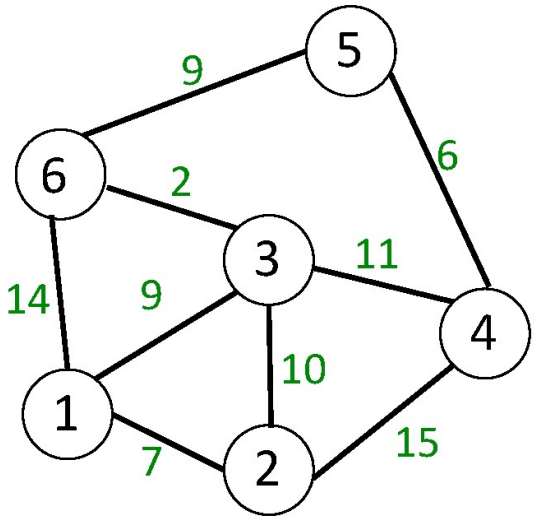
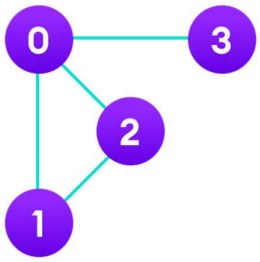
A **graph** is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices.



More precisely, a graph is a data structure (V, E) that consists of

- A collection of vertices V
- A collection of edges E, represented as ordered pairs of vertices (u,v)



Vertices and edges in the graph,

V = {0, 1, 2, 3}

E = {(0,1), (0,2), (0,3), (1,2)}

G = {V, E}

Mercy Technologies

Graph Terminology

- **Adjacency**: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them.
- **Path**: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.
- **Directed Graph**: A graph in which an edge (u,v) doesn't necessarily mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge

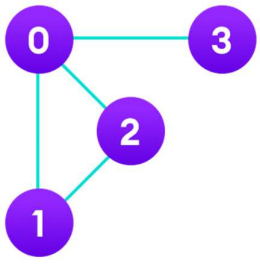
Graph Representation

Graphs are commonly represented in two ways:

1. Adjacency Matrix

An adjacency matrix is a 2D array of V x V vertices. Each row and column represent a vertex.

If the value of any element `a[i][j]` is 1, it represents that there is an edge connecting vertex i and vertex j.
The adjacency matrix for the graph we created above is



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

Since it is an undirected graph, for edge (0,2), we also need to mark edge (2,0); making the adjacency matrix symmetric about the diagonal.

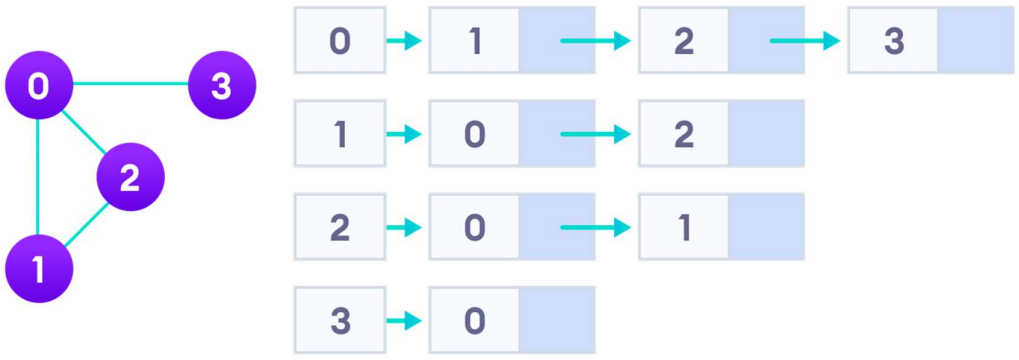
Edge lookup(checking if an edge exists between vertex A and vertex B) is extremely fast in adjacency matrix representation but we have to reserve space for every possible link between all vertices(V x V), so it requires more space.

2. Adjacency List

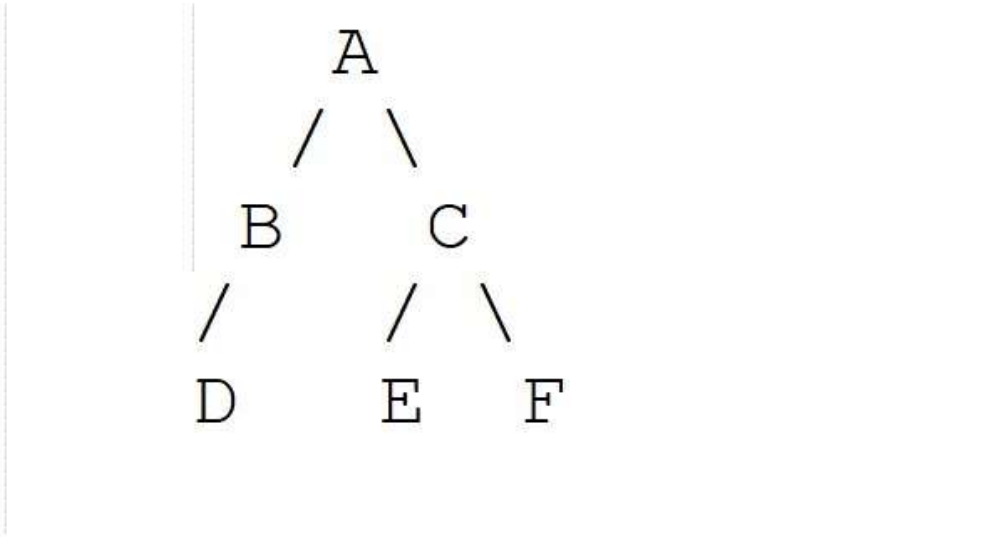
An adjacency list represents a graph as an array of linked lists.

The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

The adjacency list for the graph we made in the first example is as follows:



An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a graph with millions of vertices, this can mean a lot of saved space.



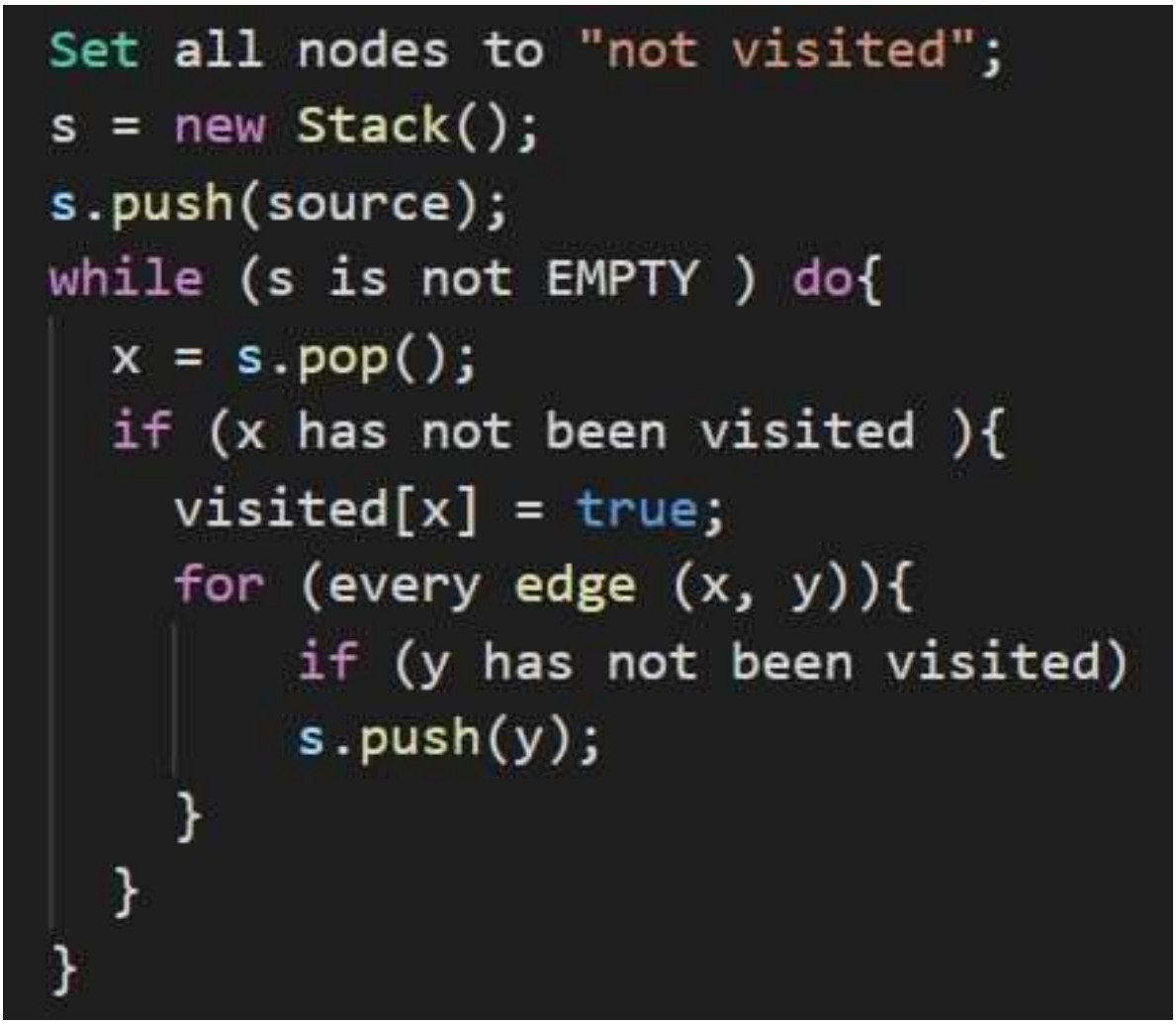
DFS=>A, B, D, C, E, F

BFS=>A, B, C, D, E, F

```
Set all nodes to "not visited";
q = new Queue();
q.enqueue(initial node);
while(q not empty) do{
  x = q.dequeue();
  if( x has not been visited ){
    visited[x] = true;
    for (every edge (x, y)){
        if (y has not been visited)
        q.enqueue(y);
    }
  }
}
```

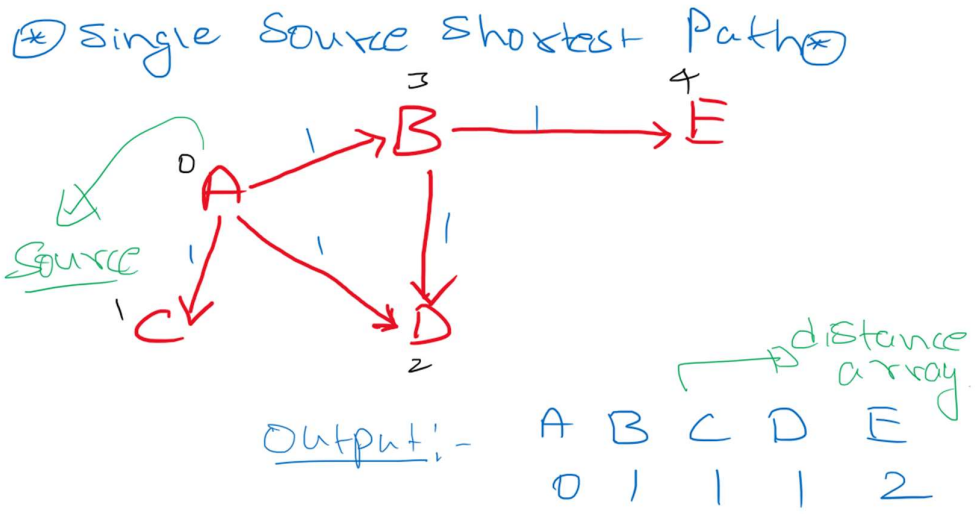
```java
public class MyGraph {
	private int numVertices;
	private LinkedList<Integer> adjLists[];
	private boolean visited[];

	MyGraph(int vertices) {
		numVertices = vertices;
		adjLists = new LinkedList[vertices];
		visited = new boolean[vertices];
		for (int i = 0; i < vertices; i++)
			adjLists[i] = new LinkedList<Integer>();
	}

	void addEdge(int source, int destination) {
		adjLists[source].add(destination);
	}

	public int getNumVertices() {
		return numVertices;
	}

	public void setNumVertices(int numVertices) {
		this.numVertices = numVertices;
	}

	public LinkedList<Integer>[] getAdjLists() {
		return adjLists;
	}

	public void setAdjLists(LinkedList<Integer>[] adjLists) {
		this.adjLists = adjLists;
	}

	public boolean[] getVisited() {
		return visited;
	}

	public void setVisited(boolean[] visited) {
		this.visited = visited;
	}

}

public class MyBFS {
	static void doBFS(MyGraph g, int source) {
		boolean[] visitedArray = g.getVisited();
		// Create a new queue for BFS
		LinkedList<Integer> queue = new LinkedList<Integer>();
		// Mark the current node as visited and enqueue it
		visitedArray[source] = true;
		queue.add(source);
		while (queue.size() > 0) {
			// pop a vertex from queue and print it
			source = queue.poll();
			System.out.print(source + " ");
			// Traverse all adj vertices,if not visited, mark visited and enqueue
			Iterator<Integer> it = g.getAdjLists()[source].listIterator();
			while (it.hasNext()) {
				int curr = it.next();
				if (!visitedArray[curr]) {
					visitedArray[curr] = true;
					queue.add(curr);
				}
			}
		}
	}
```

```
        public static void main(String args[]) {
                MyGraph g = new MyGraph(6);
                g.addEdge(0, 1);
                g.addEdge(0, 2);
                g.addEdge(1, 3);
                g.addEdge(2, 4);
                g.addEdge(2, 5);
                System.out.println("Following is Breadth First Traversal");
                doBFS(g, 0);
        }
}
```



```
public class MyDFS {
	static void doDFS(MyGraph g, int source) {
		boolean[] visitedArray = g.getVisited();
		visitedArray[source] = true;
		System.out.print(source + " ");
		Iterator itr = g.getAdjLists()[source].listIterator();
		while (itr.hasNext()) {
			int adj_node = (int) itr.next();
			if (!visitedArray[adj_node])
				doDFS(g, adj_node);
		}
	}

	public static void main(String args[]) {
		MyGraph g = new MyGraph(6);
		g.addEdge(0, 1);
		g.addEdge(0, 2);
		g.addEdge(1, 3);
		g.addEdge(2, 4);
		g.addEdge(2, 5);
		System.out.println("Following is Depth First Traversal");
		doDFS(g, 0);
	}
}
```

Single Source Shortest Path problem for unweighted graphs: -



```java
public class MyGraphWithDistance extends MyGraph {
	private int distance[];

	MyGraphWithDistance(int vertices) {
		super(vertices);
		distance = new int[vertices];
		for (int i = 0; i < distance.length; i++) {
			distance[i] = -1;
		}
	}

	public int[] getDistance() {
		return distance;
	}

	public void setDistance(int[] distance) {
		this.distance = distance;
	}

}
public class MySSSP {
	static void getSSSP(MyGraphWithDistance g, int source) {
		boolean[] visitedArray = g.getVisited();
		int[] distanceArray = g.getDistance();
		LinkedList<Integer> queue = new LinkedList<Integer>();
		visitedArray[source] = true;
		// Mark distance of source to be 0
		distanceArray[source] = 0;
		queue.add(source);
		while (queue.size() > 0) {
			// pop a vertex from queue and print it
			source = queue.poll();
			System.out.print(source + " ");
			// Traverse all adj vertices,if not visited, mark visited and enqueue
			Iterator<Integer> it = g.getAdjLists()[source].listIterator();
			while (it.hasNext()) {
				int curr = it.next();
				if (!visitedArray[curr]) {
					visitedArray[curr] = true;
					distanceArray[curr] = distanceArray[source] + 1;
					queue.add(curr);
				}
			}
		}
	}
	private static void printShortestPath(MyGraphWithDistance g) {
		System.out.println("\nFollowing is Shortest path using BFS");
		int[] distanceArray = g.getDistance();
		for (int i = 0; i < distanceArray.length; i++) {
			System.out.print(distanceArray[i] + " ");
		}
	}

	public static void main(String args[]) {
		MyGraphWithDistance g = new MyGraphWithDistance(5);
		g.addEdge(0, 1);
		g.addEdge(0, 2);
		g.addEdge(0, 3);
		g.addEdge(3, 2);
		g.addEdge(3, 4);
		System.out.println("Following is Breadth First Traversal");
		getSSSP(g, 0);
		printShortestPath(g);
	}

}
```

Mercy Technologies