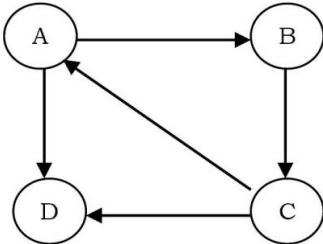
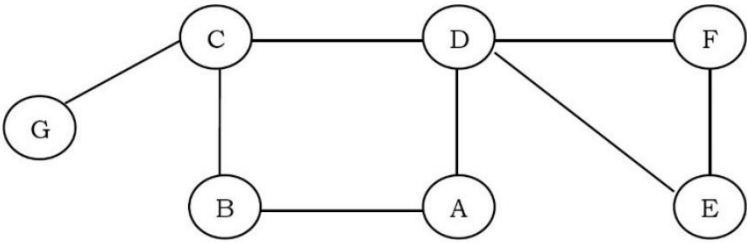


Name	Hatim Yusuf Sawai
UID no.	2021300108
Experiment No.	8

AIM:	Program on Graphs
------	-------------------

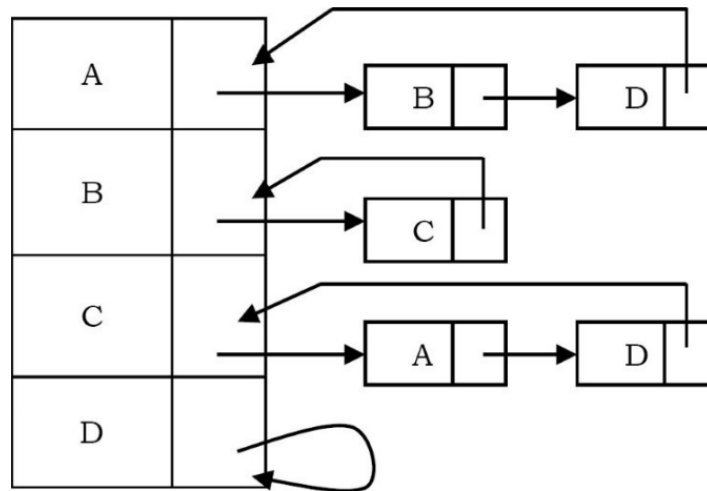
Program

PROBLEM STATEMENT:	Write a program to implement both graph traversal methods: bfs & dfs in a graph using adjacency lists method
--------------------	---

THEORY:	<p>Graph: A graph is a pair (V, E), where V is a set of nodes, called vertices, and E is a collection of pairs of vertices, called edges. • Vertices and edges are positions and store elements</p> <p>Directed Graph: • Contain directed edges , can be traversed only in a specific direction</p>  <pre> graph TD A((A)) --> B((B)) A((A)) --> D((D)) B((B)) --> C((C)) C((C)) --> D((D)) </pre> <p>Undirected Graph: • Contain undirected edges , can be traversed bidirectionally</p>  <pre> graph LR G((G)) --- C((C)) C((C)) --- B((B)) C((C)) --- D((D)) B((B)) --- A((A)) A((A)) --- D((D)) D((D)) --- F((F)) F((F)) --- E((E)) </pre>
---------	---

Graph Representation Using Adjacency List:

In this representation all the vertices connected to a vertex v are listed on a linked list for that vertex v . That means, for each vertex v we use a linked list and list nodes represents the connections between v and other vertices to which v has an edge. The total number of linked lists is equal to the number of vertices in the graph.



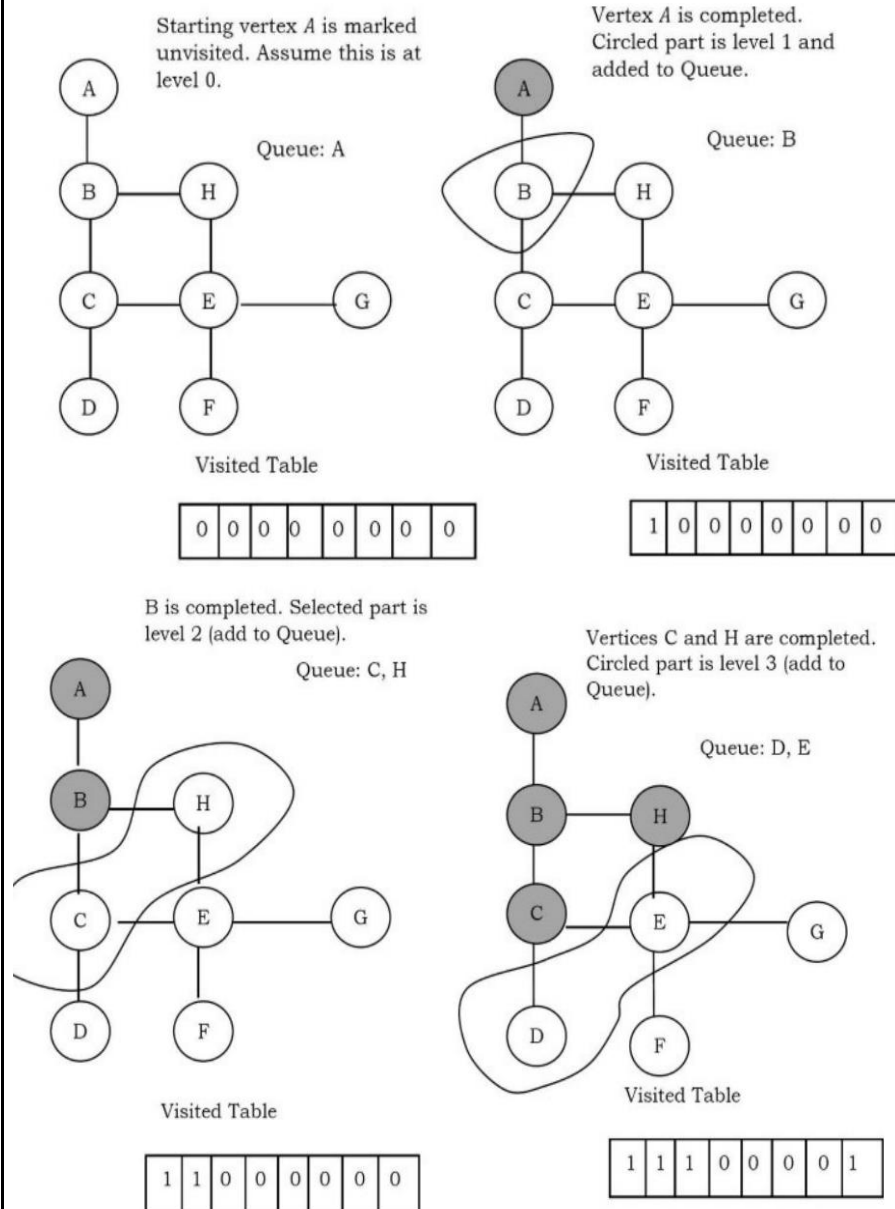
Graph Traversals:

There are 2 types of traversal techniques:

1. BFS (Breadth First) Search:

The BFS algorithm works like level - order traversal of the trees. BFS works level by level. Initially, it starts at a given vertex, which is at level 0. In the first stage it visits all vertices at level 1 (that means, vertices whose distance is 1 from the start vertex of the graph). In the second stage, it visits all vertices at the second level. These new vertices are the ones which are adjacent to level 1 vertices. BFS continues this process until all the levels of the graph are completed. Generally, queue data structure is used for storing the vertices of a level.

Example:

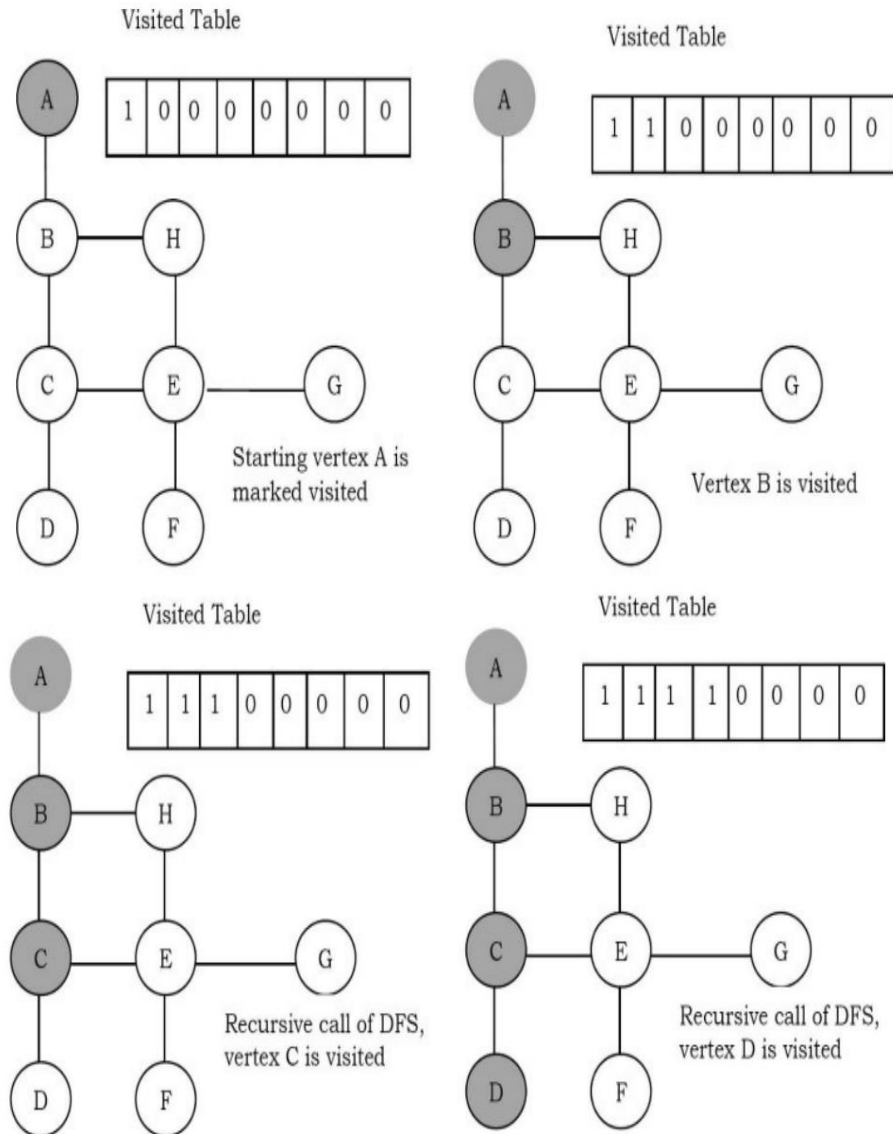


2. DFS {Depth First} Search:

DFS algorithm works in a manner similar to preorder traversal of the trees. Like preorder traversal, internally this algorithm also uses stack. Initially all vertices are marked unvisited (false). The DFS algorithm starts at a vertex u in the graph. By starting at vertex u it considers the edges from u to other vertices. If the edge leads to an already visited vertex, then backtrack to current vertex u . If an edge

leads to an unvisited vertex, then go to that vertex and start processing from that vertex. That means the new vertex becomes the current vertex. Follow this process until we reach the dead-end. At this point start backtracking. The process terminates when backtracking leads back to the start vertex.

Example:



ALGORITHM:

1. Create **Graph Class** with **arraylist<arrayList>** adj member
2. Define constructor which takes in **vertices** and initializes array lists
3. create **methods** for Graph Traversal

addEdge Method:

1. takes source & destination and adds corresponding values in **arrayList**

printGraph Method:

1. for i=0 to kadj.size:
 Print i
 1. for j=0 to j<adj[i].size:
 2. print adj.get(i).get(j)
2. the loops print the graph representation in adjacency list form

BFS Method:

1. Create **bool[]** visited array, **int[]** parent array & **int[]** level array
2. create integer queue q (from utils)
3. **add s to q**, make parent[s] as **-1**, visited[s] as **true** & level[s] as **0**
4. while queue is not empty:
5. v = q.remove() -> **dequeue** element from the queue
6. **print v & level[v]**
7. for i=0 to i<adj.get(v).size():
 1. w = adj.get(v).get(i)
 2. if w is **not in** visited[]:
 3. level[w] = level[v]+1
 4. add w to q, visited[w] = true & parent[w] = v
8. end of for loop

DFS Method:

1. Create **bool[]** visited, **int[]** parent, **int[]** dfs, **int[]** start, **int[]** end
2. Create **Stack<Integer>** from (Utils)
3. Initialize int time = 1, j=0
4. Push s to stack, make start[s] = time, visited[s] = true, parent[s] = -1
5. Make dfs[j++] = s

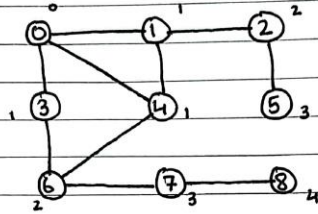
	<ol style="list-style-type: none">6. While the stack is not empty:7. $v = \text{st.peek}()$ → get top element of stack8. $\text{flag} = \text{false}$9. for $i=0$ to $i < \text{adj.get}(v).\text{size}()$10. $w = \text{adj.get}(v).\text{get}(i)$;11. if w is not in visited array:12. push w to stack, make $\text{start}[w] = ++\text{time}$, $\text{visited}[w] = \text{true}$, $\text{parent}[w] = v$13. Set flag to true, $\text{dfs}[j] = w$;14. Increment value of j then break15. if flag is false:16. Set $\text{end}[v] = ++\text{time}$;17. pop from stack18. end of while loop19. print dfs array in order with the start & end arrays to display order with timestamps
--	--

PROBLEM SOLVING:

HATIM SAWAI

8] GRAPH - TRAVERSAL

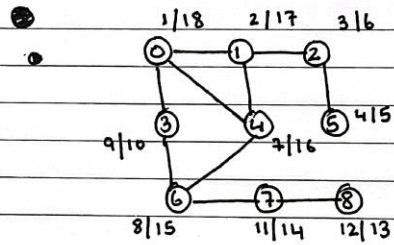
Graph:



BFS: 0, 1, 3, 4, 2, 6, 5, 7, 8

level 0 level 1 level 2 level 3 level 4

Graph:



DFS: 0, 1, 2, 5, 4, 6, 3, 7, 8

start = 1, end = 18 (2x9v)

Stack:

8 → pop

7 → pop

3 → pop

6 → pop

4 → pop

58 → pop

2 → pop

1 → pop

0 → pop

PROGRAM:**GraphCheck.java:**

```
import java.util.Scanner;
import graphds.LLGraph;
public class GraphCheck {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the number of vertices: ");
        int v = sc.nextInt();
        LLGraph g = new LLGraph(v);
        System.out.print("Enter the number of edges: ");
        int e = sc.nextInt();
        for(int i=0; i<e; i++) {
            System.out.println("For Edge "+(i+1));
            System.out.print("Enter the source and destination: ");
            int s = sc.nextInt();
            int d = sc.nextInt();
            g.addEdge(s,d);
        }
        System.out.println("The Graph is:");
        g.printGraph();
        while(true) {
            System.out.print("Enter the source for Traversal: ");
            int s = sc.nextInt();
            System.out.println("Choose the type of Traversal:\n1. BFS\n2.
DFS");
            int ch = sc.nextInt();
            switch(ch) {
                case 1:
                    System.out.println("BFS Traversal:");
                    g.BFS(s);
                    break;
                case 2:
                    System.out.println("DFS Traversal:");
                    g.DFS(s);
                    break;
                default:
```



```

        System.out.println("Invalid Choice!");
    }
    System.out.print("Do you want to continue? (0=no/1=yes): ");
    int c = sc.nextInt();
    if(c==0) {
        break;
    }
}
sc.close();
}
}

```

LLGraph.java:

```

package graphds;
import java.util.*;
public class LLGraph {
    private ArrayList<ArrayList<Integer>> adj;
    public LLGraph(int v) {
        adj = new ArrayList<ArrayList<Integer>>(v);
        for(int i=0; i<v; i++) {
            adj.add(new ArrayList<Integer>());
        }
    }
    public void addEdge(int s, int d) {
        adj.get(s).add(d);
        adj.get(d).add(s);
    }
    public void printGraph() {
        for(int i=0; i<adj.size(); i++) {
            System.out.print(i+" -> ");
            for(int j=0; j<adj.get(i).size(); j++) {
                System.out.print(adj.get(i).get(j)+" ");
            }
            System.out.println();
        }
    }
}

```

```

public void BFS(int s) {
    boolean[] visited = new boolean[adj.size()];
    int[] parent = new int[adj.size()];
    int[] level = new int[adj.size()];
    Queue<Integer> q = new LinkedList<Integer>();
    q.add(s);
    parent[s] = -1;
    visited[s] = true;
    level[s] = 0;
    while(!q.isEmpty()) {
        int v = q.remove();
        System.out.println(v+"(level "+level[v]+"")");
        for(int i=0;i<adj.get(v).size();i++) {
            int w = adj.get(v).get(i);
            if(!visited[w]) {
                level[w] = level[v]+1;
                q.add(w);
                visited[w] = true;
                parent[w] = v;
            }
        }
    }
}

public void DFS(int s) {
    boolean[] visited = new boolean[adj.size()];
    int[] parent = new int[adj.size()];
    Stack<Integer> st = new Stack<Integer>();
    int dfs[] = new int[adj.size()];
    int[] start = new int[adj.size()];
    int[] end = new int[adj.size()];
    int time = 1,j=0;
    st.push(s);
    start[s] = time;
    visited[s] = true;
    parent[s] = -1;
    dfs[j++] = s;

```

```

while(!st.isEmpty()) {
    int v = st.peek();
    boolean flag = false;
    for(int i=0;i<adj.get(v).size();i++) {
        int w = adj.get(v).get(i);
        if(!visited[w]) {
            st.push(w);
            start[w] = ++time;
            visited[w] = true;
            parent[w] = v;
            flag = true;
            dfs[j] = w;
            j++;
            break;
        }
    }
    if(!flag) {
        end[v] = ++time;
        st.pop();
    }
}

System.out.println("DFS Traversal:");
for(int i=0;i<j;i++) {
    System.out.println(dfs[i]+" (start: "+start[dfs[i]]+", end:
"+end[dfs[i]]+"");
}
}
}

```

OUTPUT:

Input:

```
PS D:\Data Structures> cd "d:\Data Struct
Enter the number of vertices: 9
Enter the number of edges: 10
For Edge 1
Enter the source and destination: 0 1
For Edge 2
Enter the source and destination: 1 2
For Edge 3
Enter the source and destination: 2 5
For Edge 4
Enter the source and destination: 1 4
For Edge 5
Enter the source and destination: 0 3
For Edge 6
Enter the source and destination: 0 4
For Edge 7
Enter the source and destination: 3 6
For Edge 8
Enter the source and destination: 4 6
For Edge 9
Enter the source and destination: 6 7
For Edge 10
Enter the source and destination: 7 8
The Graph is:
```

Graph Representation:

```
Enter the source and destination: 7
The Graph is:
0 -> 1 3 4
1 -> 0 2 4
2 -> 1 5
3 -> 0 6
4 -> 1 0 6
5 -> 2
6 -> 3 4 7
7 -> 6 8
8 -> 7
Enter the source for Traversal: █
```

BFS:

```
Enter the source for Traversal: 0
Choose the type of Traversal:
1. BFS
2. DFS
1
BFS Traversal:
0(level 0)
1(level 1)
3(level 1)
4(level 1)
2(level 2)
6(level 2)
5(level 3)
7(level 3)
8(level 4)
Do you want to continue? (0=no/1=yes): 1
```

DFS:

```
Enter the source for Traversal: 0
Choose the type of Traversal:
1. BFS
2. DFS
2
DFS Traversal:
DFS Traversal:
0 (start: 1, end: 18)
1 (start: 2, end: 17)
2 (start: 3, end: 6)
5 (start: 4, end: 5)
4 (start: 7, end: 16)
6 (start: 8, end: 15)
3 (start: 9, end: 10)
7 (start: 11, end: 14)
8 (start: 12, end: 13)
Do you want to continue? (0=no/1=yes):
```

CONCLUSION:

In this experiment, we learned how to implement bfs & dfs traversal algorithms on a graph data structure.