# Depth First Search (DFS) using Stack - Documentation

## 1. Introduction

Depth-First Search (DFS) is a fundamental graph traversal algorithm that explores as far as possible along each branch before backtracking. It is used in Artificial Intelligence, pathfinding, compilers, and network analysis.

## 2. Working Principle

DFS can be implemented using either recursion or an explicit stack. In this iterative version, a stack data structure is used to remember the next vertex to visit. The algorithm continues until all vertices connected to the start node have been visited.

## 3. Python Implementation

def dfs\_stack(graph, start):  
 visited = set() # To keep track of visited nodes  
 stack = [start] # Initialize stack with the starting node  
  
 while stack: # Loop until the stack becomes empty  
 node = stack.pop() # Take (pop) the top element from the stack  
  
 if node not in visited:  
 print(node, end=' ') # Process the node (print or store it)  
 visited.add(node) # Mark it as visited  
  
 # Add neighbors to stack in reverse order  
 for neighbor in reversed(graph[node]):  
 if neighbor not in visited:  
 stack.append(neighbor)  
  
graph = {  
 'A': ['B', 'C'],  
 'B': ['D', 'E'],  
 'C': ['F'],  
 'D': [],  
 'E': ['F'],  
 'F': []  
}  
  
dfs\_stack(graph, 'A')

## 4. Example Execution

Input Graph:

A → B, C  
B → D, E  
C → F  
E → F

Start Node: A

Output: A B D E F C

## 5. Step-by-Step Dry Run

Step 1: Stack = [A], Visited = {}  
Step 2: Pop A → Print A, Add to Visited → Push C, B  
Step 3: Pop B → Print B, Add to Visited → Push E, D  
Step 4: Pop D → Print D  
Step 5: Pop E → Print E → Push F  
Step 6: Pop F → Print F  
Step 7: Pop C → Print C  
  
Final Output: A B D E F C

## 6. Time and Space Complexity

• Time Complexity: O(V + E) where V is the number of vertices and E is the number of edges.  
• Space Complexity: O(V) due to the stack and visited set.

## 7. Applications of DFS

• Finding connected components in a graph  
• Topological sorting  
• Solving puzzles like mazes  
• Detecting cycles in a graph  
• Pathfinding in AI and games

# ****Tree Traversals: Inorder, Preorder, and Postorder****

## ****1. Introduction****

Tree traversal refers to the process of visiting each node in a tree data structure exactly once in a specific order.  
In binary trees, there are three main depth-first traversal methods:

* **Inorder Traversal (Left → Root → Right)**
* **Preorder Traversal (Root → Left → Right)**
* **Postorder Traversal (Left → Right → Root)**

These traversal techniques help in displaying, modifying, or processing tree nodes efficiently.

## ****2. Python Implementation****

class Node:

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

self.key = key

self.left = None

self.right = None

def inorder(root):

if root:

inorder(root.left)

print(root.key, end=" ")

inorder(root.right)

def preorder(root):

if root:

print(root.key, end=" ")

preorder(root.left)

preorder(root.right)

def postorder(root):

if root:

postorder(root.left)

postorder(root.right)

print(root.key, end=" ")

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')

print("Inorder Traversal:")

inorder(root)print("\nPreorder Traversal:")

preorder(root)print("\nPostorder Traversal:")

postorder(root)

## ****3. Example Tree Structure****

The binary tree created in the above code looks like this:

A

/ \

B C

/ \ /

D E F

## ****4. Types of Traversal and Explanation****

### 🔹 ****Inorder Traversal (Left → Root → Right)****

**Process:**

* Visit the left subtree.
* Visit the root node.
* Visit the right subtree.

**Example Execution:**

D → B → E → A → F → C

**Output:**

Inorder Traversal: D B E A F C

### ****Preorder Traversal (Root → Left → Right)****

**Process:**

* Visit the root node.
* Visit the left subtree.
* Visit the right subtree.

**Example Execution:**

A → B → D → E → C → F

**Output:**

Preorder Traversal: A B D E C F

### ****Postorder Traversal (Left → Right → Root)****

**Process:**

* Visit the left subtree.
* Visit the right subtree.
* Visit the root node.

**Example Execution:**

D → E → B → F → C → A

**Output:**

Postorder Traversal: D E B F C A

## ****5. Step-by-Step Example (Preorder Traversal)****

Let’s trace **Preorder (Root → Left → Right)**:

| **Step** | **Action** | **Node Visited** | **Output Sequence** |
| --- | --- | --- | --- |
| 1 | Start at Root | A | A |
| 2 | Go to Left Child | B | A B |
| 3 | Go to Left Child of B | D | A B D |
| 4 | Backtrack → Right Child of B | E | A B D E |
| 5 | Backtrack → Right Child of A | C | A B D E C |
| 6 | Go to Left Child of C | F | A B D E C F |

✅ **Final Output:**

A B D E C F

## ****6. Output of All Traversals****

Inorder Traversal: D B E A F CPreorder Traversal: A B D E C FPostorder Traversal: D E B F C A

## ****7. Time and Space Complexity****

| **Type** | **Description** |
| --- | --- |
| **Time Complexity** | O(n) → Each node is visited exactly once |
| **Space Complexity** | O(h) → Depends on tree height (stack frames in recursion) |

## ****8. Applications of Each Traversal****

| **Traversal Type** | **Applications** |
| --- | --- |
| **Inorder** | Used in Binary Search Trees to get data in sorted order |
| **Preorder** | Used to copy or serialize a tree; create prefix expressions |
| **Postorder** | Used to delete a tree or evaluate postfix expressions |

✅ **Summary:**  
All three traversals visit every node, but the **order of visiting** changes their output and use case.  
They form the foundation for **tree-based algorithms, expression evaluation, and data structure operations.**

窗体底端