Lecture Notes

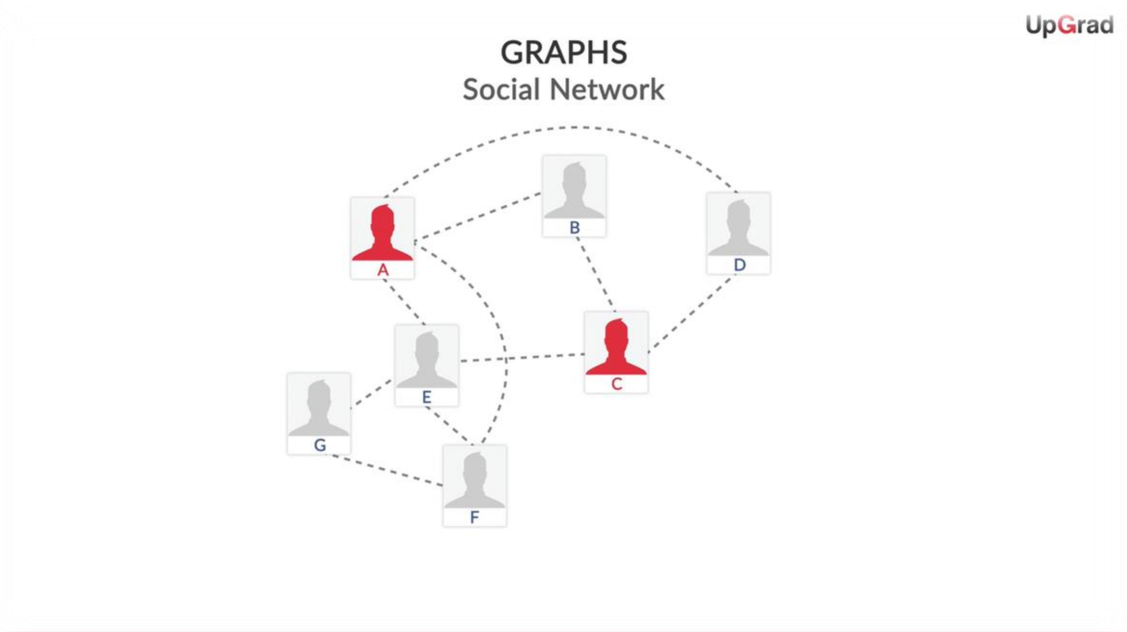
Graphs & Graph Algorithm

Session 1: Graphs

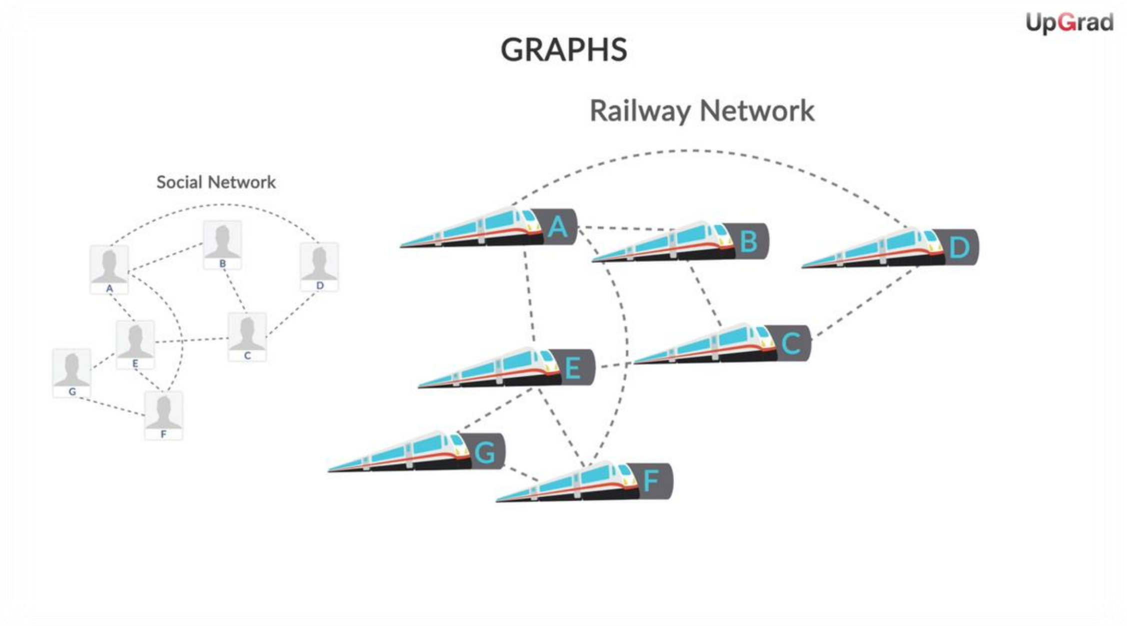
Introduction to graphs

In this session, you have learnt how graph data structures can represent the relationship between different entities. Here, the entities are called as nodes(vertices) and the relationship is called as edges(arcs).

Example 1: You have seen the example of social network, in which different people are represented as nodes and the edges connecting between them represent the friendship relation. The absence of an edge between A and C means that they are not friends

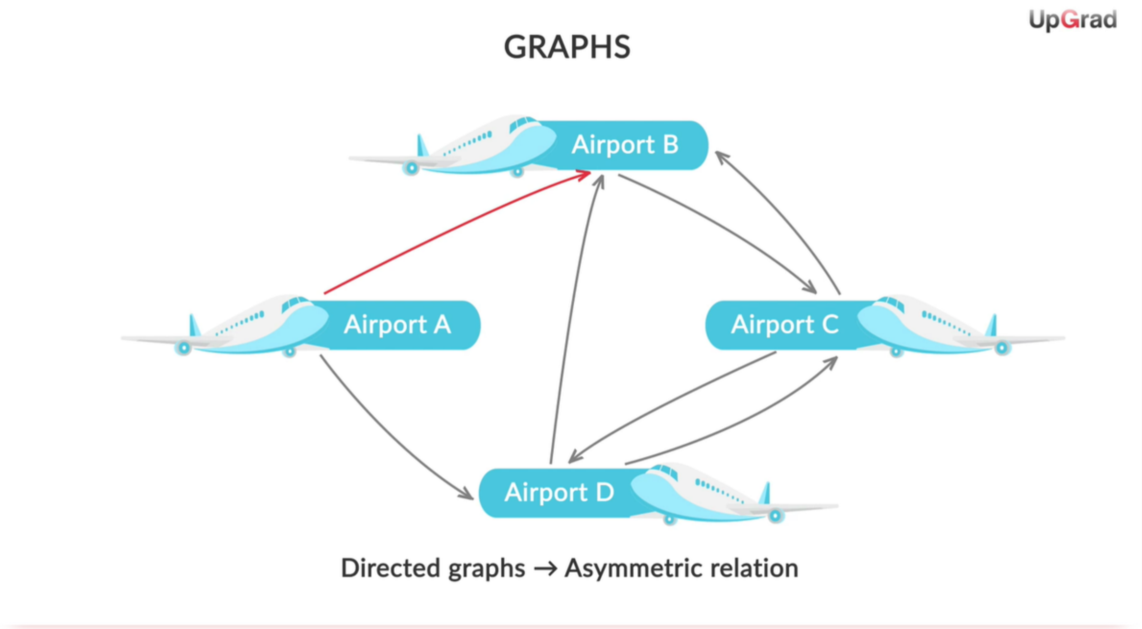


Example 2: Then, you have seen a railway network represented as a graph structure, where the railway stations are nodes and the edges connecting between different stations represent the railway track.



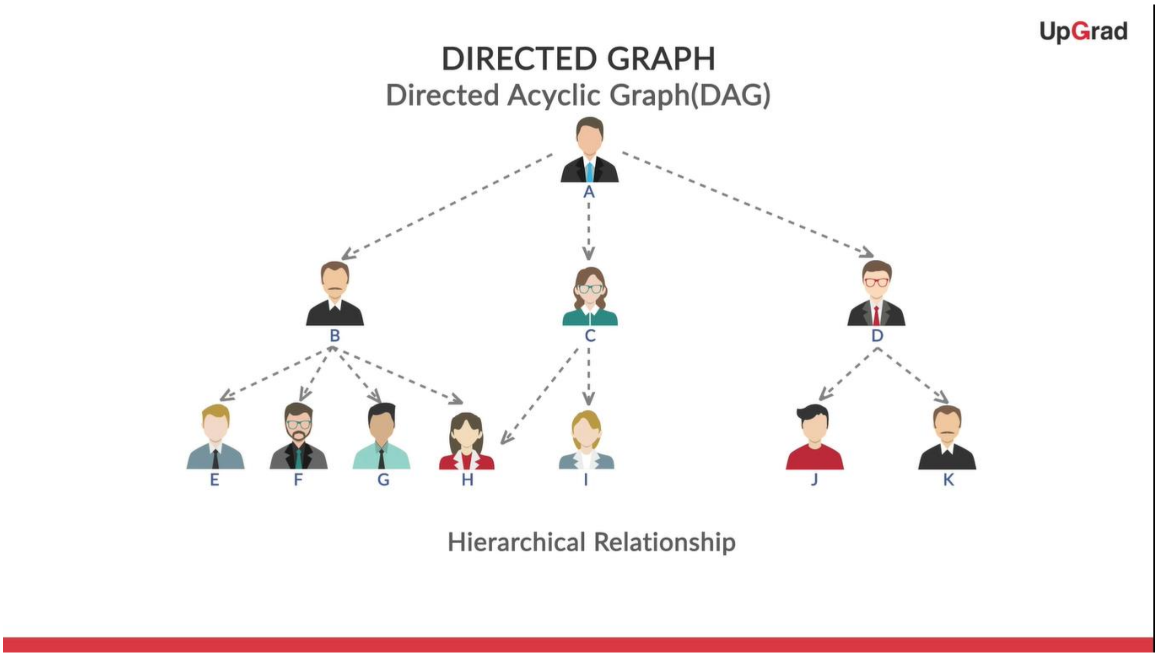
Based on the relationship between two nodes, graphs can be broadly classified as follows-

- Undirected graphs – symmetric relationship
  Example: The friendship relation among different friends in the social network represents undirected graphs. The edge connecting between A and B means A is a friend of B and vice-versa.
- Directed graphs – asymmetric relationship
  Example:
  The flight network among different airports can be represented using directed graph. In the below image, the directed arrow between Airport A and Airport B indicates a flight running from A to B but not from B to A.



In the above directed graph image, you can observe that if you start from Airport B and traverse through Airport C, Airport D and return back to Airport B. This is called as a cycle in the graph where you reach the same node after traversing across different nodes along the connected edges.

In the absence of such cycle in the directed graph can further be classified as directed acyclic graph.



You have also learnt that the directed acyclic graph is different from trees, in the above image node H has more than one parent node i.e., node B & C whereas in a tree data structure, each child node can have only one parent node.

Terminology

You have also learnt certain common terms while working on graphs as follows,

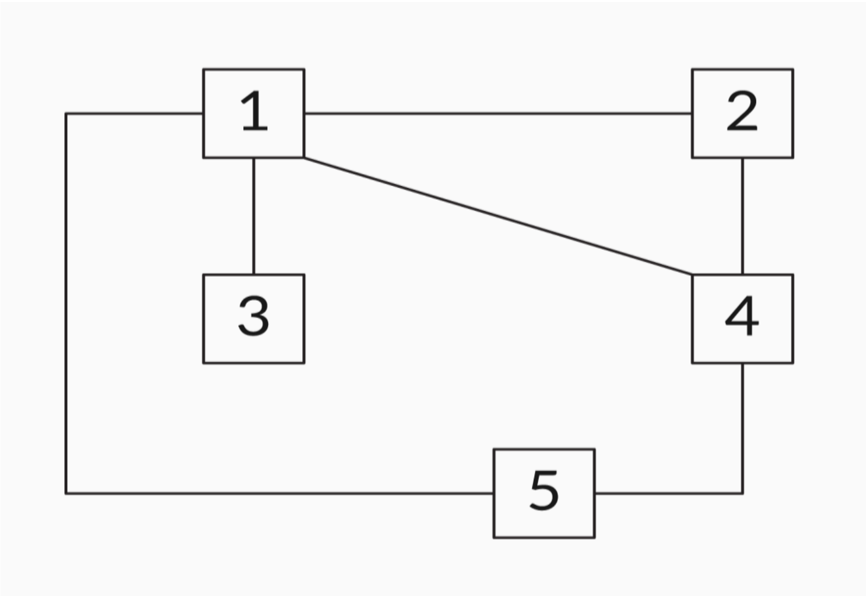
Neighbours: If two nodes are adjacent to each other and connected by an edge, then those nodes are called neighbours.

Degree: The number of edges that are connected to a node is called the degree of the node.

In case of directed graphs, this can be classified into:

- In-degree: The number of incoming edges to a node
- Out-degree: The number of outgoing edges from a node

  In the following undirected graph, degree of each node



Nodes	Neighbours	Degree
Node 1	{2, 3, 4, 5}	4
Node 2	{1, 4}	2
Node 3	{1}	1
Node 4	{1, 2, 5}	3
Node 5	{1, 4}	2

Path: When a series of vertices are connected by a sequence of edges between two specific nodes in a graph, the sequence is called a path. For example, in the above graph, {2, 1, 4, 5} indicates the path between the nodes 2 and 5, and the intermediate nodes are 1 and 4.

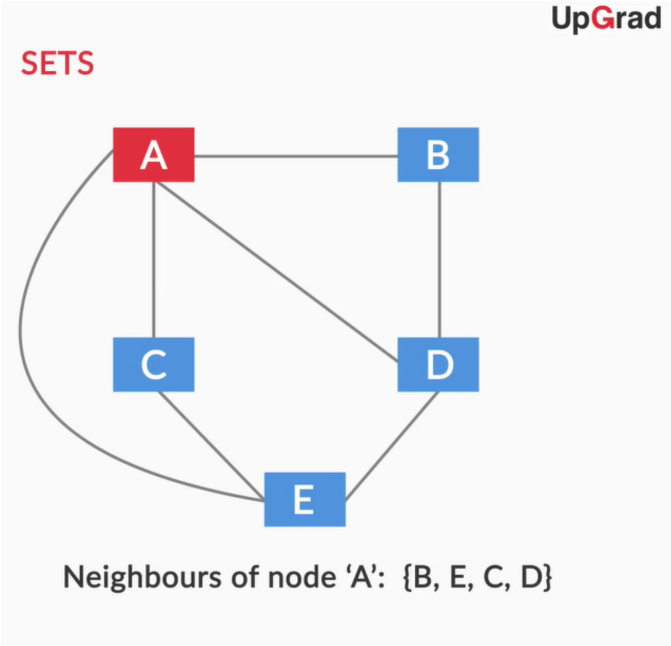
Sets and Maps

Before learning the traversal techniques of graph data structure, you have learnt the collection types sets and maps which are deliberately used to implement traversal techniques in Java.

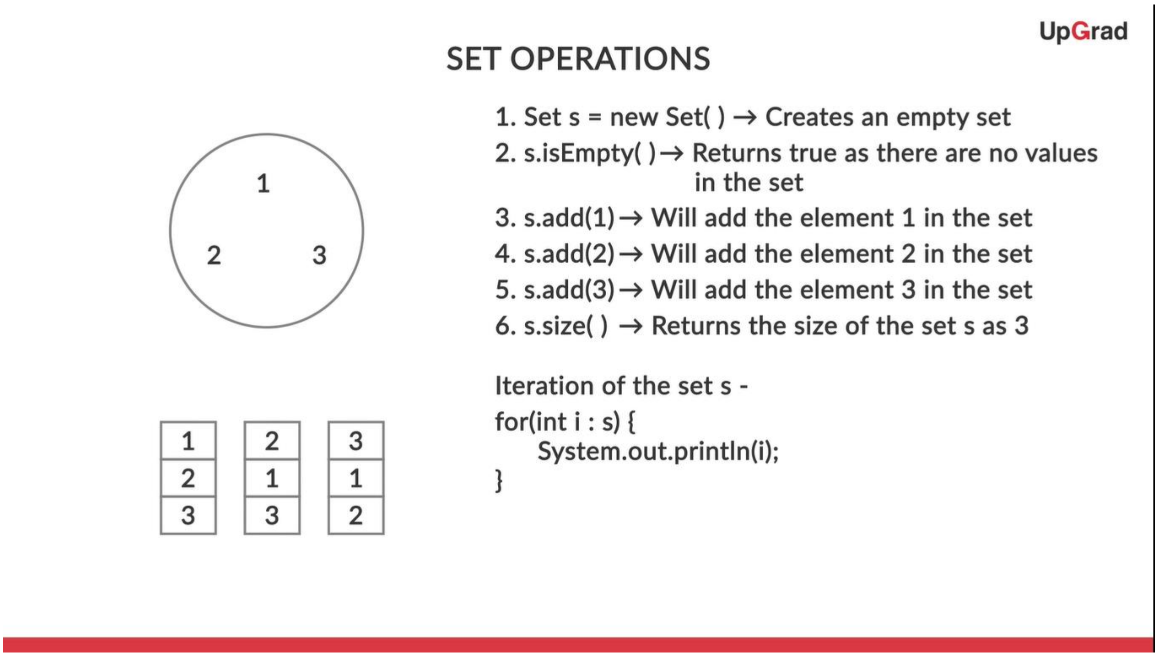
Sets

A set is a collection of unordered elements without any duplicates. It models from mathematical abstraction.

In the context of graph data structure, sets can be used to store the neighbours of a given node



You have also seen different operations performed on a given set which will be helpful in the implementation of traversal techniques of graph data structure.



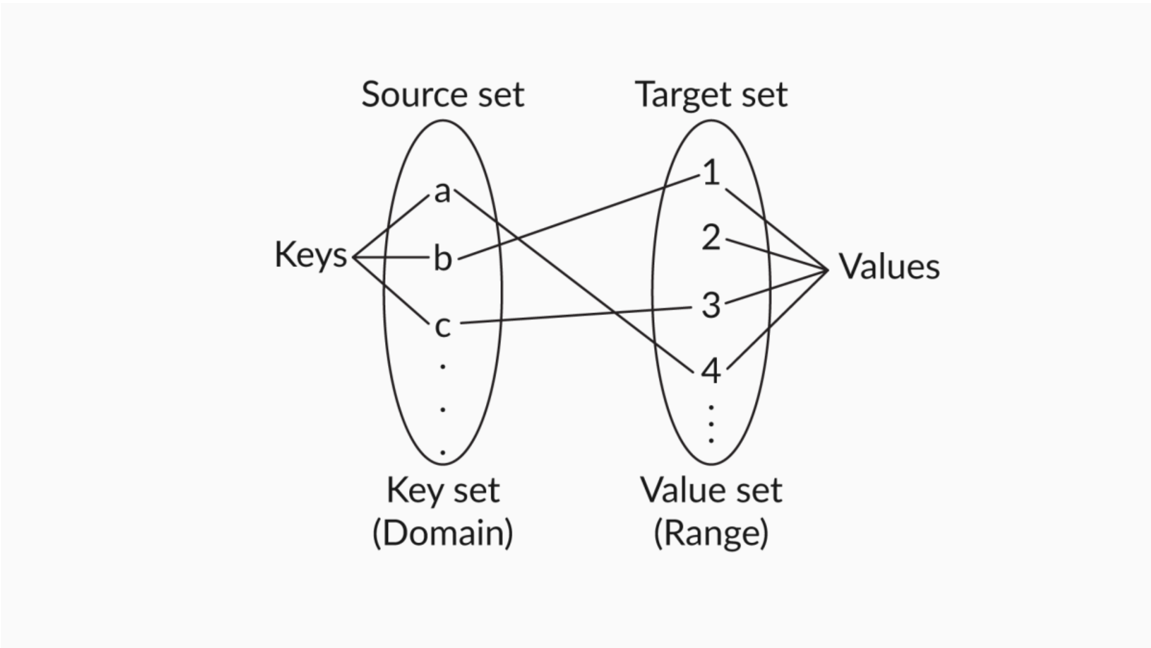
Properties:

1. Unlike lists and stacks, the elements present in a set do not follow any particular order. They are randomly present in the set.
2. The elements are not repeated in a given set.

Methods	Description
add(ele)	Adds an element to the set
clear()	Removes all elements from the set
contains(ele)	Returns true if a specified element is in the set
isEmpty()	Returns true if the set is empty
remove(ele)	Removes a specific element from the set
size()	Returns the number of elements in the set

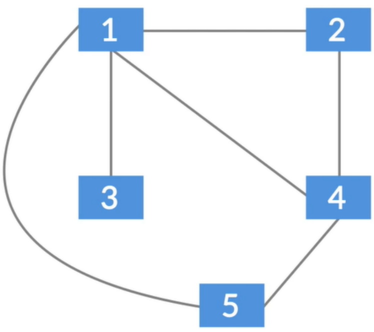
Maps

Maps is a collection type where it provides connection or mapping between the elements of source set(domain) and target set(range).



In the context of graph data structure, maps can be used to relate all the neighbours of a given node as follows, where nodes are keys and neighbours are values.

MAPS



Nodes(Keys)	Neighbours(Values)
1	{2, 3, 4, 5}
2	{1, 4}
3	{1}
4	{1, 2, 5}
5	{1, 4}

You have also seen different operations performed on maps which are deliberately used in the traversal techniques of graph data structure

UpGrad

MAP OPERATIONS

Price List

Commodity	Price (in Rupees/kg)
Potato	30
Onion	25

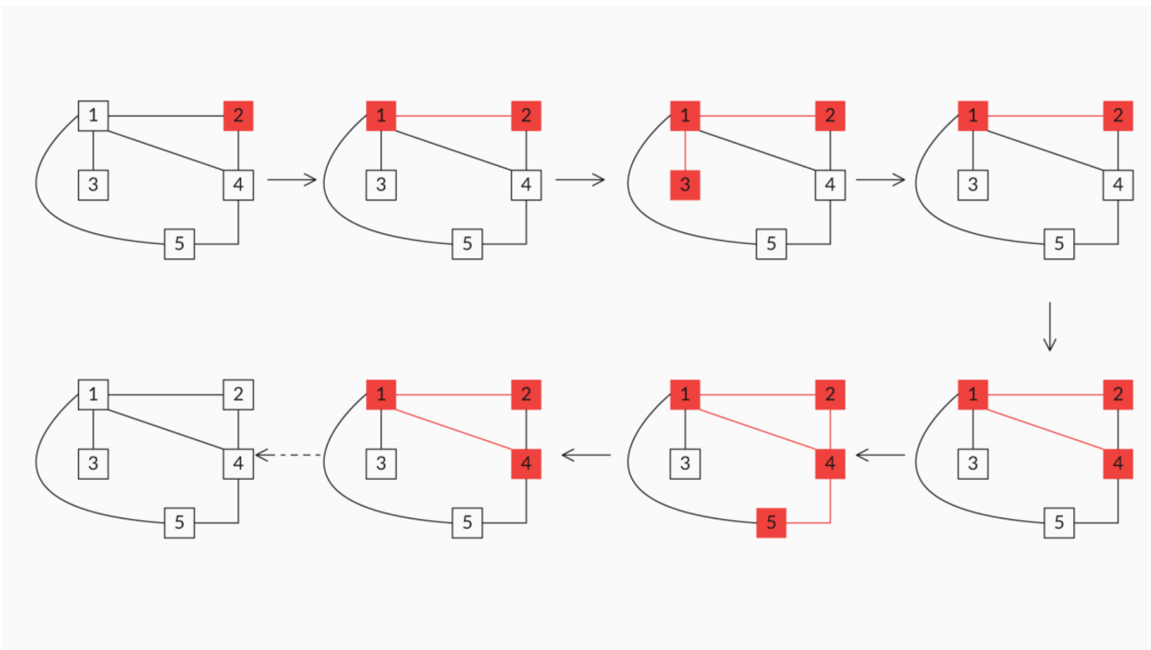
pl.get(potato) = returns the price of potatoes as 30
pl.put(onion, 25) = create a new entry
pl.put(potato, 35) = updates the corresponding entry in the price list

Depth-First Search - I

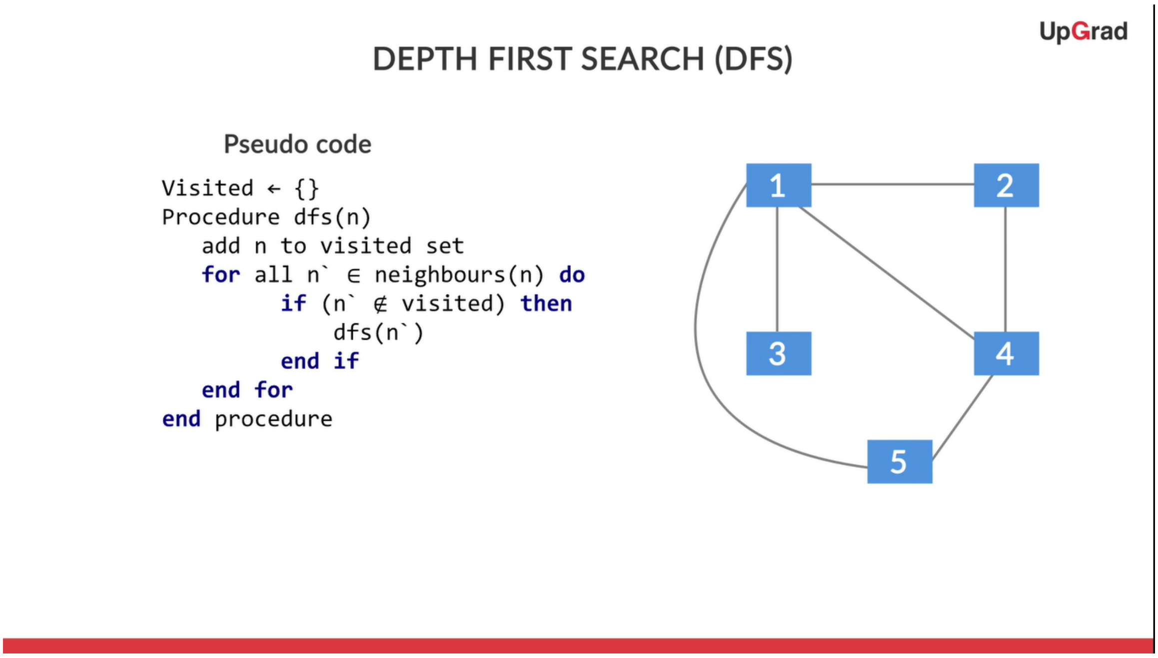
Depth-first search (DFS) is a traversal algorithm. From the start node, it traverses through any one of its neighbours and explores the farthest possible node in each branch before backtracking.

Backtracking happens when the search algorithm reaches a node where there are no neighbours to visit, or all the neighbours have already been visited. Then, the DFS algorithm traces back to the previous node and traverses any neighbour if it is left unvisited. Thus, backtracking helps to traverse through all the connected nodes in the graph and trace back to the start node.

The depth first search can be visualized in the following image, where the DFS starts from node 2 in the graph.



Now, you have the learnt the pseudo code for depth first search algorithm,



Then, professor has explained the pseudo code in detail using the same graph example.

The pseudocode for the depth-first search algorithm is as follows:

```
Procedure bfs(n)
  Q ← new Queue
  Visited ← { }
  enqueue (Q, n)
  Add n to visited set
  While Q is not empty
     n  ← dequeue(Q)
     for all n` ∈ neighbours(n)
        if (n` ∉ visited) then
           enqueue(Q, n`)
           add n` to visited set
        end if
     end for
  end while
end procedure
```

Step 1: The start node of the DFS algorithm is added to the visited set.

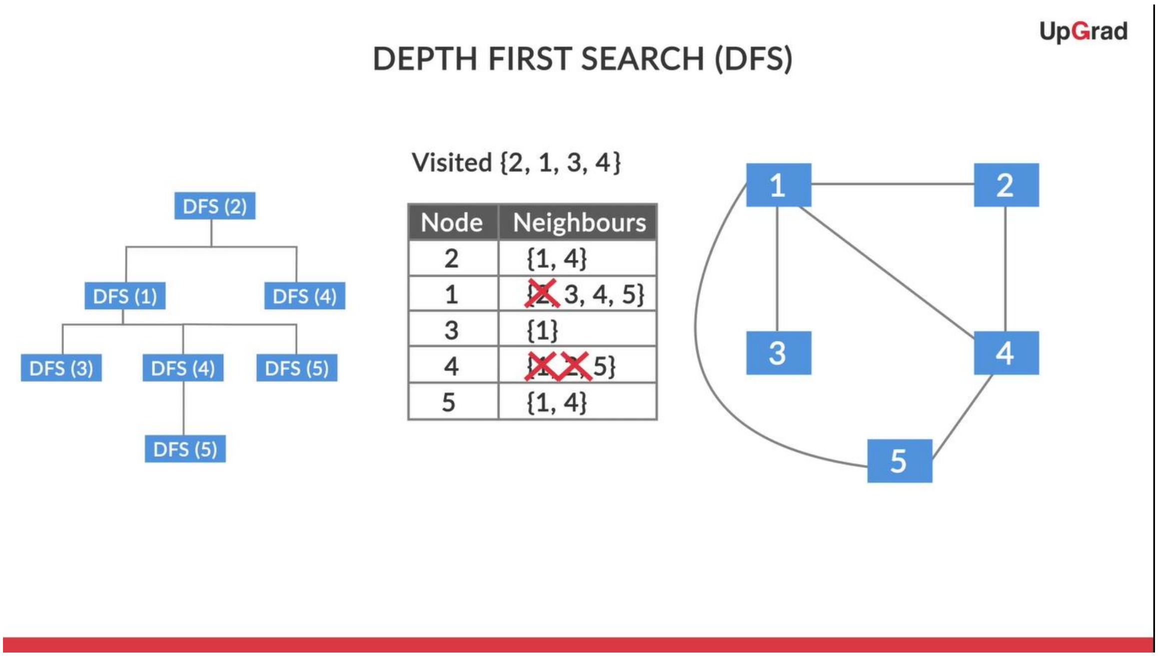
Step 2: The ‘for’ loop instruction set is executed for all the neighbours of the start node.

Step 3: Then if the neighbour node is unvisited, only then it recursively calls the dfs() method.

Step 4: Now, the unvisited neighbour node becomes the start node and repeats the above steps.

Step 5: The DFS traversal reaches a node from where there are no unvisited neighbour nodes; here, the recursive algorithm backtracks to the earlier traversed nodes and visits the remaining nodes.

Thus, the depth-first search recursively visits all the nodes along one branch and then backtracks to the unvisited neighbour nodes. Once all the nodes connected from the start node are visited, the algorithm ends.



In graphs, the visited nodes record must be maintained in order to avoid going in an infinite loop along the cycles present in the graph. This is the major difference between the DFS algorithm of graphs and trees, where you need not maintain the record of visited nodes.

Depth-First Search - II

Now, you have learnt the Java implementation of depth-first search on an application to record the order in which nodes are visited during depth-first search given the start node.

The professor has explained how DFS is implemented in the following code,

```java
import java.util.Set;
import java.util.HashSet;
import java.util.Map;
import java.util.HashMap;

public class DFS1 {

    public static void main(String[] args) {
        MyGraph<Integer> graph = new AdjacencyList<>();

        Node<Integer> n1 = graph.addNode(1);
        Node<Integer> n2 = graph.addNode(2);
        Node<Integer> n3 = graph.addNode(3);
        Node<Integer> n4 = graph.addNode(4);
        Node<Integer> n5 = graph.addNode(5);

        try {
            graph.addEdge(n1, n2);
            graph.addEdge(n1, n3);
            graph.addEdge(n1, n4);
            graph.addEdge(n1, n5);
            graph.addEdge(n2, n4);
```

```java
            graph.addEdge(n4, n5);

            Map<Node<Integer>, Integer> dfs_nums = DFS1.dfs(graph, n2);
            for (Node<Integer> n : dfs_nums.keySet()) {
                System.out.println(n + " : " + dfs_nums.get(n));
            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    public static Map<Node<Integer>, Integer> dfs(MyGraph<Integer> graph, Node<Integer> node)
throws Exception {
        Map<Node<Integer>, Integer> dfs_nums = new HashMap<Node<Integer>, Integer>();
        DFS1.dfs_rec(graph, node, dfs_nums);
        return dfs_nums;
    }

    private static void dfs_rec(MyGraph<Integer> graph, Node<Integer> node, Map<Node<Integer>,
Integer> dfs_nums) throws Exception {
        if (dfs_nums.containsKey(node)) {
            return;
        }
        dfs_nums.put(node, dfs_nums.size());
        for (Node<Integer> neighbour : graph.getAllNeighbours(node)) {
            DFS1.dfs_rec(graph, neighbour, dfs_nums);
        }
    }
}
```

MyGraph - It is the interface which represents graph abstract data type

AdjacencyList - This is the data structure which implements graph ADT

AddNode () - This method helps to add nodes to the variable 'graph'

AddEdge (n1, n2) - This method helps to add edge between the nodes n1, n2

dfs_nums – This variable maps each node to the order in which they have been visited during depth-first search.

dfs () method takes two parameters:

- graph, on which traversal to be done
- node, from which traversal to begin

dfs_rec() method takes three parameters:

- graph, on which traversal to be done
- node, from which traversal to begin
- dfs_nums, maps the relation between order of visit and nodes

So far, you have understood the depth-first search algorithm and its implementation but in order to perform the traversal technique, you need to implement graph structure in Java. For which you have learnt three different classes to implement graph data structure.

- Node class
- Edge class
- MyGraph interface

Node class is a generic class that represents all the nodes present in a graph

```
import java.util.Set;
import java.util.HashSet;
import java.util.Map;
import java.util.HashMap;

public class Node<T> {
    private T value;

    public Node(T value) {
        this.value = value;
    }

    public String toString() {
        return this.value.toString();
    }

}
```

Edge class has two variables representing the nodes that are connected through an edge and all the edges in a simple undirected graph

```
import java.util.Set;
import java.util.HashSet;
import java.util.Map;
import java.util.HashMap;

public class Edge<T> {
    public final Node<T> n1;
    public final Node<T> n2;

    public Edge(Node<T> n1, Node<T> n2) {
        this.n1 = n1;
        this.n2 = n2;
    }

    public boolean equals(Edge<T> e) {
        return (this.n1.equals(e.n1) && this.n2.equals(n2)) ||
               (this.n2.equals(e.n1) && this.n1.equals(n2));
    }
}
```

MyGraph is an interface that contains the various methods such as

- getAllNodes() – This method helps to retrieve all the nodes present in graph
- addNode() – This method adds nodes to the graph
- addEdge() – This method adds an edge between two nodes
- getAllNeighbours() – This method helps to find all the neighbours of a given node

```
import java.util.Set;
import java.util.HashSet;
import java.util.Map;
import java.util.HashMap;

public interface MyGraph<T> {

    Set<Node<T>> getAllNodes();

    Node<T> addNode(T e);

    /*
      throws exception when either of the nodes is not a member of the graph.
    */
    Edge<T> addEdge(Node<T> n1, Node<T> n2) throws Exception;

    /*
```

```
      throws exception when the node is not a member of the graph.
    */
    Set<Node<T>> getAllNeighbours(Node<T> node) throws Exception;
}
```

Depth-First Search - III

You have seen another application of depth- first search which is to find all the reachable nodes from a given node. In this application, professor has demonstrated how depth first search reaches to all the connected nodes in a graph.

```
import java.util.Set;
import java.util.HashSet;
import java.util.Map;
import java.util.HashMap;

public class DFS2 {

    public static void main(String[] args) {
        MyGraph<Integer> graph = new AdjacencyList<Integer>();

        Node<Integer> n1 = graph.addNode(1);
        Node<Integer> n2 = graph.addNode(2);
        Node<Integer> n3 = graph.addNode(3);
        Node<Integer> n4 = graph.addNode(4);
        Node<Integer> n5 = graph.addNode(5);
        Node<Integer> n6 = graph.addNode(6);

        try {
            graph.addEdge(n1, n2);
            graph.addEdge(n1, n3);
            graph.addEdge(n1, n4);
            graph.addEdge(n1, n5);
            graph.addEdge(n2, n4);
            graph.addEdge(n3, n4);
            graph.addEdge(n3, n5);
            graph.addEdge(n4, n5);
            graph.addEdge(n6, n5);
            Set<Node<Integer>> reachable = DFS2.dfs(graph, n2);
            for (Node<Integer> n : reachable) {
                System.out.println(n);
            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    public static Set<Node<Integer>> dfs(MyGraph<Integer> graph, Node<Integer> n) throws
Exception {
        Set<Node<Integer>> reachable = new HashSet<Node<Integer>>();
        DFS2.dfs_rec(graph, n, reachable);
        return reachable;
    }

    private static void dfs_rec(MyGraph<Integer> graph, Node<Integer> node, Set<Node<Integer>>
reachable) throws Exception {
        if (reachable.contains(node)) {
            return;
        }
        reachable.add(node);
        for (Node<Integer> neighbour : graph.getAllNeighbours(node)) {
            DFS2.dfs_rec(graph, neighbour, reachable);
        }
    }
}
```

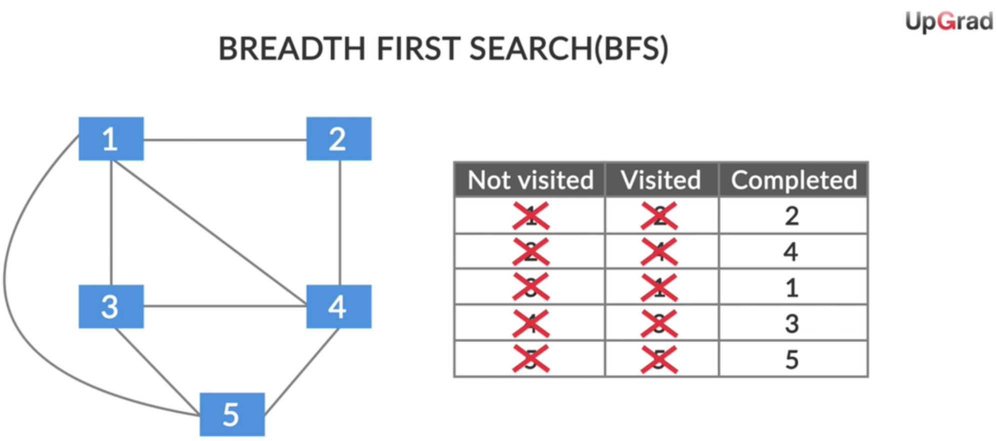
Breadth-First Search - I

Breadth-first search is a traversing algorithm where traversing starts from the start node and then explores the immediate neighbours of the start node; then the traversing moves towards the next-level neighbours of the graph structure. As the name suggests, the traversal across the graph happens breadthwise.

To implement the breadth-first search, you need to consider the stage of each node. Nodes, in general, are considered to be in three different stages as,

- Not visited
- Visited
- Completed

In BFS algorithm, nodes are marked as visited during the traversal, in order to avoid the infinite loops caused because of the possibilities of cycles in a graph structure.



Breadth-First Search - II

Having understood the breadth-first search traversal technique, professor has explained the pseudo code of breadth-first search algorithm.The pseudocode is as follows:

```
Procedure bfs(n)
   Q ← new Queue
   Visited ← { }
   enqueue (Q, n)
   Add n to visited set
   While Q is not empty
       n  ← dequeue(Q)
       for all n` ∈ neighbours(n)
           if (n` ∉ visited) then
               enqueue(Q, n`)
               add n` to visited set
           end if
       end for
   end while
end procedure
```

Step 1: The start node is enqueued and also marked as visited in the following set of instructions,

- enqueue (Q, n)
- Add n to visited set

Step 2: 'While' loop instruction set is executed when the queue is not empty

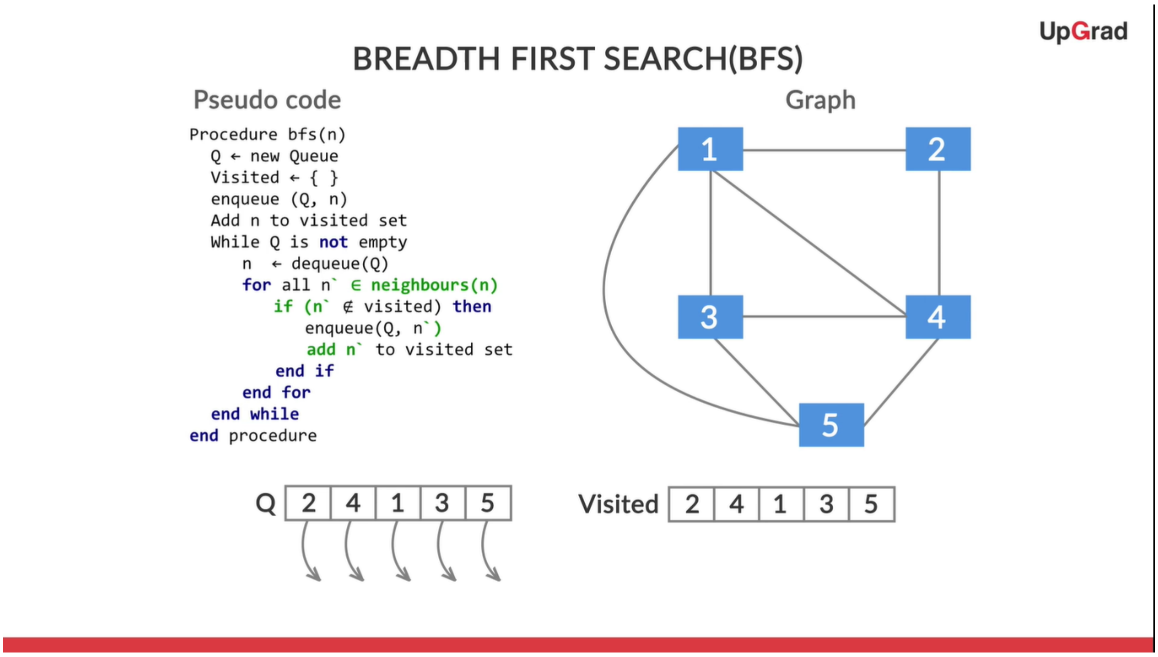
Step 3: For each iteration of while loop, a node gets dequeued

Step 4: Now 'for' loop runs till all the unvisited neighbours of the dequeued node(n) are enqueued and also marked as visited.

Step 5: For the first iteration of the while loop, all the neighbour nodes of the start node are enqueued and on the second iteration, all the next level unvisited neighbour nodes of one of the neighbour node are enqueued.

This way, all the neighbour nodes are enqueued and visited level wise from the start node and after a certain number of iterations, all the nodes are dequeued and the algorithm ends.

Then professor has explained the pseudo code of breadth-first search algorithm on the graph example as follows,



Now, breadth-first search algorithm is implemented in Java to find the different levels of nodes based on breadth-first traversal of graph from a given start node.

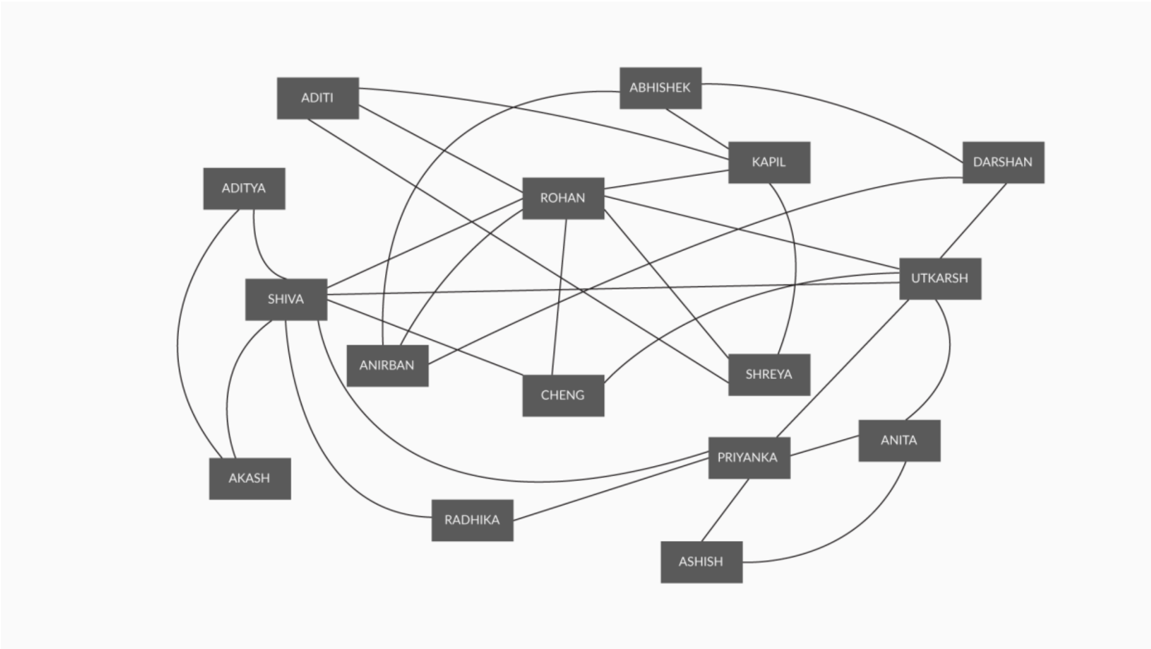
Professor has explained the implementation of BFS in the following code,

```java
import java.util.Set;
import java.util.HashSet;
import java.util.Map;
import java.util.HashMap;
import java.util.Queue;
import java.util.LinkedList;

public class BFS {

    public static void main(String[] args) {
        MyGraph<Integer> graph = new AdjacencyList<Integer>();
```

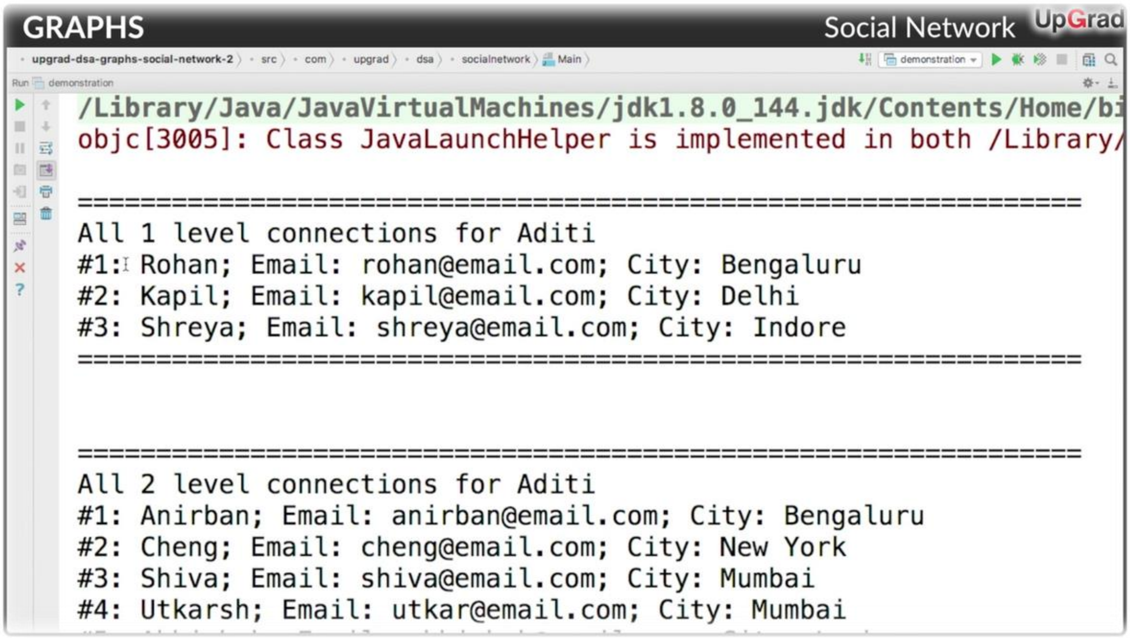
```java
        Node<Integer> n1 = graph.addNode(1);
        Node<Integer> n2 = graph.addNode(2);
        Node<Integer> n3 = graph.addNode(3);
        Node<Integer> n4 = graph.addNode(4);
        Node<Integer> n5 = graph.addNode(5);

        try {
            graph.addEdge(n1, n2);
            graph.addEdge(n1, n3);
            graph.addEdge(n1, n4);
            graph.addEdge(n1, n5);
            graph.addEdge(n2, n4);
            graph.addEdge(n3, n4);
            graph.addEdge(n3, n5);
            graph.addEdge(n4, n5);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
        Map<Node<Integer>, Integer> levels;
        try {
            levels = BFS.bfs(graph, n2);
            for (Node<Integer> n : levels.keySet()) {
                System.out.println(n + " : " + levels.get(n));
            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
            System.exit(1);
        }

    }

    public static Map<Node<Integer>, Integer> bfs(MyGraph<Integer> graph, Node<Integer> n)
throws Exception {
        Queue<Node<Integer>> queue = new LinkedList<Node<Integer>>();
        Map<Node<Integer>, Integer> levels = new HashMap<Node<Integer>, Integer>();
        queue.add(n);
        levels.put(n, 0);
        while (queue.isEmpty() == false) {
            Node<Integer> nextNode = queue.remove();
            Set<Node<Integer>> neighbours = graph.getAllNeighbours(nextNode);
            for (Node<Integer> neighbour : neighbours) {
                if (levels.containsKey(neighbour) == false) {
                    queue.add(neighbour);
                    levels.put(neighbour, levels.get(nextNode) + 1);
                }
            }
        }
        return levels;
    }
}
```

Industry Demonstration - I

The industry relevance of the two traversal techniques with respect to the graphs has been discussed using a real-world example on social networks such as Facebook, LinkedIn. In order to perform the applications of traversal techniques, we have considered a small set of people and connections between them, as shown in the following graph

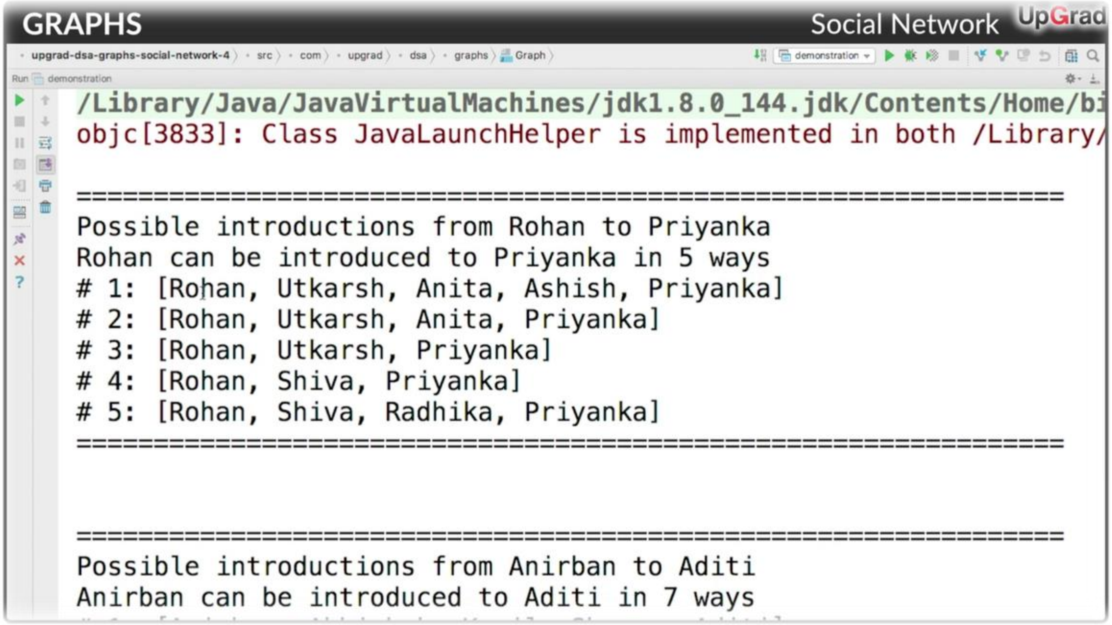


The first application discussed on the social network is to find the different level of connections of a person in the social network, you have learnt to implement the application in Java using breadth-first search algorithm to find different level of connections of a person in the social network

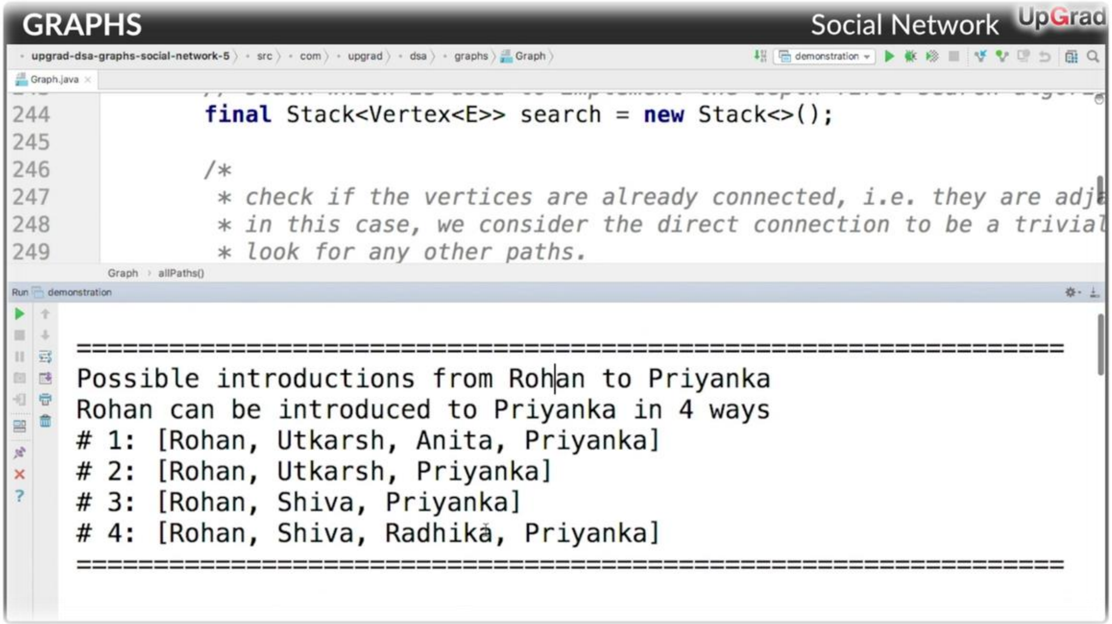


Industry Demonstration - II

In the second real-world application, depth first search algorithm is implemented to find how a person can be introduced to another person along the path connected with other people in social network. You have also learnt the java implementation of the application and the output is as follows,



On the same implementation of DFS, industry expert has introduced depth limit such that only 1st level, 2nd level, 3rd level connections can only be introduced to a person. In which case the output is as follows,



Summary

In this session, you have learnt

- What is a graph ADT?
- Different types of graphs
    - Undirected graph
    - Directed graph
        - Directed acyclic graph

- Differences between graphs and trees
- Depth-first search
    - Pseudocode
    - Applications
        - Compute the order of visit of all nodes in DFS traversal
        - Compute the set of reachable nodes from a given start node
        - Compute the different ways of introducing one person to another in a social network

- Breadth-First search
    - Pseudocode
    - Applications
        - Compute the level of each node in BFS traversal
        - Compute the nth level connections in a social network

Session 2: Graph Algorithms

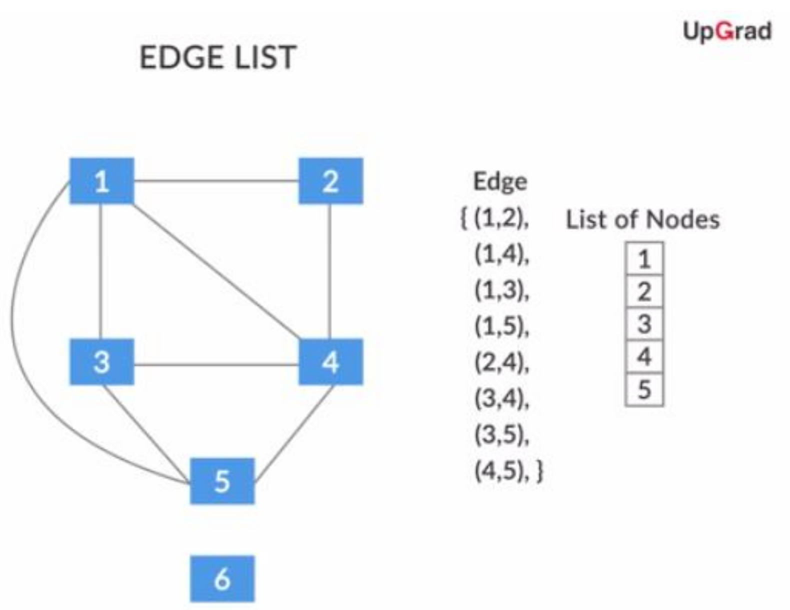
Let's revisit the topics learnt in Welcome to the session on Graph Algorithms.

In this session we gave you a brief introduction on what are the things you will be learning in this session on Graph Algorithm.

You were told that you will be learning about the following-

- Edge list
- Adjacency matrix
- Adjacency list
- Dijkstra's algorithm
- How to apply all of the above in real life

Introduction to Edge Lists



In this segment you learnt about the first graph ADT implementation i.e. an edge list and how it is implemented using the MyGraph interface.

This is the graph which we have been working with. There are these edges between -

- 1 and 2
- 1 and 4
- 3 and 4 and so on
- So all we did was to maintain a list of these edges. So the edge list class would have an attribute named edges, and it's just a list of all the edges, and in this case it suffices to represent the edges as pairs of the end point nodes.

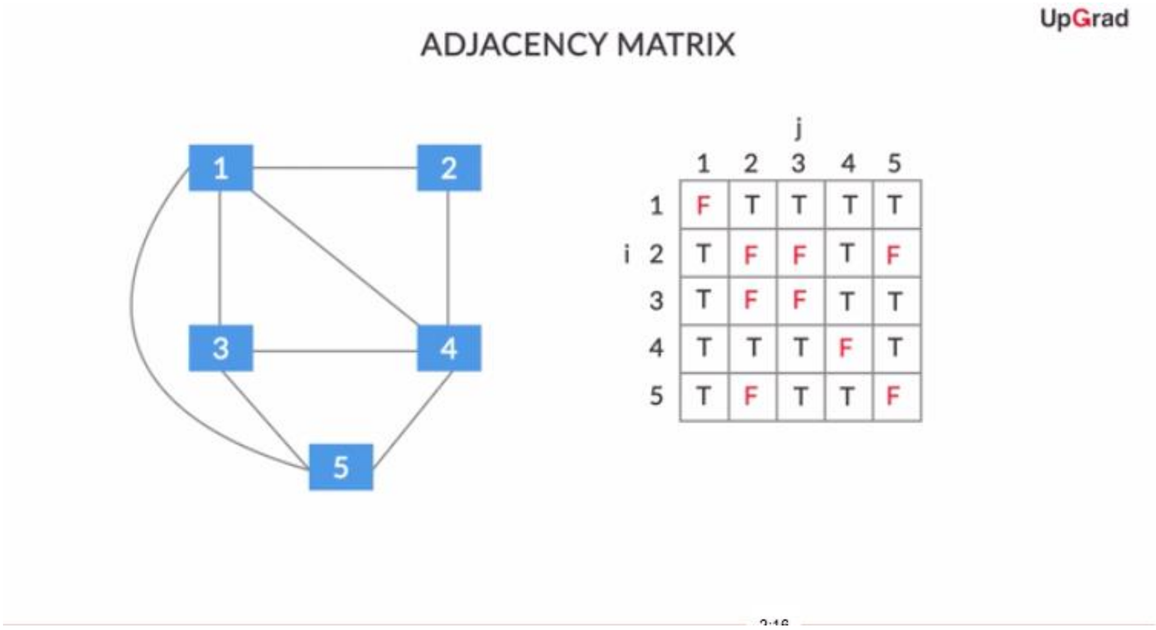
For example, the edge between 1 and 2 is merely a pair of the two nodes.
You also learnt that an edge list fails in a scenario when you have an isolated edge in your graph.

Introduction to Adjacency Matrix

In this segment you were introduced to another graph implementation: the 'adjacency matrix'. As the name suggests, an adjacency matrix is a two-dimensional boolean matrix that represents a connection between nodes.

The two-dimensional matrix is a Boolean matrix. Here, the number of rows and the number of columns are the same. It's a square matrix and that is equal to the number of nodes in the graph. Each row and each column corresponds to a particular node.

If you want to represent that there exists an edge between the node I and node J then you would place a true value in the cell corresponding to the Ith row and Jth column.



In this segment you saw how you can represent an adjacency matrix using a two-dimensional matrix of boolean values.

Adjacency Matrix Implementation

In this segment you learnt about the MyGraph interface, which has four functions, namely

- getAllNodes
- addNode
- addEdge
- getAllNeighbors

Let's recap what these functions do, in detail-

- **getAllNodes**: This function converts a list of nodes into a set of nodes.
- **add Node** This function returns a newly created node.
- **addEdge** method returns a newly created edge between node n1 and n2. So in order to find the row and the column corresponding to the nodes n1 and n2, we have to find out what is their position in the node list. And that we do by using the Index Of operation.
- **getallNeighbors** method returns the set of nodes which are neighbors to the node known, and its implementation also happens to be fairly simple. All you have to do is to find out the row which corresponds to the node that we are interested in.

To recap the code for the four methods were as follows: -

- **getAllNodes**

```java
public Set<Node<T>>getAllNodes(){
    Set<Node<T>>set=new HashSet<Node<T>>();
    for(Node n:this.nodes){
        set.add(n);
    }
    return set;
}
```

- **addNode**

```java
public Node<T> addNode(T e){
    Node<T> newNode=new Node<T>(e);
    this.nodes.add(newNode);
    boolean[][]m=new boolean[this.nodes.size()][this.nodes.size()];
    for(int i=0;i<m.length;i++){
        for(int j=0;j<m.length;j++){
            m[i][j]=false;
        }
    }
    for(int i=0;i< this.adjacencyMatrix.length;i++){
        for(int j=0;j< this.adjacencyMatrix.length;j++){
            m[i][j]=this.adjacencyMatrix[i][j];
        }
    }
    this.adjacencyMatrix=m;
    return newNode;
}
```

- **addEdge**

```java
public Set<Node<T>> getAllNeighbours(Node<T> node) throws Exception {
    if(this.nodes.contains(node) == false) {
        throw new Exception("node not contained in this graph.");
    }
    Set<Node<T>> neighbours = new HashSet<Node<T>>();
    int row = this.nodes.indexOf(node);
    for(int i = 0; i < this.adjacencyMatrix.length; i++) {
        if(this.adjacencyMatrix[row][i] == true) {
            neighbours.add(this.nodes.get(i));
        }
    }
    return neighbours;
}
```

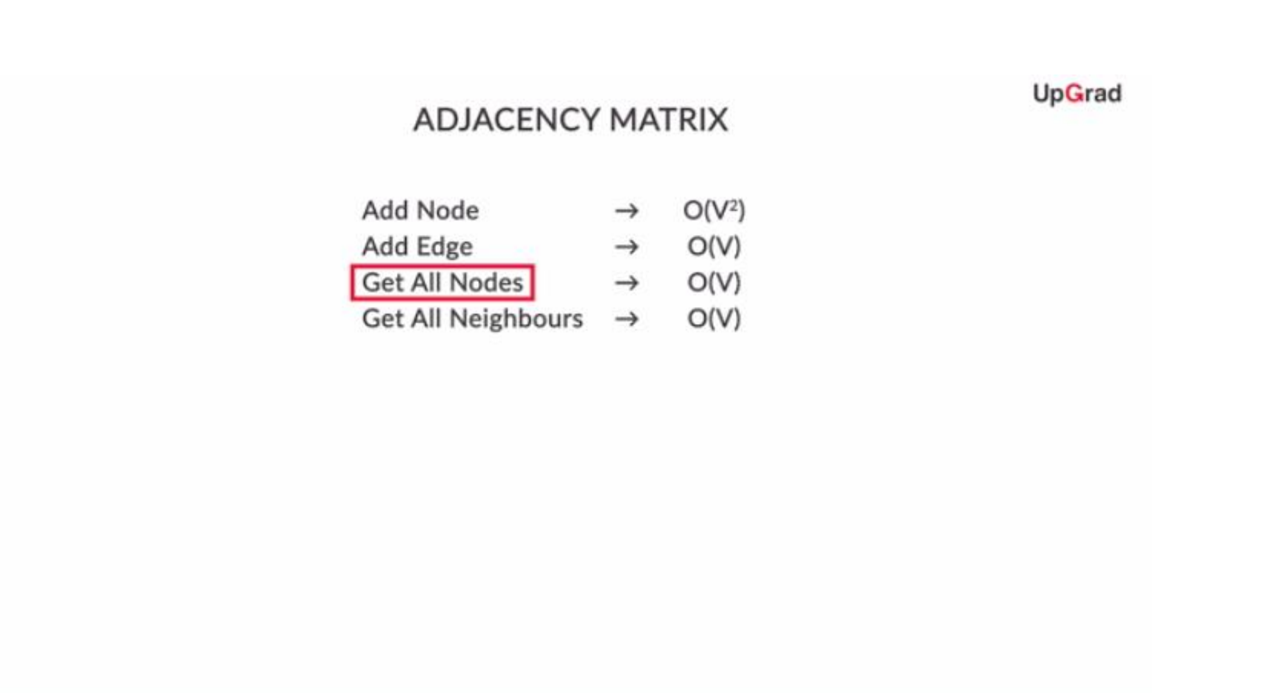
- **getAllNeighbors**

```java
public Set<Node<T>>getAllNeighbours(Node<T> node)throws Exception{
    if(this.nodes.contains(node)==false){
        throw new Exception("node not contained in this graph.");
    }
    Set<Node<T>>neighbours=new HashSet<Node<T>>();
    int row=this.nodes.indexOf(node);
    for(int i=0;i< this.adjacencyMatrix.length;i++){
        if(this.adjacencyMatrix[row][i]==true){
            neighbours.add(this.nodes.get(i));
        }
    }
    return neighbours;
}
```

Performance Characteristics of Adjacency Matrix

The most important aspect of any algorithm is its performance characteristics, which determine how it will perform in a given situation or circumstance. Therefore, you need to take into account an algorithm's performance when you make your choices.

In this segment you calculated the time complexity for an adjacency matrix implementation, specifically the time complexity for the following four methods: addNode, addEdge, getAllNodes, and getAllNeighbours. You also explored the performance characteristics of these methods.

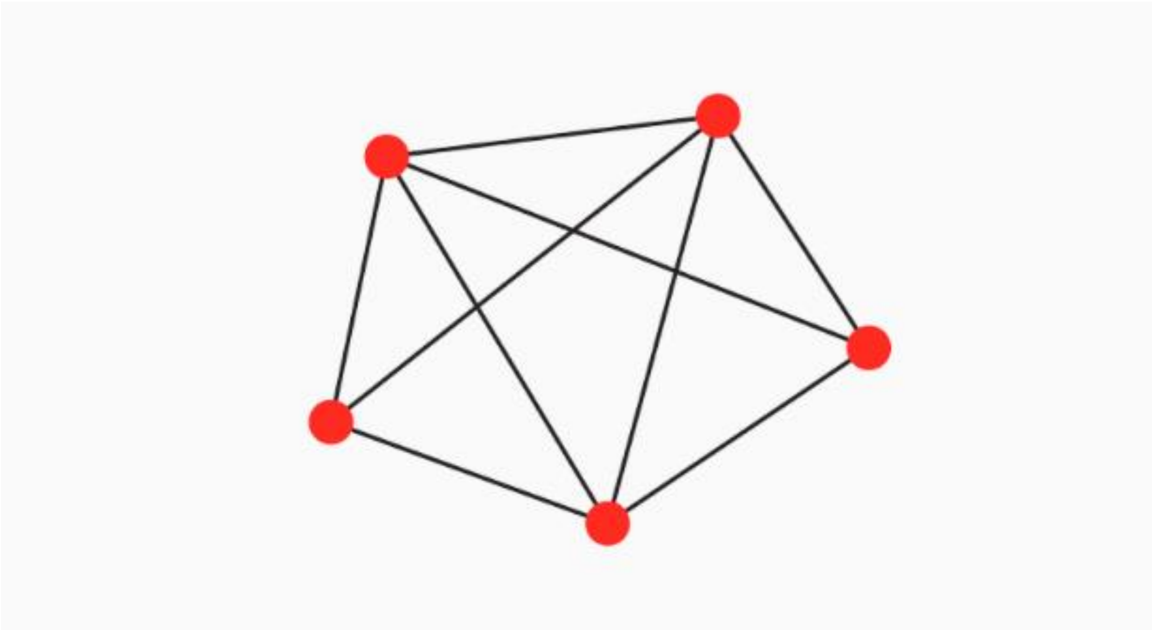


In this segment you learnt

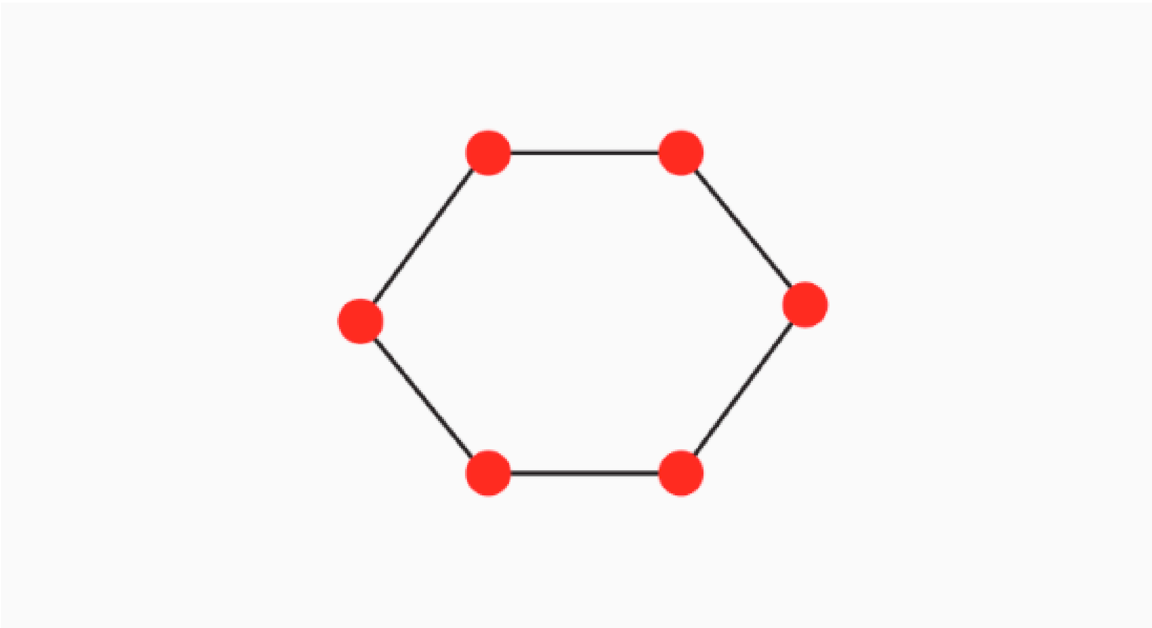
- How to calculate the time complexity of each of the methods in an adjacency matrix implementation
- The time complexities for the methods turn out to be
- For addNode method - O(V * V)
- For addEdge method - O(V)
- For getAllNodes method - O(V)
- For getAllNeighbors - O(V)
- The practices you can follow to improve the time complexities of the following methods: addEdge and getAllNeighbors

You also learnt about 'dense graphs' and 'sparse graphs' in detail.

Dense graphs: Dense graphs are densely connected, which means they have the maximum number of edges between nodes, i.e. there is an edge from each node to every other node. Given below is an example of a dense graph.



Sparse graphs: Sparse graphs are connected graphs with the minimum or a small number of edges connecting nodes. In sparse graphs, there may or may not be an edge between two nodes. Here, usually, the number of edges is n, which is also the number of vertices.



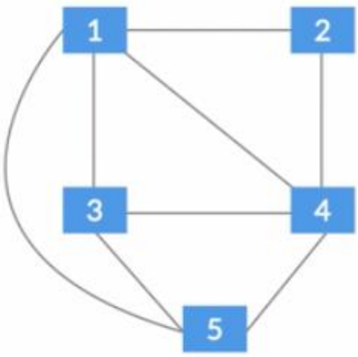
Introduction to and Implementation of Adjacency Lists

In this segment, you learnt about our third and last graph ADT implementation: the ‘Adjacency list’. You learnt the following:

- The purpose behind why we came up with this third method
- The data structure that you need to use for an adjacency list implementation (you will also see the Java code for the same)
- How to distinguish when you should choose an adjacency list over the other two implementations

UpGrad

ADJACENCY LIST



Key	Value
1	{ 2, 3, 4, 5 }
2	{ 1, 4 }
3	{ 1, 4, 5 }
4	{ 1, 2, 3, 5 }
5	{ 1, 3, 4 }

Map

- At the end of this segment, you know how this list is implemented and how it is different from an edge list and an adjacency matrix. You also saw the Java implementation of an adjacency list by using the MyGraph interface. After attempting the questions below, you are sure to get an even clearer idea of all of this.

Performance Characteristics

So far you learnt about, edge lists, adjacency matrix, and adjacency list. You explored the data structures required for each of these implementations. You even worked out the performance characteristics of an adjacency matrix in great detail. Now, it's time to compare these three graph ADT representations in terms of their performance characteristics.

UpGrad

GRAPHS

Performance Characteristics

	EL	AM	AL
Get all nodes	O(1)	O(V)	O(1)
Add Node	O(E)	O(V^2)	O(1)
Add Edge	O(1)	O(V)	O(1)
Get all neighbours	O(E)	O(V)	O(1)
Space complexity	O(V + E)	V + V^2 O(V^2)	O(V + E)

In this segment you saw situations in which particular algorithms do not perform as well as others; you will also had a look at the improvements (if any) that can be made to the performance characteristics of the graph ADT implementations.

GRAPHS

Performance Characteristics

	EL	AM	AL
Get all nodes	O(1)	O(V)	O(1)
Add Node	O(E)	O(V^2)	O(1)
Add Edge	O(1)	O(V)	O(1)
Get all neighbours	O(E)	O(V)	O(1)
Space complexity	O(V + E)	V + V^2 O(V^2)	O(V + E)

1. Its possible to improve the time performance of get all nodes and add edge in adjacency matrix
2. With proper implementation of the set of nodes in the adjacency matrix, you would actually drop the O(V) in case of get all nodes to O(1)
3. You could drop the O(V) of add edge to O(N)
4. For space complexity of sparse graphs we can use edge lists and adjacency lists
5. For space complexity of dense graphs, use an adjacency matrix

To sum up everything from this segment, you made a detailed comparison between the three graph implementations. So, the video in this segment was basically about the core of all the implementations you learnt about, where you explored the performance characteristics of each implementation.

At the end adjacency list was concluded as behaving the most optimally, but, again, this depends on your requirements.

Introduction to Dijkstra’s Algorithm

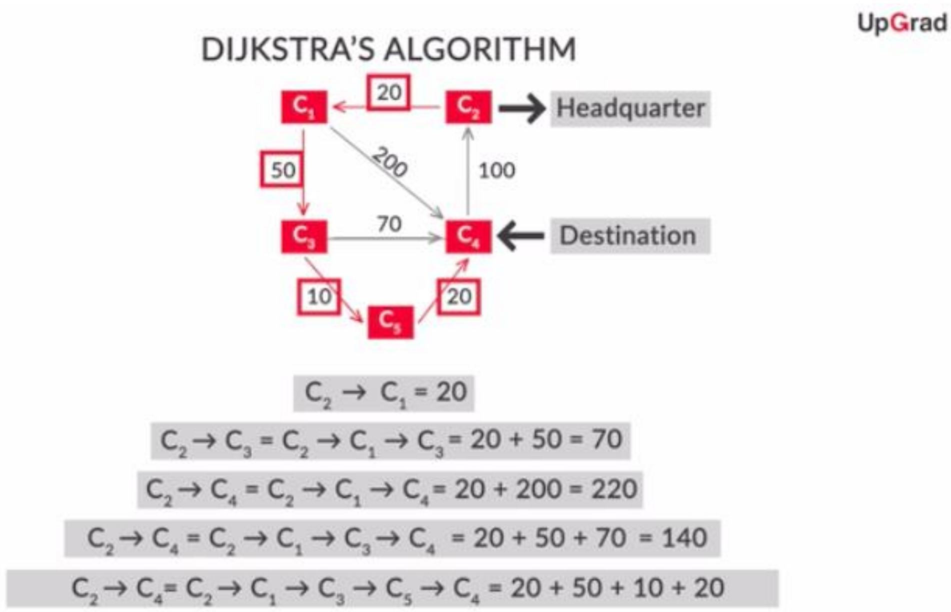
In this segment you were introduced to the famous Graph Algorithm i.e. Dijkstra’s Algorithm .

This algorithm was named after Edsker Dijkstra who was the inventor of this algorithm.

- What does Dijkstra's shortest path algorithm do?
  - Given a graph g and a node in that graph n, it computes the shortest distances from n to all the other nodes of that graph.
- Dijkstra's algorithm falls in the broad category of greedy algorithms.
- Dijkstra's algorithm will work only on graphs with positive or non-negative edge weights

Dijkstra's shortest path algorithm has a wide variety of applications.

- We introduced you to one of them, considering that you are a logistic company which supplies various products or various parcels to a number of other places or other ports.
- you would be interested to send your parcels to various ports through the path which cost you the least and therefore you would be interested to find out the paths which-have the least cost for each of the other ports to which you want to send your parcels.
- Assuming C2 to be the headquarter we wanted to send the parcel from C2 to all the other ports such that it would cost us the least to reach each destination from C2.



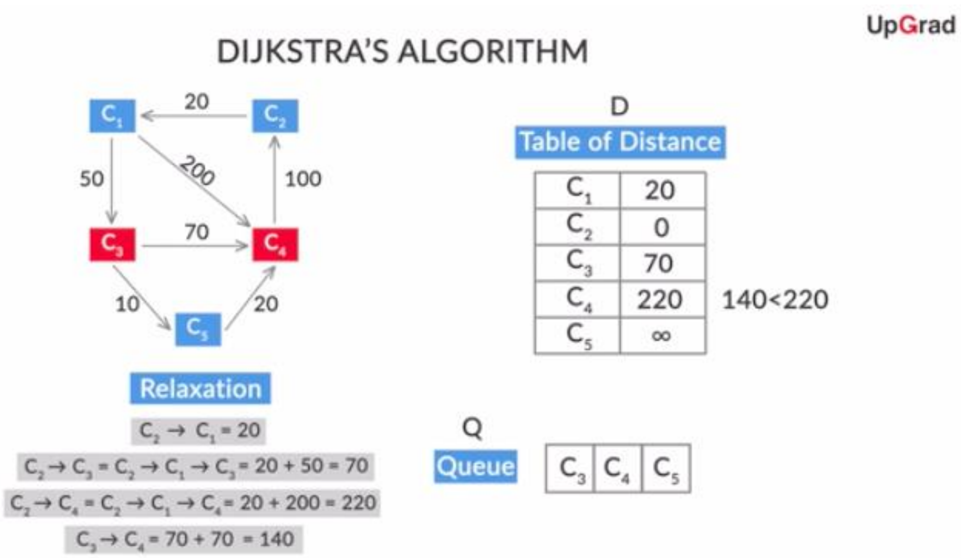
So, to sum up in this segment—

- You learnt about a new algorithm, Dijkstra's algorithm, which calculates the shortest path from a source to a destination.

Dijkstra's Algorithm

In this segment, you saw the following:

- Maintaining two data structures
  - Table of distance which was an array
  - Priority Queue
- How to make updates in the table of distance, including conditional updates
- When the priority queue comes into the picture and when you use the dequeue and enqueue operations on this priority queue



So, to sum up in this segment, you learnt the following:

- The concept of edge relaxation or relaxation
- How to make conditional updates in the table of distances, i.e. how relaxation works
- How you reach your final answer at the end of the while loop

You were also given the pseudocode of the Dijkstra's algorithm is as follows:

```
Procedure Dijkstra(graph, node)
  While Q is not empty
    Dequeue the first element from priority queue
    nextNode ← front element of queue after previous deque
    if nextNode is not null
      Relax the edges if necessary
    end if
  end while
end procedure
```

Step 1: 'While' loop instruction set is executed when the queue is not empty

Step 2: For each iteration of while loop, the node in the front gets dequeued

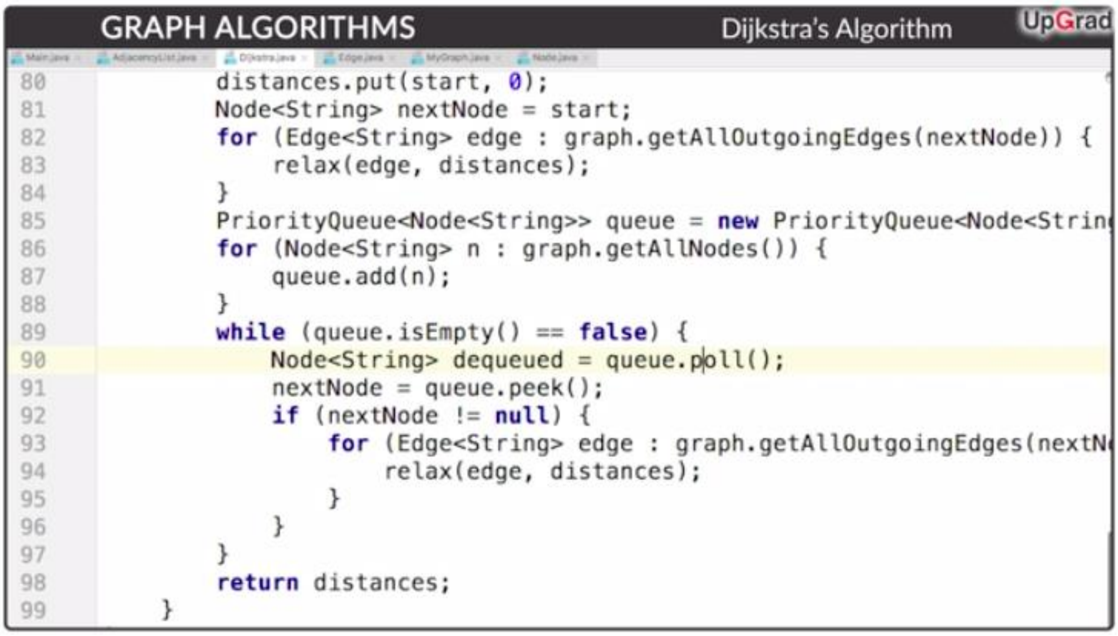
Step 3: nextNode will store the node in front of priority queue after the last deque

Step 4: We will check if the value of nextNode is null or not, if It is not null we will proceed and do edge relaxation on the nodes where require.

This way, we will end up getting the shortest distances from the given node to all the other nodes in our Table of Distance.

Dijkstra's Implementation

This segment we saw how Dijkstra's algorithm helps us in making updates to the distances in the Table of distance and the priority queue using the concept of Edge relaxation. The pseudocode that we saw earlier we saw it working here through actual Java code.



The main logic for dijkstra's goes inside the While loop which looked like this-

```
while (queue.isEmpty() == false) {
    Node<String> dequeued = queue.poll();
    nextNode = queue.peek();
    if (nextNode != null) {
        for (Edge<String> edge : graph.getAllOutgoingEdges(nextNode)) {
            relax(edge, distances);
        }
    }
}
return distances;
}
```

In the end of this segment we saw that our java code was working fine and the values obtained in our Table of Distances was exactly the same as we had seen before.