DSA **Lab Manual 8**
Part A

Detailed description of BFS and DFS algorithms with examples is given below. Read it carefully and by using your **graph** class implementations (Adjacency List + Adjacency Matrix), implement the following methods:

1. BFS graph traversal (Using Queue + Adjacency Matrix)
2. BFS graph traversal (Using Queue + Adjacency List)
3. DFS graph traversal (Using Stack + Adjacency Matrix)
4. DFS graph traversal (Using Stack + Adjacency List)
5. DFS graph traversal (Using Recursion + Adjacency Matrix)
6. DFS graph traversal (Using Recursion + Adjacency List)

Breadth First Search (BFS)

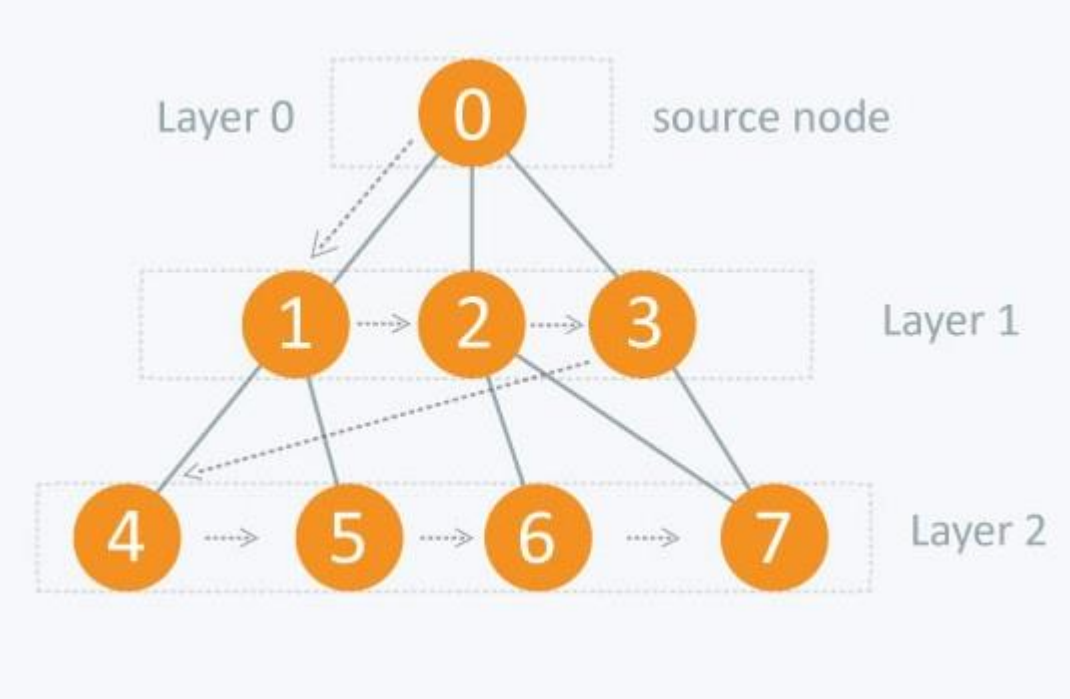
There are many ways to traverse graphs. BFS is the most commonly used approach.

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layer-wise thus exploring the neighbor nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbor nodes.

As the name BFS suggests, you are required to traverse the graph breadth-wise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

Consider the following diagram.



The distance between the nodes in layer 1 is comparatively lesser than the distance between the nodes in layer 2. Therefore, in BFS, you must traverse all the nodes in layer 1 before you move to the nodes in layer 2.

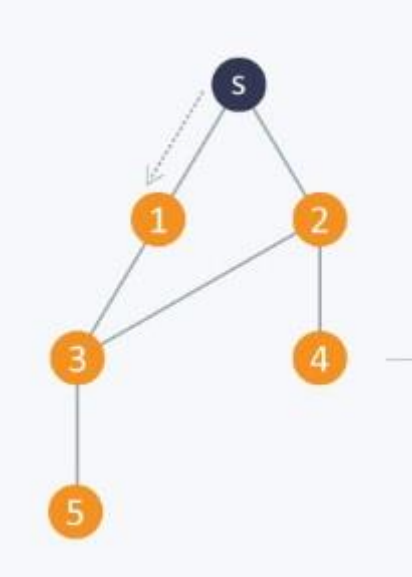
Traversing child nodes

A graph can contain cycles, which may bring you to the same node again while traversing the graph. To avoid processing of same node again, use a boolean array which marks the node after it is processed. While visiting the nodes in the layer of a graph, store them in a manner such that you can traverse the corresponding child nodes in a similar order.

In the earlier diagram, start traversing from 0 and visit its child nodes 1, 2, and 3. Store them in the order in which they are visited. This will allow you to visit the child nodes of 1 first (i.e. 4 and 5), then of 2 (i.e. 6 and 7), and then of 3 (i.e. 7) etc.

To make this process easy, use a queue to store the node and mark it as 'visited' until all its neighbors (vertices that are directly connected to it) are marked. The queue follows the First In First Out (FIFO) queuing method, and therefore, the neighbors of the node will be visited in the order in which they were inserted in the node i.e. the node that was inserted first will be visited first, and so on.

One more Example



The traversing will start from the source node and push *s* in queue. *s* will be marked as 'visited'.

First iteration

- s will be popped from the queue
- Neighbors of s i.e. 1 and 2 will be traversed
- 1 and 2, which have not been traversed earlier, are traversed. They will be:
    - Pushed in the queue
    - 1 and 2 will be marked as visited

Second iteration

- 1 is popped from the queue
- Neighbors of 1 i.e. s and 3 are traversed
- s is ignored because it is marked as 'visited'
- 3, which has not been traversed earlier, is traversed. It is:
    - Pushed in the queue
    - Marked as visited

Third iteration

- 2 is popped from the queue

- Neighbors of 2 i.e. s, 3, and 4 are traversed
- 3 and s are ignored because they are marked as 'visited'
- 4, which has not been traversed earlier, is traversed. It is:
    - Pushed in the queue
    - Marked as visited

Fourth iteration

- 3 is popped from the queue
- Neighbors of 3 i.e. 1, 2, and 5 are traversed
- 1 and 2 are ignored because they are marked as 'visited'
- 5, which has not been traversed earlier, is traversed. It is:
    - Pushed in the queue
    - Marked as visited

Fifth iteration

- 4 will be popped from the queue
- Neighbors of 4 i.e. 2 is traversed
- 2 is ignored because it is already marked as 'visited'

Sixth iteration

- 5 is popped from the queue
- Neighbors of 5 i.e. 3 is traversed
- 3 is ignored because it is already marked as 'visited'

The queue is empty and it comes out of the loop. All the nodes have been traversed by using BFS.

If all the edges in a graph are of the same weight, then BFS can also be used to find the minimum distance between the nodes in a graph.

Pseudocode

```
BFS (G, s)                    //Where G is the graph and s is the source node
      let Q be queue.
      Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.

      mark s as visited.
      while ( Q is not empty)
            //Removing that vertex from queue,whose neighbour will be visited now
            v  =  Q.dequeue( )

          //processing all the neighbours of v
          for all neighbours w of v in Graph G
               if w is not visited
                         Q.enqueue( w )              //Stores w in Q to further visit its
neighbour
                         mark w as visited.
```

Depth First Search (DFS)

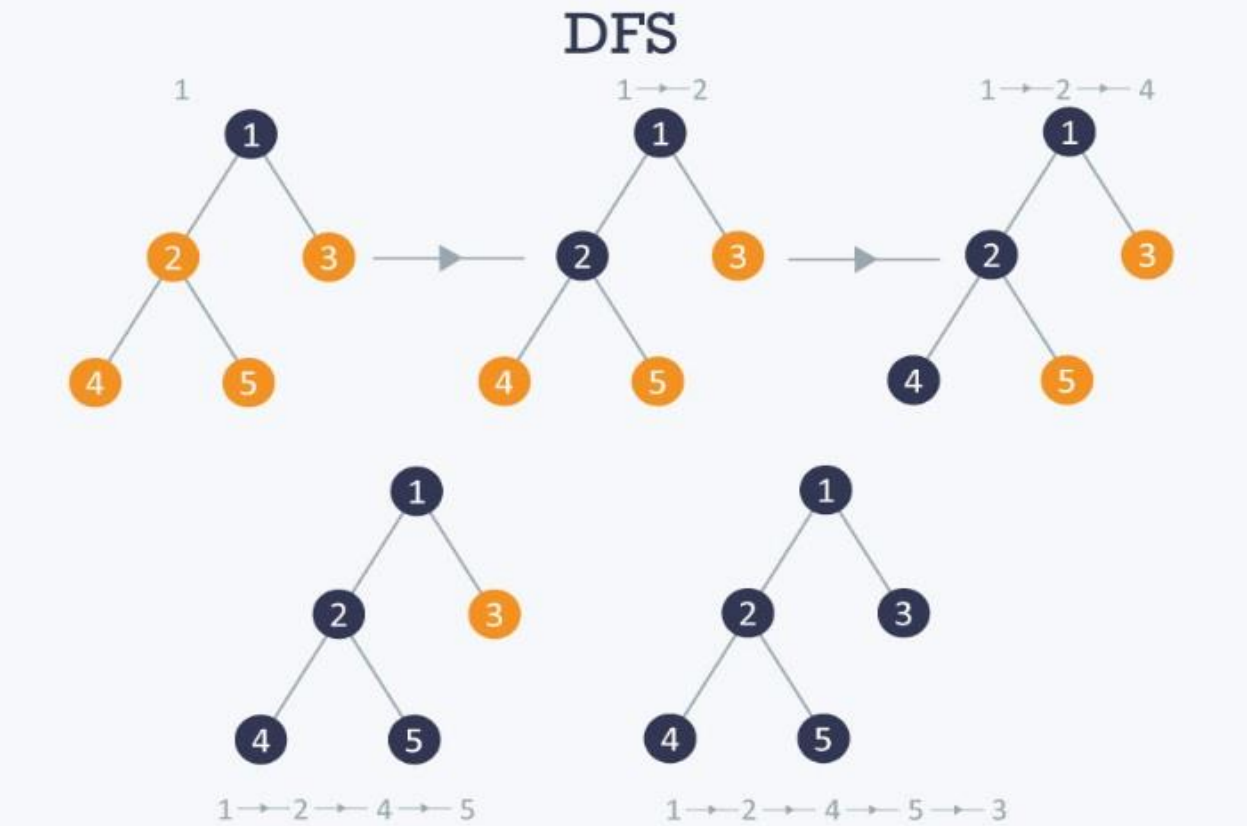
The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using **stacks**.

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**.




```
    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)
```