

PRACTICAL NO. 8

AIM: Study representation of trees and graphs in memory. What are different traversal techniques?

THEORY:

Trees and graphs can be represented in memory using different data structures. Here are some common ways to represent them:

Arrays: In this representation, each node of the tree or graph is assigned a unique index in an array, and the edges are represented using the indices of the nodes they connect.

Linked Lists: In this representation, each node is represented as a struct containing the node's data and a pointer to the next node in the list. For trees, this can be a binary tree where each node has two pointers to its left and right child.

Adjacency Matrix: In this representation, a two-dimensional array is used where the value in the (i,j) th cell represents the weight of the edge connecting nodes i and j .

Adjacency List: In this representation, each node is associated with a list of its neighboring nodes. This is useful when the graph is sparse, meaning there are relatively few edges compared to the number of nodes.

Traversal techniques are methods for visiting each node of a tree or graph in a specific order. Here are some common traversal techniques:

Depth-First Search (DFS): This technique visits all the descendants of a node before visiting its siblings. There are two types of DFS: pre-order (root, left, right), in-order (left, root, right), and post-order (left, right, root).

Breadth-First Search (BFS): This technique visits all the nodes at a given level before moving on to the next level.

Topological Sort: This technique is used for directed acyclic graphs (DAGs) and finds a linear ordering of the nodes that respects the direction of the edges.

In C++, these data structures and traversal techniques can be implemented using classes, structs, and functions. The choice of implementation depends on the specific problem being solved.

Code:

```
void inOrderTraversal(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    inOrderTraversal(root->left);
    cout << root->val << " ";
    inOrderTraversal(root->right);
}
```

Code:

```
void preOrderTraversal(TreeNode* root) {  
    if (root == nullptr) {  
        return;  
    }  
    cout << root->val << " ";  
    preOrderTraversal(root->left);  
    preOrderTraversal(root->right);  
}
```

```
void postOrderTraversal(TreeNode* root) {  
    if (root == nullptr) {  
        return;  
    }  
    postOrderTraversal(root->left);  
    postOrderTraversal(root->right);  
    cout << root->val << " ";  
}
```

```
void dfs(int u, vector<vector<int>>& graph, vector<bool>& visited) {  
    visited[u] = true;  
    for (int v : graph[u]) {
```

```
    if (!visited[v]) {  
        dfs(v, graph, visited);  
    }  
}  
}
```

```
void dfsTraversal(vector<vector<int>>& graph) {  
    int n = graph.size();  
    vector<bool> visited(n, false);  
    for (int i = 0; i < n; i++) {  
        if (!visited[i]) {  
            dfs(i, graph, visited);  
        }  
    }  
}
```

```
void bfsTraversal(vector<vector<int>>& graph, int start) {  
    int n = graph.size();  
    vector<bool> visited(n, false);  
    queue<int> q;  
    q.push(start);
```

```
visited[start] = true;

while (!q.empty()) {

    int u = q.front();

    q.pop();

    for (int v : graph[u]) {

        if (!visited[v]) {

            visited[v] = true;

            q.push(v);

        }

    }

}
```

Tree Code:

```
#include <iostream>

using namespace std;
```

```
// Definition of a binary tree node
```

```
struct Node
```

```
{
```

```
    int data;
```

```
    Node *left;
```

```
    Node *right;
```

```
};
```

```
// Function to create a new node
```

```
Node *newNode(int data)
```

```
{
```

```
    Node *newNode = new Node;
```

```
    newNode->data = data;
```

```
    newNode->left = NULL;
```

```
    newNode->right = NULL;
```

```
    return newNode;
```

```
}
```

```
// Function to create a new
```

```
// Function to insert a new node
```

```
Node *insert(Node *root, int data)
```

```
{
```

```
    if (root == NULL)
```

```
    {
```

```
        return newNode(data);
```

```
}  
else  
{  
    if (root->left == NULL)  
    {  
        root->left = insert(root->left, data);  
    }  
    else if (root->right == NULL)  
    {  
        root->right = insert(root->right, data);  
    }  
    else if (rand() % 2 == 0)  
    {  
        root->left = insert(root->left, data);  
    }  
    else  
    {  
        root->right = insert(root->right, data);  
    }  
    return root;  
}  
}
```

```
// Function to traverse the tree in pre-order
```

```
void preorder(Node *root)
```

```
{  
    if (root != NULL)  
    {  
        cout << root->data << " ";  
        preorder(root->left);  
        preorder(root->right);  
    }  
}
```

```
// Function to traverse the tree in inorder
```

```
void inorder(Node *root)
```

```
{  
    if (root != NULL)  
    {  
        inorder(root->left);  
        cout << root->data << " ";  
        inorder(root->right);  
    }  
}
```

```
// Function to traverse the tree in post-order
```



```
void postorder(Node *root)
{
    if (root != NULL)
    {
        postorder(root->left);
        postorder(root->right);
        cout << root->data << " ";
    }
}
```

```
int main()
{
    // Creating a binary tree
    Node *root = NULL;

    int data;
    char choice;

    int num;

    cout << "Enter the number of nodes : ";
    cin >> num;

    cout << "Enter root data : ";
    cin >> data;

    root = newNode(data);
```

```
for (int i = 0; i < num-1 ; i++){  
    cout << "Enter node data : ";  
    cin >> data;  
    insert(root, data);  
}
```

```
cout << "Pre-order traversal: ";  
preorder(root);  
cout << endl;
```

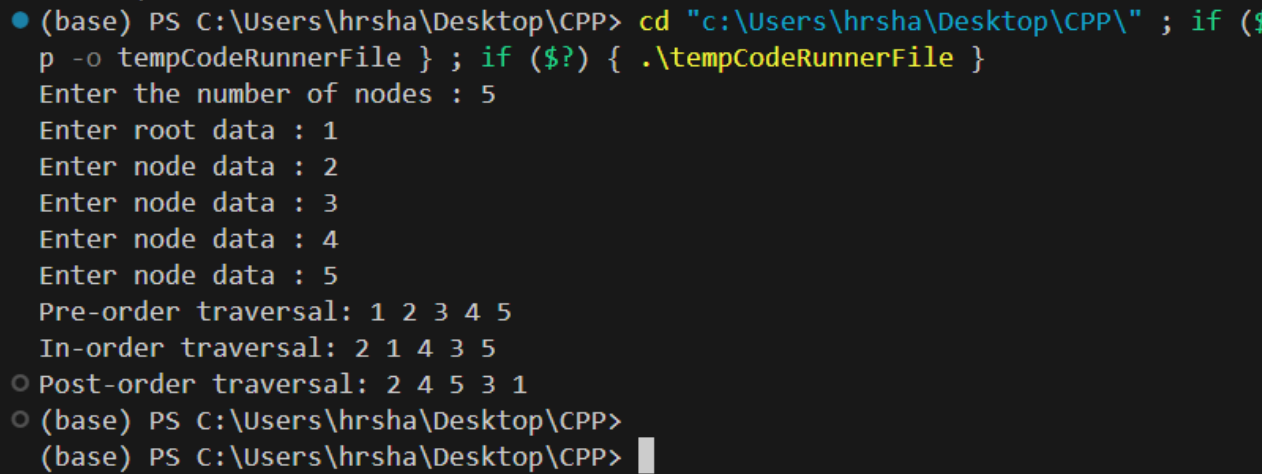
```
cout << "In-order traversal: ";  
inorder(root);  
cout << endl;
```

```
cout << "Post-order traversal: ";  
postorder(root);  
cout << endl;
```

```

return 0;
}

```



```

• (base) PS C:\Users\hrsha\Desktop\CPP> cd "c:\Users\hrsha\Desktop\CPP\" ; if ($?) { g++ -o tempCodeRunnerFile *.cpp ; if ($?) { .\tempCodeRunnerFile }
Enter the number of nodes : 5
Enter root data : 1
Enter node data : 2
Enter node data : 3
Enter node data : 4
Enter node data : 5
Pre-order traversal: 1 2 3 4 5
In-order traversal: 2 1 4 3 5
Post-order traversal: 2 4 5 3 1
(base) PS C:\Users\hrsha\Desktop\CPP>
(base) PS C:\Users\hrsha\Desktop\CPP>

```

CONCLUSION:

Thus we have successfully executed programs .