LAPORAN PRAKTIKUM
MODUL 9
TREE

Disusun Oleh:

Satria Putra Dharma Prayudha - 21104036

SE07-02

Dosen :

Wahyu Andi Saputra, S.Pd., M.Eng

PROGRAM STUDI S1 SOFTWARE ENGINEERING

FAKULTAS INFORMATIKA

TELKOM UNIVERSITY PURWOKERTO

2024

A. Tujuan

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

B. Landasan Teori

Landasan teori ini berdasarkan pada modul pembelajaran praktikum struktur data kali ini

2.1 Rekursif

- Pengertian Rekursif :

Rekursif adalah suatu proses di mana fungsi atau prosedur memanggil dirinya sendiri untuk menyelesaikan permasalahan. Konsep ini memungkinkan sebuah masalah besar dipecah menjadi masalah yang lebih kecil dengan pola yang sama, hingga akhirnya mencapai kondisi dasar (base case). Contohnya, dalam perhitungan faktorial atau traversal tree, rekursif dapat digunakan untuk menyelesaikan masalah dengan lebih ringkas.

Manfaat rekursif meliputi:

- Readability: Mempermudah pembacaan kode.
- Modularity: Memecah masalah kompleks menjadi bagian kecil.
- Reusability: Fungsi rekursif dapat digunakan berulang kali tanpa menulis ulang logika yang sama.

- Kriteria Rekursif

Agar sebuah fungsi dapat disebut rekursif, fungsi tersebut harus memenuhi dua kriteria:

- Kondisi Dasar (Base Case): Kondisi yang memastikan fungsi berhenti memanggil dirinya sendiri.
- Kondisi Rekursif (Recursive Case): Bagian dari fungsi yang memanggil dirinya sendiri dengan parameter yang lebih sederhana atau lebih kecil.

- Contoh: Fungsi pangkat dua:

```
int pangkat_2(int x) {
    if (x == 0) return 1;    // Base Case
    return 2 * pangkat_2(x - 1);  // Recursive Case
}
```

- Kekurangan Rekursif

Walaupun rekursif dapat menyederhanakan logika, terdapat beberapa kelemahan:

- Membutuhkan lebih banyak memori untuk menyimpan activation record setiap pemanggilan fungsi.
- Memerlukan waktu eksekusi tambahan untuk penanganan activation record.
- Tidak efisien dibandingkan metode iteratif untuk masalah tertentu.

- Contoh Rekursif:
  - Fungsi Faktorial:

```
int faktorial(int n) {
    if (n == 0 || n == 1) return 1;  // Base case
    return n * faktorial(n - 1);    // Recursive case
}
```

- Traversal Tree

```
void inOrder(Node* root) {
    if (root != nullptr) {
        inOrder(root->left);
        cout << root->key;
        inOrder(root->right);
    }
}
```

2.2 Tree

- Pengertian Tree

Tree adalah struktur data non-linear yang digunakan untuk merepresentasikan data dalam bentuk hierarki. Tree terdiri dari simpul-simpul (nodes) yang terhubung dengan hubungan parent-child. Tree memiliki sifat-sifat berikut:

- Akar (Root): Simpul tertinggi dalam tree yang tidak memiliki orang tua.
- Simpul Daun (Leaf): Simpul tanpa anak.
- Path (Lintasan): Urutan simpul dari akar ke simpul tertentu.
- Tinggi (Height): Panjang lintasan terpanjang dari akar ke daun.

- Jenis-Jenis Tree
  - Ordered Tree: Urutan anak-anak pada simpul penting.
  - Binary Tree: Setiap simpul memiliki maksimal dua anak:
  - Complete Binary Tree: Semua level, kecuali yang terakhir, penuh terisi.
  - Binary Search Tree (BST): Anak kiri lebih kecil dari orang tua, dan anak kanan lebih besar.
  - AVL Tree: Binary Search Tree yang menjaga perbedaan tinggi antara subtree kiri dan kanan tidak lebih dari 1.
  - Heap Tree: Tree dengan aturan:
    - Minimum Heap: Orang tua lebih kecil dari anak-anaknya.
    - Maximum Heap: Orang tua lebih besar dari anak-anaknya.

Operasi-Operasi dalam Binary Search Tree

1. **Insert (Penyisipan):**

Menambahkan elemen ke tree sesuai aturan BST:

- Jika nilai lebih kecil dari simpul orang tua, masukkan ke anak kiri.
- Jika nilai lebih besar, masukkan ke anak kanan.

2. **Delete (Penghapusan):**

   Penghapusan simpul dalam BST memiliki tiga kasus:

   - *Simpul Daun:* Hapus langsung.
   - *Simpul dengan satu anak:* Anak menggantikan posisi simpul yang dihapus.
   - *Simpul dengan dua anak:* Gunakan simpul terkecil di subtree kanan (*successor*) atau terbesar di subtree kiri (*predecessor*).

3. **Search (Pencarian):**

   Menggunakan algoritma *binary search* dengan membandingkan nilai target dengan nilai simpul, kemudian melanjutkan ke subtree kiri atau kanan berdasarkan hasil perbandingan.

Traversal pada Binary Tree

Traversal adalah proses mengunjungi semua simpul dalam tree:

- **Pre-order:** Akar → Anak Kiri → Anak Kanan.
- **In-order:** Anak Kiri → Akar → Anak Kanan.
- **Post-order:** Anak Kiri → Anak Kanan → Akar.

Operasi Tambahan dalam BST

1. **Most-Left Node:**

   Simpul terkecil dalam BST, ditemukan dengan mengikuti anak kiri dari akar hingga mencapai simpul tanpa anak kiri lagi.

2. **Most-Right Node:**

   Simpul terbesar dalam BST, ditemukan dengan mengikuti anak kanan dari akar hingga mencapai simpul tanpa anak kanan lagi.

C. Guided

a. Tree

Code :

Output :

```
PS D:\Kuliah\Struktur Data\Github\10_Tree_Bagian_1\21104036_Satria Putra Dharma Prayudha\Guided\output> & .\'guided.exe'

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:\Kuliah\Struktur Data\Github\10_Tree_Bagian_1\21104036_Satria Putra Dharma Prayudha\Guided\output>
```

Penjelasan : Pada program ini, beberapa node ditambahkan ke pohon biner menggunakan fungsi insertLeft dan insertRight. Node 'B' ditambahkan sebagai anak kiri root, 'G' sebagai anak kanan root, 'A' sebagai anak kiri dari 'B', 'D' sebagai anak kanan dari 'B', 'C' sebagai anak kiri dari 'D', dan 'E' sebagai anak kanan dari 'D'. Setelah penambahan node, dilakukan traversal pohon dengan tiga metode: pre-order, in-order, dan post-order, dan hasilnya ditampilkan ke layar. Pre-order traversal mengunjungi node dalam urutan root, left, right; in-order traversal mengunjungi node dalam urutan left, root, right; dan post-order traversal mengunjungi node dalam urutan left, right, root.

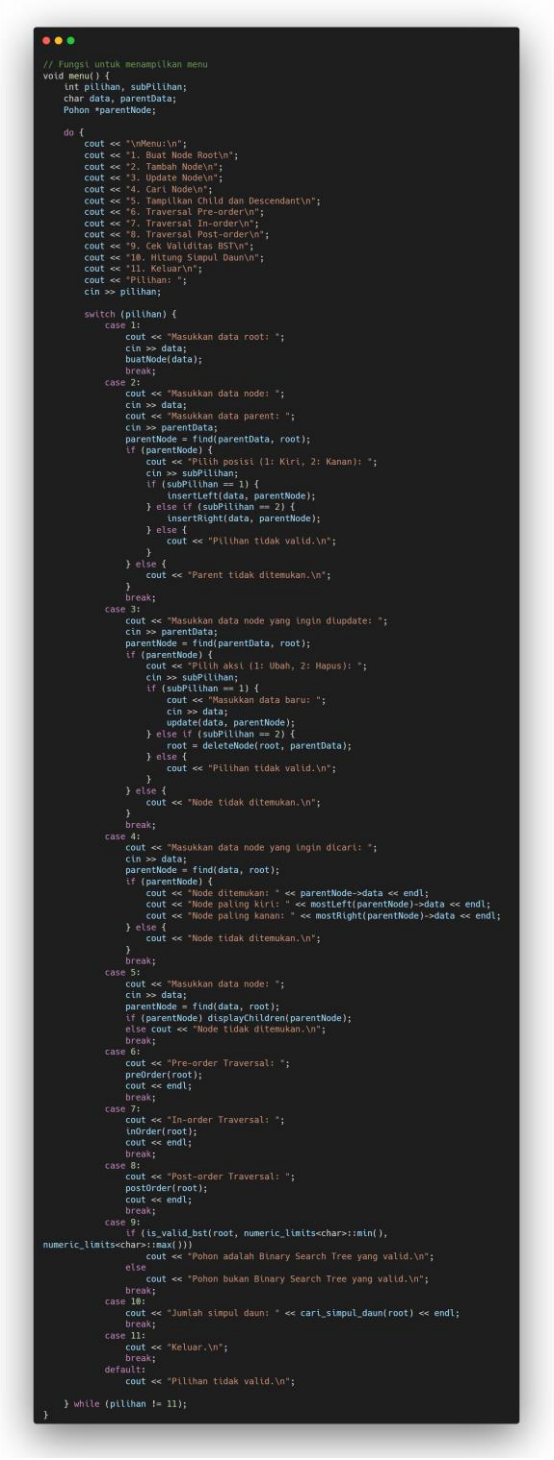
Program juga menampilkan node paling kiri dan paling kanan dari pohon menggunakan fungsi mostLeft dan mostRight. Data dari node-node ini ditampilkan ke layar. Selanjutnya, program menghapus node 'D' dari pohon menggunakan fungsi deleteNode. Setelah penghapusan, dilakukan kembali in-order traversal untuk menampilkan struktur pohon yang baru setelah penghapusan node 'D'. Hasil traversal ini juga ditampilkan ke layar, menunjukkan perubahan struktur pohon setelah penghapusan node.

D. Unguided

Kode Keseluruhan :

a. **Penambahan Menu Interaktif Menampilkan Node Child dan Descendant Dari Node**

Code:



```
// Fungsi untuk menampilkan menu
void menu() {
    int pilihan, subPilihan;
    char data, parentData;
    Pohon *parentNode;

    do {
        cout << "\nMenu:\n";
        cout << "1. Buat Node Root\n";
        cout << "2. Tambah Node\n";
        cout << "3. Update Node\n";
        cout << "4. Cari Node\n";
        cout << "5. Tampilkan Child dan Descendant\n";
        cout << "6. Traversal Pre-order\n";
        cout << "7. Traversal In-order\n";
        cout << "8. Traversal Post-order\n";
        cout << "9. Cek Validitas BST\n";
        cout << "10. Hitung Simpul Daun\n";
        cout << "11. Keluar\n";
        cout << "Pilihan: ";
        cin >> pilihan;

        switch (pilihan) {
            case 1:
                cout << "Masukkan data root: ";
                cin >> data;
                buatNode(data);
                break;
            case 2:
                cout << "Masukkan data node: ";
                cin >> data;
                cout << "Masukkan data parent: ";
                cin >> parentData;
                parentNode = find(parentData, root);
                if (parentNode) {
                    cout << "Pilih posisi (1: Kiri, 2: Kanan): ";
                    cin >> subPilihan;
                    if (subPilihan == 1) {
                        insertLeft(data, parentNode);
                    } else if (subPilihan == 2) {
                        insertRight(data, parentNode);
                    } else {
                        cout << "Pilihan tidak valid.\n";
                    }
                } else {
                    cout << "Parent tidak ditemukan.\n";
                }
                break;
            case 3:
                cout << "Masukkan data node yang ingin diupdate: ";
                cin >> parentData;
                parentNode = find(parentData, root);
                if (parentNode) {
                    cout << "Pilih aksi (1: Ubah, 2: Hapus): ";
                    cin >> subPilihan;
                    if (subPilihan == 1) {
                        cout << "Masukkan data baru: ";
                        cin >> data;
                        update(data, parentNode);
                    } else if (subPilihan == 2) {
                        root = deleteNode(root, parentData);
                    } else {
                        cout << "Pilihan tidak valid.\n";
                    }
                } else {
                    cout << "Node tidak ditemukan.\n";
                }
                break;
            case 4:
                cout << "Masukkan data node yang ingin dicari: ";
                cin >> data;
                parentNode = find(data, root);
                if (parentNode) {
                    cout << "Node ditemukan: " << parentNode->data << endl;
                    cout << "Node paling kiri: " << mostLeft(parentNode)->data << endl;
                    cout << "Node paling kanan: " << mostRight(parentNode)->data << endl;
                } else {
                    cout << "Node tidak ditemukan.\n";
                }
                break;
            case 5:
                cout << "Masukkan data node: ";
                cin >> data;
                parentNode = find(data, root);
                if (parentNode) displayChildren(parentNode);
                else cout << "Node tidak ditemukan.\n";
                break;
            case 6:
                cout << "Pre-order Traversal: ";
                preOrder(root);
                cout << endl;
                break;
            case 7:
                cout << "In-order Traversal: ";
                inOrder(root);
                cout << endl;
                break;
            case 8:
                cout << "Post-order Traversal: ";
                postOrder(root);
                cout << endl;
                break;
            case 9:
                if (is_valid_bst(root, numeric_limits<char>::min(),
numeric_limits<char>::max()))
                    cout << "Pohon adalah Binary Search Tree yang valid.\n";
                else
                    cout << "Pohon bukan Binary Search Tree yang valid.\n";
                break;
            case 10:
                cout << "Jumlah simpul daun: " << cari_simpul_daun(root) << endl;
                break;
            case 11:
                cout << "Keluar.\n";
                break;
            default:
                cout << "Pilihan tidak valid.\n";

    } while (pilihan != 11);
}
```

```
// Menampilkan child dan descendant dari node tertentu
void displayChildren(Pohon *node) {
    if (!node) return;
    cout << "\nNode " << node->data << " memiliki child: ";
    if (node->left) cout << "Kiri: " << node->left->data << " ";
    if (node->right) cout << "Kanan: " << node->right->data << " ";
    cout << "\nDescendant dari node " << node->data << ": ";
    preOrder(node);
    cout << endl;
}

// Bagian dari menu untuk menampilkan child dan descendant
case 5:
    cout << "Masukkan data node: ";
    cin >> data;
    parentNode = find(data, root);
    if (parentNode) displayChildren(parentNode);
    elsecout << "Node tidak ditemukan.\n";
    break;
```

Output:

```
Menu:
1. Buat Node Root
2. Tambah Node
3. Update Node
4. Cari Node
5. Tampilkan Child dan Descendant
6. Traversal Pre-order
7. Traversal In-order
8. Traversal Post-order
9. Cek Validitas BST
10. Hitung Simpul Daun
11. Keluar
Pilihan: 5
Masukkan data node: S

Node S memiliki child: Kiri: A Kanan: U
Descendant dari node S: S A U
```

Penjelasan: Pada modifikasi pertama, ditambahkan menu interaktif yang memungkinkan pengguna untuk memasukkan data tree secara dinamis. Pengguna dapat membuat node root, menambahkan node baru sebagai anak kiri atau kanan dari node tertentu, mengubah data node, dan mencari node. Selain itu, ditambahkan juga fungsi displayChildren yang menampilkan child dan descendant dari node yang diinputkan oleh pengguna. Fungsi ini menggunakan traversal pre-order untuk mencetak semua descendant dari node yang diberikan. Menu ini memberikan

antarmuka yang mudah digunakan untuk mengelola pohon biner.

b. Fungsi Rekursif - Memeriksa Properti Binary Search Tree

Code:

```
// Memeriksa apakah pohon memenuhi properti Binary Search Tree
bool is_valid_bst(Pohon *node, char min_val, char max_val) {
    if (!node) 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);
}

// Bagian dari menu untuk memeriksa validitas BST
case 9:
    if (is_valid_bst(root, numeric_limits<char>::min(), numeric_limits<char>::max()))
        cout << "Pohon adalah Binary Search Tree yang valid.\n";
    else
        cout << "Pohon bukan Binary Search Tree yang valid.\n";
    break;
```

Output:

Valid

```
Node S memiliki child: Kiri: A Kanan: U
Descendant dari node S: S A U
```

```
Menu:
1. Buat Node Root
2. Tambah Node
3. Update Node
4. Cari Node
5. Tampilkan Child dan Descendant
6. Traversal Pre-order
7. Traversal In-order
8. Traversal Post-order
9. Cek Validitas BST
10. Hitung Simpul Daun
11. Keluar
Pilihan: 9
Pohon adalah Binary Search Tree yang valid.
```

Tidak Valid

```
Node A memiliki child: Kiri: Z
Descendant dari node A: A Z
```

```
Menu:
1. Buat Node Root
2. Tambah Node
3. Update Node
4. Cari Node
5. Tampilkan Child dan Descendant
6. Traversal Pre-order
7. Traversal In-order
8. Traversal Post-order
9. Cek Validitas BST
10. Hitung Simpul Daun
11. Keluar
Pilihan: 9
Pohon bukan Binary Search Tree yang valid.
```

Penjelasan: Modifikasi kedua melibatkan penambahan fungsi rekursif is_valid_bst yang memeriksa apakah suatu pohon memenuhi properti Binary Search Tree (BST). Fungsi ini bekerja dengan memeriksa setiap node untuk memastikan bahwa nilai data di node tersebut berada dalam rentang yang valid (lebih besar dari nilai minimum dan lebih kecil dari nilai maksimum). Jika semua node memenuhi kondisi ini, maka pohon tersebut adalah BST yang valid. Fungsi ini diuji pada berbagai pohon, baik yang valid maupun tidak valid sebagai BST, melalui menu interaktif yang memungkinkan pengguna untuk memeriksa validitas pohon yang mereka buat.

c. Fungsi Rekursif - Menghitung Jumlah Simpul Daun

Code:

```
// Menghitung jumlah simpul daun dalam Binary Tree
int cari_simpul_daun(Pohon *node) {
    if (!node) return 0;
    if (!node->left && !node->right) return 1;
    return cari_simpul_daun(node->left) + cari_simpul_daun(node->right);
}

// Bagian dari menu untuk menghitung jumlah simpul daun
case 10:
    cout << "Jumlah simpul daun: " << cari_simpul_daun(root) << endl;
    break;
```

Output:

```
Pre-order Traversal: S A U

Menu:
1. Buat Node Root
2. Tambah Node
3. Update Node
4. Cari Node
5. Tampilkan Child dan Descendant
6. Traversal Pre-order
7. Traversal In-order
8. Traversal Post-order
9. Cek Validitas BST
10. Hitung Simpul Daun
11. Keluar
Pilihan: 10
Jumlah simpul daun: 2
```

Penjelasan: Modifikasi ketiga adalah penambahan fungsi rekursif cari_simpul_daun yang menghitung jumlah simpul daun dalam pohon biner. Simpul daun adalah node yang tidak memiliki anak kiri maupun kanan. Fungsi ini bekerja dengan menghitung jumlah simpul daun di setiap sub-pohon kiri dan kanan secara rekursif dan mengembalikan jumlah totalnya. Menu interaktif menyediakan opsi bagi pengguna untuk menghitung dan menampilkan jumlah simpul daun dalam pohon yang mereka buat, sehingga memudahkan analisis struktur pohon.

E. Kesimpulan

Dalam praktik ini, Diperoleh pemahaman yang mendalam tentang penggunaan rekursif dan struktur data tree dalam pemrograman. Rekursif membantu menyelesaikan masalah secara modular dengan memecahnya menjadi bagian-bagian kecil, seperti traversal tree dan pemeriksaan validitas Binary Search Tree (BST), meskipun perlu dengan adanya pertimbangan terkait dengan efisiensi memori. Selain itu, penerapan dari Binary Tree memungkinkan pengelolaan data secara hierarkis melalui operasi seperti penyisipan, penghapusan, dan pencarian. Melalui praktikum ini, didapatkan belajar menganalisis dan memodifikasi struktur tree, seperti menghitung node daun dan memvalidasi BST, yang menunjukkan pentingnya tree dalam menyelesaikan masalah yang muncul.