Graph Day-02

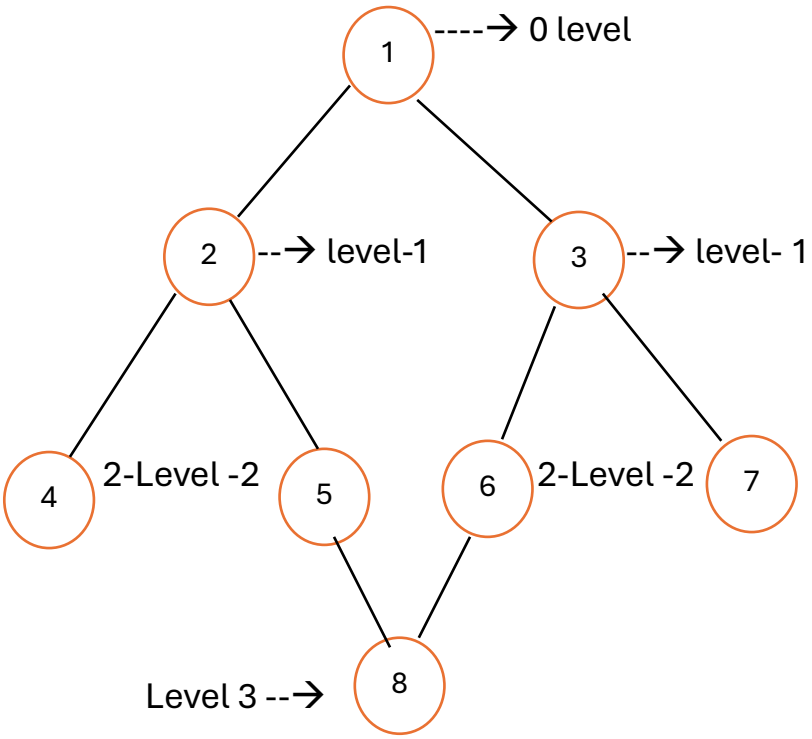
Traversal Technique in Graph:

1. Breath First Search (BFS)
2. Depth First Search (DFS)

Breath First Search (BFS):

In BFS we do the traversal level wise, and in graph we can start with any node, means starting node can be any node

Let's take a starting node = 1



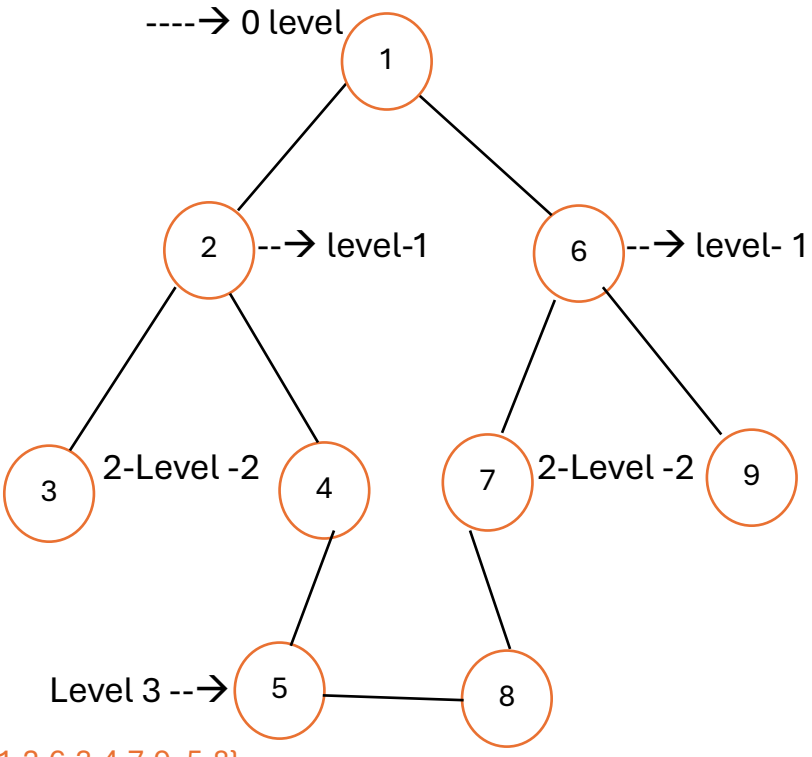
Traversing on node from starting node 1 = [1,2,3,4,5,6,7,8]

Level 0 – 1

Level 1 – 2,3

level 2 – 4,5,6,7

level 3 - 8



Traverse: {1,2,6,3,4,7,9, 5,8}

Step to be done in BFS Traversal:

- We need to initialize visited array the size of visited array will be the total node +1 (n+1) and Queue Data Structure
- Take a starting node and put into the Queue data structure
- After putting into the queue then visit that node in visited array
- Take out the element from the queue, as it's starting node so find out its neighbor
- Whatever neighbor we will get take it and put all the neighbors in queue data structure one by one and simultaneously visit it in visited array

Step -1

Starting node = 1

Let's put it in to the queue =

1									

Visited_array =

1	0	0	0	0	0	0	0	0	0
1	2	3	4	5	6	7	8	9	

Adj list

1 – [2,6]	6 – [1,7,9]
2 – [1,3,4]	7 – [6,8]
3 – [2]	8 – [5,7]
4 – [2, 5]	9 – [6]
5 – [4,8]	

Step -2 take out the element from the Queue and find it's neighbor

Taking 1 from the Queue

1 --- [2,6]

Starting node = 1

Let's put it into the queue =

2	6								

Traverse: {1}

Visited_array =

1	1	0	0	0	0	1	0	0	0
1	2	3	4	5	6	7	8	9	

Step – 3: again take the element from the queue and find out it's neighbor

2 -→ [1,3,4]

Let's visit it if not visited and put it into the queue

1 is already visited 3, 4 not visited yet

Traverse: {1,2}

Visited_array =

1	1	1	1	0	0	1	0	0	0
1	2	3	4	5	6	7	8	9	

Let's put 3 & 4 into the queue,...

Let's put it into the queue =

	6	3	4						

Step – 4: again take the element from the queue and find out it's neighbor

6 -→ [1,7,9]

Traverse: {1,2,6}

Let's visit it if not visited and put it into the queue

1 is already visited 7,9 not visited yet

Visited_array =

1	1	1	1	0	0	1	1	0	1
1	2	3	4	5	6	7	8	9	

Let's put 7 & 9 into the queue,...

Let's put it into the queue =

		3	4	7	9				

Step – 5: again, take the element from the queue and find out it's neighbor

3 -→ [2]

Let's visit it if not visited and put it into the queue

2 is already visited

		4	7	9				

Traverse: {1,2,6,3}

Step – 6: again take the element from the queue and find out it's neighbor

4 -→ [2,5]

Traverse: {1,2,6,3,4}

Let's visit it if not visited and put it into the queue

2 is already visited 5 not visited yet

Visited_array =

1	1	1	1	1	0	1	1	0	1
1	2	3	4	5	6	7	8	9	

Let's put 5 into the queue,...

	7	9	5	

Step – 7: again take the element from the queue and find out it's neighbor

7 -→ [6,8]

Traverse: {1,2,6,3,4,7}

Let's visit it if not visited and put it into the queue

6,8 not visited yet

Visited_array =

1	1	1	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8	9	

queue,...

		9	5	6	8	

Step – 8: again take the element from the queue and find out it's neighbor

9 -→ [6]

Traverse: {1,2,6,3,4,7,9}

6 is already visited

Visited_array =

1	1	1	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8	9	

the queue,...

	5	6	8	

Adj list

1 – [2,6]
2 – [1,3,4]
3 – [2]
4 – [2, 5]
5 – [4,8]
6 – [1,7,9]
7 – [6,8]
8 – [5,7]
9 – [6]

Step – 9: again take the element from the queue and find out it's neighbor

5 -→ [4,8]

Traverse: {1,2,6,3,4,7,9, 5}

4,8 is already visited

Visited_array =

1	1	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8	9

		6	8	

Step – 10: again take the element from the queue and find out it's neighbor

6 -→ [1,7,9]

Traverse: {1,2,6,3,4,7,9, 5,6}

6 is already visited

Visited_array =

1	1	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8	9

			8	

Step – 11: again take the element from the queue and find out it's neighbor

8 -→ [5,7]

Traverse: {1,2,6,3,4,7,9, 5,8}

5,7 is already visited

Visited_array =

1	1	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8	9

Now queue become empty

Traverse: {1,2,6,3,4,7,9, 5,8}

Sudo Code

```
BFS_traversal (V,adj):
    Vis = [0]*V  ## let's initialize the visited array
    Bfs_traversal = [] ## here we will store the bfs_traversal
    Queue = [] # queue data structure
    ## now let's mark the starting node as visited
    Vis[0] = 1
    # now let's process the node until queue is empty
    While queue:
        Node = queue.popleft()
        bfs_traversal.append(node)
        ### Now let's Explore the adj list of the current node
        for neighbor in adj[node]:
            if not vis[neighbor]:
                vis[neighbor] = 1
                queue.append(neighbor)
    return bfs_traverse
```

Time Complexity: O(n) ← while loop + O(2*E) ← Total degree ➔ O(n+2E)

Space Complexity: O(n)

```python
from collections import deque
def bfs(V,adj,start):
    ## now 1st lets initilize the list which keep track of visited node
    vis = [0]*(V+1)
    ## 2nd thing we need another list which store the traversal
    bfs_traversal = []

    q = deque([start])

    ## now let's mark the starting node as visited
    vis[start] = 1

    ### Now let's process the node until queue is empty
    while q:
        node = q.popleft()
        bfs_traversal.append(node)

        ## second thing is that let's find out it's neighbor
        for neighbor in adj[node]:
            if not vis[neighbor]:
                vis[neighbor] = 1
                q.append(neighbor)
    return bfs_traversal

V = 9
adj_list = [[],[2,6],[1,3,4],[2],[2,5],[4,8],[1,7,9],[6,8],[5,7],[6]]
bfs(V,adj_list,1)
```

✓ 0.0s

```
[1, 2, 6, 3, 4, 7, 9, 5, 8]
```

Depth First Search

Depth-First Search (DFS) is a fundamental algorithm used for traversing or searching through graph data structures. It starts at a chosen node (often called the root in a tree) and explores as far as possible along each branch before backtracking.

In DFS, the algorithm follows a path from the starting node to an adjacent node, continuing along the depth of the graph until it reaches a node with no unvisited neighbors. At this point, it backtracks to the previous node to explore other unvisited paths. This process is repeated until all nodes in the graph have been visited.

DFS can be implemented using either a recursive approach (using function calls) or an iterative approach (using a stack).

Characteristics:

- Traversal Type: Depth-first (explores deeply along a branch before moving to another branch).

- Data Structure: Uses a stack (either implicit via recursion or explicit).

- 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 used for recursion or backtracking.

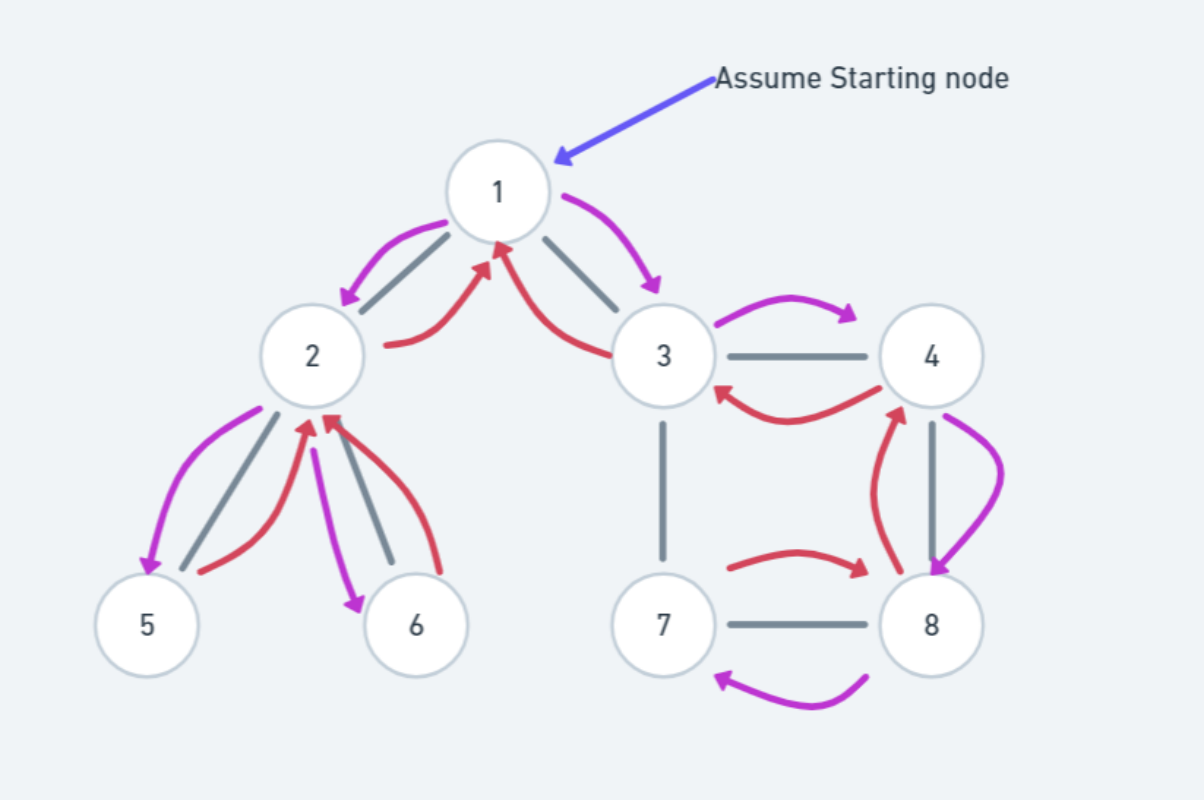
DFS is commonly used for:

- Finding connected components in a graph

- Detecting cycles

- Solving puzzles (e.g., mazes)

- Topological sorting



In this graph i took starting node as ‘1’ , we can take any node as starting node

Let’s Write the Adj List

Sudo Code

```
dfs_ traversal (node, adj):
        vis[node] = 1 ## Visiting Starting node
        list.add(node)
        for neighbor in adj:
                if not vis[neighbor]:
                        dfs_traversal(neighbor)
        return list
```

Time Complexity: O(V+E)

- **V**: Each node is visited once.
- **E**: Each edge is traversed once during the DFS traversal.

Space Complexity: O(V) + O(V) + O(V) = O(3V) = O(V)

- **O(V)** for the vis list.
- **O(V)** for the traversal_list.
- **O(V)** for the recursion stack in the worst case.