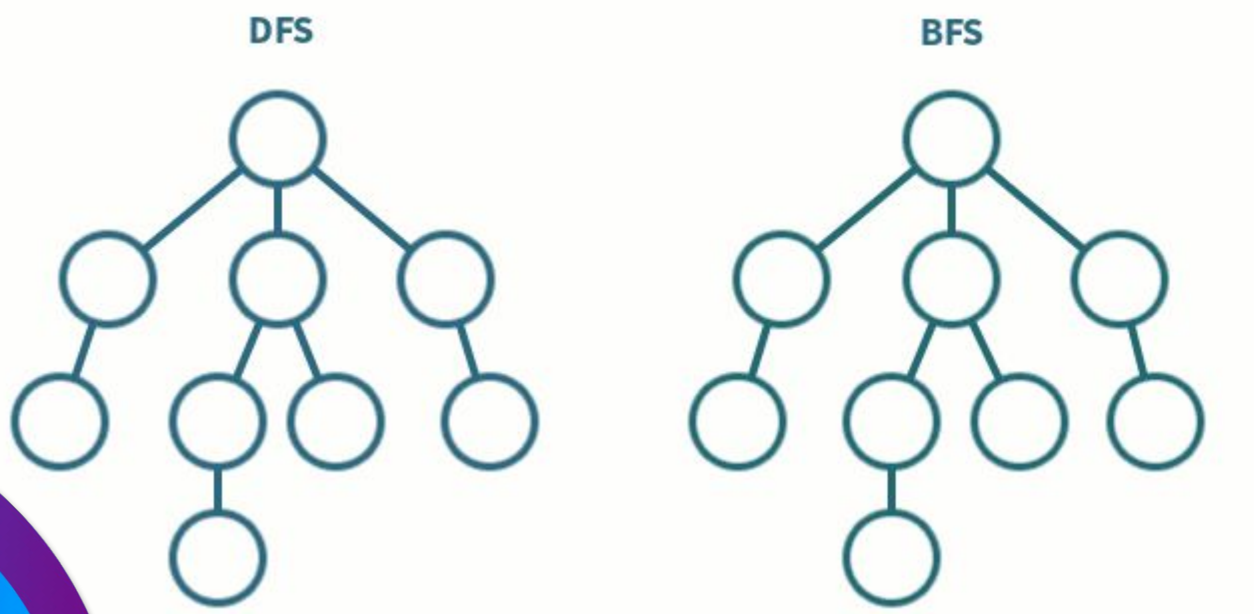
# BFS



BFS:

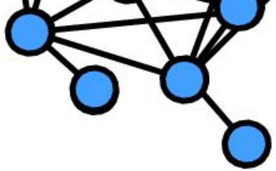
DFS:

BFS:

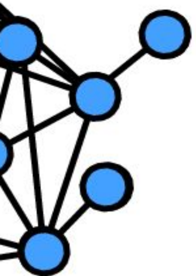
# Lecture Flow

1) Pre-requisites
2) Definition
3) Applications of BFS
4) BFS Variations
5) Practice questions
6) Quote of the day

# Pre-requisites



- Introduction to Graph
- Basic understanding of Queue Data Structure
- DFS graph traversal algorithm

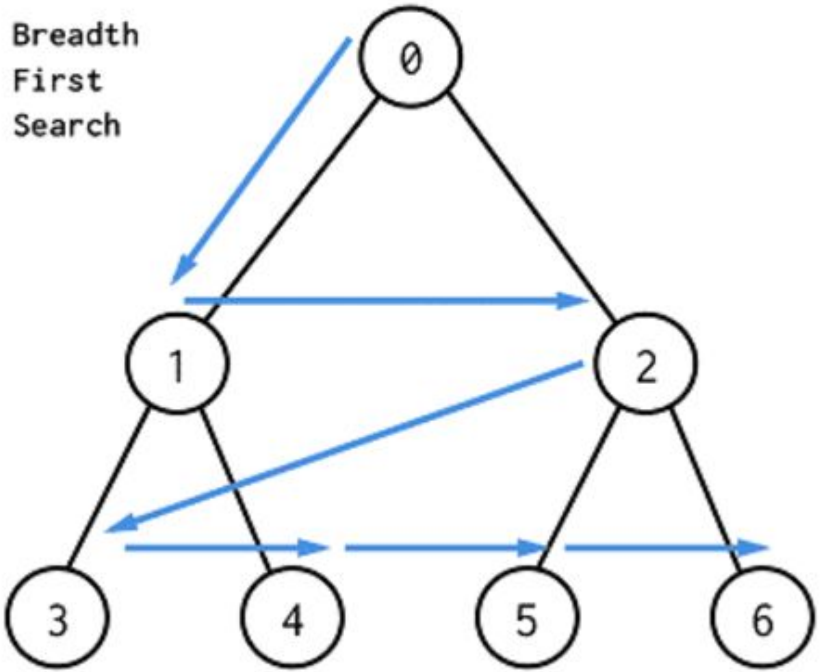


# Introduction

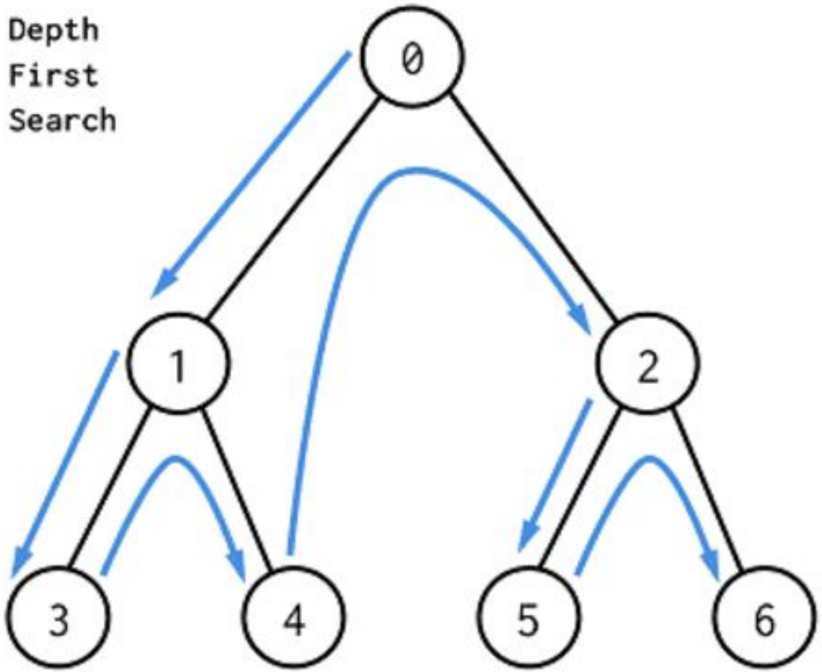
- **BFS (Breadth-First Search)** is a graph traversal algorithm that visits all the nodes of a graph in **breadth-first order.**

- It visits all the nodes at **a given level** before moving on to **the next level**.

- It starts at the root node and explores all the **neighboring nodes** before moving on to the next level.

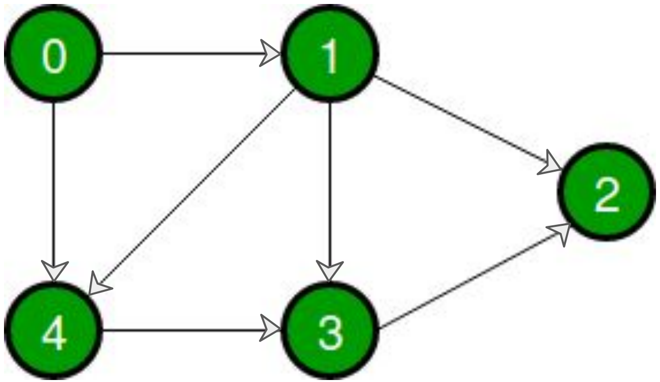


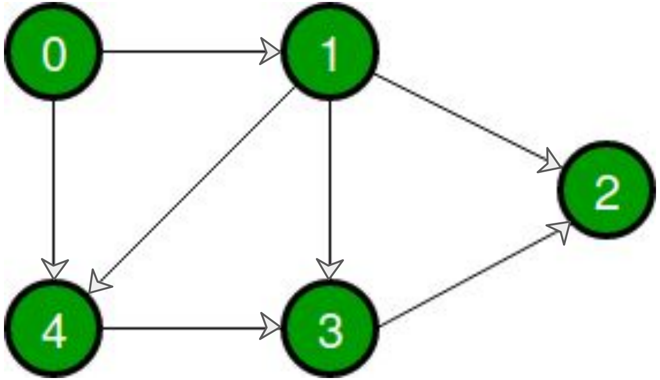

Output: 0, 1, 2, 3, 4, 5, 6




Output: 0, 1, 3, 4, 2, 5, 6

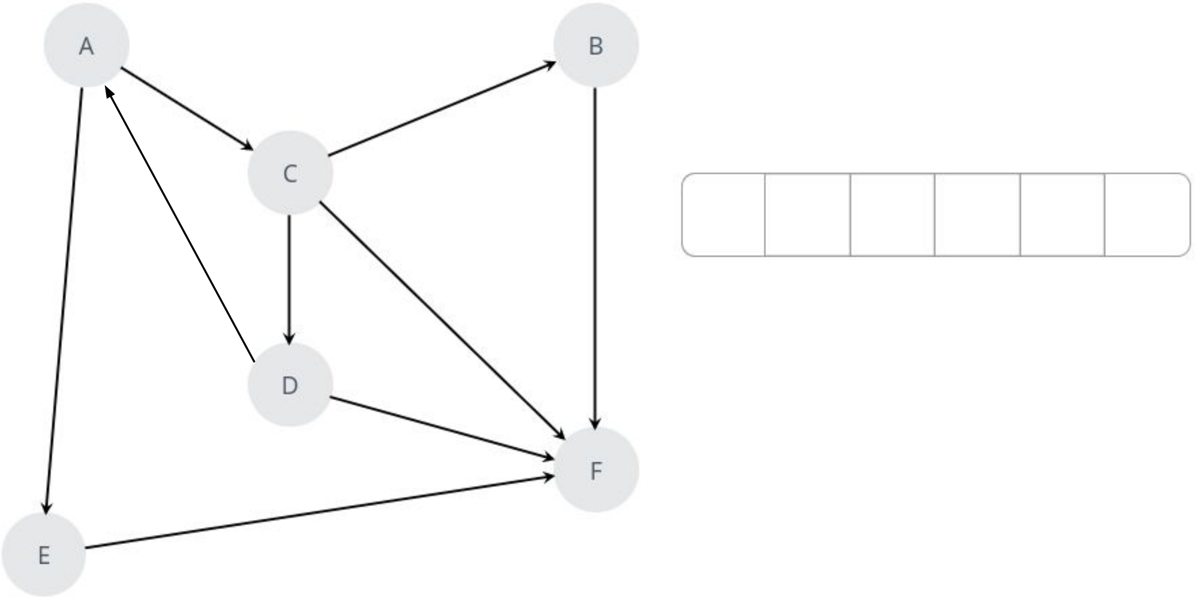
What is the bfs traversal of the graph if we start at 0?



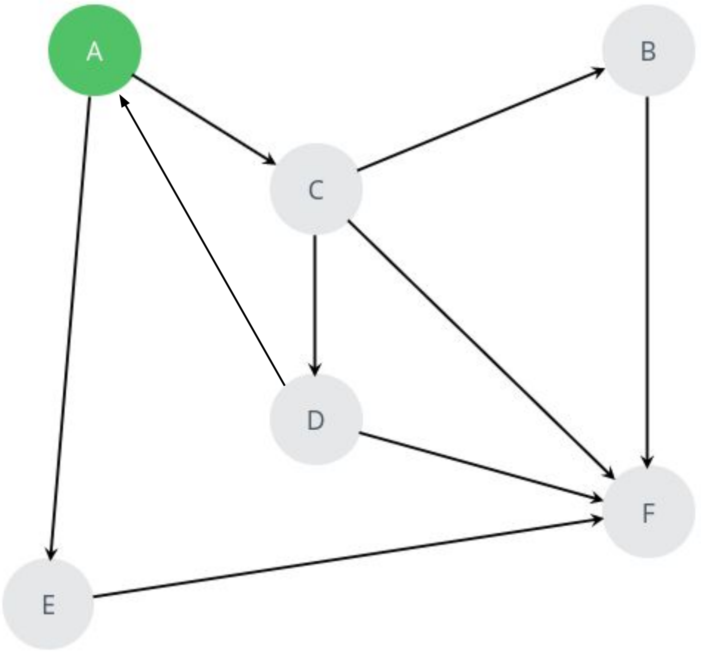



**Answer:** 0, 1, 4, 2, 3

# Visualization

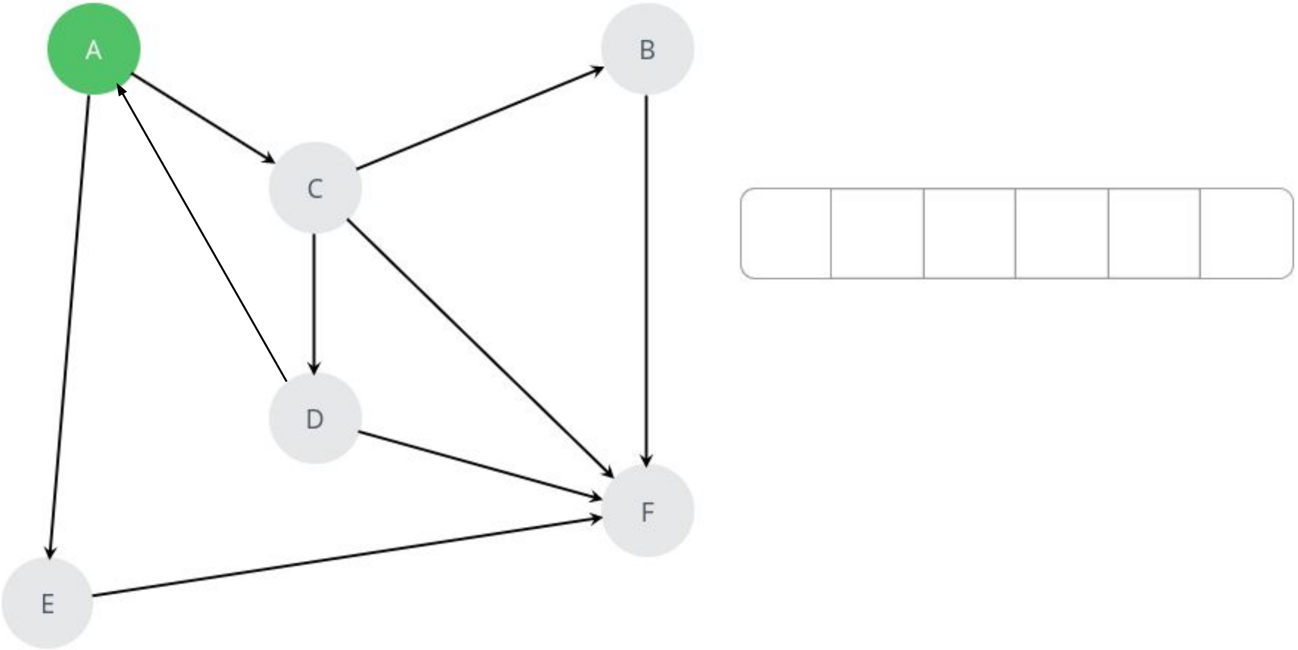



**Steps:**
Let us look at the details of how a breadth-first search works.



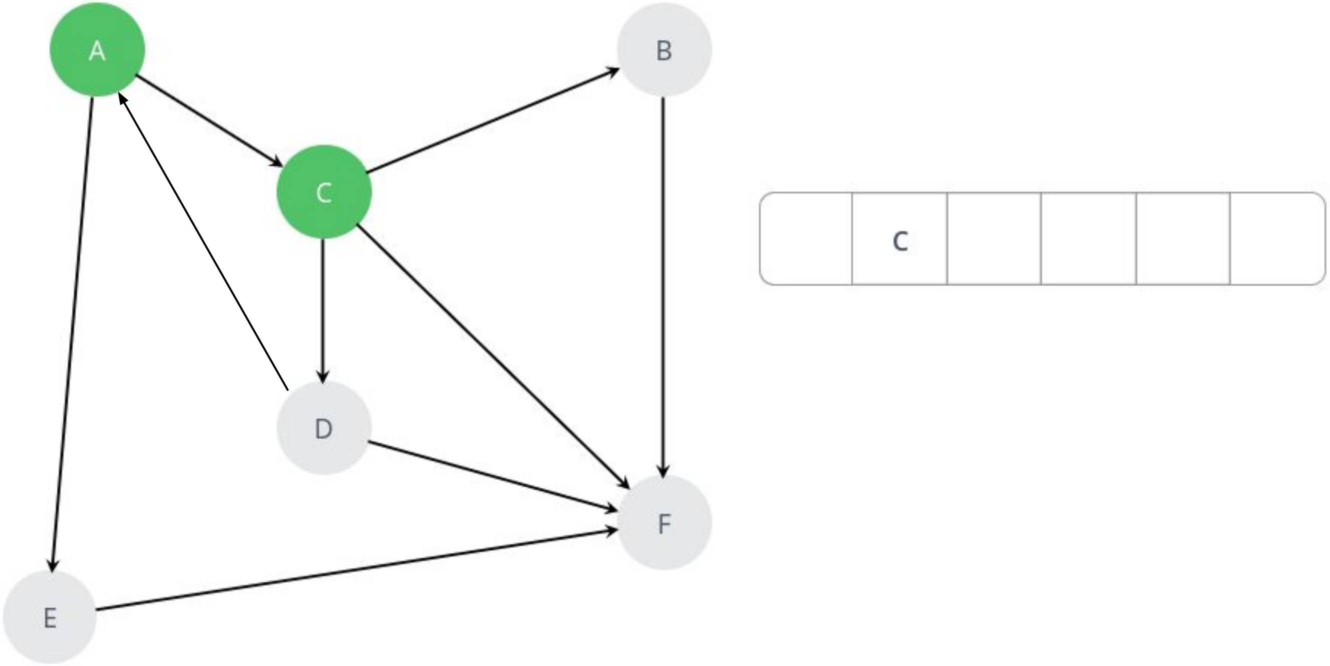

| | | | | | |
|---|---|---|---|---|---|
| A | | | | | |

**Steps:**
Mark and enqueue A

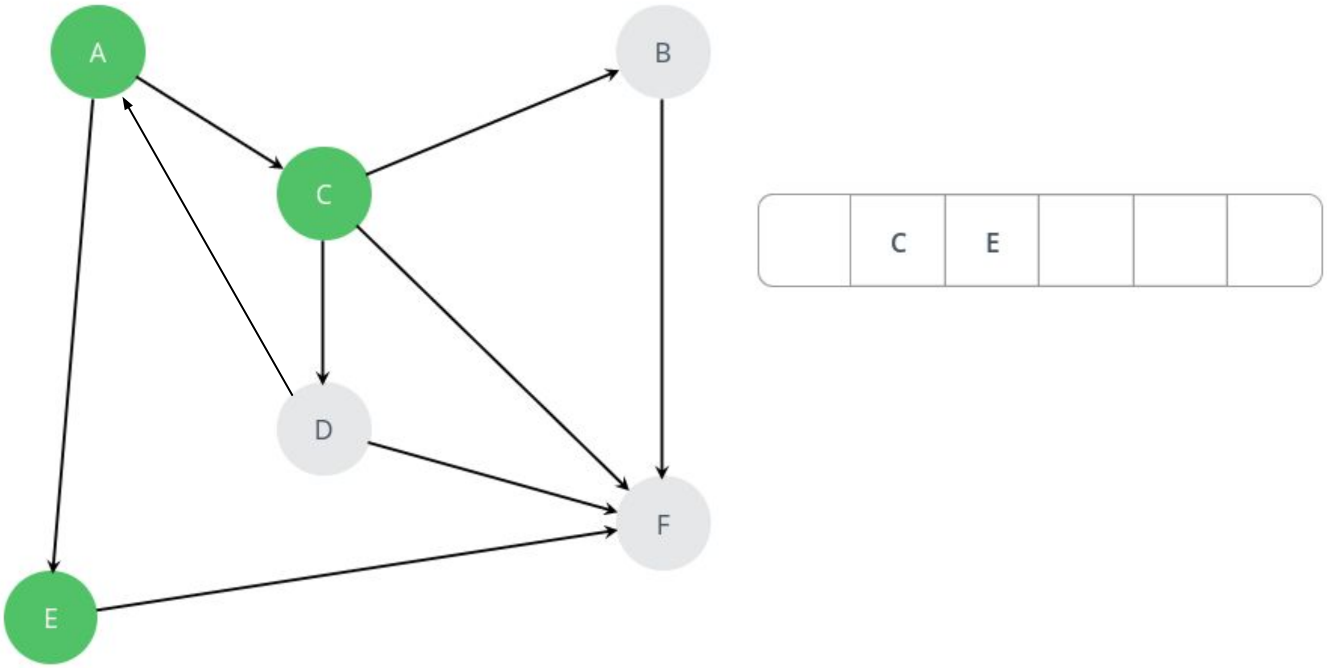



**Steps:**

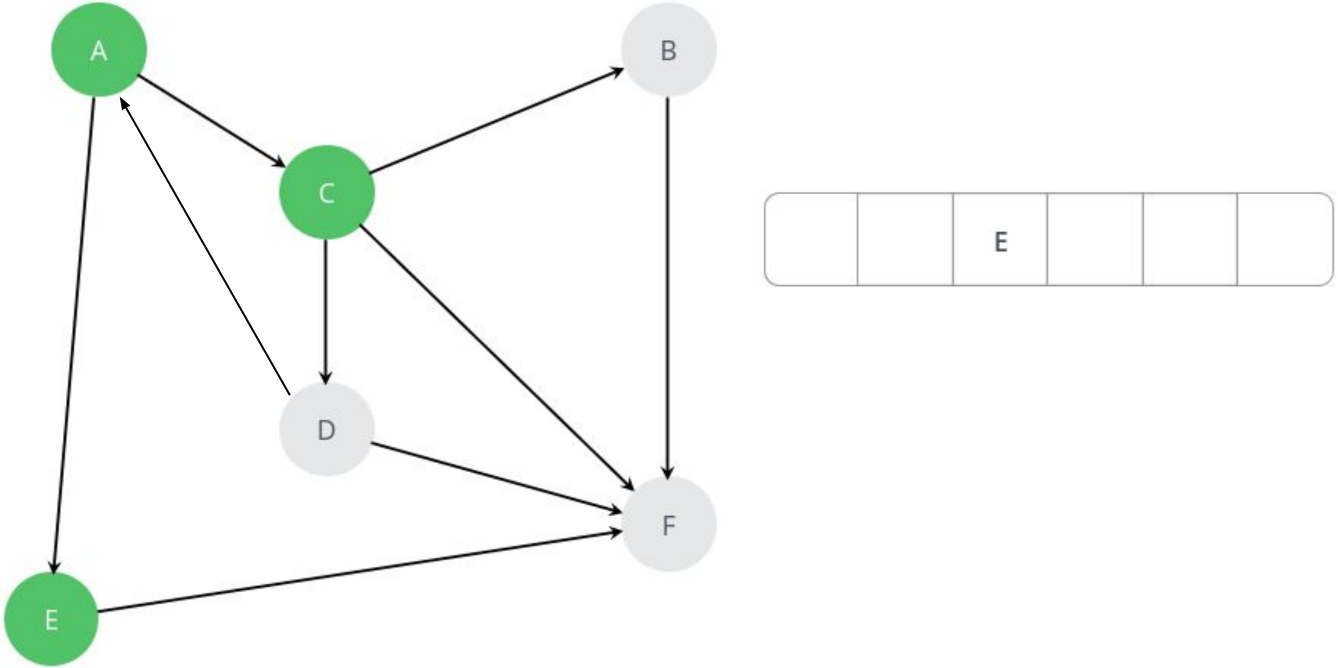
Dequeue A




**Steps:**
Mark and enqueue C

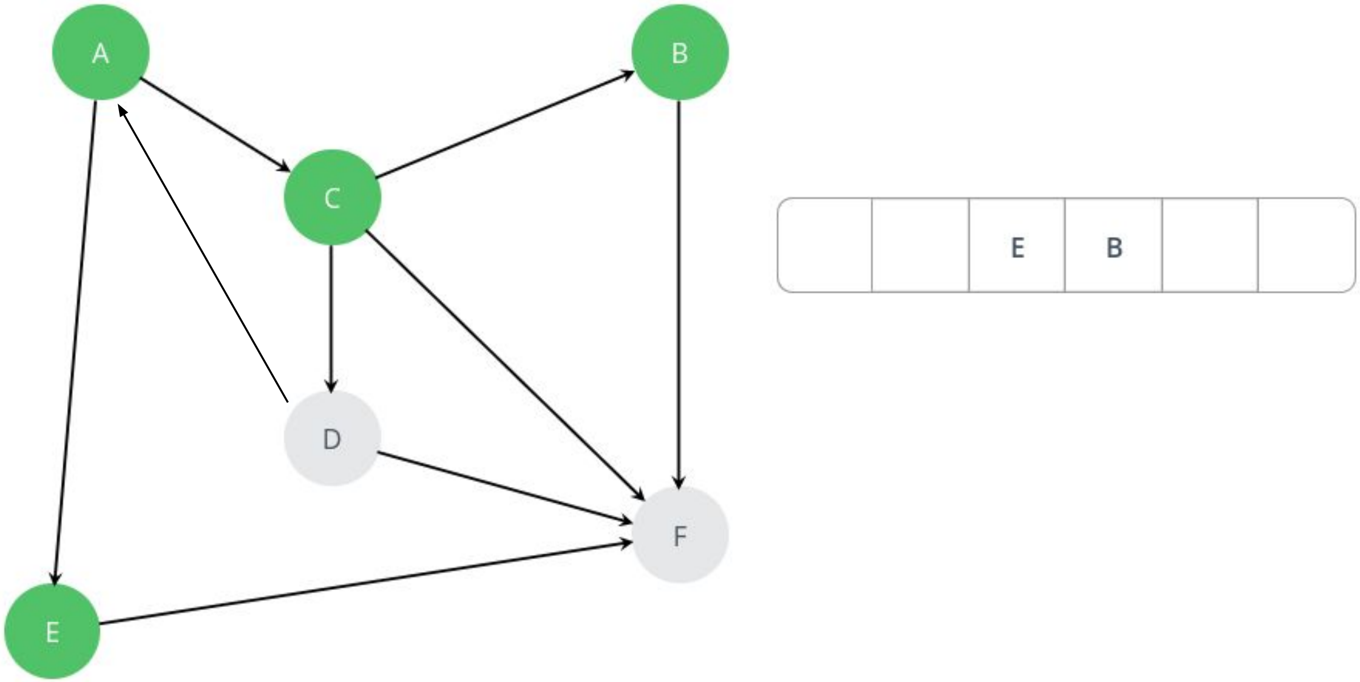



**Steps:**
Mark and enqueue E

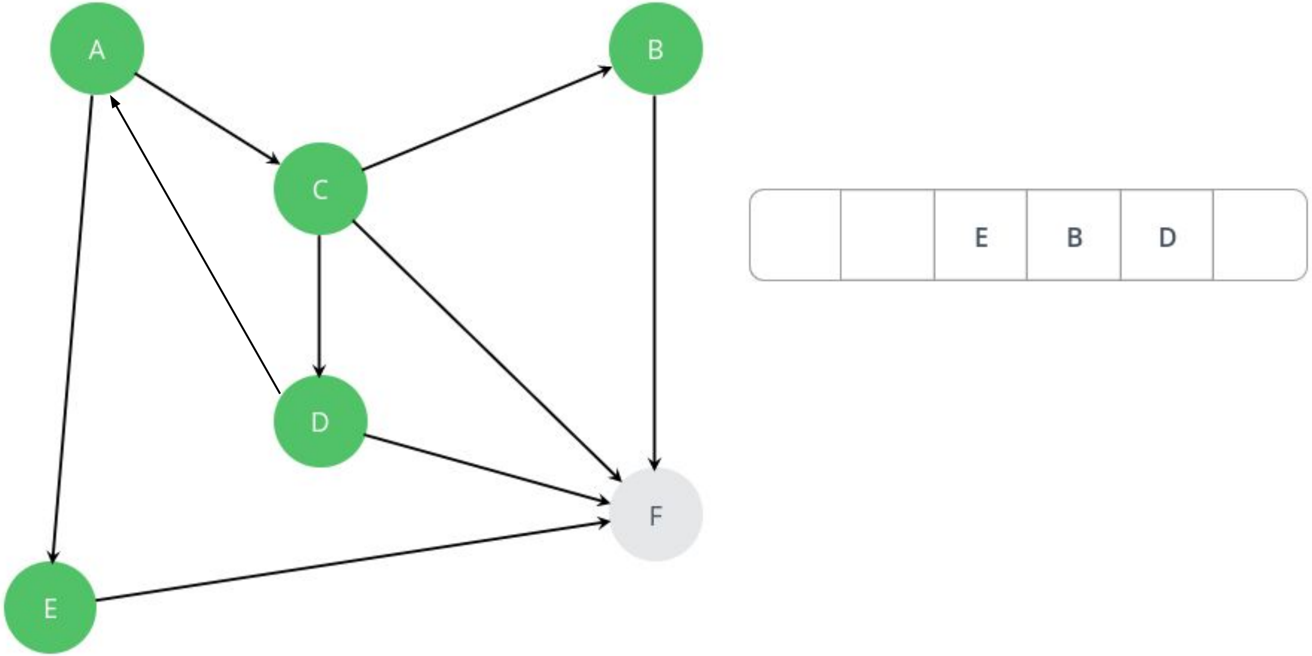



**Steps:**

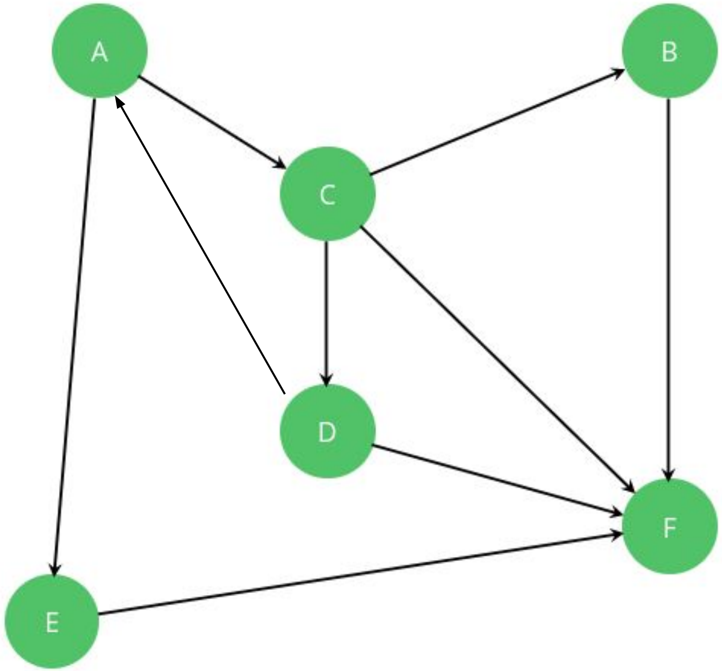
Dequeue C




**Steps:**
Mark and enqueue B



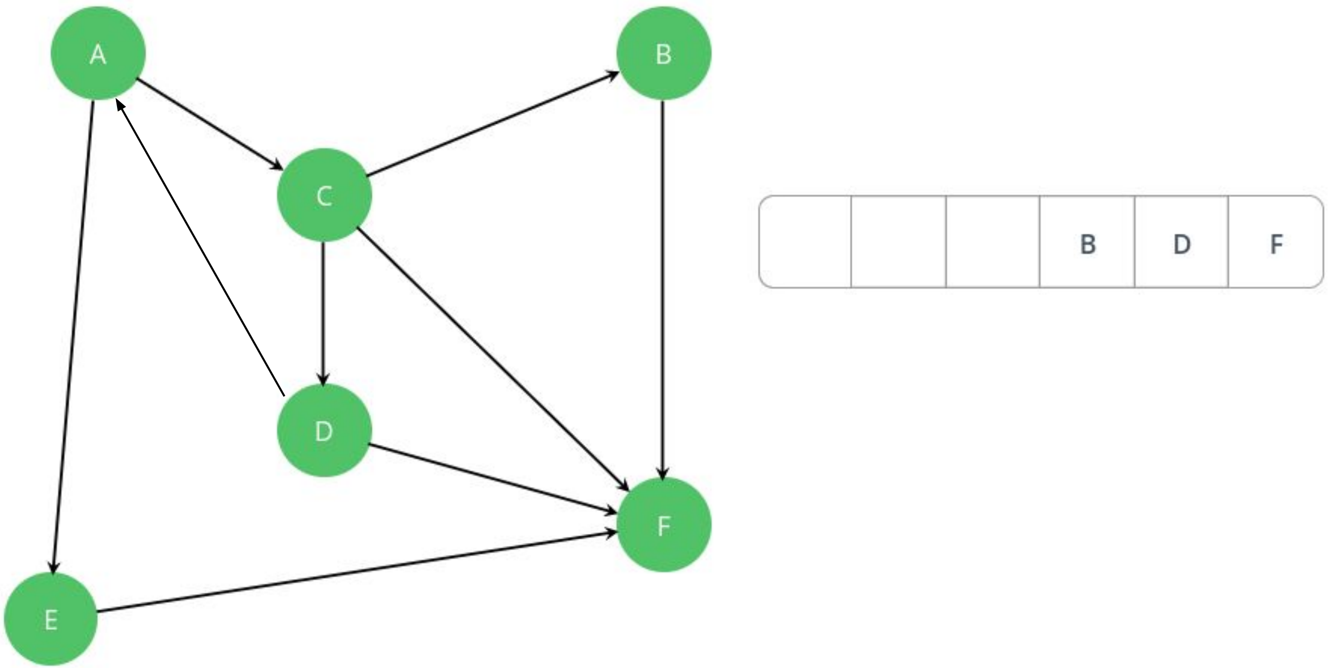

**Steps:**
Mark and enqueue D




| | | | | | |
|---|---|---|---|---|---|
| | | E | B | D | F |

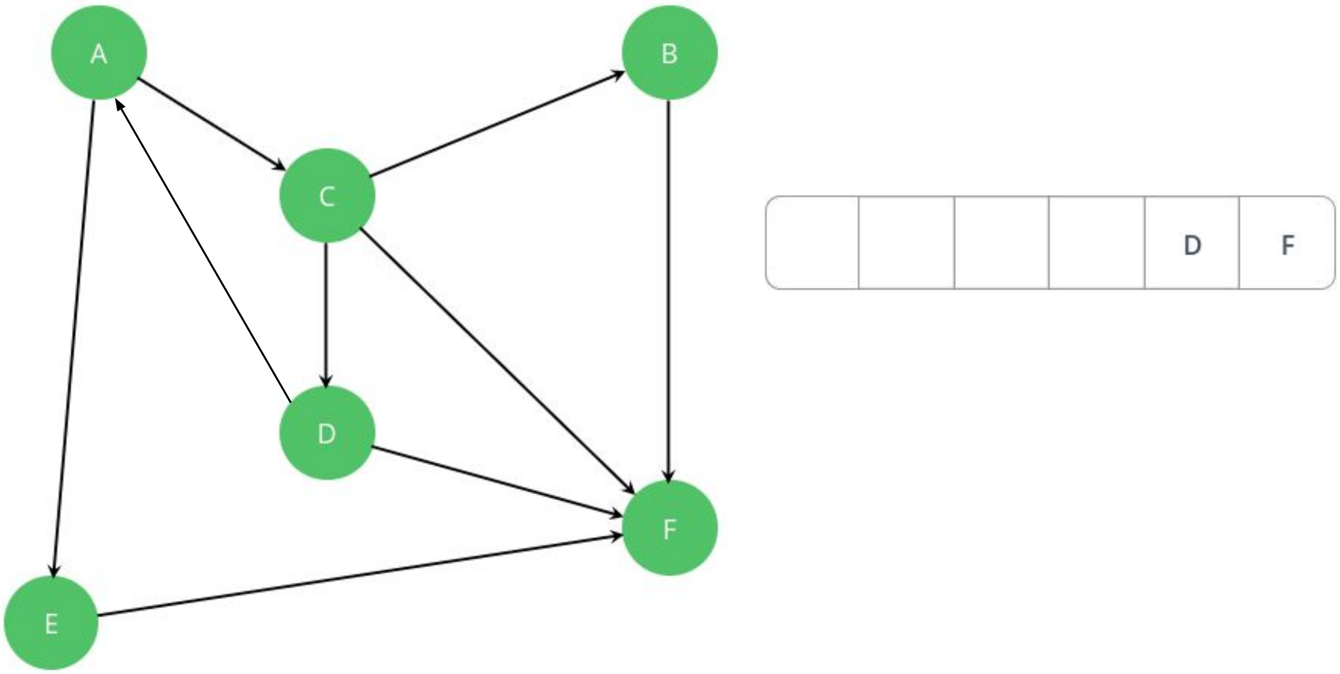
**Steps:**

Mark and enqueue F

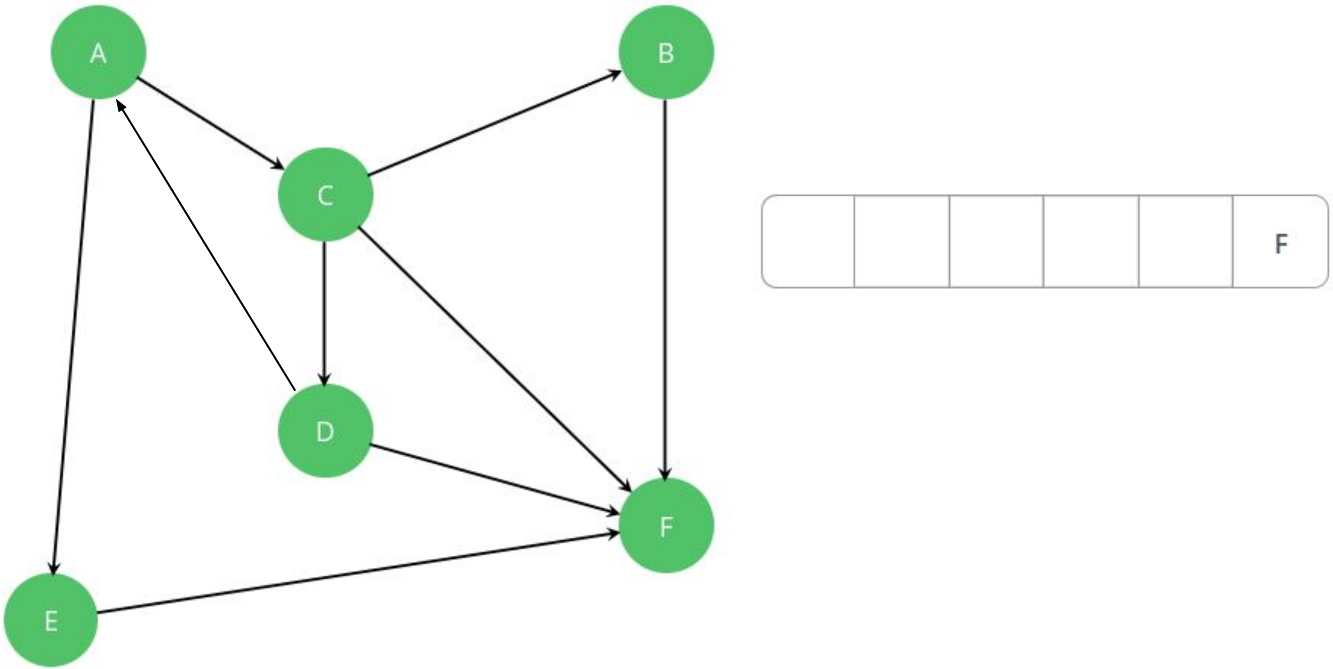



**Steps:**

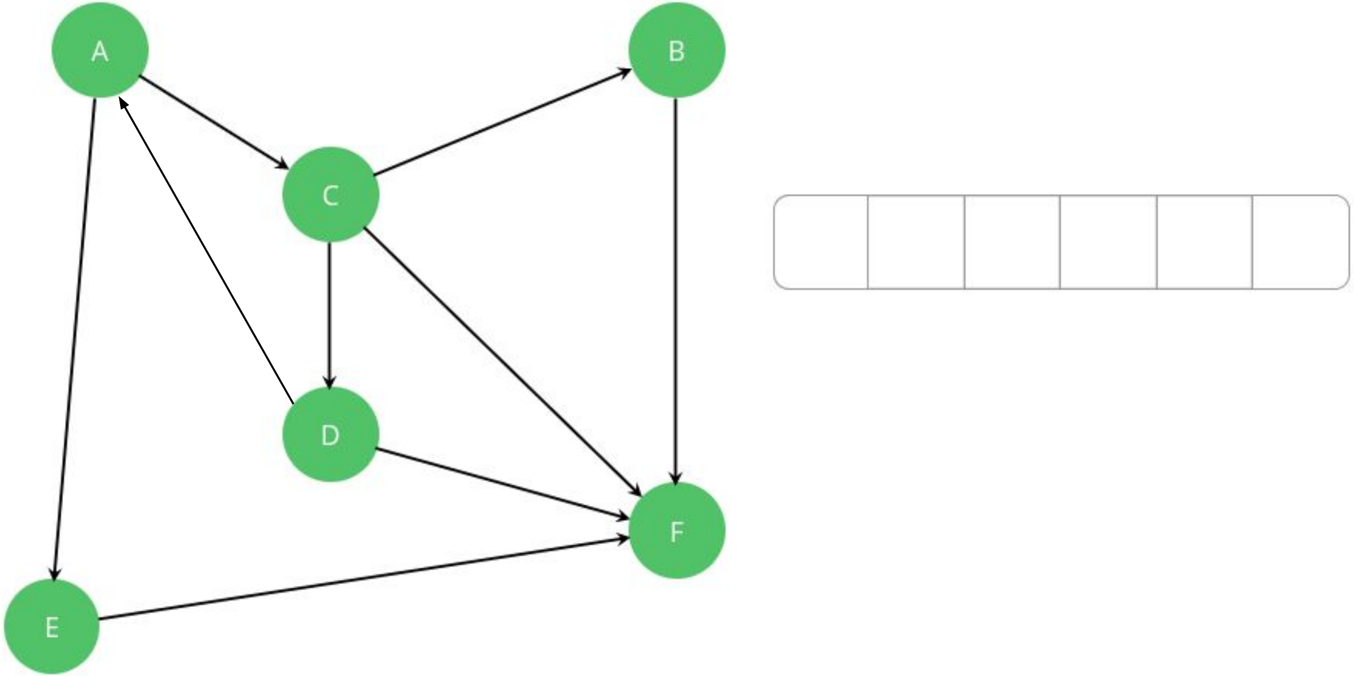
Dequeue E




**Steps:**
Dequeue B




**Steps:**
Dequeue D




**Steps:**
Dequeue F

```python
def bfs(graph, node) => list:
    visited = set([node])
    queue = deque([node])
    _list = []
    while queue:
        node = queue.popleft()
        _list.append(node)

        for neighbour in graph[node]:
            if neighbour not in visited:
                visited.add(neighbour)
                queue.append(neighbour)

    return _list
```

## Time Complexity

- We have **V nodes** and **E edges**
- We will only **visit a node once**, and if it's **a complete graph** we will also **use every edge**.

Time Complexity = **O (V + E)**

## Space Complexity

- We have **V nodes** and we only visit each node at most once.
- **the space complexity is just the space required to input them in the visited set.**

Space Complexity = **O (V)**

# Here is an example

# 965. Univalued Binary Tree

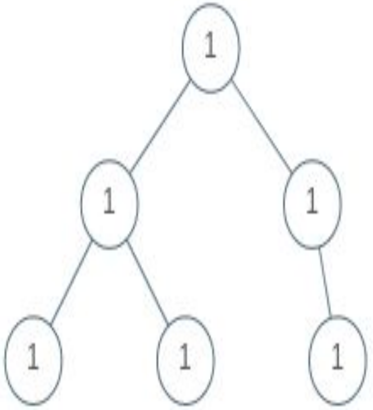
## 965. Univalued Binary Tree

Easy Topics Companies

A binary tree is **uni-valued** if every node in the tree has the same value.

Given the `root` of a binary tree, return `true` *if the given tree is* ***uni-valued****, or* `false` *otherwise.*

**Example 1:**




```
Input: root = [1,1,1,1,1,null,1]
Output: true
```

# Approach

1. Initialize a variable val who has the same value as the root
2. Initialize a queue with the root node.
3. Loop through each level of the tree( as long as queue is not empty):
    - Check if the popped node value is same as val
    - If it is not then return False immediately
    - Else Add the children of all nodes in the level to the queue.
4. Return True

# Implementation

```
def isUnivalTree(root):
        val =root.val
        queue=deque([root])

        While queue:
            node=queue.popleft()
            if node.val != val:
                return False
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        return True
```

## Time Complexity

- We have **V nodes and we visit each node exactly once.**
- Since the number of edges is constant for every node, we can say that **E = 2V**. Which we ignore.

Time Complexity = **O (V)**

## Space Complexity

- We store at most M nodes in the queue, where **M is the maximum number of nodes in a level** of the binary tree

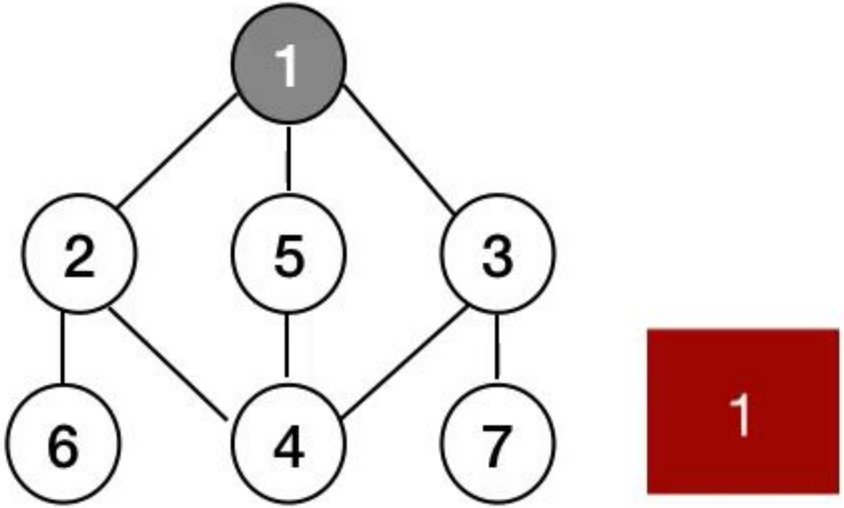
Space Complexity = **O (M)**

# Applications

# 1. Level Order Traversal

# Level order traversal

- BFS can be used for level order traversal by **visiting nodes based on their distance from the root node.**



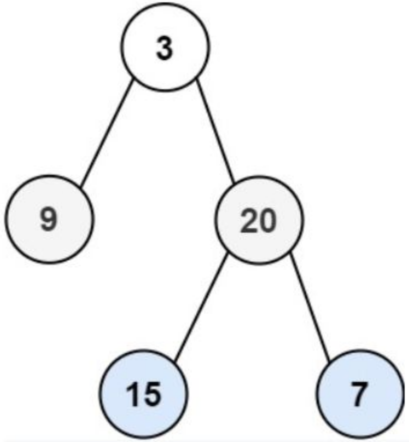
# Binary Tree Level Order Traversal

102. Binary Tree Level Order Traversal

Medium 12944 257 Add to List Share

Given the `root` of a binary tree, return *the level order traversal of its nodes' values*. (i.e., from left to right, level by level).

**Example 1:**




```
Input: root = [3,9,20,null,null,15,7]
Output: [[3],[9,20],[15,7]]
```

# Approach

1. **Initialize** your **queue**,**visited** set, **current level array**, **current level** you are currently at.
2. Check the **current level** with **global level**:
    - If They are **not equal**, **add** current level **array** to answer and set the current level **array** to **empty**.
3. **Add current** node to current level **array**.
4. **Traverse** through the **neighbours** and insert **neighbour's** with their **level** in to your queue.
5. Repeat step **2, 3 , 4** until you left with **empty queue.**

```python
def levelOrder(self, root):
    levels = []
    level = 0
    queue = deque([(root, 0)])
    currLevel = []

    while queue:
        node, nodeLevel = queue.popleft()

        if not node:
            continue

        if nodeLevel != level:
            levels.append(currLevel.copy())
            level += 1
            currLevel = []

        currLevel.append(node.val)
        queue.append((node.left, level + 1))
        queue.append((node.right, level + 1))

    if currLevel:
        levels.append(currLevel.copy())

    return levels
```

# Other Implementation/ Using for loop to track level

```
def levelOrder(self, root):
    levels = []
    queue = deque([root])

    while queue:

        levels.append([node.val for node in queue])

        for _ in range(len(queue)):

            node = queue.popleft()

            if node.left : queue.append(node.left)
            if node.right: queue.append(node.right)

    return levels
```

## Time Complexity

- We have **V nodes and we visit each node exactly once.**
- Since the number of edges is constant for every node, we can say that **E = 2V**. Which we ignore.

Time Complexity = **O (V)**

## Space Complexity

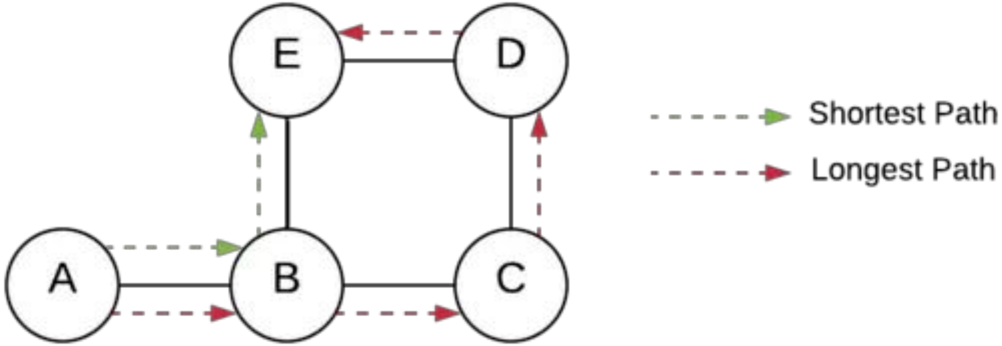
- We store at most M nodes in the queue, where **M is the maximum number of nodes in a level** of the binary tree

Space Complexity = **O (M)**

# 2.Finding shortest path in unweighted graph

# Problem

Given an unweighted undirected graph, we have to find the shortest path from the given source to the given destination using the Breadth-First Search algorithm.



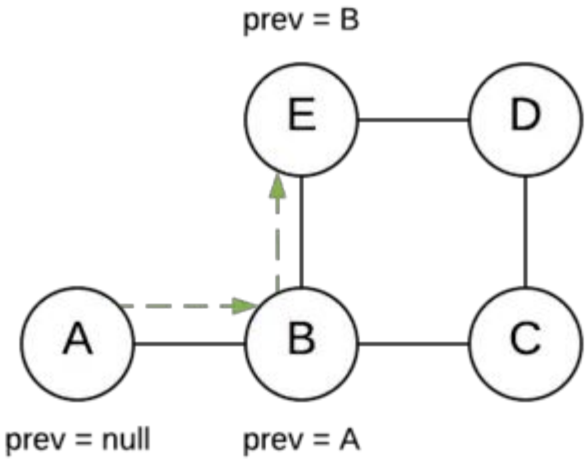
# Approach

- Use BFS to traverse a graph or tree.
- Compare visited nodes with the end node. Stop BFS when the end node is found.
- Use either a cleared queue or a boolean flag to mark the end of BFS.

# How to trace path from start to end node?

# Trace path

- We use **"prev"** array or dictionary to **store the reference** of the **preceding node.**
- Using the "prev" value, we can trace the route back from the end node to the starting node.



# The shortest path

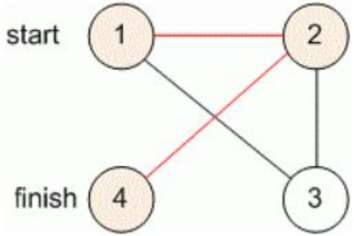
The undirected graph is given. Find the shortest path from vertex **a** to vertex **b**.

## Input data

The first line contains two integers **n** and **m** (**1 ≤ n ≤ 5** * `10^4`, **1 ≤ m ≤** `10^5`) - the number of vertices and edges. The second line contains two integers **a** and **b** - the starting and ending point correspondingly. Next **m** lines describe the edges.

## Output data

If the path between **a** and **b** does not exist, print **-1**. Otherwise print in the first line the length **l** of the shortest path between these two vertices in number of edges, and in the second line print **l + 1** numbers - the vertices of this path.



# Approach

1. In order to track the shortest path we can use **path tracing**. We can either maintain a **list** or a **dictionary**. Let's assume we're using a dictionary.
2. Once we start our traversal, while we are going through the **neighbors** of that certain node, we can make the **neighbors** a **key**, and their **parent** i.e. the current node the **value**.
3. Now if our target is **not** in our dictionary then we can return **-1**
4. Otherwise we can return the shortest path by **going all the way back** to the initial node.

```python
def shortestPath():
    graph = defaultdict(list)
    nodes, edges = map(int, input().split())
    source, target = map(int, input().split())
    prev = {source: -1}
    queue = deque([source])
    answer = []

    for _ in range(edges):
        node1, node2 = map(int, input().split())
        graph[node1].append(node2)
        graph[node2].append(node1)

    while  queue:
        node = queue.popleft()
        if node == target:
            break

        for neighbour in graph[node]:
            if neighbour not in prev:
                prev[neighbour] = node
                queue.append(neighbour)
```

```python
# this section is for printing the path, and comes after the
code on the left
if target not in prev:
    print(-1)
    return

current = target
answer = []

while  current != -1:
    answer.append(current)
    current = prev[current]

print(len(answer) - 1)
print(*answer[::-1])
```

## Time Complexity

- We have **V nodes** and **E edges**
- We will only **visit a node once**, and if it's **a complete graph** we will also **use every edge**.

  Time Complexity = **O (V + E)**

## Space Complexity

- We have **V nodes** and **E edges**
- We will store **the nodes**, and also **the edges**.

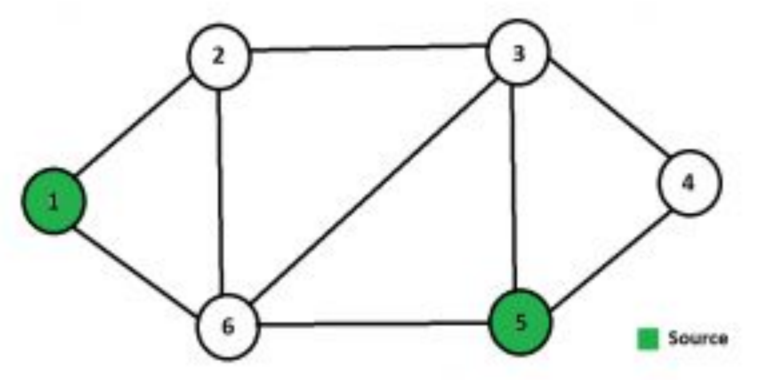
  Space Complexity = **O (V + E)**

# BFS Variations

# 1.Multi-Source BFS

# Multi-source BFS

- If we have **multiple starting locations** for our BFS, there is **nothing stopping** us from **appending** all those locations into our starting queue.



# Rotting Oranges

## 994. Rotting Oranges

Medium 10237 341 Add to List Share

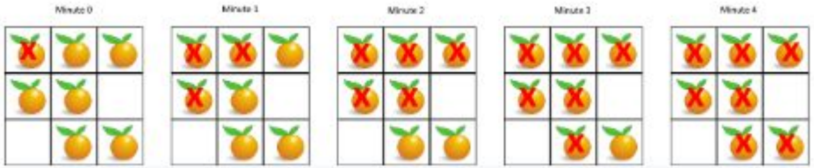
You are given an `m x n` `grid` where each cell can have one of three values:

- `0` representing an empty cell,
- `1` representing a fresh orange, or
- `2` representing a rotten orange.

Every minute, any fresh orange that is **4-directionally adjacent** to a rotten orange becomes rotten.

Return *the minimum number of minutes that must elapse until no cell has a fresh orange*. If *this is impossible, return* `-1`.

**Example 1:**



```
Input: grid = [[2,1,1],[1,1,0],[0,1,1]]
Output: 4
```

**Example 2:**

```
Input: grid = [[2,1,1],[0,1,1],[1,0,1]]
Output: -1
Explanation: The orange in the bottom left
corner (row 2, column 0) is never rotten,
because rotting only happens 4-directionally.
```

# Approach

1. **Initialize** your queue.
2. Traverse through the **grid and insert all rotten orange** indices in to your queue.
3. After the insertions, while we still have elements in our queue, and as long as we have remaining fresh oranges in our grid we will continue our iteration.
    a. In here we will go through **every element in our queue,** and we will pop the leftmost element.
    b. As long as that cell is not in bound and the grid is containing a fresh orange.
        i. We change that orange to rotten, and we **append the new grid** in to our queue.
    c. Once the above steps are completed then we can **decrease the number of fresh** oranges that we have.

**4.** Now once we are done with going through the queue we can return the minimum time taken to make all oranges in that grid rotten.

```python
def orangesRotting(self, grid):
        q = deque()
        time, fresh = 0, 0
        n, m = len(grid), len(grid[0])
        for r in range(n):
            for c in range(m):
                if grid[r][c] == 1:
                    fresh += 1
                if grid[r][c] == 2:
                    q.append([r,c])
        directions = [[0,1], [1,0], [0,-1], [-1,0]]
        while q and fresh > 0:
            for i in range(len(q)):
                r, c = q.popleft()
                for dr, dc in directions:
                    row, col = dr + r, dc + c
                    if (row < 0 or row == n or col < 0 or col == m or grid[row][col] != 1):
                        continue
                    grid[row][col] = 2
                    q.append([row, col])
                    fresh -= 1
            time += 1
        return time if fresh == 0 else -1
```

## Time Complexity

- Let **N** be **number of rows** and **M** be **number of columns**.
- We are traversing through the entire grid. Which would take a time complexity of **O(N * M).**

## Space Complexity

- Let **N** be **number of rows** and **M** be **number of columns**.
- At worst we might have the entire grid in our queue if all grid positions are occupied by either rotten or fresh oranges, so the space complexity will also be **O(N * M).**

# 2. BFS with State Storing

# BFS with state storing

- In some problems, a cell can be revisited with new information. To keep track of previously visited cells with different states, we mark each cell visited with a combination of its state and cell_id.
- In other words, We revisit same node but with different information.

# Shortest Path with Alternating Colors

## 1129. Shortest Path with Alternating Colors

Medium 3080 165 Add to List Share

You are given an integer `n`, the number of nodes in a directed graph where the nodes are labeled from `0` to `n - 1`. Each edge is red or blue in this graph, and there could be self-edges and parallel edges.

You are given two arrays `redEdges` and `blueEdges` where:

- `redEdges[i] = [ai, bi]` indicates that there is a directed red edge from node $a_i$ to node $b_i$ in the graph, and
- `blueEdges[j] = [uj, vj]` indicates that there is a directed blue edge from node $u_j$ to node $v_j$ in the graph.

Return an array `answer` of length `n`, where each `answer[x]` is the length of the shortest path from node `0` to node `x` such that the edge colors alternate along the path, or `-1` if such a path does not exist.

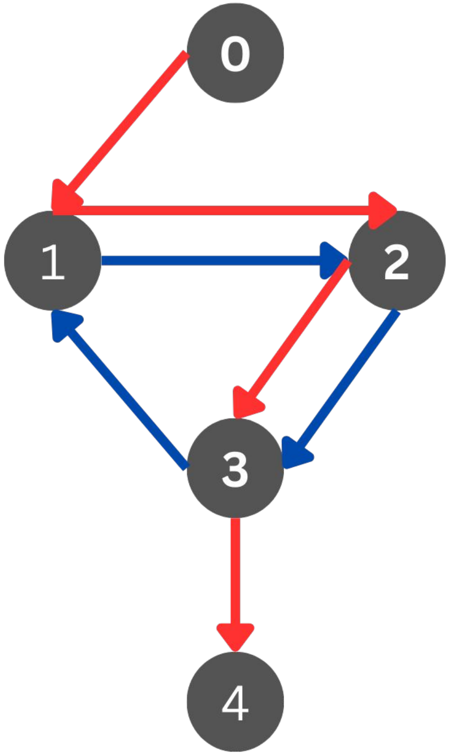
**Example 1:**

```
Input: n = 3, redEdges = [[0,1],[1,2]], blueEdges = []
Output: [0,1,-1]
```

**Example 2:**

```
Input: n = 3, redEdges = [[0,1]], blueEdges = [[2,1]]
Output: [0,1,-1]
```

# Is there a valid path to node “4”?



# Approach

1. Create a new graph but for every node, we we have two types of edges, red edges, and blue edges.
2. Initialize a queue with the root node, the starting color, and the starting distance.
3. Loop through the queue:
   - Compare the current distance for the node, with the distance in the answer array.
   - Add the neighbours in the opposite color list of the node
4. Return the list of distances

```python
def shortestAlternatingPaths(self, n, redEdges, blueEdges):
    graph = [[[], []] for _ in range(n)]
    for a, b in redEdges:
        graph[a][0].append(b)
    for a, b in blueEdges:
        graph[a][1].append(b)
    queue = deque([(0, 0), (0, 1)])
    visited = set([(0, 0), (0, 1)])
    answer = [-1 for _ in range(n)]
    dist = 0
    while queue:
        for _ in range(len(queue)):
            node, color = queue.popleft()
            if answer[node] == -1:
                answer[node] = dist
            alternative = 1 - color

            for neighbour in graph[node][alternative]:
                if (neighbour, alternative) not in visited:
                    visited.add((neighbour, alternative))
                    queue.append((neighbour, alternative))
        dist += 1
    return answer
```

## Time Complexity

- We have **V nodes** and **E edges**
- We will only **visit a node once**, and if it’s **a complete graph** we will also **use every edge**.

  Time Complexity = **O (V + E)**

## Space Complexity

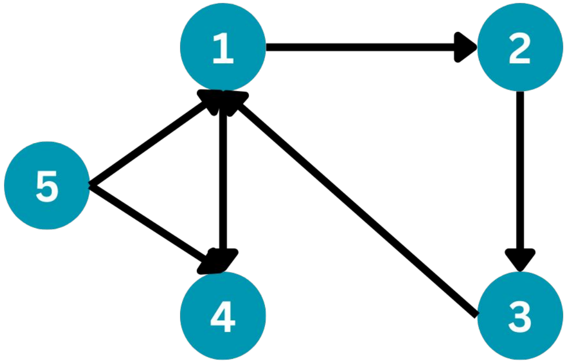
- We have **V nodes** and **E edges**
- We will store **the nodes**, and also **the edges**.

  Space Complexity = **O (V + E)**

# Common Pitfalls

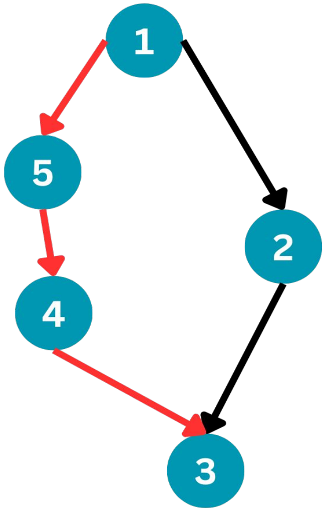
# Not maintaining visited nodes

It is important to keep **track** of **visited nodes** to avoid **revisiting them** and getting stuck in **an infinite loop**.



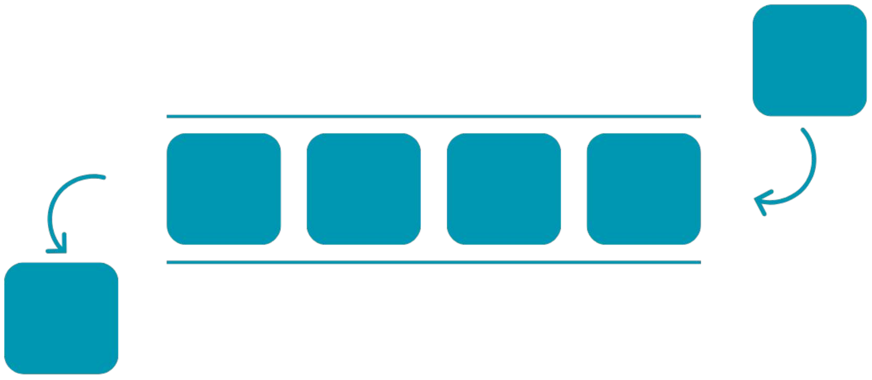
## Improper handling of visited nodes

In certain situations, it's possible to return to a cell with **new data**. While we might label a node as **visited** based on earlier visits, we may not always consider it as such when encountering it again with new information.



## Using the wrong data structure

BFS requires a data structure that allows for **FIFO** (first in, first out) access, such as a queue. If you use a data structure that **does not allow** for FIFO access, such as a **stack**, you may **not get the correct** traversal order.



## Not Checking for Goal State

It's essential to incorporate checks for the **goal state** within the BFS algorithm. Failure to include this check can lead to **unnecessary exploration** of the entire search space, even after finding the solution.

# Practice Problems

- Shortest Path in Binary Matrix
- Keys and Rooms
- Open the lock
- 01 Matrix
- Map of highest peak
- As Far from Land as Possible
- All Nodes Distance K in Binary Tree
- Nearest Exit from Entrance in Maze
- Snakes and Ladders
- Rotting Oranges
- Race Car
- Bus Routes
- Word Ladder

# Quote of the day

*"Exploration is really the essence of the human spirit."*

**Frank Borman**