Artificial Intelligence

Department of Computer Science

University of Engineering and Technology, Lahore

LAB-06
DFS

CLO-01& CLO3

CLO	0	1	2	3	4
CLO1,3	0 for no problem solved	4.0 Marks for problem 1 solved	4.0 Marks for problem 2 solved	6.0 Marks for problem 2 solved	6.0 Marks for problem 2 solved

Class Learning Outcomes

- Students will clearly understand the implementation of Depth First Search in Python.

1. Concept Map

1.1Depth First Search

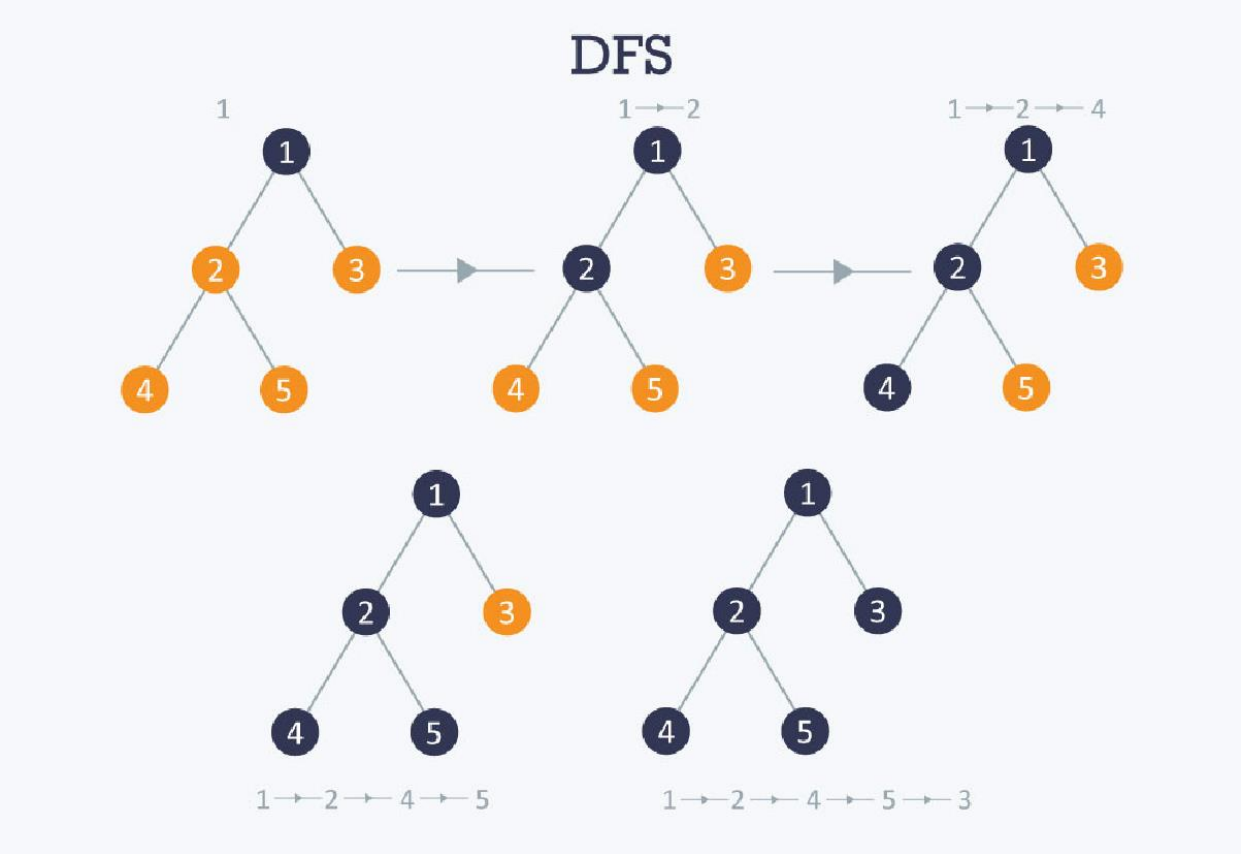
Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

The basic idea is as follows:

- Pick a starting node and push all its adjacent nodes into a stack.
- Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.
- Repeat this process until the stack is empty.
- However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once.
- If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

Pseudocode

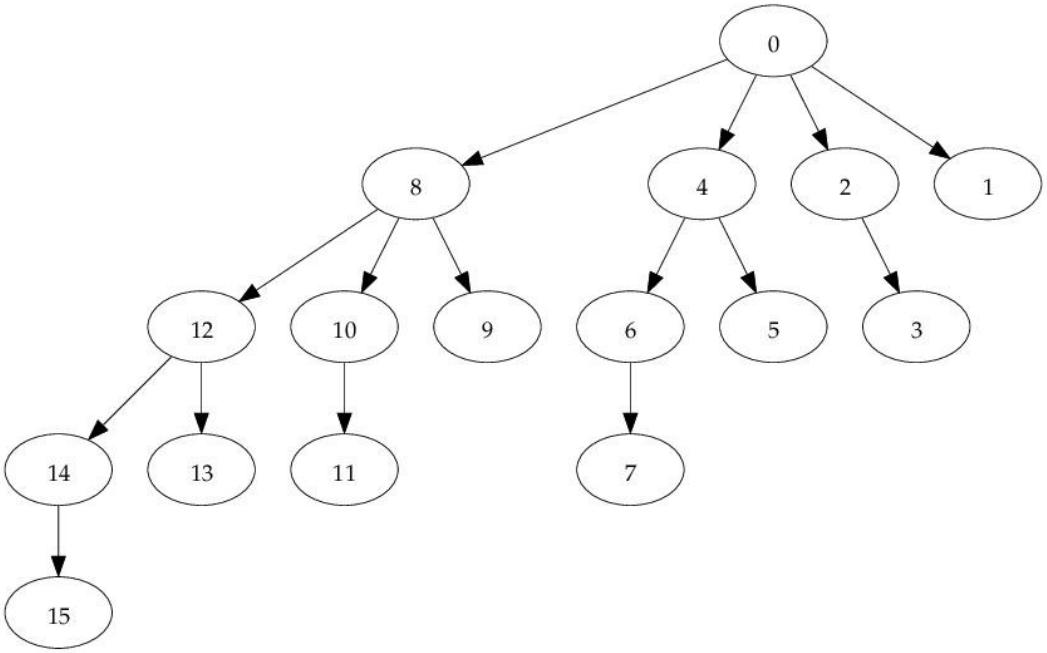
```
DFS-iterative (G, s):                                    //Where G is graph and s is source vertex
  let S be stack
  S.push( s )              //Inserting s in stack
  mark s as visited.
  while ( S is not empty):
      //Pop a vertex from stack to visit next
      v  =  S.top( )
     S.pop( )
     //Push all the neighbours of v in stack that are not visited
    for all neighbours w of v in Graph G:
        if w is not visited :
                 S.push( w )
                mark w as visited


DFS-recursive(G, s):
    mark s as visited
    for all neighbours w of s in Graph G:
        if w is not visited:
            DFS-recursive(G, w)
```



Exercise Questions:

1.1 Depth First Search using Stack.

You are given the following tree whose starting node is 0 and the Goal node is 7. You are required to implement the following graph in Python and then apply the following search a) Depth First Search using Stack.

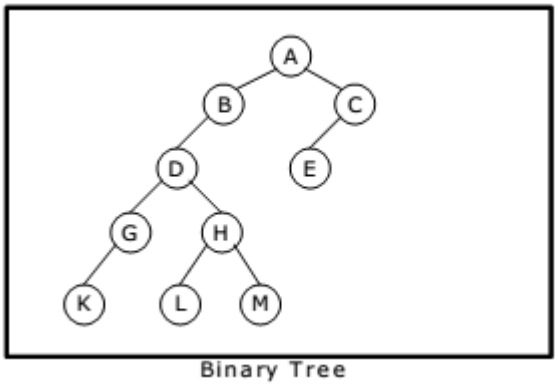


1.2 Binary Tree Traversal

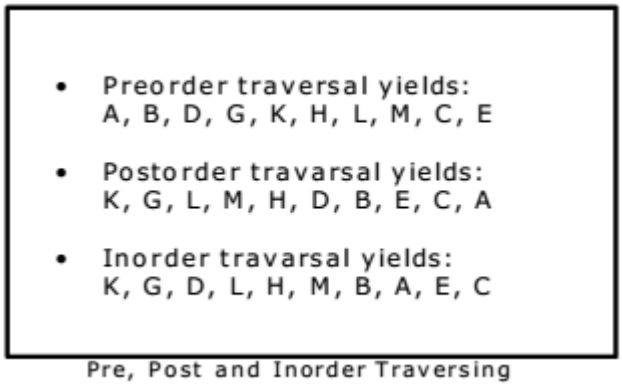
Problem: Given a binary tree, perform an in-order, pre-order, or post-order traversal.

Approach:

- Use recursion or an explicit stack.
- For in-order: Visit left subtree, root, right subtree.
- For pre-order: Visit root, left subtree, right subtree.
- For post-order: Visit left subtree, right subtree, root.



Binary Tree



Pre, Post and Inorder Traversing

1.3 Word Search

The **Word Search** problem involves determining if a given word can be formed by sequentially adjacent cells in a 2D grid of characters. You can move horizontally or vertically to form the word, and you cannot reuse the same cell in a single word formation.

Problem Statement
Given a 2D board of characters and a string word, return true if the word exists in the grid and false otherwise.

Example
Input:

Input	Output
board = [['A', 'B', 'C', 'E'], ['S', 'F', 'C', 'S'], ['A', 'D', 'E', 'E']] word = "ABCCED"	True

Approach

1. **DFS Exploration**:
    - Start from each cell in the grid and initiate a DFS search.
    - If the first letter of the word matches the character in the cell, proceed to check the adjacent cells.
    - Use a visited marker to keep track of the cells already included in the current path to avoid reusing them.
2. **Boundary Conditions**:
    - Ensure that you do not go out of the grid bounds or revisit a cell.
    - Check if the entire word has been matched successfully.
3. **Backtracking**:
    - If a cell does not lead to a valid path for the word, backtrack and mark it unvisited.

1.4 N-Queens problem

The N-Queens problem is a classic problem in computer science and combinatorial optimization. The task is to place N queens on an N×N chessboard such that no two queens threaten each other, meaning:

- No two queens share the same row.
- No two queens share the same column.
- No two queens are on the same diagonal.

DFS Approach for N-Queens:
DFS can be used to explore all possible placements of queens, with backtracking when an invalid placement is encountered.
Key Idea:

- We explore each row one by one.
- In each row, we try placing a queen in every column.
- Before placing the queen, we check whether placing it would violate any of the constraints (same column or diagonal).
- If placing the queen is valid, we move to the next row and repeat the process.
- If we successfully place queens in all rows, we have found a valid solution.

Input	Output
Number of queens = n = 4	.Q.. ...Q Q... ..Q. ..Q. Q... ...Q .Q..

Helping Materials

-

1. Helpful Links

https://www.tutorialspoint.com/prolog_in_artificial_intelligence/index.asp
https://stackoverflow.com/questions/27065774/depth-first-search-algorithm-prolog

2. Implementation of DFS in C++

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

// Graph class represents a directed graph
// using adjacency list representation
class Graph
{
    int V;    // No. of vertices

    // Pointer to an array containing
    // adjacency lists
    list<int> *adj;

    // A recursive function used by DFS
    void DFSUtil(int v, bool visited[]);
public:
    Graph(int V);   // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // DFS traversal of the vertices
    // reachable from v
    void DFS(int v);
};
```

```
Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and
    // print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent
    // to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}
void Graph::DFS(int v)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function
    // to print DFS traversal
    DFSUtil(v, visited);
}
```

...

```
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Depth First Traversal"
            " (starting from vertex 2) \n";
    g.DFS(2);

    return 0;
}
```