

LAPORAN PRAKTIKUM
Modul 9
“TREE”



Disusun Oleh:

Ahmad Al - Farizi - 2311104054

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

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.

2. Landasan Teori

1. Pengertian Rekursif

Secara harfiah, rekursif berarti suatu proses pengulangan sesuatu dengan cara kesamaan-diri atau suatu proses yang memanggil dirinya sendiri. Prosedur dan fungsi merupakan sub program yang sangat bermanfaat dalam pemrograman, terutama untuk program atau proyek yang besar. Manfaat penggunaan sub program antara lain adalah:

- meningkatkan readability, yaitu mempermudah pembacaan program
- meningkatkan modularity, yaitu memecah sesuatu yang besar menjadi modul-modul atau bagian-bagian yang lebih kecil sesuai dengan fungsinya, sehingga mempermudah pengecekan, testing dan lokalisasi kesalahan.
- meningkatkan reusability, yaitu suatu sub program dapat dipakai berulang kali dengan hanya memanggil sub program tersebut tanpa menuliskan perintah-perintah yang semestinya diulang-ulang.

Sub Program Rekursif adalah sub program yang memanggil dirinya sendiri selama kondisi pemanggilan dipenuhi. Prinsip rekursif sangat berkaitan erat dengan bentuk induksi matematika.

2. Kriteria Rekursif

Dengan melihat sifat sub program rekursif di atas maka sub program rekursif harus memiliki:

- Kondisi yang menyebabkan pemanggilan dirinya berhenti (disebut kondisi khusus atau special condition)
- Pemanggilan diri sub program (yaitu bila kondisi khusus tidak dipenuhi).

Secara umum bentuk dari sub program rekursif memiliki statemen kondisional:

- if kondisi khusus tak dipenuhi
- then panggil diri-sendiri dengan parameter yang sesuai
- else lakukan instruksi yang akan dieksekusi bila kondisi khusus dipenuhi.

Sub program rekursif umumnya dipakai untuk permasalahan yang memiliki langkah penyelesaian yang terpola atau langkah-langkah yang teratur. Bila kita memiliki suatu permasalahan dan kita mengetahui algoritma penyelesaiannya, kadang-kadang sub program rekursif menjadi pilihan kita bila memang memungkinkan untuk dipergunakan. Secara algoritmis (dari segi algoritma, yaitu bila kita mempertimbangkan penggunaan memori, waktu eksekusi sub program) sub program rekursif sering bersifat tidak efisien.

Dengan demikian sub program rekursif umumnya memiliki efisiensi dalam penulisan perintah, tetapi kadang tidak efisien secara algoritmis. Meskipun demikian banyak pula permasalahan-permasalahan yang lebih sesuai diselesaikan dengan cara rekursif (misalnya dalam pencarian / searching, yang akan dibahas pada pertemuan-pertemuan yang akan datang).

3. Kekurangan Rekursif

Konsep penggunaan yang terlihat mudah karena fungsi rekursif dapat menyederhanakan solusi dari suatu permasalahan, sehingga sering kali menghasilkan bentuk algoritma dan program yang lebih singkat dan lebih mudah dimengerti. Kendati demikian, penggunaan rekursif memiliki beberapa kekurangan antara lain:

- Memerlukan memori yang lebih banyak untuk menyimpan activation record dan variabel lokal. Activation record diperlukan waktu proses kembali kepada pemanggil
- Memerlukan waktu yang lebih banyak untuk menangani activation record.

Secara umum gunakan penyelesaian rekursif hanya jika:

- Penyelesaian sulit dilaksanakan secara iteratif.
- Efisiensi dengan cara rekursif sudah memadai.
- Efisiensi bukan masalah dibandingkan dengan kejelasan logika program.

4. Contoh Rekursif

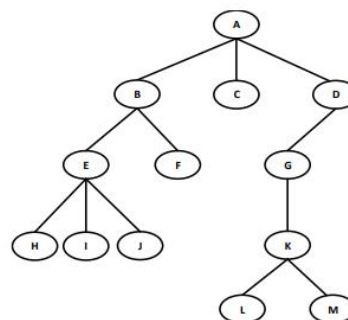
Berikut adalah contoh program untuk rekursif menghitung nilai faktorial sebuah bilangan.

Algoritma	C++
Program rekursif_factorial Kamus faktor, n : integer function faktorial (input: a: integer) Algoritma input(n) faktor = faktorial(n) output(faktor) function faktorial (input: a: integer) kamus algoritma if (a == 1 a == 0) then → 1 else if (a > 1) then → a* faktorial(a-1) else → 0	<pre> #include <conio.h> #include <iostream> long int faktorial(long int a); main() { long int faktor; long int n; cout<<"Masukkan nilai faktorial "; cin>>n; faktor =faktorial(n); cout<<n<<"!="<<faktorial<<endl; getch(); } long int faktorial(long int y){ if (a==1 a==0){ return(1); }else if (a>1){ return(a*faktorial(a-1)); }else{ return 0; } } </pre>

5. Pengertian Tree

Kita telah mengenal dan mempelajari jenis – jenis strukur data yang linear, seperti: list, stack dan queue. Adapun jenis struktur data yang kita pelajari kali ini adalah struktur data yang non-linear (non – linear data structure) yang disebut tree. Tree digambarkan sebagai suatu graph tak berarah terhubung dan tidak terhubung dan tidak mengandung sirkuit. Karakteristik dari suatu tree T adalah:

- T kosong berarti empty tree
- Hanya terdapat satu node tanpa pendahulu, disebut akar (root)
- Semua node lainnya hanya mempunyai satu node pendahulu.



Gambar 10-1 Tree

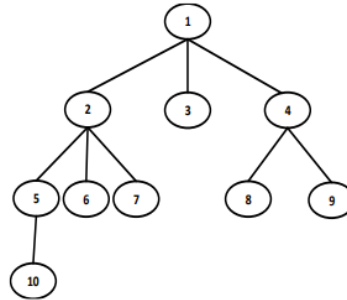
Berdasarkan gambar diatas dapat digambarkan beberapa terminologinya, yaitu

1. Anak (*child* atau *children*) dan Orangtua (*parent*). B, C, dan D adalah anak-anak simpul A, A adalah Orangtua dari anak-anak itu.
2. Lintasan (*path*). Lintasan dari A ke J adalah A, B, E, J. Panjang lintasan dari A ke J adalah 3.
3. Saudara kandung (*sibling*). F adalah saudara kandung E, tetapi G bukan saudara kandung E, karena orangtua mereka berbeda.
4. Derajat(*degree*). Derajat sebuah simpul adalah jumlah pohon (atau jumlah anak) pada simpul tersebut. Derajat A = 3, derajat D = 1 dan derajat C = 0. Derajat maksimum dari semua simpul merupakan derajat pohon itu sendiri. Pohon diatas berderajat 3.
5. Daun (*leaf*). Simpul yang berderajat nol (atau tidak mempunyai anak) disebut daun. Simpul H, I, J, F, C, L, dan M adalah daun.
6. Simpul Dalam (*internal nodes*). Simpul yang mempunyai anak disebut simpul dalam. Simpul B, D, E, G, dan K adalah simpul dalam.
7. Tinggi (*height*) atau Kedalaman (*depth*). Jumlah maksimum *node* yang terdapat di cabang *tree* tersebut. Pohon diatas mempunyai tinggi 4.

6. Jenis – Jenis Tree

A. Ordered Tree

Yaitu pohon yang urutan anak-anaknya penting.



Gambar 10-2 Ordered Tree

B. Binary Tree

Setiap node di Binary Tree hanya dapat mempunyai maksimum 2 children tanpa pengecualian. Level dari suatu tree dapat menunjukkan berapa kemungkinan jumlah maximum nodes yang terdapat pada tree tersebut. Misalnya, level tree adalah r , maka node maksimum yang mungkin adalah 2^r .

- Complete Binary Tree

Suatu binary tree dapat dikatakan lengkap (complete), jika pada setiap level yang mempunyai jumlah maksimum dari kemungkinan node yang dapat dipunyai, dengan pengecualian node terakhir. Complete tree T_n yang unik memiliki n nodes. Untuk menentukan jumlah left children dan right children tree T_n di node K dapat dilakukan dengan cara:

1. Menentukan left children: $2 \cdot K$
2. Menentukan right children: $2 \cdot (K + 1)$
3. Menentukan parent: $\lceil K/2 \rceil$

- Extended Binary Tree

Suatu binary tree yang terdiri atas tree T yang masing-masing node-nya terdiri dari tepat 0 atau 2 children disebut 2-tree atau extended binary tree. Jika setiap node N mempunyai 0 atau 2 children disebut internal nodes dan node dengan 0 children disebut external nodes.

- Binary Search Tree

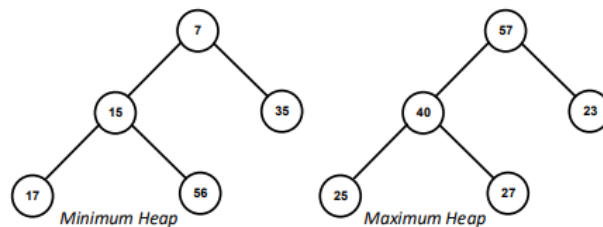
Binary search tree adalah Binary tree yang terurut dengan ketentuan: 1. Semua LEFTCHILD harus lebih kecil dari parent-nya. 2. Semua RIGHTCHILD harus lebih besar dari parentnya dan leftchild-nya.

- AVL Tree

Adalah binary search tree yang mempunyai ketentuan bahwa maximum perbedaan height antara subtree kiri dan subtree kanan adalah 1.

- Heap Tree

Adalah tree yang memenuhi persamaan berikut: $R[i] < r[2i]$ and $R[i] < r[2i+1]$. Heap juga disebut Complete Binary Tree, karena jika suatu node mempunyai child, maka jumlah child-nya harus selalu dua. Minimum Heap: jika parent-nya selalu lebih kecil daripada kedua children-nya. Maximum Heap: jika parent-nya selalu lebih besar daripada kedua children-nya.



7. Operasi – Operasi dalam Binary Search Tree

A. Insert

- Jika node yang akan di-insert lebih kecil, maka di-insert pada Left Subtree.
- Jika lebih besar, maka di-insert pada Right Subtree.

B. Update

Jika setelah diupdate posisi/lokasi node yang bersangkutan tidak sesuai dengan ketentuan, maka harus dilakukan dengan proses REGENERASI agar tetap memenuhi kriteria Binary Search Tree.

C. Search

Proses pencarian elemen pada binary tree dapat menggunakan algoritma rekursif binary search. Berikut adalah algoritma binary search:

- Pencarian pada binary search tree dilakukan dengan menaruh pointer

dan membandingkan nilai yang dicari dengan node awal (root)

- Jika nilai yang dicari tidak sama dengan node, maka pointer akan diganti ke child dari node yang ditunjuk:
 - a. Pointer akan pindah ke child kiri bila, nilai dicari lebih kecil dari nilai node yang ditunjuk saat itu.
 - b. Pointer akan pindah ke child kanan bila, nilai dicari lebih besar dari nilai node yang ditunjuk saat itu.
- Nilai node saat itu akan dibandingkan lagi dengan nilai yang dicari dan apabila belum ditemukan, maka perulangan akan kembali ke tahap 2.
- Pencarian akan berhenti saat nilai yang dicari ketemu, atau pointer menunjukan nilai null.

3. Guided

1. Guided 9

Program ini merupakan implementasi dari struktur data pohon biner dalam bahasa C++. Pertama, struktur data Pohon didefinisikan untuk menyimpan karakter data dan pointer ke anak kiri, anak kanan, serta induk dari setiap node. Kemudian, beberapa fungsi utama diimplementasikan: `init()` untuk menginisialisasi pohon sebagai kosong, `isEmpty()` untuk mengecek apakah pohon kosong, `buatNode()` untuk membuat node baru sebagai root jika pohon masih kosong, `insertLeft()` dan `insertRight()` untuk menambahkan node baru sebagai anak kiri atau kanan dari node tertentu. Ada juga fungsi `update()` untuk mengubah data dalam sebuah node, `find()` untuk mencari node dengan data tertentu, serta tiga fungsi traversal pohon: `preOrder()`, `inOrder()`, dan `postOrder()`. Program juga mencakup fungsi `deleteNode()` untuk menghapus node dengan data tertentu, serta fungsi `mostLeft()` dan `mostRight()` untuk menemukan node paling kiri dan kanan dalam pohon. Pada fungsi `main()`, pohon biner dibentuk dengan beberapa node, dan dilakukan traversal pre-order, in-order, dan post-order untuk menampilkan isi pohon. Program juga menunjukkan cara menghapus sebuah node dan melakukan traversal in-order setelah penghapusan.

Kode Program:

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

/// PROGRAM BINARY TREE

// Struktur data pohon biner untuk menyimpan data dan pointer ke anak kiri,
// kanan, dan induk
struct Pohon {
    char data;           // Data yang disimpan di node (tipe char)
    Pohon *left, *right; // Pointer ke anak kiri dan anak kanan
    Pohon *parent;       // Pointer ke node induk
};

// Variabel global untuk menyimpan root (akar) pohon dan node baru
Pohon *root, *baru;

// Inisialisasi pohon agar kosong
void init() {
    root = NULL; // Mengatur root sebagai NULL (pohon kosong)
}

// Mengecek apakah pohon kosong
bool isEmpty() {
    return root == NULL; // Mengembalikan true jika root adalah NULL
}

// Membuat node baru sebagai root pohon
void buatNode(char data) {
    if (isEmpty()) { // Jika pohon kosong
        root = new Pohon{data, NULL, NULL, NULL}; // Membuat node baru
        // sebagai root
        cout << "\nNode " << data << " berhasil dibuat menjadi root." << endl;
    } else {
        cout << "\nPohon sudah dibuat." << endl; // Root sudah ada, tidak
        // membuat node baru
    }
}

// Menambahkan node baru sebagai anak kiri dari node tertentu
Pohon* insertLeft(char data, Pohon *node) {
    if (node->left != NULL) { // Jika anak kiri sudah ada
        cout << "\nNode " << node->data << " sudah ada child kiri!" << endl;
    }
}
```



```
        return NULL; // Tidak menambahkan node baru
    }
    // Membuat node baru dan menghubungkannya ke node sebagai anak kiri
    baru = new Pohon{data, NULL, NULL, node};
    node->left = baru;
    cout << "\nNode " << data << " berhasil ditambahkan ke child kiri " <<
node->data << endl;
    return baru; // Mengembalikan pointer ke node baru
}

// Menambahkan node baru sebagai anak kanan dari node tertentu
Pohon* insertRight(char data, Pohon *node) {
    if (node->right != NULL) { // Jika anak kanan sudah ada
        cout << "\nNode " << node->data << " sudah ada child kanan!" << endl;
        return NULL; // Tidak menambahkan node baru
    }
    // Membuat node baru dan menghubungkannya ke node sebagai anak
    kanan
    baru = new Pohon{data, NULL, NULL, node};
    node->right = baru;
    cout << "\nNode " << data << " berhasil ditambahkan ke child kanan " <<
node->data << endl;
    return baru; // Mengembalikan pointer ke node baru
}

// Mengubah data di dalam sebuah node
void update(char data, Pohon *node) {
    if (!node) { // Jika node tidak ditemukan
        cout << "\nNode yang ingin diubah tidak ditemukan!" << endl;
        return;
    }
    char temp = node->data; // Menyimpan data lama
    node->data = data; // Mengubah data dengan nilai baru
    cout << "\nNode " << temp << " berhasil diubah menjadi " << data <<
endl;
}

// Mencari node dengan data tertentu
void find(char data, Pohon *node) {
    if (!node) return; // Jika node tidak ada, hentikan pencarian

    if (node->data == data) { // Jika data ditemukan
```

```
        cout << "\nNode ditemukan: " << data << endl;
        return;
    }
    // Melakukan pencarian secara rekursif ke anak kiri dan kanan
    find(data, node->left);
    find(data, node->right);
}

// Traversal Pre-order (Node -> Kiri -> Kanan)
void preOrder(Pohon *node) {
    if (!node) return; // Jika node kosong, hentikan traversal
    cout << node->data << " "; // Cetak data node saat ini
    preOrder(node->left);    // Traversal ke anak kiri
    preOrder(node->right);   // Traversal ke anak kanan
}

// Traversal In-order (Kiri -> Node -> Kanan)
void inOrder(Pohon *node) {
    if (!node) return; // Jika node kosong, hentikan traversal
    inOrder(node->left); // Traversal ke anak kiri
    cout << node->data << " "; // Cetak data node saat ini
    inOrder(node->right); // Traversal ke anak kanan
}

// Traversal Post-order (Kiri -> Kanan -> Node)
void postOrder(Pohon *node) {
    if (!node) return; // Jika node kosong, hentikan traversal
    postOrder(node->left); // Traversal ke anak kiri
    postOrder(node->right); // Traversal ke anak kanan
    cout << node->data << " "; // Cetak data node saat ini
}

// Menghapus node dengan data tertentu
Pohon* deleteNode(Pohon *node, char data) {
    if (!node) return NULL; // Jika node kosong, hentikan

    // Rekursif mencari node yang akan dihapus
    if (data < node->data) {
        node->left = deleteNode(node->left, data); // Cari di anak kiri
    } else if (data > node->data) {
        node->right = deleteNode(node->right, data); // Cari di anak kanan
    } else {
```

```
// Jika node ditemukan
if (!node->left) { // Jika tidak ada anak kiri
    Pohon *temp = node->right; // Anak kanan menggantikan posisi node
    delete node;
    return temp;
} else if (!node->right) { // Jika tidak ada anak kanan
    Pohon *temp = node->left; // Anak kiri menggantikan posisi node
    delete node;
    return temp;
}

// Jika node memiliki dua anak, cari node pengganti (successor)
Pohon *successor = node->right;
while (successor->left) successor = successor->left; // Cari node terkecil
di anak kanan
node->data = successor->data; // Gantikan data dengan successor
node->right = deleteNode(node->right, successor->data); // Hapus
successor
}
return node;
}

// Menemukan node paling kiri
Pohon* mostLeft(Pohon *node) {
    if (!node) return NULL; // Jika node kosong, hentikan
    while (node->left) node = node->left; // Iterasi ke anak kiri hingga mentok
    return node;
}

// Menemukan node paling kanan
Pohon* mostRight(Pohon *node) {
    if (!node) return NULL; // Jika node kosong, hentikan
    while (node->right) node = node->right; // Iterasi ke anak kanan hingga
mentok
    return node;
}

// Fungsi utama
int main() {
    init(); // Inisialisasi pohon
    buatNode('F'); // Membuat root dengan data 'F'
    insertLeft('B', root); // Menambahkan 'B' ke anak kiri root
}
```

```

insertRight('G', root); // Menambahkan 'G' ke anak kanan root
insertLeft('A', root->left); // Menambahkan 'A' ke anak kiri dari 'B'
insertRight('D', root->left); // Menambahkan 'D' ke anak kanan dari 'B'
insertLeft('C', root->left->right); // Menambahkan 'C' ke anak kiri dari 'D'
insertRight('E', root->left->right); // Menambahkan 'E' ke anak kanan dari
'D'

// Traversal pohon
cout << "\nPre-order Traversal: ";
preOrder(root);
cout << "\nIn-order Traversal: ";
inOrder(root);
cout << "\nPost-order Traversal: ";
postOrder(root);

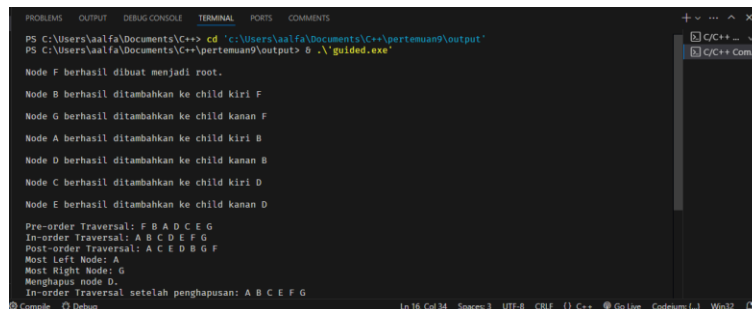
// Menampilkan node paling kiri dan kanan
cout << "\nMost Left Node: " << mostLeft(root)->data;
cout << "\nMost Right Node: " << mostRight(root)->data;

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

return 0;
}

```

Output dari Kode Program:



```

PS C:\Users\aa1fa\Documents\C++> cd 'c:\Users\aa1fa\Documents\C++\pertemuan9\output'
PS C:\Users\aa1fa\Documents\C++\pertemuan9\output> g++ .\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

```

4. Unguided

1. Nomor 1

Program di bawah ini merupakan implementasi dari struktur data binary tree menggunakan bahasa C++, menyediakan fitur untuk membuat, menambah, dan menampilkan node pada binary tree. Node terdiri dari data bertipe char serta pointer untuk anak kiri, anak kanan, dan parent. Program dimulai dengan fungsi `init()` untuk menginisialisasi root menjadi NULL. Fungsi `buatNode()` membuat root baru jika belum ada, sedangkan fungsi `findNode()` mencari node berdasarkan data yang diberikan. Fungsi `insertLeft()` dan `insertRight()` digunakan untuk menambahkan anak kiri atau kanan ke node tertentu, dengan memeriksa apakah node sudah memiliki anak di sisi tersebut. Fungsi `tampilkanChild()` menampilkan anak kiri dan kanan dari node yang dipilih, sementara `tampilkanDescendants()` merekursi seluruh anak (descendant) dari node tertentu. Program memiliki menu interaktif yang membuat pengguna untuk mengelola binary tree, seperti membuat root, menambah anak, menampilkan anak, dan menampilkan semua descendant node tertentu. Proses berjalan dalam loop hingga pengguna memilih opsi keluar.

Kode Program:

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

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

Pohon *root = NULL;

void init() {
    root = NULL;
}
```

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

void buatNode(char data) {
    if (isEmpty()) {
        root = new Pohon{data, NULL, NULL, NULL};
        cout << "Node " << data << " berhasil dibuat sebagai root.\n";
    } else {
        cout << "Root sudah ada.\n";
    }
}

Pohon* findNode(Pohon *node, char data) {
    if (!node) return NULL;
    if (node->data == data) return node;
    Pohon *leftResult = findNode(node->left, data);
    if (leftResult) return leftResult;
    return findNode(node->right, data);
}

Pohon* insertLeft(char data, Pohon *node) {
    if (node->left != NULL) {
        cout << "Node " << node->data << " sudah memiliki anak kiri.\n";
        return NULL;
    }
    Pohon *baru = new Pohon{data, NULL, NULL, node};
    node->left = baru;
    cout << "Node " << data << " berhasil ditambahkan ke anak kiri " <<
        node->data << ".\n";
    return baru;
}
```

```
Pohon* insertRight(char data, Pohon *node) {
    if (node->right != NULL) {
        cout << "Node " << node->data << " sudah memiliki anak kanan.\n";
        return NULL;
    }
    Pohon *baru = new Pohon{data, NULL, NULL, node};
    node->right = baru;
    cout << "Node " << data << " berhasil ditambahkan ke anak kanan " <<
        node->data << ".\n";
    return baru;
}

void tampilkanChild(Pohon *node) {
    if (!node) {
        cout << "Node tidak ditemukan.\n";
        return;
    }
    cout << "Child dari " << node->data << ":\n";
    if (node->left) cout << "Kiri: " << node->left->data << "\n";
    else cout << "Kiri: NULL\n";
    if (node->right) cout << "Kanan: " << node->right->data << "\n";
    else cout << "Kanan: NULL\n";
}

void tampilkanDescendants(Pohon *node) {
    if (!node) return;
    if (node->left || node->right) cout << node->data << ": ";
    if (node->left) cout << node->left->data << " ";
    if (node->right) cout << node->right->data << " ";
    cout << "\n";
    tampilkanDescendants(node->left);
}
```

```
tampilkanDescendants(node->right);
}

void menu() {
    int pilihan;
    char data, parent_data;
    Pohon *parent_node;

    do {
        cout << "\nMenu Binary Tree:";
        cout << "\n1. Buat Root";
        cout << "\n2. Tambahkan Anak Kiri";
        cout << "\n3. Tambahkan Anak Kanan";
        cout << "\n4. Tampilkan Child";
        cout << "\n5. Tampilkan Descendants";
        cout << "\n6. Keluar";
        cout << "\nPilih: ";
        cin >> pilihan;

        switch (pilihan) {
            case 1:
                cout << "Masukkan data root: ";
                cin >> data;
                buatNode(data);
                break;

            case 2:
                cout << "Masukkan data parent: ";
                cin >> parent_data;
                cout << "Masukkan data anak kiri: ";
                cin >> data;
                parent_node = findNode(root, parent_data);
```



```
        if (parent_node) insertLeft(data, parent_node);
        else cout << "Node " << parent_data << " tidak ditemukan.\n";
        break;

    case 3:
        cout << "Masukkan data parent: ";
        cin >> parent_data;
        cout << "Masukkan data anak kanan: ";
        cin >> data;
        parent_node = findNode(root, parent_data);
        if (parent_node) insertRight(data, parent_node);
        else cout << "Node " << parent_data << " tidak ditemukan.\n";
        break;

    case 4:
        cout << "Masukkan data node: ";
        cin >> data;
        parent_node = findNode(root, data);
        tampilkanChild(parent_node);
        break;

    case 5:
        cout << "Masukkan data node: ";
        cin >> data;
        parent_node = findNode(root, data);
        tampilkanDescendants(parent_node);
        break;

    case 6:
        cout << "Keluar.\n";
        break;
```

```

    default:

        cout << "Pilihan tidak valid.\n";

    }

    } while (pilihan != 6);

}

int main() {

    init();

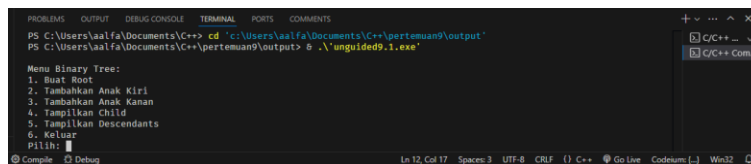
    menu();

    return 0;

}

```

Output dari Kode Program:



```

PS C:\Users\aaifa\Documents\C++> cd 'c:\Users\aaifa\Documents\C++\pertemuan9\output'
PS C:\Users\aaifa\Documents\C++\pertemuan9\output> B .\unguided9.1.exe

Menu Binary Tree:
1. Buat Root
2. Tambahkan Anak Kiri
3. Tambahkan Anak Kanan
4. Tampilkan Child
5. Tampilkan Descendants
6. Keluar
Pilih:

```

2. Nomor 2

Program di bawah ini memeriksa apakah suatu binary tree valid sebagai binary search tree (BST) atau tidak. Node dibuat menggunakan fungsi `createNode`, dengan setiap node memiliki data, anak kiri, dan anak kanan. Validasi BST dilakukan oleh fungsi `is_valid_bst`, yang memeriksa apakah nilai setiap node berada dalam rentang nilai minimum dan maksimum yang diperbolehkan, sambil merekursi anak kiri dan kanan dengan memperbarui batas rentang. Program memvalidasi dua pohon: satu valid sebagai BST dan satu tidak valid, lalu mencetak hasilnya.

Kode Program:

```

#include <iostream>

#include <climits>

using namespace std;

struct Pohon {

```

```
int data;
Pohon *left, *right;
};

Pohon* createNode(int data) {
    Pohon* node = new Pohon;
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

bool is_valid_bst(Pohon* node, int min_val, int 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);
}

int main() {
    Pohon* root = createNode(10);
    root->left = createNode(5);
    root->right = createNode(20);
    root->left->left = createNode(2);
    root->left->right = createNode(8);

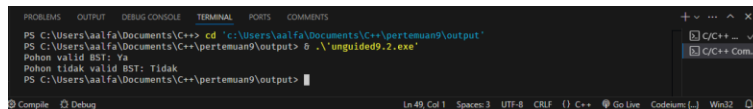
    Pohon* root_invalid = createNode(10);
    root_invalid->left = createNode(5);
    root_invalid->right = createNode(20);
    root_invalid->left->left = createNode(12);
```

```
cout << "Pohon valid BST: " << (is_valid_bst(root, INT_MIN,
    INT_MAX) ? "Ya" : "Tidak") << endl;

cout << "Pohon tidak valid BST: " << (is_valid_bst(root_invalid,
    INT_MIN, INT_MAX) ? "Ya" : "Tidak") << endl;

return 0;
}
```

Output Kode Program:



```
PS C:\Users\aa1fa\Documents\C++> cd c:\Users\aa1fa\Documents\C++\pertemuan9\output
PS C:\Users\aa1fa\Documents\C++\pertemuan9\output> g++ .\unguided9.2.exe
Pohon valid BST: Ya
Pohon tidak valid BST: Tidak
PS C:\Users\aa1fa\Documents\C++\pertemuan9\output>
```

3. Nomor 3

Program di bawah ini menghitung jumlah simpul daun (leaf nodes) pada sebuah binary tree. Fungsi createNode() digunakan untuk membuat node baru dengan data tertentu. Fungsi rekursif cari_simpul_daun menghitung simpul daun dengan memeriksa apakah node tidak memiliki anak kiri maupun kanan, lalu menjumlahkan hasil dari rekursi pada anak kiri dan kanan. Program membuat sebuah pohon contoh, menghitung jumlah simpul daun, dan mencetak hasilnya.

Kode Program:

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

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

Pohon* createNode(char data) {
    Pohon* node = new Pohon;
    node->data = data;
    node->left = node->right = NULL;
```

```
        return node;
    }

    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);
    }

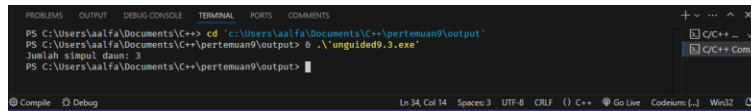
    int main() {
        Pohon* root = createNode('A');
        root->left = createNode('B');
        root->right = createNode('C');
        root->left->left = createNode('D');
        root->left->right = createNode('E');
        root->right->right = createNode('F');

        /*
            Pohon:
                A
              /\
             B  C
            /\  \
           D  E  F
        */

        cout << "Jumlah simpul daun: " << cari_simpul_daun(root) << endl;

        return 0;
    }
}
```

Output dari Kode Program:



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
PS C:\Users\aaifa\Documents\C++\pertemuan9\output>
PS C:\Users\aaifa\Documents\C++\pertemuan9\output> g++ -std=c++11 unguided9_3.exe
Jumlah simpul daun: 3
PS C:\Users\aaifa\Documents\C++\pertemuan9\output>
```

5. Kesimpulan

Rekursif adalah teknik pemrograman di mana sebuah fungsi memanggil dirinya sendiri untuk menyelesaikan masalah, dengan keuntungan seperti meningkatkan keterbacaan, modularitas, dan penggunaan kembali kode. Namun, rekursif memerlukan lebih banyak memori dan waktu eksekusi. Tree adalah struktur data non-linear yang terdiri dari node yang terhubung tanpa sirkuit, dengan berbagai jenis seperti binary tree, complete binary tree, extended binary tree, binary search tree, AVL tree, dan heap tree. Operasi pada binary search tree meliputi insert, update, dan search, yang memanfaatkan algoritma rekursif untuk efisiensi dan kejelasan logika program.

