

# Name: Tahir Fareed

# Roll No: SU92-BSAIM-F24-036

# Section: BSAI (3A)

# Subject: AI Lab

# Task : 05

Artificial Intelligence Lab 5

**1. Breadth First Search (BFS)**

Breadth First Search (BFS) is a graph traversal algorithm that explores all the neighbors of a node before moving to the next level neighbors. It uses a queue to keep track of nodes to visit.

## Algorithm:

1. Start with the root node and mark it as visited.  
2. Place the root node in a queue.  
3. While the queue is not empty:  
 a. Dequeue a node from the queue.  
 b. Visit all unvisited neighbors of this node.  
 c. Mark them visited and enqueue them.

## Python Implementation:

import collections  
  
def bfs(graph, root):  
 visited, queue = set(), collections.deque([root])  
 visited.add(root)  
  
 while queue:  
 vertex = queue.popleft()  
 print(str(vertex) + ' ', end='')  
  
 for neighbour in graph[vertex]:  
 if neighbour not in visited:  
 visited.add(neighbour)  
 queue.append(neighbour)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 graph = {0: [1, 2], 1: [2], 2: [3], 3: [1, 2]}  
 print('Following is Breadth First Traversal:')  
 bfs(graph, 0)

## Explanation:

In this code, a graph is represented as a dictionary. BFS starts from the root node, visits it, then explores its neighbors level by level. The 'collections.deque' is used to implement the queue efficiently.

# 2. Depth First Search (DFS)

Depth First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It can be implemented using recursion or an explicit stack.

## Algorithm:

1. Start with the root node and mark it as visited.  
2. Explore an unvisited neighbor and mark it as visited.  
3. Recursively continue until no more unvisited neighbors remain.  
4. Backtrack to previous nodes to explore other branches.

## Python Implementation:

def dfs(graph, start, visited=None):  
 if visited is None:  
 visited = set()  
 visited.add(start)  
 print(start)  
  
 for next in graph[start] - visited:  
 dfs(graph, next, visited)  
 return visited  
  
graph = {  
 '0': set(['1', '2']),  
 '1': set(['0', '3', '4']),  
 '2': set(['0']),  
 '3': set(['1']),  
 '4': set(['2', '3'])  
}  
dfs(graph, '0')

## Explanation:

In this DFS code, recursion is used. The algorithm starts from a given node, marks it as visited, and then recursively visits all unvisited neighbors. The traversal continues deep into the graph before backtracking.

**Question:2.**

**Research about Tree Traversals (Inorder, Preorder, Postorder) and implement them using DFS.**

**Answer :**

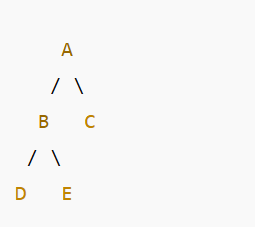
In computer science, tree traversal refers to the process of visiting every node in a tree structure exactly once, in some order. For binary trees (each node has at most a left and a right child), three very common depth-first traversal orders are:

1. Preorder
2. Inorder
3. Postorder

These are all forms of DFS (Depth-First Search) on a tree.

The difference among them is when you “visit” (process) the current node: whether before recursing on children, between, or after.

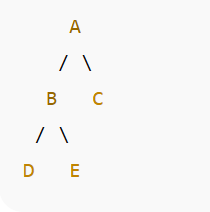
**Preorder Traversal (Root → Left → Right)**

* Order: Visit the root node first, then traverse the left subtree, then traverse the right subtree.
* In pseudocode:
* Preorder(node):
* if node is null:
* return
* visit(node)
* Preorder(node.left)
* Preorder(node.right)
* Use-cases / intuition:
  + When you want to process the parent node before its children (top-down).
  + Useful for serializing a tree, or cloning a tree (because you record structure as you go).
  + In expression trees, gives prefix (Polish notation) if the node values are operators / operands
  + **Example:**
  + ****

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

* Order: Traverse the left subtree first, then visit the root, then traverse the right subtree.
* Pseudocode:
* Inorder(node):
* if node is null:
* return
* Inorder(node.left)
* visit(node)
* Inorder(node.right)
* Use-cases / intuition:
  + In a binary search tree (BST), inorder traversal returns the nodes in sorted (ascending) order. Because for each node, all nodes in its left subtree are smaller, and those in the right subtree are larger.
  + In expression trees, yields infix notation (with parentheses if needed).
  + Good when you want a “middle” visit — i.e. treat left subtree first, then the node, then right subtree.

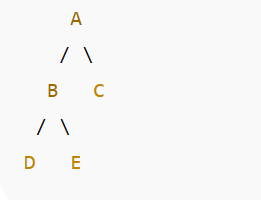
**Example:**

****

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

* Order: Traverse the left subtree, then the right subtree, and finally visit the root node.
* Pseudocode:
* Postorder(node):
* if node is null:
* return
* Postorder(node.left)
* Postorder(node.right)
* visit(node)
* Use-cases / intuition:
  + When you want to process children before their parent (bottom-up).
  + Useful in deleting or freeing a tree: delete children first, then the parent.
  + In expression trees, gives postfix (Reverse Polish Notation).

**Example:**

****