkan efisiensi pencarian. Selain itu, ada pula Heap, yang merupakan pohon biner dengan properti tertentu, yaitu nilai pada setiap node anak lebih kecil atau lebih besar dari node induknya, tergantung pada tipe heap (min-heap atau max-heap). Trie digunakan untuk merepresentasikan data berupa string dan sangat efisien dalam pencarian kata, misalnya dalam sistem autocomplete.

Operasi dasar pada tree mencakup traversal, insertion, deletion, dan searching. Traversal adalah proses mengunjungi setiap node dalam tree dengan menggunakan metode tertentu seperti pre-order, in-order, post-order, atau level-order. Insertion adalah operasi untuk menyisipkan node baru, yang dilakukan dengan mengikuti aturan tertentu tergantung pada jenis pohon yang digunakan. Deletion atau penghapusan node dilakukan dengan mempertimbangkan beberapa skenario, misalnya jika node yang akan dihapus memiliki dua anak atau tidak. Searching digunakan untuk mencari elemen tertentu dalam tree, dengan memanfaatkan aturan yang ada pada jenis tree yang digunakan.

C. Guided

Code

Guided.cpp > ...

```cpp
#include <iostream>
using namespace std;

struct Pohon
{
    char data;
    Pohon *left, *right;
    Pohon *parent;
};

Pohon *root, *baru;

Tabnine | Edit | Test | Explain | Document | Ask
void init()
{
    root = NULL;
}

bool isEmpty()
{
    return root == NULL;
}

Tabnine | Edit | Test | Explain | Document | Ask
void buatNode(char data)
{
    if (isEmpty())
    {
        root = new Pohon{data, NULL, NULL, NULL};
        cout << "\nNode " << data << " berhasil dibuat menjadi root" << endl;
    }
    else
    {
        cout << "\nPohon sudah dibuat" << endl;
    }
}
```

```
Pohon *insertLeft(char data, Pohon *node)
{
    if (node->left != NULL)
    {
        cout << "\nNode " << node->data << " sudah ada child kiri" << endl;
        return NULL;
    }
    baru = new Pohon{data, NULL, NULL, node};
    node->left = baru;
    cout << "\nNode " << data << " berhasil ditambahkan ke child kiri dari "
    << node->data << endl;

    return baru;
}
```

```
Pohon *insertRight(char data, Pohon *node)
{
    if (node->right != NULL)
    {
        cout << "\nNode " << node->data << " sudah ada child kanan" << endl;
        return NULL;
    }
    baru = new Pohon{data, NULL, NULL, node};
    node->right = baru;
    cout << "\nNode " << data << " berhasil ditambahkan ke child kanan dari "
        << node->data << endl;
    return baru;
}
```

```
void update(char data, Pohon *node)
{
    if (!node)
    {
        cout << "\nNode yang ingin diubah tidak ditemukan" << endl;
        return;
    }
    char temp = node->data;
    node->data = data;
    cout << "\nNode " << temp << " berhasil diubah menjadi " << data << endl;
}
```

```
77  void find(char data, Pohon *node)
78  {
79      if (!node)
80          return;
81
82      if (node->data == data)
83      {
84          cout << "\nNode ditemukan: " << data << endl;
85          return;
86      }
87
88      find(data, node->left);
89      find(data, node->right);
90  }
91
```

```
92  void preOrder(Pohon *node)
93  {
94      if (!node)
95          return;
96      cout << node->data << " ";
97      preOrder(node->left);
98      preOrder(node->right);
99  }
100
    Tabnine | Edit | Test | Explain | Document | Ask
101 void inOrder(Pohon *node)
102 {
103     if (!node)
104         return;
105     inOrder(node->left);
106     cout << node->data << " ";
107     inOrder(node->right);
108 }
```

```
110  void postOrder(Pohon *node)
111  {
112      if (!node)
113          return;
114      postOrder(node->left);
115      postOrder(node->right);
116      cout << node->data << " ";
117  }
118
```

```
Pohon *deleteNode(Pohon *node, char data)
{
    if (!node)
        return NULL;

    if (data < node->data)
    {
        node->left = deleteNode(node->left, data);
    }
    else if (data > node->data)
    {
        node->right = deleteNode(node->right, data);
    }
    else
    {
        if (!node->left)
        {
            Pohon *temp = node->right;
            delete node;
            return temp;
        }
        else if (!node->right)
        {
            Pohon *temp = node->left;
            delete node;
            return temp;
        }

        Pohon *successor = node->right;
        while (successor->left)
            successor = successor->left;
        node->data = successor->data;
        node->right = deleteNode(node->right, successor->data);
    }
    return node;
}
```

```
Pohon *mostLeft(Pohon *node)
{
    if (!node)
        return NULL;
    while (node->left)
        node = node->left;
    return node;
}

Tabnine | Edit | Test | Explain | Document | Ask
Pohon *mostRight(Pohon *node)
{
    if (!node)
        return NULL;
    while (node->right)
        node = node->right;
    return node;
}
```

```cpp
int main()
{
    init();
    buatNode('F');
    insertLeft('B', root);
    insertRight('G', root);
    insertLeft('A', root->left);
    insertRight('D', root->left);
    insertLeft('C', root->left->right);
    insertLeft('E', root->left->right);

    cout << "Pre-order traversal: ";
    preOrder(root);
    cout << "\nIn-order traversal: ";
    inOrder(root);
    cout << "\nPost-order traversal: ";
    postOrder(root);

    cout << "\nMost Left Node: " << mostLeft(root)->data;
    cout << "\nMost Right Node: " << mostRight(root)->data;

    cout << "\nMenghapus node D.";
    root = deleteNode(root, 'D');
    cout << "\nIn-order traversal setelah penghapusan: ";
    inOrder(root);

    return 0;
}
```

Output

```
Node F berhasil dibuat menjadi root

Node B berhasil ditambahkan ke child kiri dari F

Node G berhasil ditambahkan ke child kanan dari F

Node A berhasil ditambahkan ke child kiri dari B

Node D berhasil ditambahkan ke child kanan dari B

Node C berhasil ditambahkan ke child kiri dari D

Node D sudah ada child kiri
Pre-order traversal: F B A D C G
In-order traversal: A B C D F G
Post-order traversal: A C D B G F
Most Left Node: A
Most Right Node: G
Menghapus node D.
In-order traversal setelah penghapusan: A B C F G
PS D:\PROJECT\C++ Project\pertemuan9>
```

Deskripsi Program

Kode di atas mengimplementasikan struktur data binary tree yang terdiri dari node yang memiliki dua anak, yaitu kiri dan kanan. Dimulai dengan mendeklarasikan struktur data Pohon, yang memiliki data, pointer ke anak kiri dan kanan, serta pointer ke parent (induk). Fungsi utama yang ada dalam kode ini adalah untuk membuat pohon, menambah node anak kiri dan kanan, melakukan traversal pohon dengan metode pre-order, in-order, dan post-order, serta menghapus node berdasarkan nilai tertentu. Proses pembuatan pohon dimulai dengan membuat node root, kemudian menambahkan node kiri dan kanan sesuai dengan struktur pohon biner. Fungsi traversal digunakan untuk mengunjungi setiap node dalam pohon dengan urutan tertentu. Fungsi deleteNode digunakan untuk menghapus node tertentu dengan mempertimbangkan tiga kondisi: jika node tersebut memiliki satu anak atau tidak memiliki anak sama sekali, maka node tersebut dapat dihapus dengan menggantinya dengan anaknya yang ada; jika node memiliki dua anak, maka node tersebut akan digantikan dengan node pengganti yang paling kiri pada subpohon kanan. Algoritma ini secara umum menunjukkan cara kerja struktur data pohon biner yang memungkinkan pencarian, penambahan, penghapusan, dan traversal data secara efisien.

D. Unguided

Code

Unguided.cpp > init()

```cpp
#include <iostream>
#include <queue>
using namespace std;

struct Pohon
{
    char data;
    Pohon *left, *right;
    Pohon *parent;
};

```

```
12  Pohon *root, *baru;
13
    Tabnine | Edit | Test | Explain | Document | Ask
14  void init()
15  {
16      root = NULL;
17  }
18
    Tabnine | Edit | Test | Explain | Document | Ask
19  bool isEmpty()
20  {
21      return root == NULL;
22  }
24  void buatNode(char data)
25  {
26      if (isEmpty())
27      {
28          root = new Pohon{data, NULL, NULL, NULL};
29          cout << "\nNode " << data << " berhasil dibuat menjadi root" << endl;
30      }
31      else
32      {
33          cout << "\nPohon sudah dibuat" << endl;
34      }
35  }
37  Pohon *insertLeft(char data, Pohon *node)
38  {
39      if (node->left != NULL)
40      {
41          cout << "\nNode " << node->data << " sudah ada child kiri" << endl;
42          return NULL;
43      }
44      baru = new Pohon{data, NULL, NULL, node};
45      node->left = baru;
46      cout << "\nNode " << data << " berhasil ditambahkan ke child kiri dari "
47          << node->data << endl;
48      return baru;
49  }
```

```
Pohon *insertRight(char data, Pohon *node)
{
    if (node->right != NULL)
    {
        cout << "\nNode " << node->data << " sudah ada child kanan" << endl;
        return NULL;
    }
    baru = new Pohon{data, NULL, NULL, node};
    node->right = baru;
    cout << "\nNode " << data << " berhasil ditambahkan ke child kanan dari "
         << node->data << endl;
    return baru;
}
void update(char data, Pohon *node)
{
    if (!node)
    {
        cout << "\nNode yang ingin diubah tidak ditemukan" << endl;
        return;
    }
    char temp = node->data;
    node->data = data;
    cout << "\nNode " << temp << " berhasil diubah menjadi " << data << endl;
}
void find(char data, Pohon *node, Pohon *&result)
{
    if (!node)
        return;

    if (node->data == data)
    {
        result = node;
        return;
    }

    find(data, node->left, result);
    find(data, node->right, result);
}
```

```
92  void preOrder(Pohon *node)
93  {
94      if (!node)
95          return;
96      cout << node->data << " ";
97      preOrder(node->left);
98      preOrder(node->right);
99  }
100
101 void inOrder(Pohon *node)
102 {
103     if (!node)
104         return;
105     inOrder(node->left);
106     cout << node->data << " ";
107     inOrder(node->right);
108 }
110 void postOrder(Pohon *node)
111 {
112     if (!node)
113         return;
114     postOrder(node->left);
115     postOrder(node->right);
116     cout << node->data << " ";
117 }
119 Pohon *deleteNode(Pohon *node, char data)
120 {
121     if (!node)
122         return NULL;
123
124     if (data < node->data)
125     {
126         node->left = deleteNode(node->left, data);
127     }
128     else if (data > node->data)
129     {
130         node->right = deleteNode(node->right, data);
131     }
```

```
    else
    {
        if (!node->left)
        {
            Pohon *temp = node->right;
            delete node;
            return temp;
        }
        else if (!node->right)
        {
            Pohon *temp = node->left;
            delete node;
            return temp;
        }
        Pohon *successor = node->right;
        while (successor->left)
            successor = successor->left;
        node->data = successor->data;
        node->right = deleteNode(node->right, successor->data);
    }
    return node;
}
Pohon *mostLeft(Pohon *node)
{
    if (!node)
        return NULL;
    while (node->left)
        node = node->left;
    return node;
}
Pohon *mostRight(Pohon *node)
{
    if (!node)
        return NULL;
    while (node->right)
        node = node->right;
    return node;
}
```

```
void showChild(Pohon *node)
{
    if (!node)
    {
        cout << "Node tidak ditemukan!" << endl;
        return;
    }
    cout << "Child kiri dari node " << node->data << ": ";
    if (node->left)
        cout << node->left->data;
    else
        cout << "Tidak ada";
    cout << "\nChild kanan dari node " << node->data << ": ";
    if (node->right)
        cout << node->right->data;
    else
        cout << "Tidak ada";
    cout << endl;
}
int cari_simpul_daun(Pohon *node)
{
    if (node == NULL)
        return 0;

    if (node->left == NULL && node->right == NULL)
        return 1;

    return cari_simpul_daun(node->left) + cari_simpul_daun(node->right);
}
void showDescendants(Pohon *node)
{
    if (!node)
    {
        cout << "Node tidak ditemukan!" << endl;
        return;
    }

    cout << "Descendants dari node " << node->data << ": ";
    queue<Pohon *> q;
    if (node->left)
        q.push(node->left);
    if (node->right)
        q.push(node->right);

```

```
    while (!q.empty())
    {
        Pohon *current = q.front();
        q.pop();
        cout << current->data << " ";
        if (current->left)
            q.push(current->left);
        if (current->right)
            q.push(current->right);
    }
    cout << endl;
}
bool is_valid_bst(Pohon *node, char min_val, char max_val)
{
    if (node == NULL)
        return true;

    if (node->data <= min_val || node->data >= max_val)
        return false;

    return is_valid_bst(node->left, min_val, node->data) &&
           is_valid_bst(node->right, node->data, max_val);
}
bool check_bst()
{
    return is_valid_bst(root, CHAR_MIN, CHAR_MAX);
}
```

```
250  int main()
251  {
252      init();
253      int choice;
254      char data, parentData;
255
256      while (true)
257      {
258          cout << "\nMenu:\n";
259          cout << "1. Buat root\n";
260          cout << "2. Tambahkan child kiri\n";
261          cout << "3. Tambahkan child kanan\n";
262          cout << "4. Tampilkan pre-order traversal\n";
263          cout << "5. Tampilkan in-order traversal\n";
264          cout << "6. Tampilkan post-order traversal\n";
265          cout << "7. Tampilkan child dari node\n";
266          cout << "8. Tampilkan descendant dari node\n";
267          cout << "9. Hapus node\n";
268          cout << "10. Periksa apakah BST valid\n";
269          cout << "11. Tampilkan jumlah simpul daun\n";
270          cout << "12. Keluar\n";
271          cout << "Masukkan pilihan (1-12): ";
272          cin >> choice;
274          switch (choice)
275          {
276          case 1:
277              cout << "Masukkan data root: ";
278              cin >> data;
279              buatNode(data);
280              break;
281          case 2:
282              cout << "Masukkan data node parent dan child kiri: ";
283              cin >> parentData >> data;
284              Pohon *parentNode = NULL;
285              find(parentData, root, parentNode);
286              if (parentNode)
287              {
288                  insertLeft(data, parentNode);
289              }
290              else
291              {
292                  cout << "Node parent tidak ditemukan!" << endl;
293              }
294              break;
```

```
295         case 3:
296             cout << "Masukkan data node parent dan child kanan: ";
297             cin >> parentData >> data;
298             find(parentData, root, parentNode);
299             if (parentNode)
300             {
301                 insertRight(data, parentNode);
302             }
303             else
304             {
305                 cout << "Node parent tidak ditemukan!" << endl;
306             }
307             break;
308         case 4:
309             cout << "Pre-order traversal: ";
310             preOrder(root);
311             cout << endl;
312             break;
313         case 5:
314             cout << "In-order traversal: ";
315             inOrder(root);
316             cout << endl;
317             break;
318         case 6:
319             cout << "Post-order traversal: ";
320             postOrder(root);
321             cout << endl;
322             break;
323         case 7:
324             cout << "Masukkan data node untuk tampilkan child: ";
325             cin >> data;
326             find(data, root, parentNode);
327             showChild(parentNode);
328             break;
329         case 8:
330             cout << "Masukkan data node untuk tampilkan descendant: ";
331             cin >> data;
332             find(data, root, parentNode);
333             showDescendants(parentNode);
334             break;
335         case 9:
336             cout << "Masukkan data node yang akan dihapus: ";
337             cin >> data;
338             deleteNode(root, data);
339             break;
```

```
            case 10:
                if (check_bst())
                    cout << "Pohon adalah BST valid." << endl;
                else
                    cout << "Pohon bukan BST valid." << endl;
                break;
            case 11:
                cout << "Jumlah simpul daun: " << cari_simpul_daun(root) << endl;
                break;
            case 12:
                return 0;
            default:
                cout << "Pilihan tidak valid!" << endl;
                break;
            }
        }

        return 0;
    }
```

Output

```
Menu:
1. Buat root
2. Tambahkan child kiri
3. Tambahkan child kanan
4. Tampilkan pre-order traversal
5. Tampilkan in-order traversal
6. Tampilkan post-order traversal
7. Tampilkan child dari node
8. Tampilkan descendant dari node
9. Hapus node
10. Keluar
Masukkan pilihan (1-10): 2
Masukkan data node parent dan child kiri: F
B

Node ditemukan: F

Node B berhasil ditambahkan ke child kiri dari F
```

```
Menu:
1. Buat root
2. Tambahkan child kiri
3. Tambahkan child kanan
4. Tampilkan pre-order traversal
5. Tampilkan in-order traversal
6. Tampilkan post-order traversal
7. Tampilkan child dari node
8. Tampilkan descendant dari node
9. Hapus node
10. Keluar
Masukkan pilihan (1-10): 3
Masukkan data node parent dan child kanan: F G

Node ditemukan: F

Node G berhasil ditambahkan ke child kanan dari F

Menu:
1. Buat root
2. Tambahkan child kiri
3. Tambahkan child kanan
4. Tampilkan pre-order traversal
5. Tampilkan in-order traversal
6. Tampilkan post-order traversal
7. Tampilkan child dari node
8. Tampilkan descendant dari node
9. Hapus node
10. Keluar
Masukkan pilihan (1-10): 4
Pre-order traversal: F B G
```

```
 Menu:
 1. Buat root
 2. Tambahkan child kiri
 3. Tambahkan child kanan
 4. Tampilkan pre-order traversal
 5. Tampilkan in-order traversal
 6. Tampilkan post-order traversal
 7. Tampilkan child dari node
 8. Tampilkan descendant dari node
 9. Hapus node
 10. Keluar
 Masukkan pilihan (1-10): 5
 In-order traversal: B F G

Menu:
1. Buat root
2. Tambahkan child kiri
3. Tambahkan child kanan
4. Tampilkan pre-order traversal
5. Tampilkan in-order traversal
6. Tampilkan post-order traversal
7. Tampilkan child dari node
8. Tampilkan descendant dari node
9. Hapus node
10. Keluar
Masukkan pilihan (1-10): 6
Post-order traversal: B G F
```

```
Menu:
1. Buat root
2. Tambahkan child kiri
3. Tambahkan child kanan
4. Tampilkan pre-order traversal
5. Tampilkan in-order traversal
6. Tampilkan post-order traversal
7. Tampilkan child dari node
8. Tampilkan descendant dari node
9. Hapus node
10. Keluar
Masukkan pilihan (1-10): 7
Masukkan data node untuk ditampilkan child-nya: F

Node ditemukan: F
Child kiri dari node F: B
Child kanan dari node F: G

Menu:
1. Buat root
2. Tambahkan child kiri
3. Tambahkan child kanan
4. Tampilkan pre-order traversal
5. Tampilkan in-order traversal
6. Tampilkan post-order traversal
7. Tampilkan child dari node
8. Tampilkan descendant dari node
9. Hapus node
10. Keluar
Masukkan pilihan (1-10): 8
Masukkan data node untuk ditampilkan descendant-nya: F

Node ditemukan: F
Descendants dari node F: B G
```

```
Menu:
1. Buat root
2. Tambahkan child kiri
3. Tambahkan child kanan
4. Tampilkan pre-order traversal
5. Tampilkan in-order traversal
6. Tampilkan post-order traversal
7. Tampilkan child dari node
8. Tampilkan descendant dari node
9. Hapus node
10. Keluar
Masukkan pilihan (1-10): 9
Masukkan data node untuk dihapus: B

Menu:
1. Buat root
2. Tambahkan child kiri
3. Tambahkan child kanan
4. Tampilkan pre-order traversal
5. Tampilkan in-order traversal
6. Tampilkan post-order traversal
7. Tampilkan child dari node
8. Tampilkan descendant dari node
9. Hapus node
10. Keluar
Masukkan pilihan (1-10): 10
Keluar...
PS D:\PROJECT\C++ Project\pertemuan9>
```

```
10. Cek apakah Pohon Valid sebagai BST
11. Hitung Jumlah Simpul Daun
12. Keluar
Masukkan pilihan: 10
Pohon ini bukan Binary Search Tree.
```

```
10. Cek apakah Pohon Valid sebagai BST
11. Hitung Jumlah Simpul Daun
12. Keluar
Masukkan pilihan: 11
Jumlah simpul daun: 2
```

Deskripsi Code

Program yang diberikan adalah implementasi dasar dari Binary Tree dalam bahasa C++. Dalam struktur data pohon biner ini, setiap node memiliki data, dan dua anak, yaitu anak kiri dan anak kanan. Pohon biner ini digunakan untuk menyimpan dan mengelola data secara hierarkis.

- Root dan Pembuatan Pohon: Program dimulai dengan membuat root pohon menggunakan fungsi buatNode. Fungsi ini hanya dapat membuat root jika pohon masih kosong. Root berfungsi sebagai titik awal pohon yang akan menampung semua node lainnya. Jika pohon sudah ada, maka root tidak akan dibuat lagi.
- Menambahkan Anak: Fungsi insertLeft dan insertRight digunakan untuk menambahkan node sebagai anak kiri atau kanan dari node yang sudah ada. Jika node yang dituju sudah memiliki anak di sisi yang ingin ditambahkan, maka node baru tidak akan ditambahkan.
- Traversal Pohon: Tiga jenis traversal pohon yang umum digunakan tersedia dalam program ini:
    1. Pre-order: Mengunjungi root terlebih dahulu, lalu anak kiri, dan anak kanan.
    2. In-order: Mengunjungi anak kiri, lalu root, dan anak kanan. Traversal ini sering digunakan untuk pohon pencarian biner karena akan menghasilkan urutan data yang terurut.
    3. Post-order: Mengunjungi anak kiri dan kanan terlebih dahulu, baru kemudian root.
- Pencarian dan Pembaruan Node: Fungsi find digunakan untuk mencari node tertentu di dalam pohon. Setelah node ditemukan, kita dapat melakukan pembaruan data pada node tersebut menggunakan fungsi update.
- Penghapusan Node: Fungsi deleteNode digunakan untuk menghapus node yang ada. Penghapusan node di pohon biner bisa dilakukan dalam tiga cara:
    1. Node tanpa anak (daun).
    2. Node dengan satu anak.
    3. Node dengan dua anak, di mana kita harus mencari pengganti node tersebut (biasanya node dengan nilai terkecil di anak kanan atau terbesar di anak kiri).
- Cek Validitas Binary Search Tree (BST): Fungsi is_valid_bst digunakan untuk memastikan apakah pohon tersebut adalah BST yang valid. BST adalah pohon biner di mana nilai anak kiri selalu lebih kecil dari nilai root, dan anak kanan selalu lebih besar. Fungsi ini memeriksa setiap node untuk memastikan aturan ini diikuti.
- Menampilkan Anak dan Keturunan: Fungsi showChild menampilkan anak kiri dan kanan dari suatu node. Sementara fungsi showDescendants menampilkan semua keturunan dari suatu node, menggunakan teknik traversal level-order (breadth-

first).

- Menghitung Jumlah Simpul Daun: Fungsi cari_simpul_daun digunakan untuk menghitung jumlah node daun, yaitu node yang tidak memiliki anak.

Secara keseluruhan, program ini memberikan berbagai fungsi dasar yang dapat digunakan untuk bekerja dengan pohon biner, seperti menambah, menghapus, mencari node, dan melakukan traversal pohon. Program ini juga memeriksa apakah pohon tersebut memenuhi aturan BST dan menghitung jumlah simpul daun.

E. Kesimpulan

Tree (Pohon) adalah struktur data non-linear yang menyimpan data dalam bentuk hierarkis, di mana setiap node dapat memiliki satu atau lebih anak, dengan satu node sebagai akar (root). Struktur ini digunakan untuk merepresentasikan hubungan yang terorganisir seperti dalam organisasi, sistem file, dan algoritma pencarian. Berbagai jenis pohon, seperti Binary Tree**,** Binary Search Tree (BST)**,** AVL Tree**,** dan Trie, memiliki karakteristik dan kegunaan yang berbeda, memungkinkan aplikasi yang lebih efisien dalam berbagai konteks. Operasi dasar dalam pohon meliputi insertion**,** deletion**,** searching**,** dan traversal, yang dapat diimplementasikan dengan cara yang berbeda tergantung pada jenis pohon yang digunakan. Keuntungan utama dari tree adalah kemampuannya untuk menyimpan dan mengakses data secara terstruktur dengan waktu eksekusi yang efisien, terutama pada pohon terurut seperti BST yang memungkinkan pencarian cepat dengan kompleksitas O(log n).

Namun, efisiensi tree sangat dipengaruhi oleh keseimbangannya. Pohon yang tidak seimbang dapat menurunkan kinerja operasi pencarian, penyisipan, dan penghapusan. Oleh karena itu, teknik untuk menjaga keseimbangan, seperti pada AVL Tree atau Red-Black Tree, sangat penting untuk memastikan bahwa operasi tetap efisien. Tree banyak digunakan dalam aplikasi dunia nyata, seperti sistem file, struktur organisasi, dan algoritma pencarian teks, menjadikannya salah satu struktur data yang fundamental dalam ilmu komputer.