# Topics to cover:

1. Graph traversal
2. Shortest path

# 1. Graph traversal

## 1.1 Breadth First Traversal/Search (BFS)

Breadth First Search (BFS) is a graph traversal algorithm that explores all the vertices in a graph at the current depth before moving on to the vertices at the next depth level. It starts at a specified vertex and visits all its neighbors before moving on to the next level of neighbors. BFS is commonly used in algorithms for pathfinding, connected components, and shortest path problems in graphs.

### 1.1.1 Relation between BFS for Graph and BFS for Tree

Breadth-First Traversal (BFS) for a graph is similar to the Breadth-First Traversal of a tree.

The only catch here is, that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

- Visited and
- Not visited

A boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a queue data structure for traversal.

## 1.1.2 BFS Applications

Because of the algorithm's flexibility, Breadth-First Search is quite useful in the real world. These are some of them:

- In a peer-to-peer network, peer nodes are discovered. Most torrent clients, such as BitTorrent, uTorrent, and qBittorent, employ this process to find "seeds" and "peers" in the network.
- The index is built using graph traversal techniques in web crawling. The procedure starts with the source page as the root node and works its way down to all secondary pages that are linked to the source page (and this process continues). Because of the reduced depth of the recursion tree, Breadth-First Search has an inherent advantage here.
- The use of GPS navigation systems using the GPS, conduct a breadth-first search to locate nearby sites.
- Cheney's technique, which employs the concept of breadth-first search, is used to collect garbage.

## 1.1.3 Breadth First Search (BFS) for a Graph Algorithm:

Let’s discuss the algorithm for the BFS:

1. **Initialization**: Enqueue the starting node into a queue and mark it as visited.
2. **Exploration**: While the queue is not empty:
    - Dequeue a node from the queue and visit it (e.g., print its value).
    - For each unvisited neighbor of the dequeued node:
        - ➢ Enqueue the neighbor into the queue.
        - ➢ Mark the neighbor as visited.
3. **Termination**: Repeat step 2 until the queue is empty.

This algorithm ensures that all nodes in the graph are visited in a breadth-first manner, starting from the starting node.

## 1.1.4 How Does the BFS Algorithm Work?

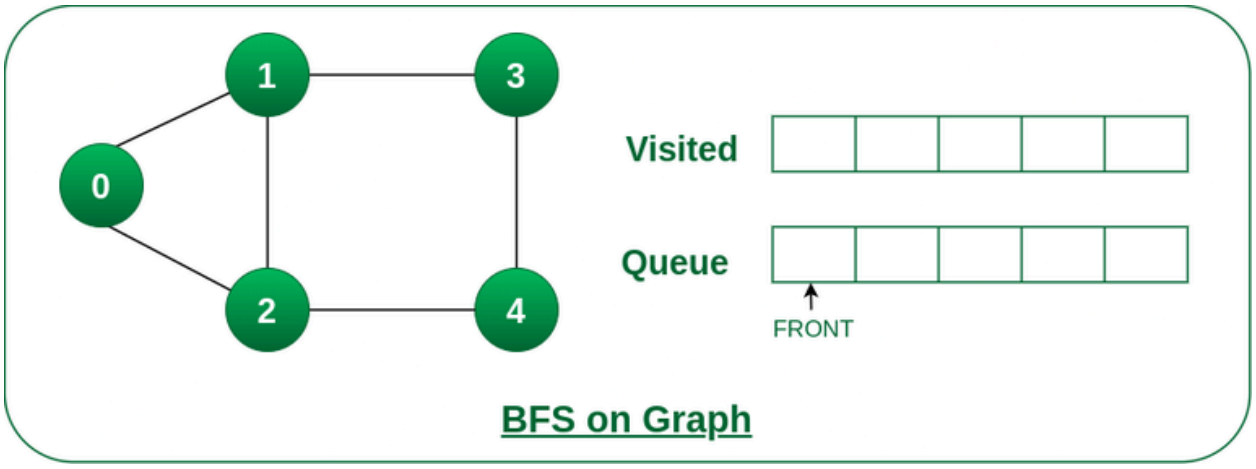
Starting from the root, all the nodes at a particular level are visited first and then the nodes of the next level are traversed till all the nodes are visited.

To do this a queue is used. All the adjacent unvisited nodes of the current level are pushed into the queue and the nodes of the current level are marked visited and popped from the queue.

### Illustration:

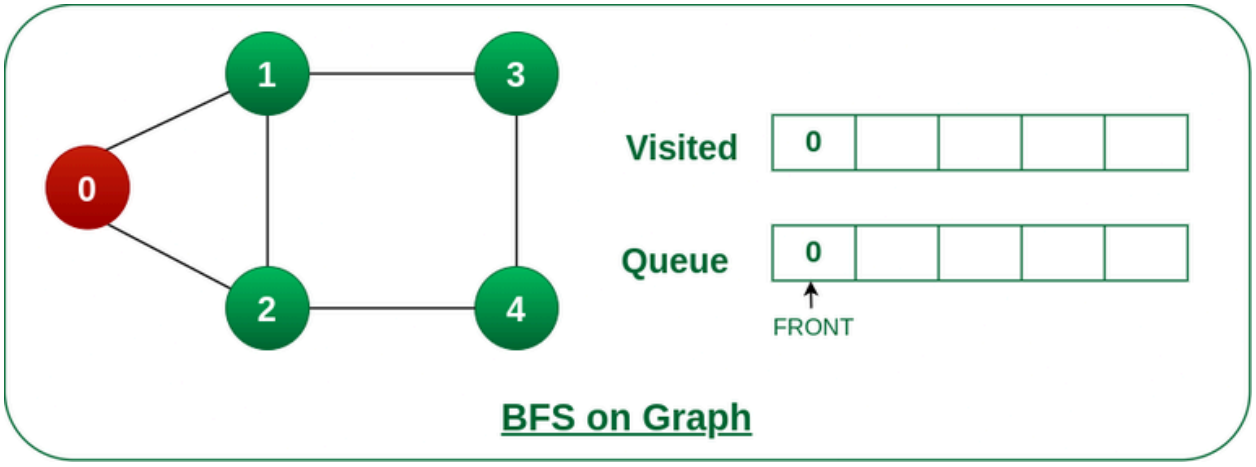
Let us understand the working of the algorithm with the help of the following example.

Step1: Initially the queue and visited arrays are empty.



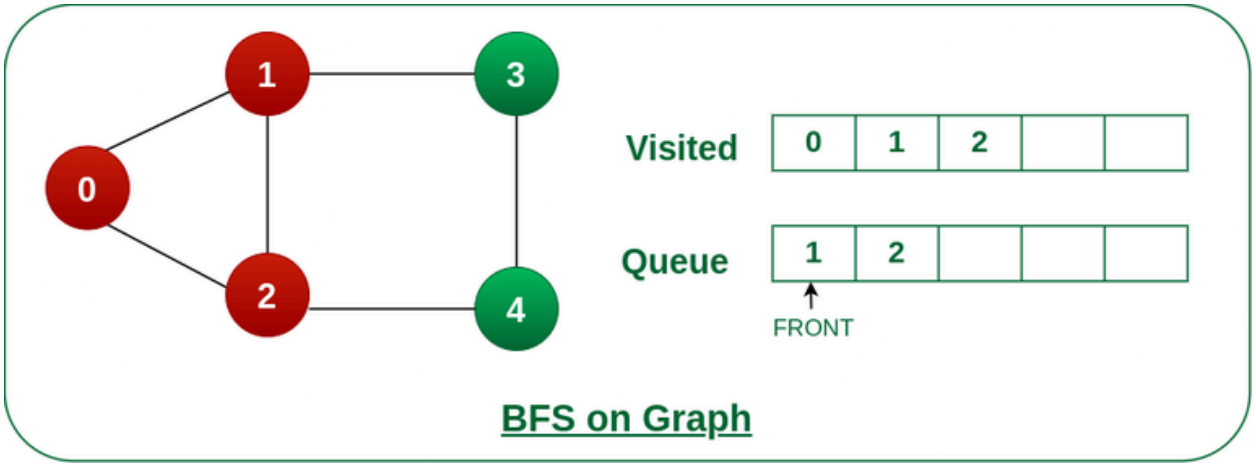

*Queue and visited arrays are empty initially.*

Step2: Push node 0 into the queue and mark it visited.



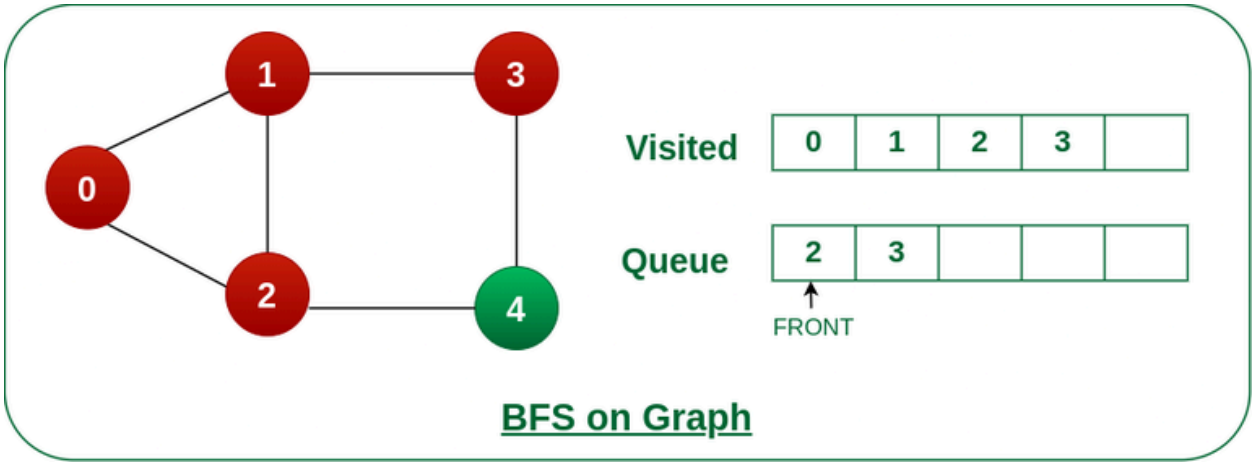

*Push node 0 into queue and mark it visited.*

Step 3: Remove node 0 from the front of the queue and visit the unvisited neighbours and push them into queue.



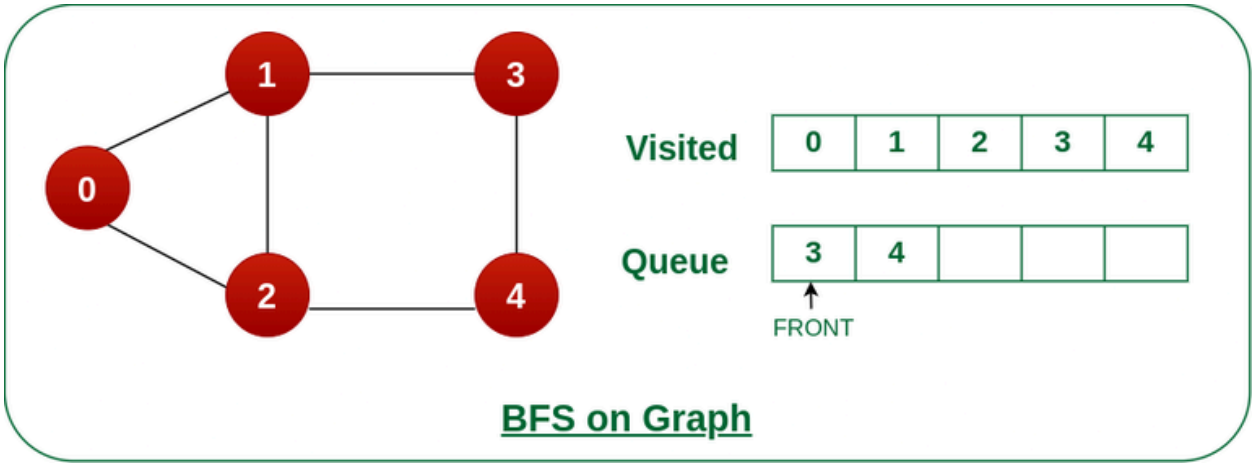

*Remove node 0 from the front of queue and visited the unvisited neighbours and push into queue.*

Step 4: Remove node 1 from the front of the queue and visit the unvisited neighbours and push them into the queue.



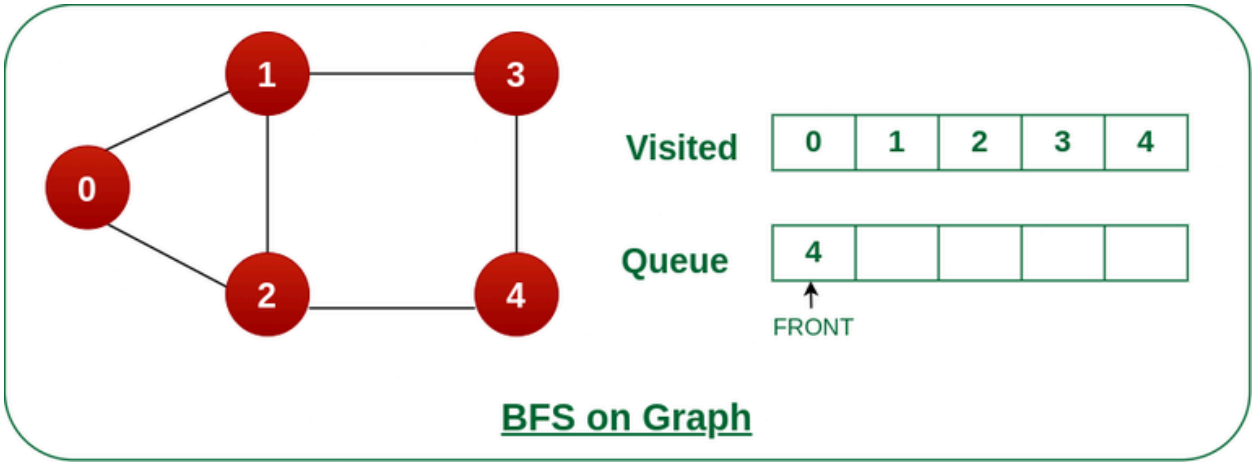

*Remove node 1 from the front of queue and visited the unvisited neighbours and push*

Step 5: Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.



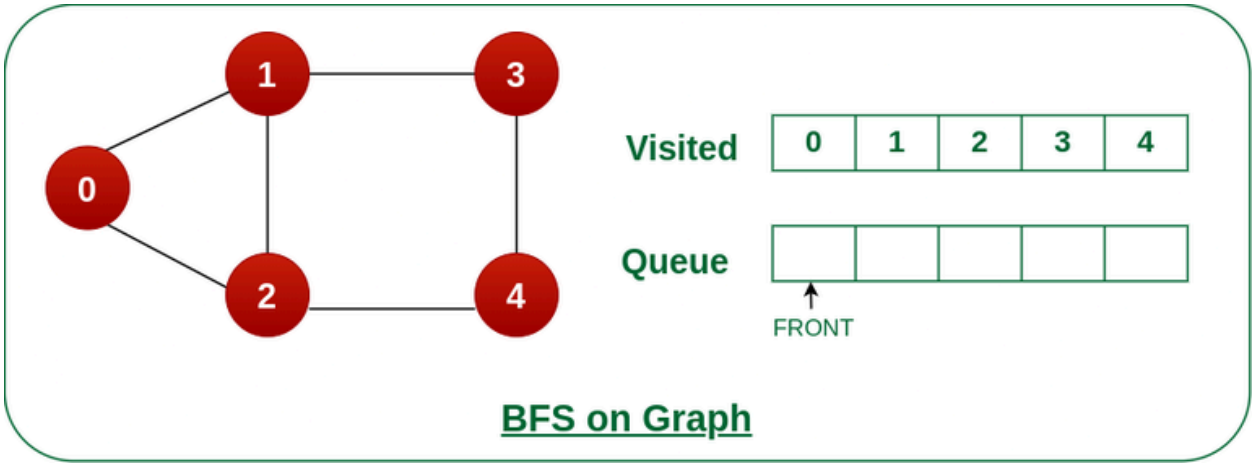

*Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.*

Step 6: Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.
As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.




*Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.*

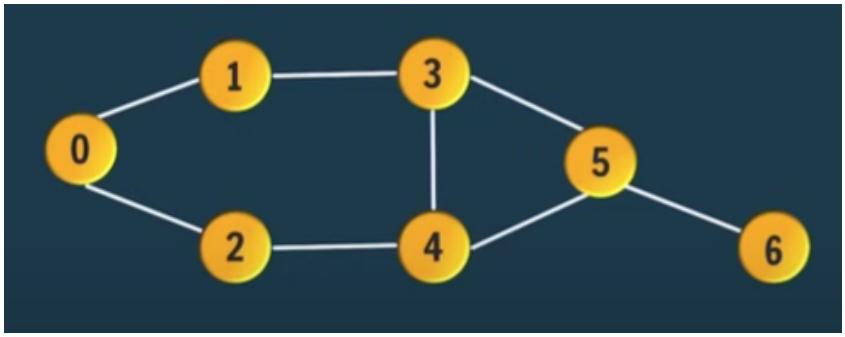
Steps 7: Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.
As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.




*Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.*

Now, Queue becomes empty, So, terminate this process of iteration.

Program: Write a program to perform BFS in the following undirected graph.




**Answer:**

```
package GraphPackage;

import java.util.*;

public class Graph7 {

	static class Edge{
		int src,dst;

		Edge(int s, int d){
			this.src=s;
			this.dst=d;
		}
	}

	public static void createGraph(ArrayList<Edge> graph[]) {
		for(int i=0;i<graph.length;i++) {
			graph[i]=new ArrayList<Edge>();
		}
		graph[0].add(new Edge(0,1));
		graph[0].add(new Edge(0,2));

		graph[1].add(new Edge(1,0));
		graph[1].add(new Edge(1,3));
```

```
        graph[2].add(new Edge(2,0));
        graph[2].add(new Edge(2,4));


        graph[3].add(new Edge(3,1));
        graph[3].add(new Edge(3,4));
        graph[3].add(new Edge(3,5));

        graph[4].add(new Edge(4,2));
        graph[4].add(new Edge(4,3));
        graph[4].add(new Edge(4,5));

        graph[5].add(new Edge(5,3));
        graph[5].add(new Edge(5,4));
        graph[5].add(new Edge(5,6));

        graph[6].add(new Edge(6,5));
}

public static void bfs(ArrayList<Edge> graph[],int V) {
        Queue<Integer> q = new LinkedList<>();
        boolean vis[] = new boolean[V];
        q.add(0);

        while(!q.isEmpty()) {
                int curr = q.remove();
                if(vis[curr]==false) {
                        System.out.print(curr+" ");
                        vis[curr]=true;

                        for(int i=0;i<graph[curr].size();i++) {
                                Edge e = graph[curr].get(i);

                                q.add(e.dst);
                        }
                }
        }
}

public static void main(String[] args) {
        int V=7;
        ArrayList<Edge> graph[] = new ArrayList[V];
        createGraph(graph);
        bfs(graph,V);
```

```
                System.out.println();
        }
}
```

Output:

```
0 1 2 3 4 5 6
```

**Time Complexity**: O(V+E), where V is the number of nodes and E is the number of edges.
**Auxiliary Space**: O(V)

# 1.2 Depth First Traversal/Search (DFS)

Depth First Traversal (or DFS) for a graph is similar to Depth First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a boolean visited array. A graph can have more than one DFS traversal.

## 1.1.1 DFS Applications

The applications of using the DFS algorithm are given as follows -

- DFS algorithm can be used to implement the topological sorting.
- It can be used to find the paths between two vertices.
- It can also be used to detect cycles in the graph.
- DFS algorithm is also used for one solution puzzles.
- DFS is used to determine if a graph is bipartite or not.

## 1.1.2 How Does the DFS Algorithm Work?

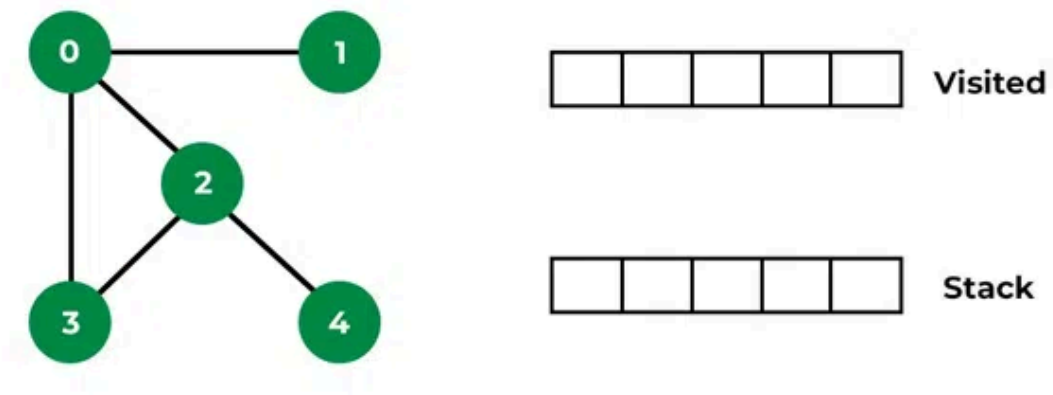
Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

Let us understand the working of Depth First Search with the help of the following illustration:

### Illustration:

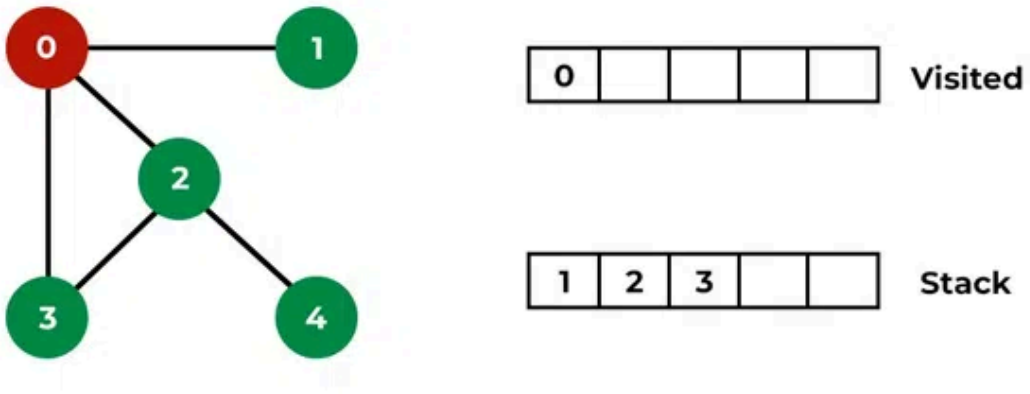
Let us understand the working of the algorithm with the help of the following example.

Step1: Initially stack and visited arrays are empty.

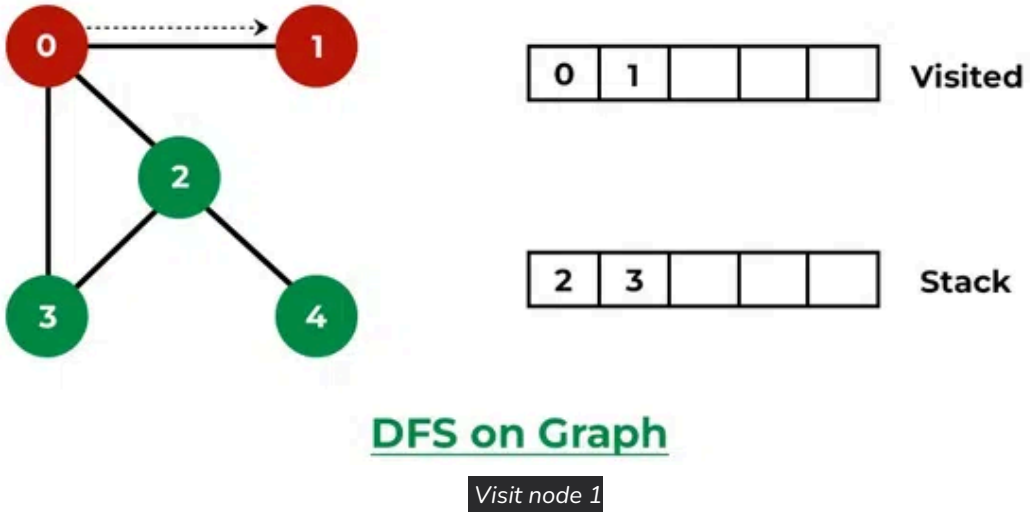



*Stack and visited arrays are empty initially.*

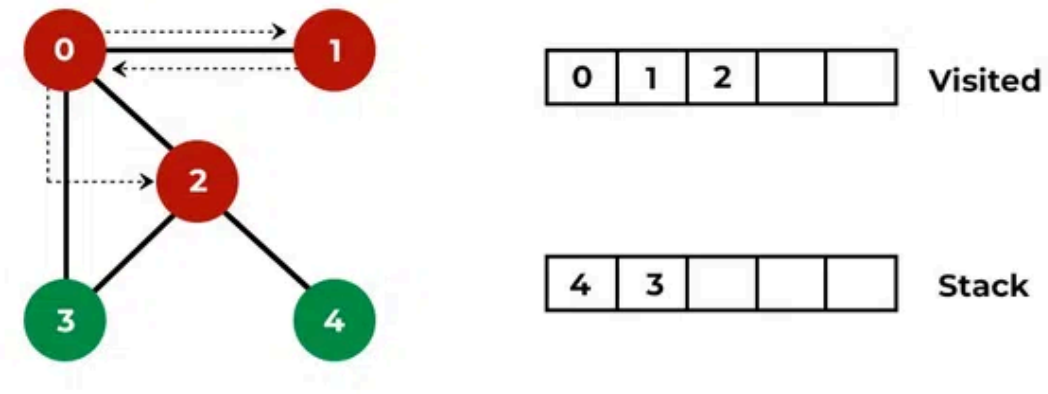
Step2: Visit 0 and put its adjacent nodes which are not visited yet into the stack.



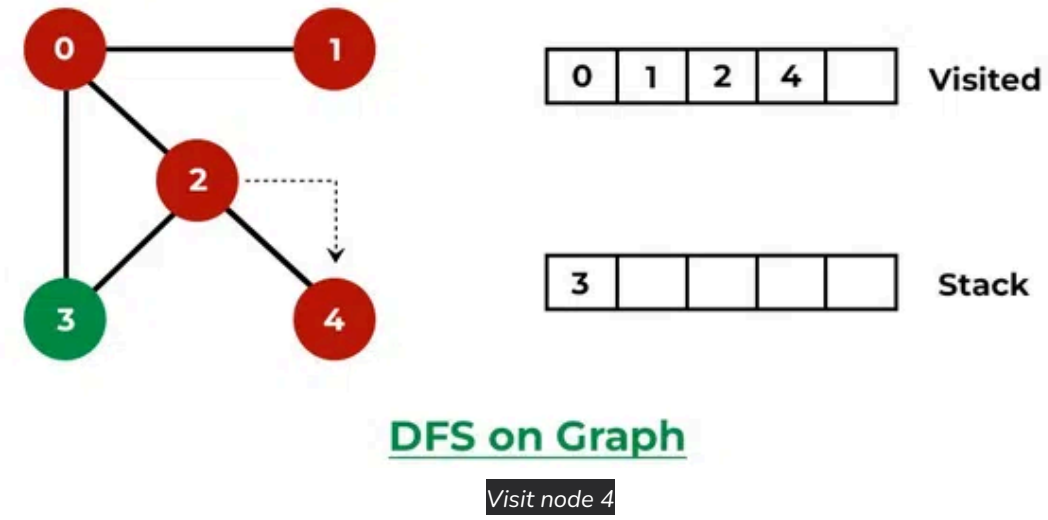

Step 3: Now, Node 1 at the top of the stack, so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



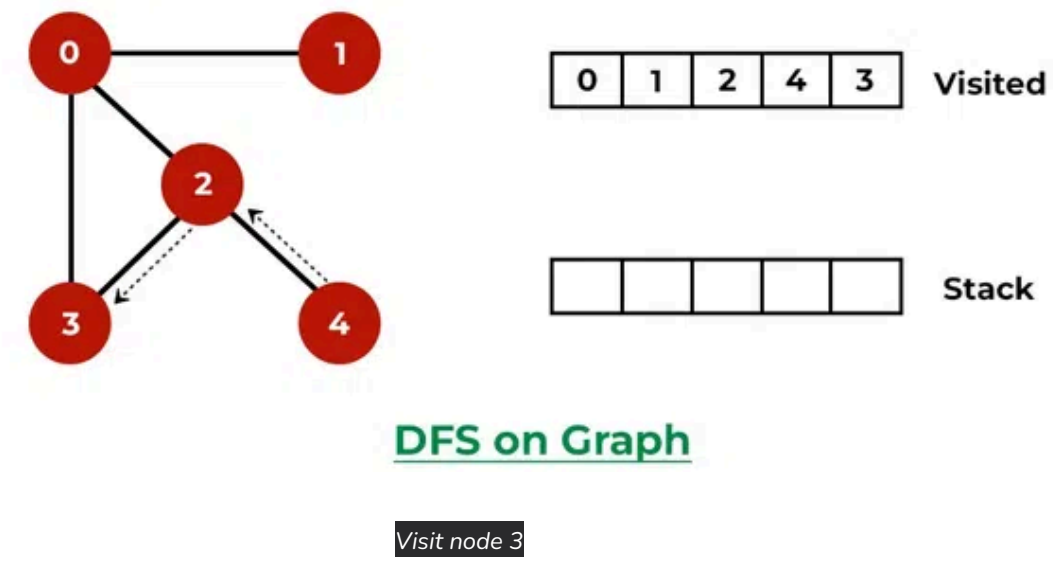
Step 4: Now, Node 2 at the top of the stack, so visit node 2 and pop it from the stack and put all of its adjacent nodes which are not visited (i.e, 3, 4) in the stack.




Step 5: Now, Node 4 at the top of the stack, so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



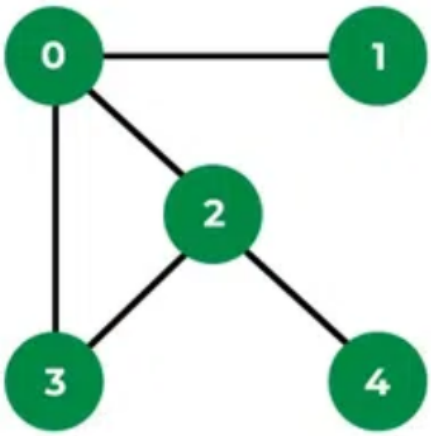
Step 6: Now, Node 3 at the top of the stack, so visit node 3 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.




Now, Stack becomes empty, which means we have visited all the nodes and our DFS traversal ends.

**Time complexity:** The time complexity of DFS is O(V + E), where V is the number of vertices and E is the number of edges.
**Auxiliary Space**: O(V)

Exercise: Write a program to perform DFS in the following undirected graph.




# 2. Shortest path

In graph theory, the **shortest path problem** is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.

The problem of finding the shortest path between two intersections on a road map may be modeled as a special case of the shortest path problem in graphs, where the vertices correspond to intersections and the edges correspond to road segments, each weighted by the length of the segment.

# What is Dijkstra's Algorithm?

Dijkstra's algorithm is used in finding the shortest path between any 2 given nodes of the graph. This algorithm only works for a weighted, undirected/directed graph and it is also called a single source shortest path problem. But it can also be used to solve multiple-source shortest path problems by simply running the algorithm for each source node. It uses a greedy Approach and always provides the optimal answer.

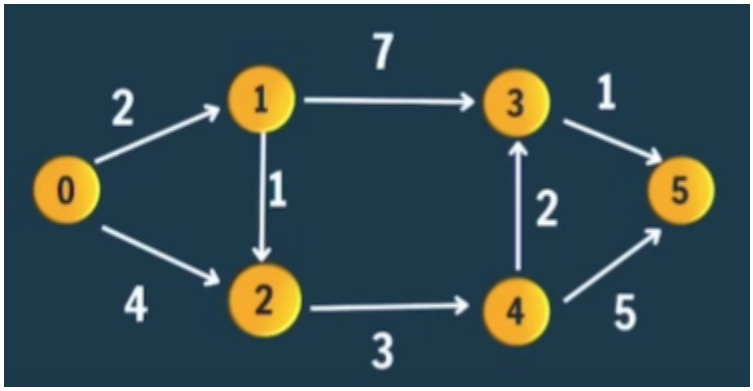
It starts working from the source node and calculates each distance to neighboring nodes (basically checking all the possible paths to the next neighboring node). It then

takes up the path to the next node which has the lowest cost out of all possible options. It repeats the same process until it reaches the destination node.

Here is the complete approach:

1. Create a visited boolean array of size = no.of vertices, where each index of the array points to each vertex.
2. Initially, all values in the array are false, depicting no vertex has been visited.
3. Then we make a priority queue PrQue, storing objects of type Pair. A pair consisting of 2 data members: vertex and weight to reach that vertex(w).
4. The priority queue arranges the objects in terms of the weight to reach that vertex(w).
5. Initially we insert a Pair(sourceVertex, 0) in the priority Queue.
6. Now till PrioQue is not empty  we follow the procedure -> remove, mark, print, add i.e
    a. Remove the topmost element from the priority queue and store it in a variable say topEle.
    b. Mark the topmost element as visited in the boolean array. If it is already visited then continue.
    c. Print the vertex value and the weight so far.
    d. Now add all the neighbors of topEle to the priority queue only and only if they have not been visited before.
7. Exit.

Now, write a program to find shortest paths of the following graph Dijkstra's algorithm using the Priority Queue.



## Answer:

```
import java.util.*;

public class DJ6V3 {

	public static class Edge{
		int src;
		int nbr;
		int wt;

		Edge(int src, int nbr, int wt){
			this.src=src;
			this.nbr=nbr;
			this.wt=wt;
		}
	}

	public static void createGraph(ArrayList<Edge> graph[]) {
		for(int i=0;i<graph.length;i++) {
			graph[i] = new ArrayList<Edge>();
		}

		graph[0].add(new Edge(0,1,2));
		graph[0].add(new Edge(0,2,4));

		graph[1].add(new Edge(1,2,1));
		graph[1].add(new Edge(1,3,7));

		graph[2].add(new Edge(2,4,3));

		graph[3].add(new Edge(3,5,1));

		graph[4].add(new Edge(4,3,2));
		graph[4].add(new Edge(4,5,5));
	}

	public static class Pair implements Comparable<Pair>{
		int vertex;
		int wsf;

		Pair(int vertex, int wsf){
			this.vertex=vertex;
			this.wsf=wsf;
```

```
		}

		public int compareTo(Pair o) {
			return this.wsf-o.wsf;
		}

	}

	public static void DJ(ArrayList<Edge> graph[], int s, int V) {
		boolean vis[] = new boolean[V];
		PriorityQueue<Pair> pq = new PriorityQueue<Pair>();
		pq.add(new Pair(s,0));

		while(!pq.isEmpty()) {
			Pair topEle = pq.remove();

			if(vis[topEle.vertex]==true) {
				continue;
			}
			vis[topEle.vertex]=true;

			System.out.println("Vertex: "+topEle.vertex+
					"\tWeight: "+topEle.wsf);

			for(Edge e: graph[topEle.vertex]) {
				if(vis[e.nbr]==false) {
					pq.add(new Pair(e.nbr,topEle.wsf+e.wt));
					//System.out.println(e.nbr+" "+(topEle.wsf+e.wt));
				}
			}
		}
	}

	public static void main(String[] args) {
		int V = 6;
		ArrayList<Edge> graph[] = new ArrayList[V];
		createGraph(graph);
		DJ(graph,0,V);
	}
}
```

Output:  source 0

```
Vertex: 0       Weight: 0
Vertex: 1       Weight: 2
Vertex: 2       Weight: 3
Vertex: 4       Weight: 6
Vertex: 3       Weight: 8
Vertex: 5       Weight: 9
```

## Exercise:

For this graph below, write a program to find the shortest path between 0 and 5.