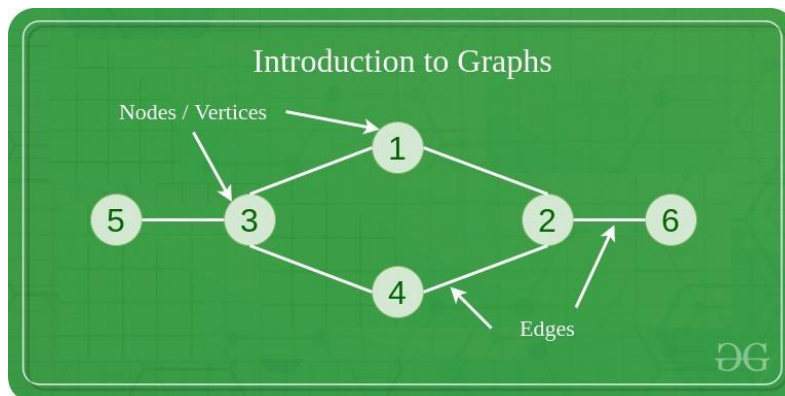


NAME: ADWAIT S PURAO
UID: 2021300101
EXP NO. : 8
AIM: To implement BFS and DFS Traversals of graph with printing levels and start and end time respectively.

## THEORY:

### What is Graph in Data Structure and Algorithms?

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(  $V$  ) and a set of edges(  $E$  ). The graph is denoted by  $G(E, V)$ .



### Components of a Graph

**Vertices:** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabelled.

**Edges:** Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labeled/unlabelled.

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.

A graph is a data structure that consists of the following two components:

1. A finite set of vertices also called as nodes.

2. A finite set of ordered pair of the form  $(u, v)$  called as edge. The pair is ordered because  $(u, v)$  is not the same as  $(v, u)$  in case of a directed graph(di-graph). The pair of the form  $(u, v)$  indicates that there is an edge from vertex  $u$  to vertex  $v$ . The edges may contain weight/value/cost.

Graphs are used to represent many real-life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, and locale.

The following two are the most commonly used representations of a graph.

1. Adjacency Matrix

2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of graph representation is situation-specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

Adjacency Matrix is a 2D array of size  $V \times V$  where  $V$  is the number of vertices in a graph. Let the 2D array be  $adj[][]$ , a slot  $adj[i][j] = 1$  indicates that there is an edge from vertex  $i$  to vertex  $j$ . Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If  $adj[i][j] = w$ , then there is an edge from vertex  $i$  to vertex  $j$  with weight  $w$ .

The adjacency matrix for the above example graph is:

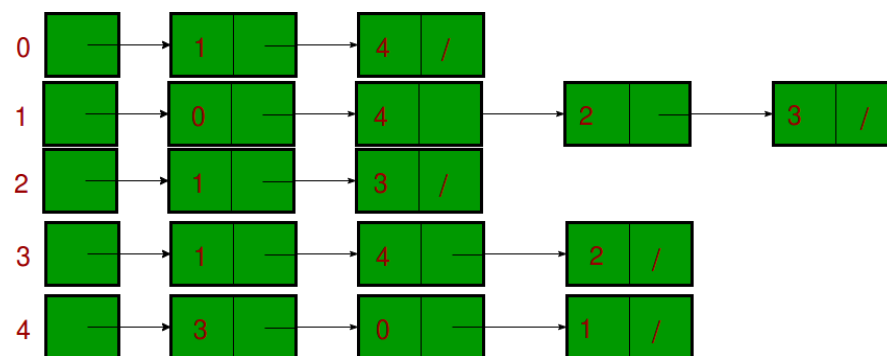
	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Pros: Representation is easier to implement and follow. Removing an edge takes  $O(1)$  time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done  $O(1)$ .

Cons: Consumes more space  $O(V^2)$ . Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is  $O(V^2)$  time. Computing all neighbors of a vertex takes  $O(V)$  time (Not efficient).

### Adjacency List:

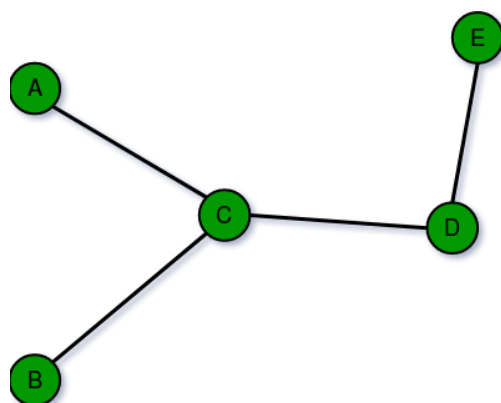
An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an array[]. An entry array[i] represents the list of vertices adjacent to the ith vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above graph.



### Types of Graphs:

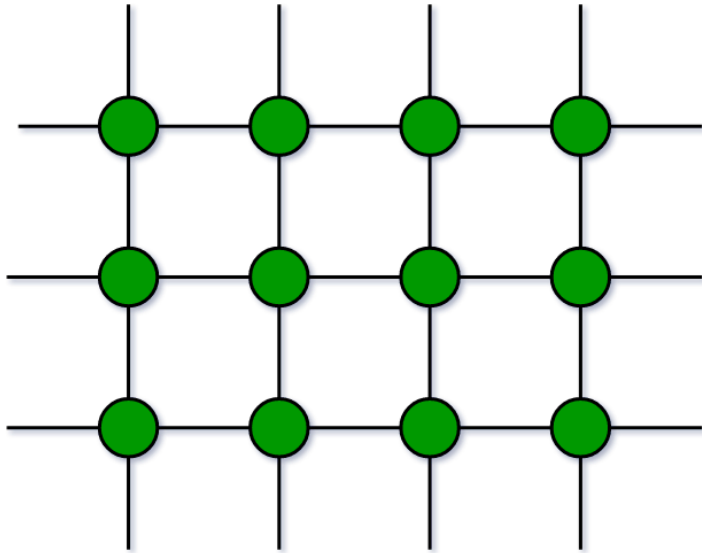
#### 1. Finite Graphs

A graph is said to be finite if it has a finite number of vertices and a finite number of edges



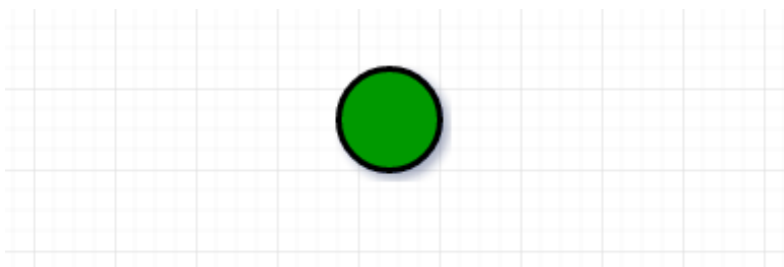
#### 2. Infinite Graph:

A graph is said to be infinite if it has an infinite number of vertices as well as an infinite number of edges.



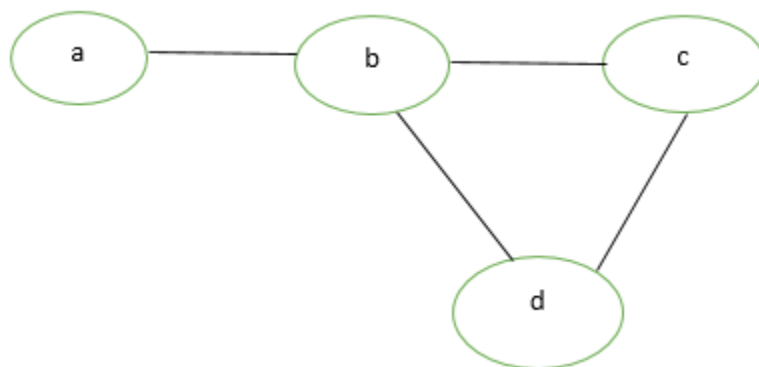
### 3. Trivial Graph:

A graph is said to be trivial if a finite graph contains only one vertex and no edge.



### 4. Simple Graph:

A simple graph is a graph that does not contain more than one edge between the pair of vertices. A simple railway track connecting different cities is an example of a simple graph.



### 5. Multi Graph:

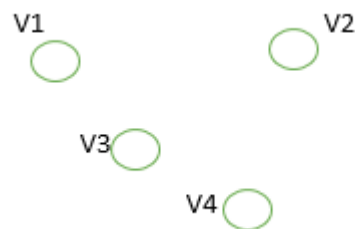
Any graph which contains some parallel edges but doesn't contain any self-loop is called a multigraph. For example a Road Map.

**Parallel Edges:** If two vertices are connected with more than one edge then such edges are called parallel edges that are many routes but one destination.

**Loop:** An edge of a graph that starts from a vertex and ends at the same vertex is called a loop or a self-loop.

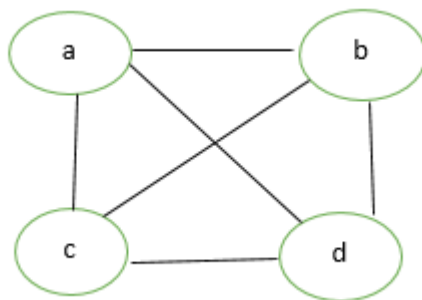
#### 6. Null Graph:

A graph of order  $n$  and size zero is a graph where there are only isolated vertices with no edges connecting any pair of vertices.



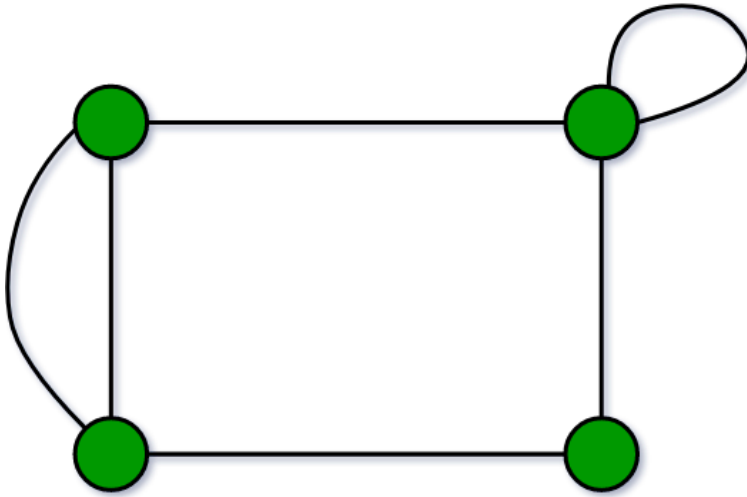
#### 7. Complete Graph:

A simple graph with  $n$  vertices is called a complete graph if the degree of each vertex is  $n-1$ , that is, one vertex is attached with  $n-1$  edges or the rest of the vertices in the graph. A complete graph is also called Full Graph.



#### 8. Pseudo Graph:

A graph  $G$  with a self-loop and some multiple edges is called a pseudo graph.



## Traversals

### Breadth first search

In this tutorial, you will learn about breadth first search algorithm. Also, you will find working examples of bfs algorithm in C, C++, Java and Python.

Traversal means visiting all the nodes of a graph. Breadth First Traversal or Breadth First Search is a recursive algorithm for searching all the vertices of a graph or tree data structure.

### BFS algorithm

A standard BFS implementation puts each vertex of the graph into one of two categories:

- Visited
- Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

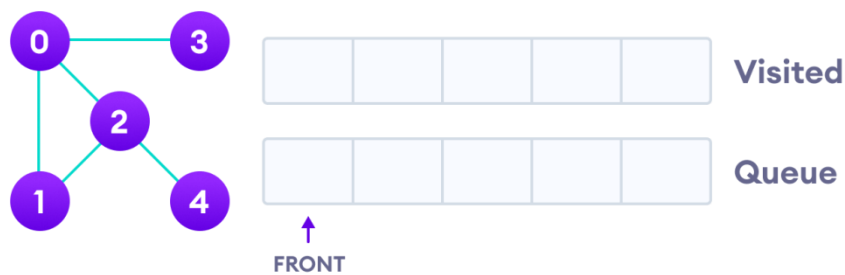
1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

### BFS example

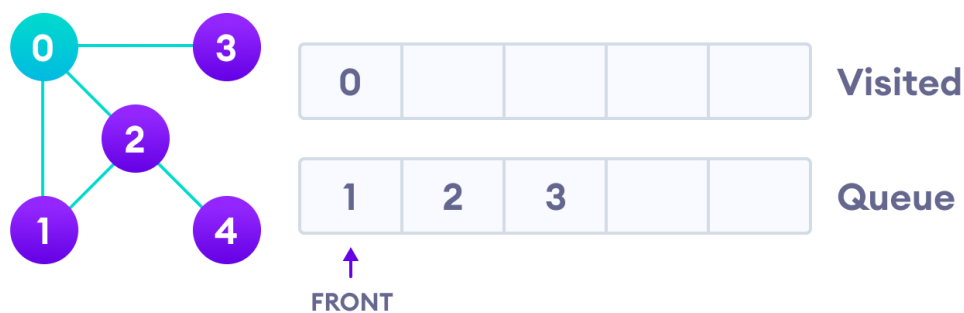
Let's see how the Breadth First Search algorithm works with an example. We use an undirected graph with 5 vertices.

Step 1:



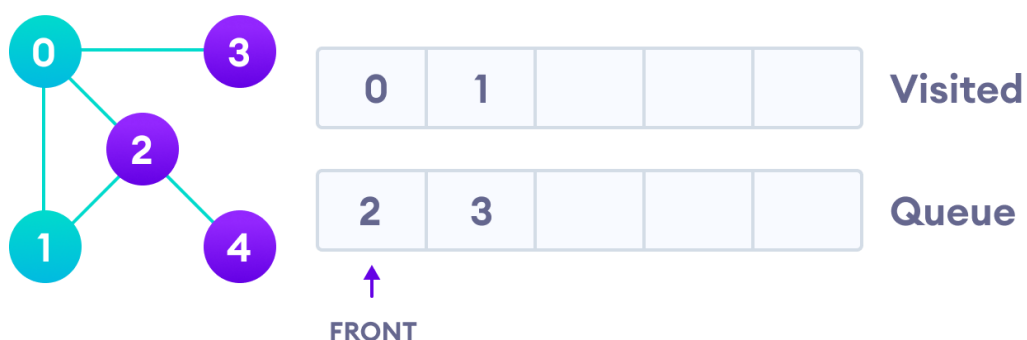
We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.

Step 2:



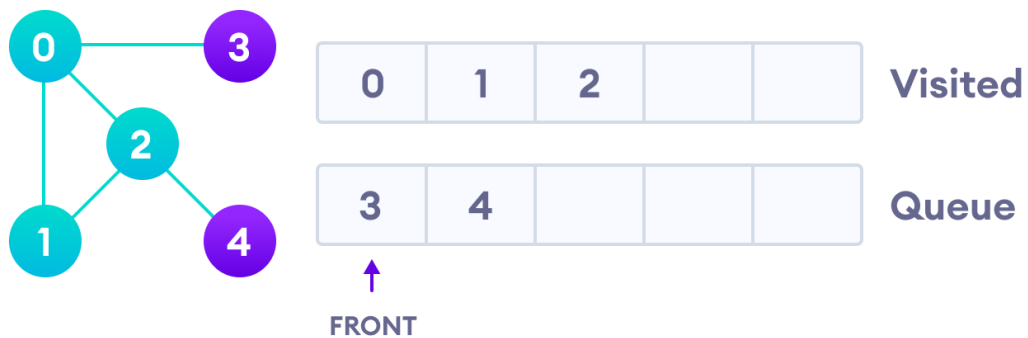
Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

Step 3:

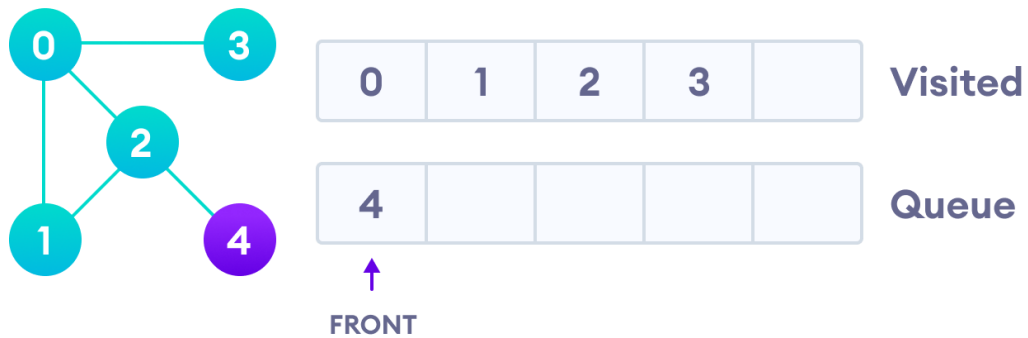


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.

Step 4:



Step 5:



Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited. We visit it.

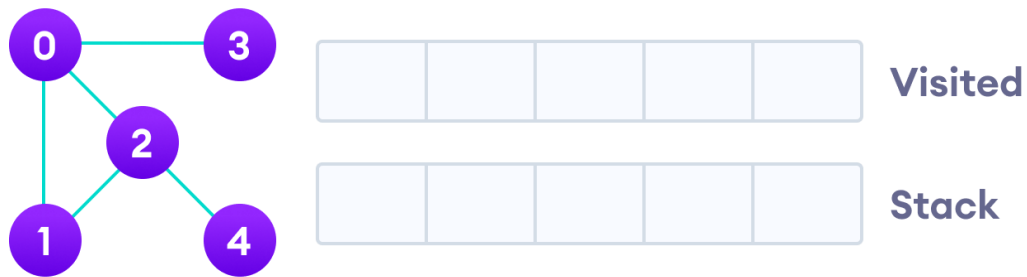
Step 6:



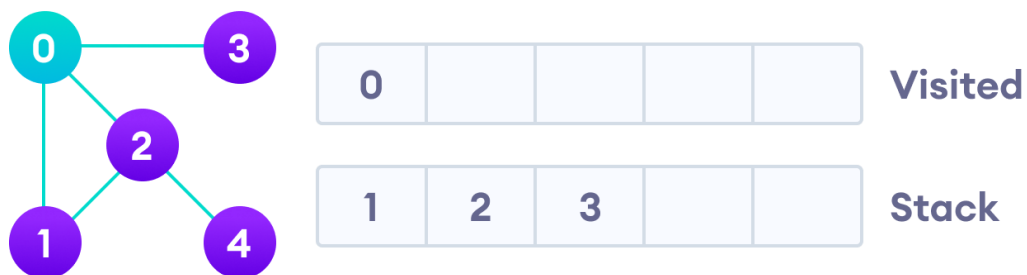
Since the queue is empty, we have completed the Breadth First Traversal of the graph.



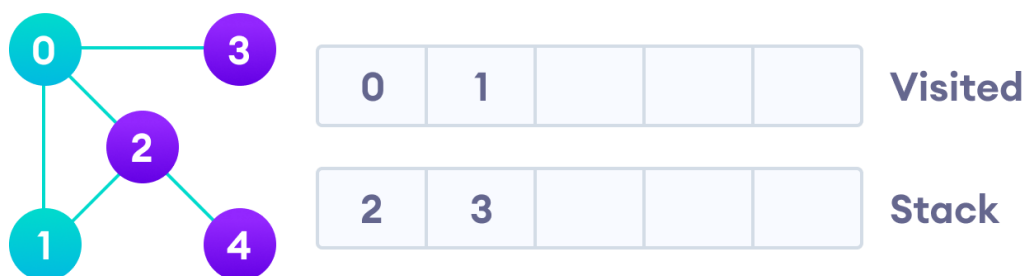
## Depth First Search Example



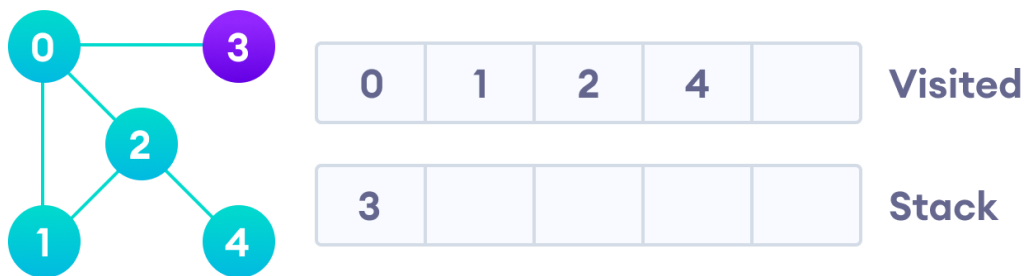
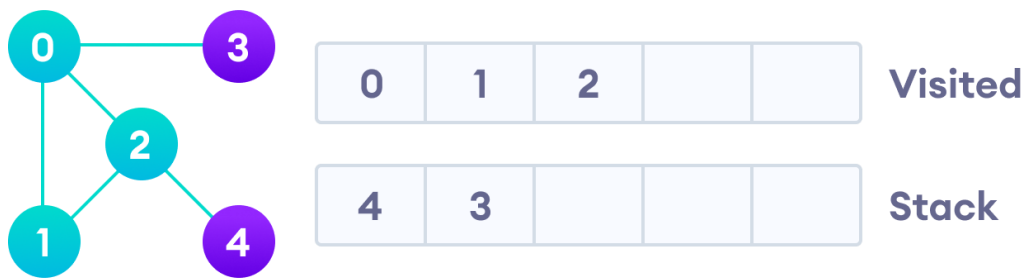
We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



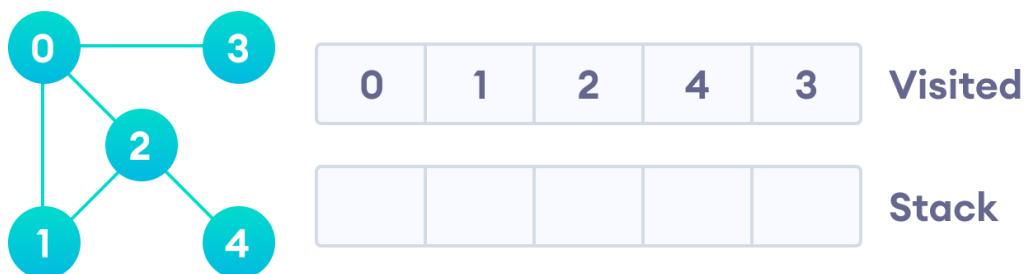
Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



#### ALGORITHM:

##### 1. Class Graph

Initialize integer variables time and ver  
arrays visiteddfs and visitedbfs and Linked List adj

Constructor of class Graph pass integer variable ver

This.ver=ver

Adj= new LinkedList[ver]

Visiteddfs=new Boolean[ver]

Visitedbfs=new Boolean[ver]

i=0 and Repeat the steps until i<ver  
Adj[i]= new LinkedList<Integer>()

Function void Add pass integer variables w and v

adj[v].add(w)

adj[w].add(v)

Function void DFS pass integer variable ver , array start and end

Start[ver]←time

Time++

Visiteddfs[ver]←true

Print(ver)

Declare a Iterator it = adj[ver].listIterator

Repeat the steps while It.hasNext() is true

Int vis←it.next()

If visiteddfs[vis] is false

Recur function DFS

End[ver]←time

Time++

Function void Print

i=1 and Repeat the steps until i<ver

Print i

For x in adj[i]

Print x

Function void BFS

Declare a Linked List queue

Visitedbfs[src]←true

Level[src]←0

Queue.add(src)

Repeat until size of queue is not equal to zero

```

Src=queue.poll()

Print src

Declare a Iterator It=adj[src].listIterator()

Repeat until It.hasNext() returns true

Int neigh←It.next()

If visitedbfs[neigh] is false

    Visitedbfs[neigh]←true

    Queue.add(neigh)

    Level[neigh]←levl[src]+1

```

Public Class DSA8

Main method

```

Take the number of vertices as input

Declare an object of graph and pass ver+1

Take input the number of edges

Declare integer arrays start , end and level with sizes ver+1

l=0 and repeat until i<number of edges

    Take input the source and destination of edges

Call function G.print to print the linked Representation of graph

Take input the vertex from where you want to start the traversal

Call function g.DFS to print the DFS Traversal of the graph

l=1 and repeat until i<=ver

    Print start time and end time of each node

Call function g.BFS and to print the BFS Traversal of graph

l=0 and repeat until i<ver

    Print number of nodes and their levels

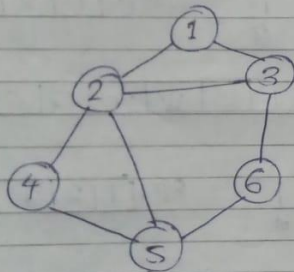
```

PROBLEM SOLVING ON CONCEPT:

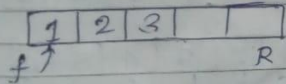
## BFS (Queue) (Level Order)

- 1 → Choose any vertex ( $V$ ) as starting pt. & add it to Queue.
- 2 → Put  $V$  to the visited list & add it to output also Dequeue it.
- 3 → Add all unvisited neighbours of  $V$  to queue until it empty.
- 4 → Repeat 2 & 3 until queue is empty.

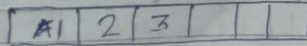
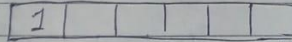
Graph



S-1:



O/P



Adwait . Purao  
2021300107

Page No. \_\_\_\_\_  
Date. \_\_\_\_/\_\_\_\_/\_\_\_\_

Q  
S-2

2	3	4	5	1
---	---	---	---	---

  
f                      R                      R

op:

1	2			
---	---	--	--	--

Visited 

A	B	C		
---	---	---	--	--

Visited 

1	2	3	4	5
---	---	---	---	---

S-3

op

3	4	5	6		
---	---	---	---	--	--

  
f                      R                      R

1	2	3			
---	---	---	--	--	--

Visited 

1	2	3	4	5	6
---	---	---	---	---	---

S-4

4	5	6	
---	---	---	--

  
f                      R                      R

op: 

1	2	3	4		
---	---	---	---	--	--

Visited 

1	2	3	4	5	6
---	---	---	---	---	---

S-5

5	6		
---	---	--	--

  
f                      R                      R

op: 

1	2	3	4	5	
---	---	---	---	---	--

Visited 

1	2	3	4	5	6
---	---	---	---	---	---

S-6

6	
---	--

  
f                      R

op: 

1	2	3	4	5	6
---	---	---	---	---	---

  
level: 0 1 1 2 2 2

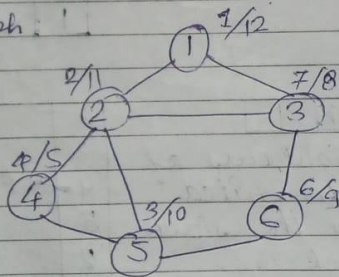
Visited: 

1	2	3	4	5	6
---	---	---	---	---	---

## DFS (Stack)

- 1 → Choose any vertex  $V$  as starting pt. & add it to stack
- 2 → Put  $V$  to visited list & print  $V$
- 3 → Add any 1 unvisited neighbour of  $V$  to stack
- 4 → If current vertex has all its neighbours already visited, pop it from stack & back track
- 5 → Check remaining vertices in the stack ~~& back track~~ for any unvisited nodes
- 6 → Repeat till stack is empty

→ Graph:



Adwait . Puro  
2021300107

Page No.

Date

S-1 o/p:

A

Visited

Stack

1

S-2

o/p:

1 2

Visited

1 2

Stack

2

1

S-3: o/p:

1 2 5

Visited

1 2 5

Stack

5

2

1

S-4 o/p:-

1 2 5 4

As all neighbours of  
4 have been visited  
pop it from the stack  
& backtrack

Stack

4 pop

5

2

1

Visited

1 2 5 4



Adwait Purao  
2021300101

Page No: \_\_\_\_\_  
Date: \_\_\_\_/\_\_\_\_/\_\_\_\_

S-5 o/p:

1	2	5	4	6	
---	---	---	---	---	--

6
5
2
1

New check  
for unvisited  
nodes from E &  
add to stack

Vis. 

1	2	5	4	6
---	---	---	---	---

S-6 o/p:

1	2	5	4	6	3
---	---	---	---	---	---

3
6
5
2
1

Vis. 

1	2	5	4	6	3
---	---	---	---	---	---

As all vertices have  
been visited pop & empty the stack

CODE:

```
import java.util.*;

class Graph{
public static int time =1;
int ver;
static boolean visitedbfs[];
static boolean visiteddfs[];
LinkedList <Integer> adj [];

Graph(int ver){
    this.ver=ver;
    adj= new LinkedList[ver];
    visitedbfs= new boolean[ver];
    visiteddfs= new boolean[ver];

    for(int i=0;i<ver;i++){
        adj[i]=new LinkedList<Integer>();
    }
}

void Add(int v,int w){
    adj[v].add(w);
    adj[w].add(v);
}

void DFS(int ver, int start [],int end []){
```

```

        start[ver]=time;
        time++;

        visiteddfs [ver]=true;
        System.out.print(ver + " ");

        Iterator <Integer> it= adj[ver].listIterator();
        while(it.hasNext()){
            int vis=it.next();
            if(!visiteddfs[vis]){
                DFS(vis,start,end);
            }
        }

        end[ver]=time;
        time++;
    }

    void print(){
        System.out.println("Adjacency List representation of Graph:");
        for (int i = 1; i < ver; i++) {
            System.out.print(i+ ":");
            for (int x : adj[i]) {
                System.out.print(x+ " ");
            }
            System.out.println();
        }
    }

    void BFS(int src, int level []){
        LinkedList<Integer> queue = new LinkedList();
        visitedbfs[src] =true;
        level[src]=0;

        queue.add(src);

        while(queue.size()!=0){
            int new_src=queue.poll();
            System.out.print(new_src + " ");

            Iterator<Integer> It= adj[src].listIterator();
            while(It.hasNext()){
                int neigh=It.next();
                if(!visitedbfs[neigh]){
                    visitedbfs[neigh]=true;
                    queue.add(neigh);
                    level[neigh]=level[src]+1;
                }
            }
        }
    }

```

```

    }
}
}

public class DSA8{
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the number of vertices:");
        int ver= sc.nextInt();
        Graph g = new Graph(ver+1);

        System.out.println("Enter the number of edges:");
        int edge=sc.nextInt();

        int start[]=new int [ver+1];
        int end[]=new int [ver+1];
        int level[]=new int [ver];

        for(int i=0;i<edge;i++){
            System.out.println("Enter the first vertex");
            int src=sc.nextInt();
            System.out.println("Enter the second vertex");
            int des=sc.nextInt();
            g.Add(src, des);
        }

        System.out.println("Representation of graph");
        g.print();
        System.out.println("Enter the vertex from where you want to start
traversal:");
        int startver=sc.nextInt();
        System.out.println();

        System.out.println("DFS TRAVERSAL");
        g.DFS(startver,start,end);
        System.out.println();

        for (int i = 1; i <= ver; i++)
            System.out.println("Vertex " + i + " START POINT "
                               + start[i] + " END POINT " + end[i]);

        System.out.println("BFS TRAVERSAL");
        g.BFS(startver,level);
        System.out.println();

        System.out.println("Nodes"+ " " + "Level");
    }
}

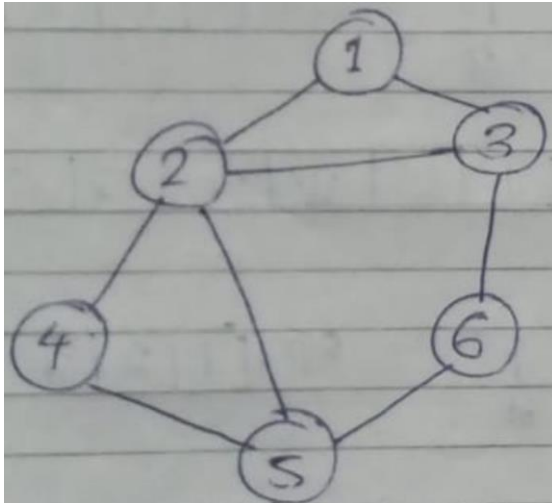
```

```

        for (int i = 0; i < ver; i++)
            System.out.println(" " + i + " --> " + level[i]);
    }
}

```

OUTPUT SCREENSHOT:



```

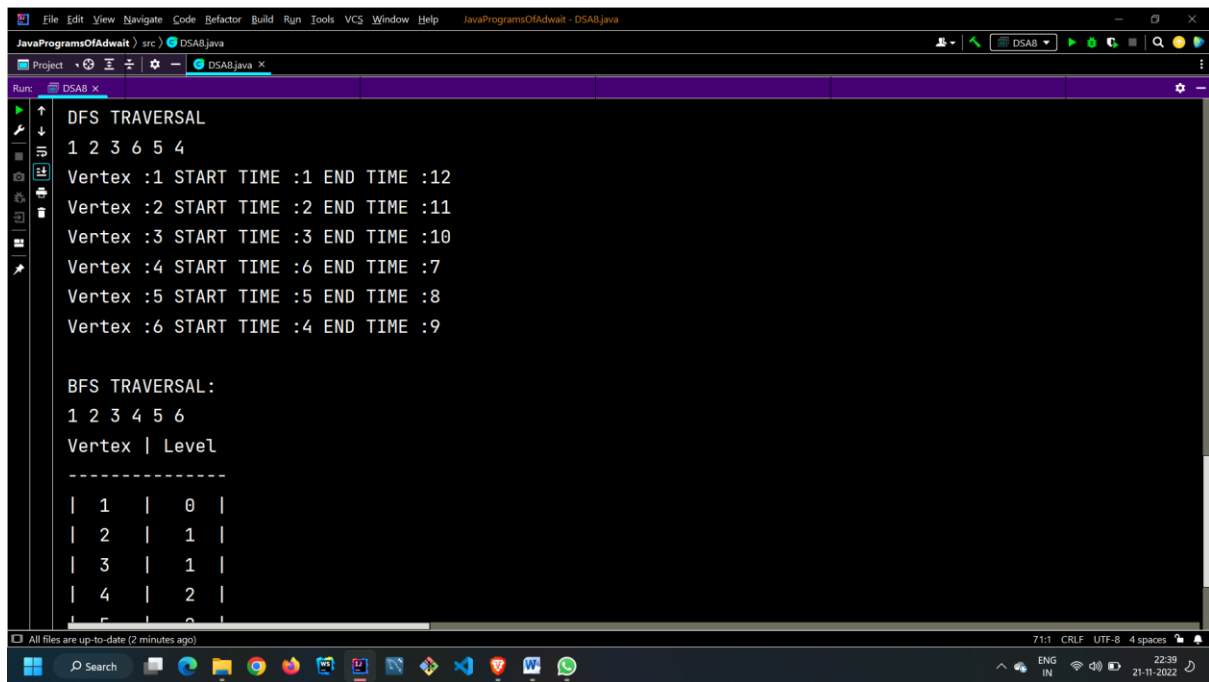
"C:\Program Files\Java\jdk-18.0.1\bin\java.exe" "-javaagent:C:\Program Files\Java\IntelliJ IDEA Community B
Enter the number of vertices:
6
Enter the number of edges:
8
Enter the first vertex
2
Enter the second vertex
1
Enter the first vertex
2
Enter the second vertex
3
Enter the first vertex
2
Enter the second vertex
4
Enter the first vertex

```

```
File Edit View Navigate Code Refactor Build Run Tools VCS Window Help JavaProgramsOfAdwait - DSAB.java
Project DSAB x
Run: DSAB x
Enter the first vertex
2
Enter the second vertex
5
Enter the first vertex
4
Enter the second vertex
5
Enter the first vertex
1
Enter the second vertex
3
Enter the first vertex
5
Enter the second vertex
6
Enter the first vertex
3
```

```
File Edit View Navigate Code Refactor Build Run Tools VCS Window Help JavaProgramsOfAdwait - DSAB.java
Project DSAB x
Run: DSAB x
Enter the first vertex
3
Enter the second vertex
6
Representation of graph
Adjacency List representation of Graph:
1:2 3
2:1 3 4 5
3:2 1 6
4:2 5
5:2 4 6
6:5 3
Enter the vertex from where you want to start traversal:
1

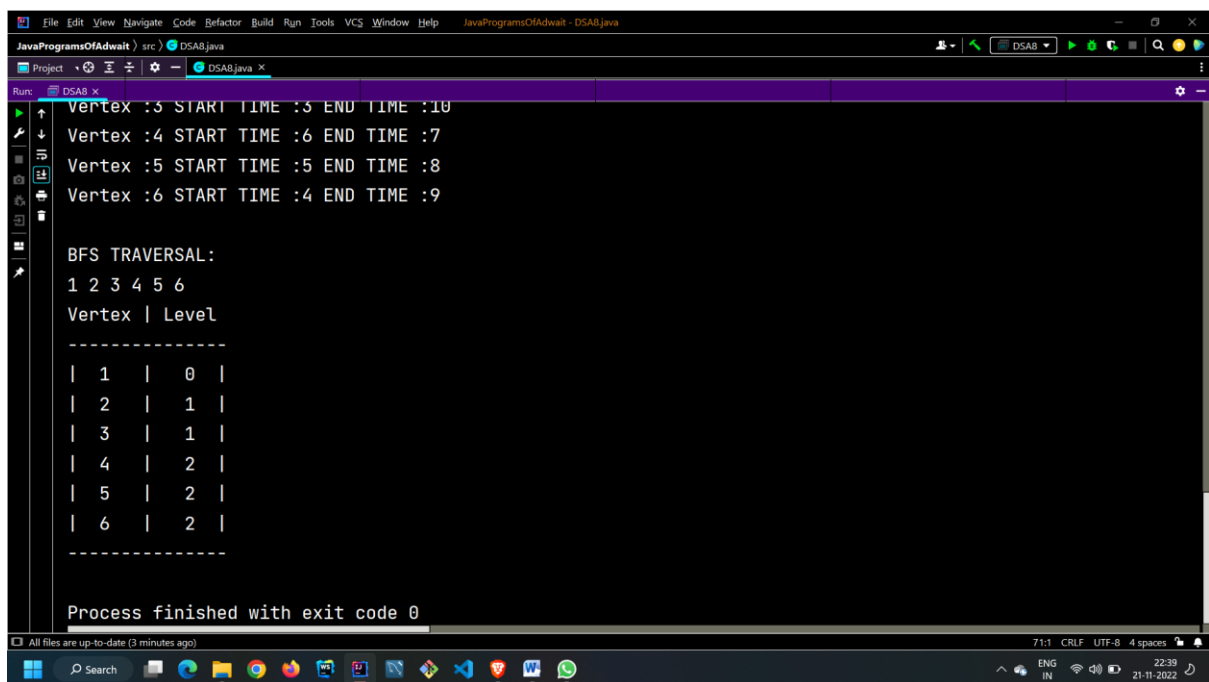
DFS TRAVERSAL
1 2 3 6 5 4
Memory: 4 START TIME: 1 END TIME: 10
```



The screenshot shows an IDE window titled "JavaProgramsOfAdwait - DSAB.java". The "Run" console displays the output of a Java program. It starts with "DFS TRAVERSAL" followed by the sequence "1 2 3 6 5 4". Then, it lists the start and end times for each vertex: Vertex :1 START TIME :1 END TIME :12, Vertex :2 START TIME :2 END TIME :11, Vertex :3 START TIME :3 END TIME :10, Vertex :4 START TIME :6 END TIME :7, Vertex :5 START TIME :5 END TIME :8, and Vertex :6 START TIME :4 END TIME :9. Below this, it shows "BFS TRAVERSAL:" followed by the sequence "1 2 3 4 5 6". Finally, it displays a table for BFS levels.

```
DFS TRAVERSAL
1 2 3 6 5 4
Vertex :1 START TIME :1 END TIME :12
Vertex :2 START TIME :2 END TIME :11
Vertex :3 START TIME :3 END TIME :10
Vertex :4 START TIME :6 END TIME :7
Vertex :5 START TIME :5 END TIME :8
Vertex :6 START TIME :4 END TIME :9

BFS TRAVERSAL:
1 2 3 4 5 6
Vertex | Level
-----
| 1 | 0 |
| 2 | 1 |
| 3 | 1 |
| 4 | 2 |
| 5 | 2 |
| 6 | 2 |
```



This screenshot shows the continuation of the BFS traversal output from the previous image. It lists the start and end times for vertices 3, 4, 5, and 6. Below this, it shows the BFS traversal sequence "1 2 3 4 5 6" and the BFS levels table, which now includes vertex 6 at level 2. At the bottom, it states "Process finished with exit code 0".

```
Vertex :3 START TIME :3 END TIME :10
Vertex :4 START TIME :6 END TIME :7
Vertex :5 START TIME :5 END TIME :8
Vertex :6 START TIME :4 END TIME :9

BFS TRAVERSAL:
1 2 3 4 5 6
Vertex | Level
-----
| 1 | 0 |
| 2 | 1 |
| 3 | 1 |
| 4 | 2 |
| 5 | 2 |
| 6 | 2 |

Process finished with exit code 0
```

## CONCLUSION:

In the above experiment we learnt about graphs their structure and their representation i.e. Linked representation and matrix representation. We also learned about the types of traversals in graph which are bfs traversal and dfs traversal and implemented them in Java.