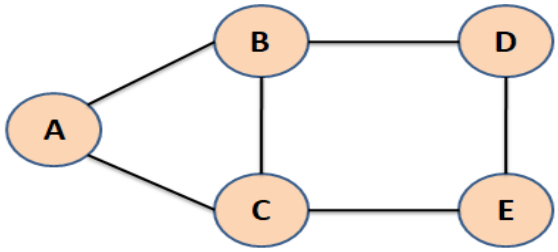
Program 1

Write a Program to implement Depth First Search (DFS) using Python.



Description:

Breadth-First Search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the root (or an arbitrary node in the case of a graph) and explores all the neighbor nodes at the present depth level before moving on to nodes at the next depth level. This means BFS explores the graph in layers, level by level.

Key Characteristics:

- **Layered Exploration:** BFS explores nodes in levels, where it first explores all nodes at the current level before moving on to the next level.
- **Queue-based:** BFS uses a queue data structure to keep track of nodes to visit next. The queue ensures nodes are explored in the correct order: first-in, first-out (FIFO).
- **Shortest Path:** In an unweighted graph, BFS guarantees the shortest path (in terms of number of edges) from the starting node to any other node.

BFS Algorithm - Step-by-Step:

1. **Initialization:**
   - Mark the starting node as visited.
   - Add the starting node to a queue.
2. **Exploration:**
   - While the queue is not empty:
     - Dequeue a node from the front of the queue.
     - Process the node (e.g., print it or store its value).
     - Visit all its unvisited neighbors:
       - Mark them as visited.
       - Enqueue them into the queue.
3. **Repeat:**
   - Continue the process until all reachable nodes have been visited.

SOURCE CODE :

```
# Input Graph
graph = {
'A' : ['B','C'],
'B' : ['A','C','D'],
'C' : ['A','B','E'],
'D' : ['B','E'],
'E' : ['C','D']
}
# To store visited nodes.
visitedNodes = []
# To store nodes in queue
queueNodes = []
# function
def bfs(visitedNodes, graph, snode):
	visitedNodes.append(snode)
	queueNodes.append(snode)
	print()
	print("RESULT :")
	while queueNodes:
		s = queueNodes.pop(0)
		print (s, end = " ")
		for neighbour in graph[s]:
			if neighbour not in visitedNodes:
				visitedNodes.append(neighbour)
				queueNodes.append(neighbour)

# Main Code
snode = input("Enter Starting Node(A, B, C, D, or E) :").upper()
# calling bfs function
bfs(visitedNodes, graph, snode)
```

Output:

Sample Output 1:

```
-------------------
Enter Starting Node(A, B, C, D, or E) :A

RESULT :
A B C D E
```

Sample Output 2:

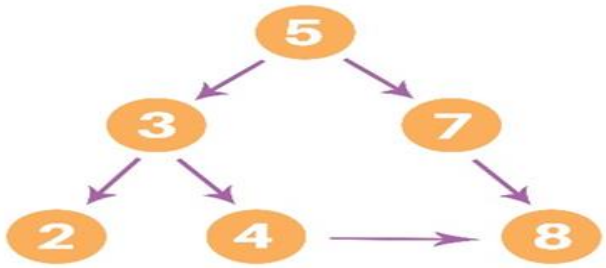
```
-------------------
Enter Starting Node(A, B, C, D, or E) :B

RESULT :
B A C D E
```

Program 2

Write a Program to implement Depth First Search (DFS) using Python.

Input Tree:



Depth First Search (DFS) - Description

Depth First Search (DFS**)** is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It starts at the root (or an arbitrary node in the case of a graph) and explores each branch of the graph deeply before visiting sibling nodes.

Key Characteristics:

- **Recursive Nature:** DFS explores one branch of the graph as deeply as possible before backtracking to explore other branches.
- **Stack-based:** DFS can be implemented using a stack data structure, either explicitly (using a stack) or implicitly (using recursion).
- **Backtracking:** If a node has no more unvisited neighbors, DFS backtracks to the most recent node that has unexplored neighbors.

DFS Algorithm - Step-by-Step

1. **Initialization:**
    - Mark the starting node as visited.
    - Push the starting node to a stack (or call the recursive function).
2. **Exploration:**
    - While the stack is not empty:
        - Pop a node from the stack.
        - Process the node (e.g., print it or store its value).
        - Visit all its unvisited neighbors:
            - Mark them as visited.
            - Push them onto the stack.
3. **Repeat:**Continue until all reachable nodes have been visited.

SOURCE CODE :

```
tree = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = set() # Set to keep track of visited nodes of tree.

def dfs(visited, tree, node):  #function for dfs
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in tree[node]:
            dfs(visited, tree, neighbour)

# Driver Code
node = input("Enter Starting Node(5, 3, 2, 4,7, or 8) :")
print("Following is the Depth-First Search")
dfs(visited, tree, node)
```

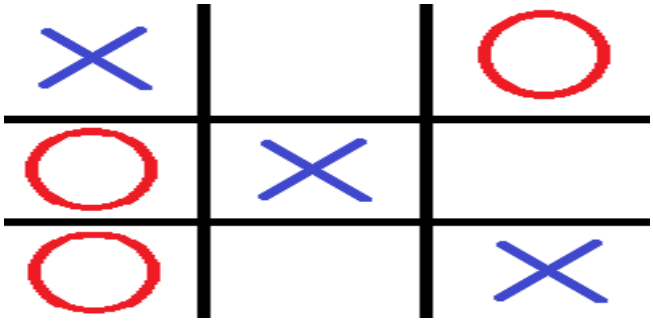
OUTPUT :

Sample Output 1:

```
Enter Starting Node(5, 3, 2, 4,7, or 8) :5
Following is the Depth-First Search
5
3
2
4
8
7
```

Program 3

Write a Program to implement Tic-Tac-Toe game using Python.



Tic-Tac-Toe Game Description:

Tic-Tac-Toe (also known as **Noughts and Crosses**) is a classic two-player board game where players take turns marking a 3x3 grid with their respective symbols (usually "X" and "O"). The goal is to get three of your symbols in a row (horizontally, vertically, or diagonally) before your opponent does.

In this description, we'll create a simple implementation of the Tic-Tac-Toe game in Python that allows two players to play interactively. The game will:

1. Display the current state of the board after each move.
2. Allow players to take turns marking spaces.
3. Check for a winner or a draw after every move.

Game Rules:

1. **Players**: Two players, one playing with "X" and the other with "O".
2. **Turns**: Players alternate turns.
3. **Victory Condition**: A player wins if they have three of their marks ("X" or "O") in a row, column, or diagonal.
4. **Draw**: If the board is full and there is no winner, the game ends in a draw.
5. **Board**: The game is played on a 3x3 grid, and the positions are numbered from 1 to 9 (1 being the top-left and 9 being the bottom-right).

SOURCE CODE :

```
 # Tuple to store winning positions.
win_positions = (
    (0, 1, 2), (3, 4, 5), (6, 7, 8),
    (0, 3, 6), (1, 4, 7), (2, 5, 8),
    (0, 4, 8), (2, 4, 6)
)
 def game(player):
    # diplay current mesh
    print("\n", " | ".join(mesh[:3]))
    print("---+---+---")
    print("", " | ".join(mesh[3:6]))
    print("---+---+---")
    print("", " | ".join(mesh[6:]))

    # Loop until player valid input cell number.
    while True:
        try:
            ch = int(input(f"Enter player {player}'s choice : "))
            if str(ch) not in mesh:
                raise ValueError
            mesh[ch-1] = player
            break
        except ValueError:
            print("Invalid position number.")
    # Return winning positions if player wins, else None.
    for wp in win_positions:
        if all(mesh[pos] == player for pos in wp):
            return wp
    return None
player1 = "X"
player2 = "O"
player = player1
mesh = list("123456789")
for i in range(9):
      won = game(player)
      if won:
            print("\n", " | ".join(mesh[:3]))
            print("---+---+---")
            print("", " | ".join(mesh[3:6]))
            print("---+---+---")
            print("", " | ".join(mesh[6:]))
            print(f"*** Player {player} won! ***")
            break
      player = player1 if player == player2 else player2
else:
    # 9 moves without a win is a draw.
    print("Game ends in a draw.")
```

OUTPUT :

Sample Output:

```
1 | 2 | 3
---+---+---
4 | 5 | 6
---+---+---
7 | 8 | 9
Enter player X's choice : 5

1 | 2 | 3
---+---+---
4 | X | 6
---+---+---
7 | 8 | 9
Enter player O's choice : 3

1 | 2 | O
---+---+---
4 | X | 6
---+---+---
7 | 8 | 9
Enter player X's choice : 1

X | 2 | O
---+---+---
4 | X | 6
---+---+---
7 | 8 | 9
Enter player O's choice : 6

X | 2 | O
---+---+---
4 | X | O
---+---+---
7 | 8 | 9
Enter player X's choice : 9
```

```
X | 2 | O
---+---+---
4 | X | O
---+---+---
7 | 8 | X
*** Player X won! ***
```