**AI LAB 5 – Depth First Search (DFS) & Tree Traversals**

**Task 1 – DFS using Stack and Node**

**Objective**

To understand how **Depth First Search (DFS)** works using a **stack** data structure instead of recursion.  
We will implement DFS step-by-step to visit all nodes in a graph.

**Concept**

Depth First Search (DFS) is a graph traversal algorithm that explores as far as possible along one branch before backtracking.  
Instead of recursion, we can use a **stack** to track nodes to visit next. The **stack** follows a **Last In First Out (LIFO)** approach.

**Algorithm**

1. Start from the given node and **push it into the stack**.
2. While the stack is not empty:
   * Pop a node from the stack.
   * If the node has not been visited, mark it as visited and print it.
   * Push all unvisited neighbors into the stack (in reverse order).

**Python Code**

# DFS Implementation using Stack and Node

graph = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F'],

'D': [],

'E': ['G'],

'F': [],

'G': []

}

def dfs\_stack(start\_node):

visited = []

stack = [start\_node]

while stack:

node = stack.pop()

if node not in visited:

print(node, end=" ")

visited.append(node)

for neighbor in reversed(graph[node]):

stack.append(neighbor)

print("DFS Traversal using Stack:")

dfs\_stack('A')

**Output**

DFS Traversal using Stack:

A B D E G C F

**Explanation**

* The program starts from node **A**.
* It uses a stack to store nodes and visits them in **depth-first order**.
* Whenever a node is visited, its connected neighbors are pushed into the stack for later visits.
* This continues until the stack is empty.

**Learning Outcome**

✅ Learned how **DFS** can be implemented **without recursion** using a **stack**.  
✅ Understood how **graph traversal** works internally.  
✅ Practiced the concept of **Last In First Out (LIFO)** using a stack.

**Task 2 – DFS Traversals: Inorder, Preorder, and Postorder**

**Objective**

To implement **three Depth First Search (DFS) traversal techniques** — Inorder, Preorder, and Postorder — for binary trees.

**Concept**

DFS is also used in **tree data structures**.  
There are three common ways to traverse a binary tree using DFS:

| **Traversal Type** | **Visiting Order** | **Description** |
| --- | --- | --- |
| **Inorder** | Left → Root → Right | Visits left subtree first, then root, then right |
| **Preorder** | Root → Left → Right | Visits root before its subtrees |
| **Postorder** | Left → Right → Root | Visits root after its subtrees |

**Python Code**

# Node class for binary tree

class Node:

def \_\_init\_\_(self, value):

self.value = value

self.left = None

self.right = None

# Inorder Traversal (Left → Root → Right)

def inorder(root):

if root:

inorder(root.left)

print(root.value, end=" ")

inorder(root.right)

# Preorder Traversal (Root → Left → Right)

def preorder(root):

if root:

print(root.value, end=" ")

preorder(root.left)

preorder(root.right)

# Postorder Traversal (Left → Right → Root)

def postorder(root):

if root:

postorder(root.left)

postorder(root.right)

print(root.value, end=" ")

# Create binary tree

root = Node('A')

root.left = Node('B')

root.right = Node('C')

root.left.left = Node('D')

root.left.right = Node('E')

root.right.left = Node('F')

root.right.right = Node('G')

# Run traversals

print("Inorder Traversal:")

inorder(root)

print("\nPreorder Traversal:")

preorder(root)

print("\nPostorder Traversal:")

postorder(root)

**Output**

Inorder Traversal:

D B E A F C G

Preorder Traversal:

A B D E C F G

Postorder Traversal:

D E B F G C A

**Explanation**

* In **Inorder**, the algorithm visits the left subtree first, then the root, and finally the right subtree.
* In **Preorder**, the root is visited first, then left and right subtrees.
* In **Postorder**, the root is visited at the end after both subtrees.