"You don't have to be a genius to code, you just have to be persistent."

Hello Everyone
Very Special Good Evening
to All of you 😊
We will Start
from 9:00 pm

🙏 Graphs 1 🙏

~~~~~~~~~~

Agenda :

~~~~~~~~~~

Introduction { 1. Introduction to Graph
2. Types of Graphs

creation { 3. How to store data in Graph

traversal { 4. BFS (Breadth First Search)

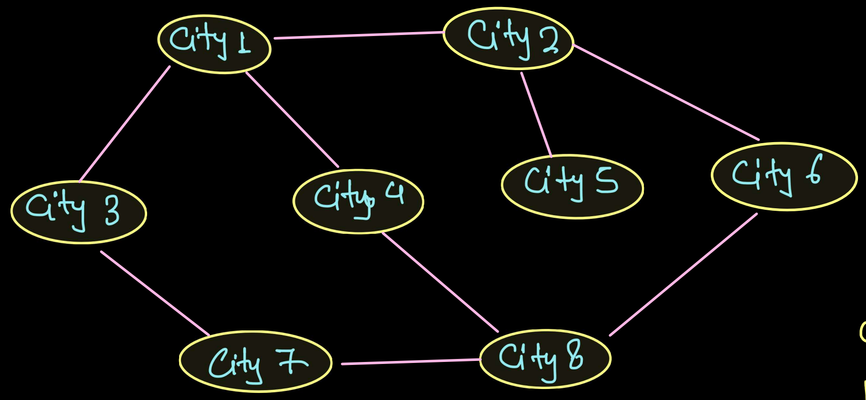
searching in Graph { 5. Is Path Available from Source to Destination

~~~~~~~~~~~~~~~~~~~~~~~~~~

Introduction to Graph

~~~~~~~~~~~~~~~~~~~~~~~~~~

Want to store information of cities and their connectivity.



With the help of graph we can store this kind of information.

cities ≡ vertex

links ≡ Edge

No. of Edges = 9

No. of Vertex = 8

Definition: Collection of vertex and edges is known as Graph.

Neighbour of 1 ⟶ city 2, city 3 and city 4

neighbour (nbr) of 8 ⟶ city 4, city 6, and city 7

Direct connected vertex are known as Neighbours.

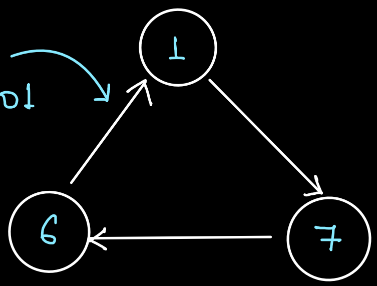
~~~~~~~~~~~~~~~~~~

Types of Graphs

~~~~~~~~~~~~~~~~~~

1. Based on type of edges:

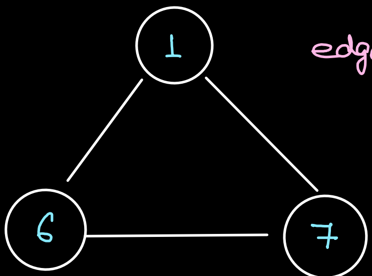
edge from 6 to 1



directed graph

[instagram, youtube scriber]

water supply from City 6 to City 1



edge available b/w 1 to 7

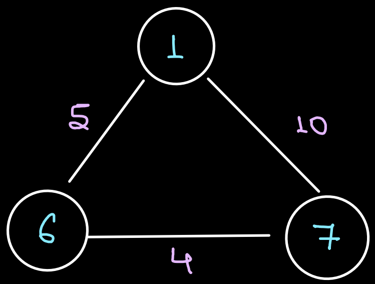
undirected graph

[LinkedIn, facebook]

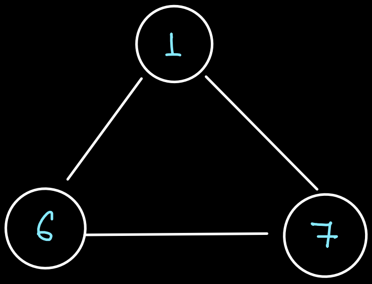
connection,

Road connection b/w city 1 to City 6

2. Based on Edge wt. present or not:

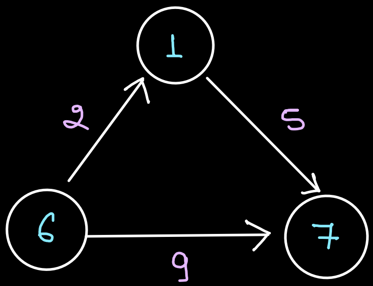


weighted graph



unweighted graph

3. Combination of above types are also possible:



directed weighted graph

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

How to store data in Graph

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

There is two famous implementation of graph is available :

1. Adjacency matrix
2. Adjacency List

1. Adjacency Matrix:

Vertex = 7 , Edges = 8

int[ ][ ] graph = new int[vtx][vtx];

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

Edge →

i →

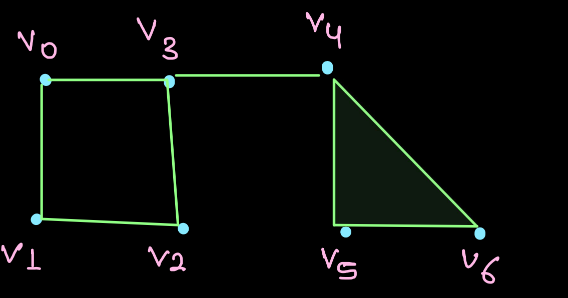
u	v	
0	3	✓
0	1	✓
2	3	✓
3	4	✓
1	2	✓
4	5	✓
4	6	✓
5	6	✓

undirected graph

```
int u = edge[i][0];
int v = edge[i][1];
// Edge b/w u & v
graph[u][v] = 1
graph[v][u] = 1
```

```
int u = edge[i][0] ;      u = 2
int v =  edge[i][1] ;     v = 3

graph[u][v] = 1 ;  →  graph[2][3] = 1
grap[v][u] = 1  ;  →  graph[3][2] = 1
```

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

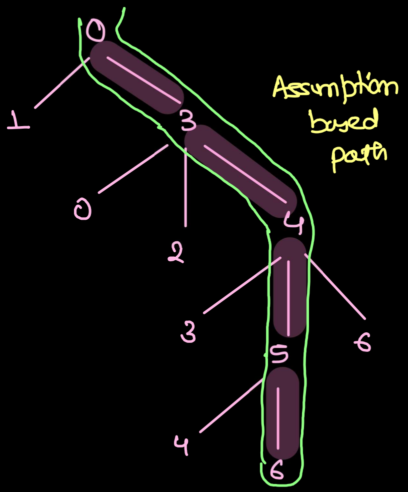


unweighted undirected graph

undirected graph.

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

source = 0 ; destination =6



directed weighted Graph:

	0	1	2	3	4	5	6
0	0	5	0	10	0	0	0
1	0	0	4	0	0	0	0
2	0	0	0	9	0	0	0
3	0	0	0	0	3	0	0
4	0	0	0	0	0	7	20
5	0	0	0	0	0	0	15
6	0	0	0	0	0	0	0

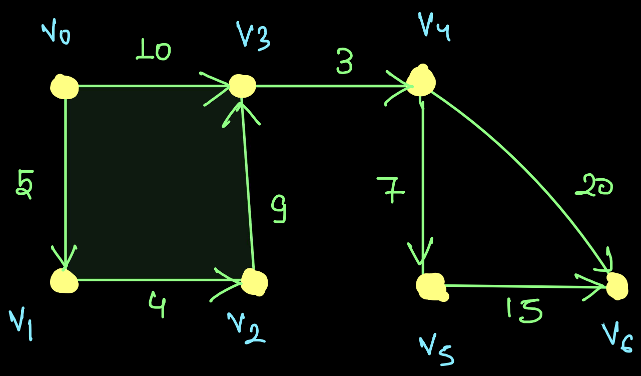
```
int u = edge[i][0];
int v = edge[i][1];
int wt = edge[i][2];
// Edge from u to v
   with weight wt'
graph[u][v] = wt;
```

Vertex = 7 , Edges = 8

Edge→

u	v	wt	
0	3	10	✓
0	1	5	✓
2	3	9	✓
3	4	3	✓
1	2	4	✓
4	5	7	✓
4	6	20	✓
5	6	15	✓

	0	1	2	3	4	5	6
0	0	5	0	10	0	0	0
1	0	0	4	0	0	0	0
2	0	0	0	9	0	0	0
3	0	0	0	0	3	0	0
4	0	0	0	0	0	7	20
5	0	0	0	0	0	0	15
6	0	0	0	0	0	0	0

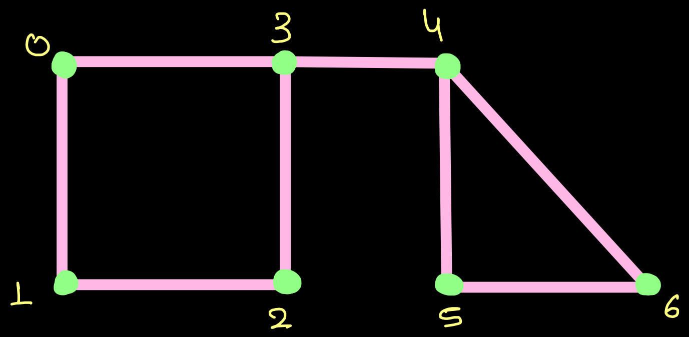


Major disadvontge of Adjacency matrix :

major disadvontge is wastage of Space. that 's why. most of the tim we will deal with adjacency list problem

2. Adjacency List:

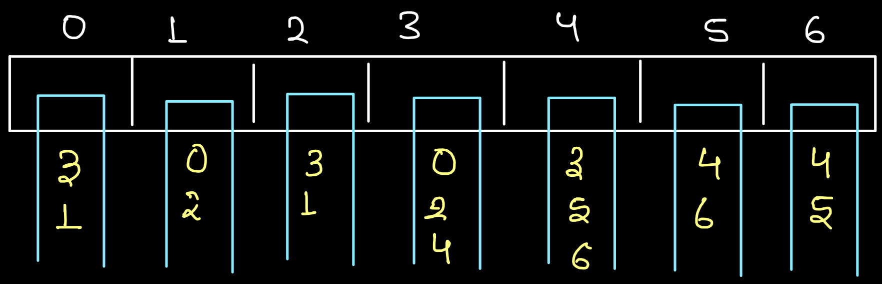
Undirected graph



Vertex: 7 , Edges = 8

Edge→

V_1	V_2	
0	3	✓
0	1	✓
2	3	✓
3	4	✓
1	2	✓
4	5	✓
4	6	✓
5	6	✓



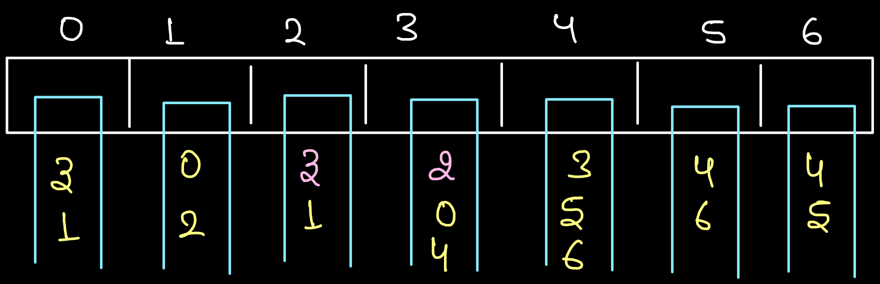
ArrayList< ArrayList<Integer>> graph

Vtx=7, Edge: 8

Edge→

i →

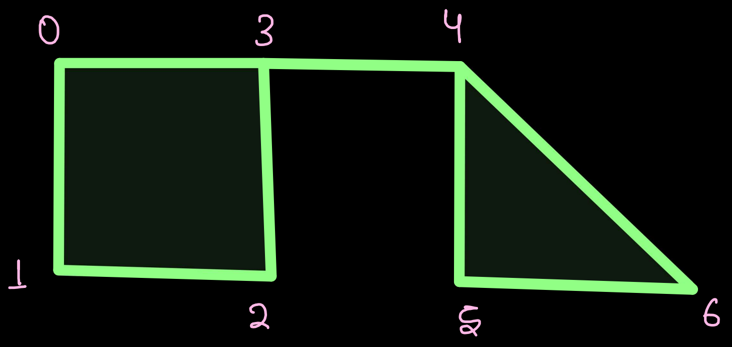
V_1	V_2	
0	3	✓
0	1	✓
2	3	✓
3	4	✓
1	2	✓
4	5	✓
4	6	✓
5	6	✓



```
int u= edge[i][0]; →2
int v= edge[i][1]; →3
graph.get(2).add(3);
graph.get(3).add(2);
```

```
int u= edge[i][0];
int v= edge[i][1];
// create edge b/w u & v
graph.get(u).add(v);
graph.get(v).add(u);
```

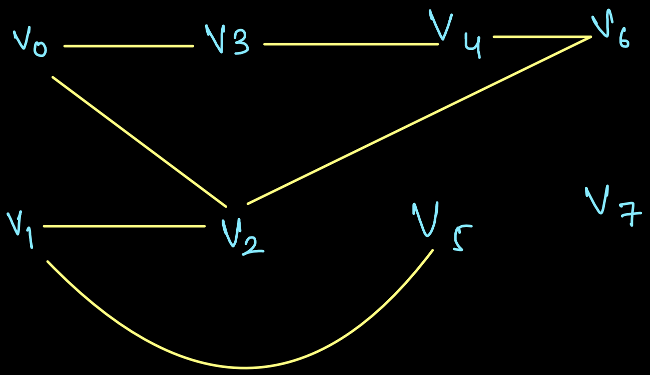
0	1	2	3	4	5	6
3	0	3	2	3	4	4
1	2	1	0	5	6	5
			4	6		

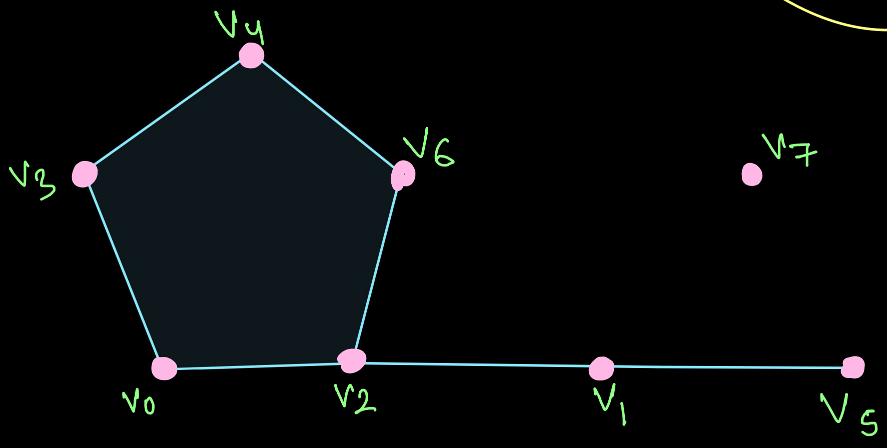


no. of vertex 8

Edge are

u→	0	2	1	3	6	2	0
v→	3	1	5	4	4	6	2





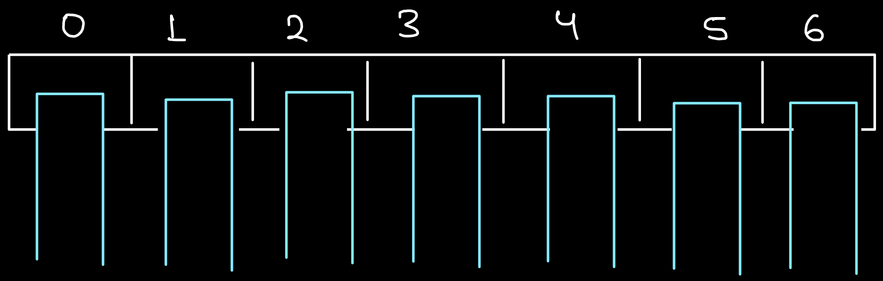
```
ArrayList<ArrayList<Integer>> graph = new AL<>();
```

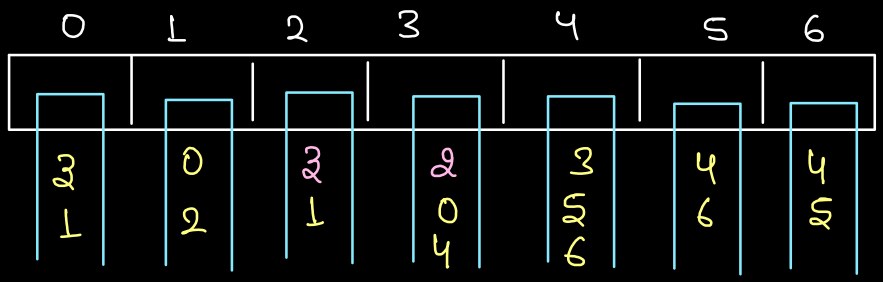
Vtx=7, Edge: 8

Edge→

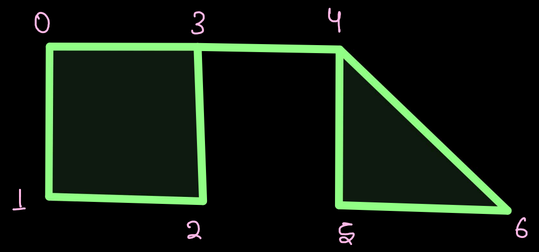
V_1	V_2
0	3
0	1
2	3
3	4
1	2
4	5
4	6
5	6

```
for(int v=0 ; v< vtx; v++){
        graph.add( new AL<>());
}
```





0 ⟶ [3, 1]
1 ⟶ [0, 2]
2 ⟶ [3, 1]
3 ⟶ [2, 0, 4]
4 ⟶ [3, 5, 6]
5 → [4, 6]
6 → [4, 5]



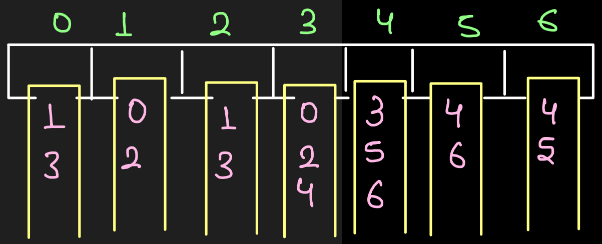
```java
import java.util.*;

class Main {

    public static void addEdge(ArrayList<ArrayList<Integer>> graph,
                int u, int v) {
        graph.get(u).add(v);
        graph.get(v).add(u);
    }

    public static void display(ArrayList<ArrayList<Integer>> graph) {
        for(int v = 0; v < graph.size(); v++) {
            System.out.print(v + " -> [ ");
            // graph.get(v) -> neighbours of vth vertex
            for(int nbr : graph.get(v)) {
                System.out.print(nbr + ", ");
            }
            System.out.println("]");
        }
    }

    public static void demo() {
        int vtx = 7;
        int edges = 8;

        ArrayList<ArrayList<Integer>> graph = new ArrayList<>();
        for(int v = 0; v < vtx; v++) {
            graph.add(new ArrayList<>());
        }

        // addEdge(graph, u, v);
        addEdge(graph, 0, 1);
        addEdge(graph, 0, 3);
        addEdge(graph, 1, 2);
        addEdge(graph, 2, 3);
        addEdge(graph, 3, 4);
        addEdge(graph, 4, 5);
        addEdge(graph, 4, 6);
        addEdge(graph, 5, 6);

        display(graph);
    }

    public static void main(String args[]) {
        demo();
    }
}
```

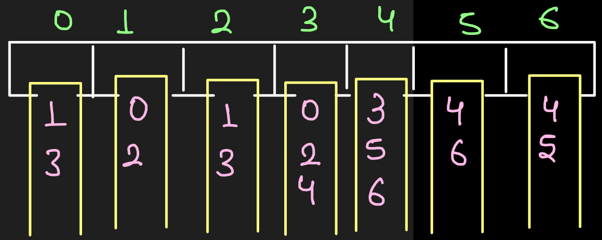
already discussed

→ Empty container created with name graph

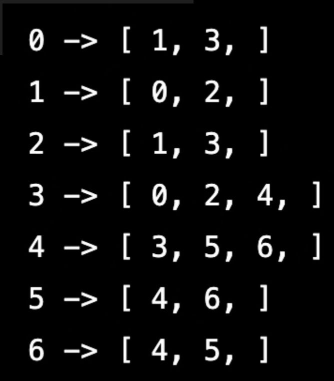
new Empty are added in graph which are representing nbr of vth vertex

0 → [1, 3]
1 → [0, 2,]
2 → [1, 3,]
3 → [0, 2, 4,]
4 → [3, 5, 6,]
5 → [4, 6,]
6 → [4, 5,]





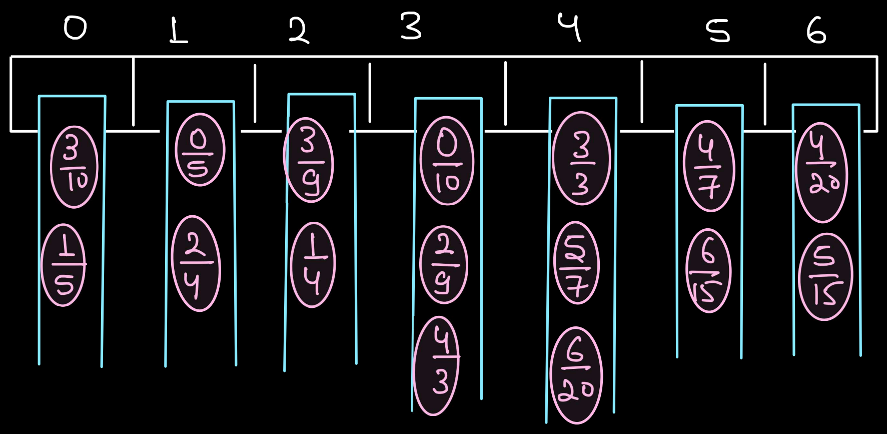
0 → [1, 3]
1 → [0, 2,]
2 → [1, 3,]
3 → [0, 2, 4,]
4 → [3, 5, 6,]
5 → [4, 6,]
6 → [4, 5,]



```
0 -> [ 1, 3, ]
1 -> [ 0, 2, ]
2 -> [ 1, 3, ]
3 -> [ 0, 2, 4, ]
4 -> [ 3, 5, 6, ]
5 -> [ 4, 6, ]
6 -> [ 4, 5, ]
```

How to make a weighted graph:

AL <AL <Pair> > graph = new AL<>();



Vtx= 7, Edge= 8

Edge→

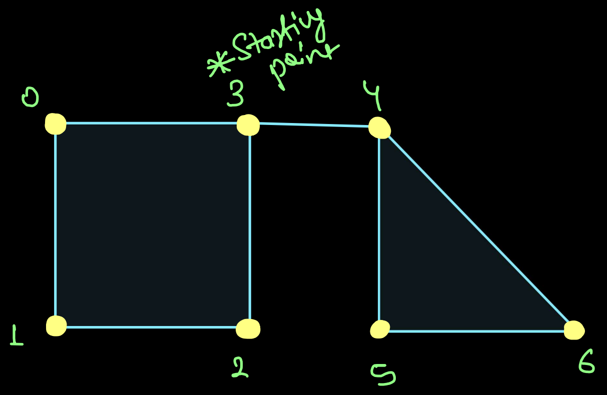
	V1	V2	wt
✓	0	3	→10
✓	0	1	→5
✓	2	3	→9
✓	3	4	→3
✓	1	2	→4
✓	4	5	→7
✓	4	6	→20
✓	5	6	→15

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

BFS (Breadth First Search) [Traversal on graph]

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

[exactly same as $\underbrace{\text{level order}}_{\text{data structure} \rightarrow \text{queue}}$ traversal of tree]



undirected graph

Steps of BFS

→ Remove
→ print
→ add unvisited nbr.
  → mark that nbr once it is visited

queue

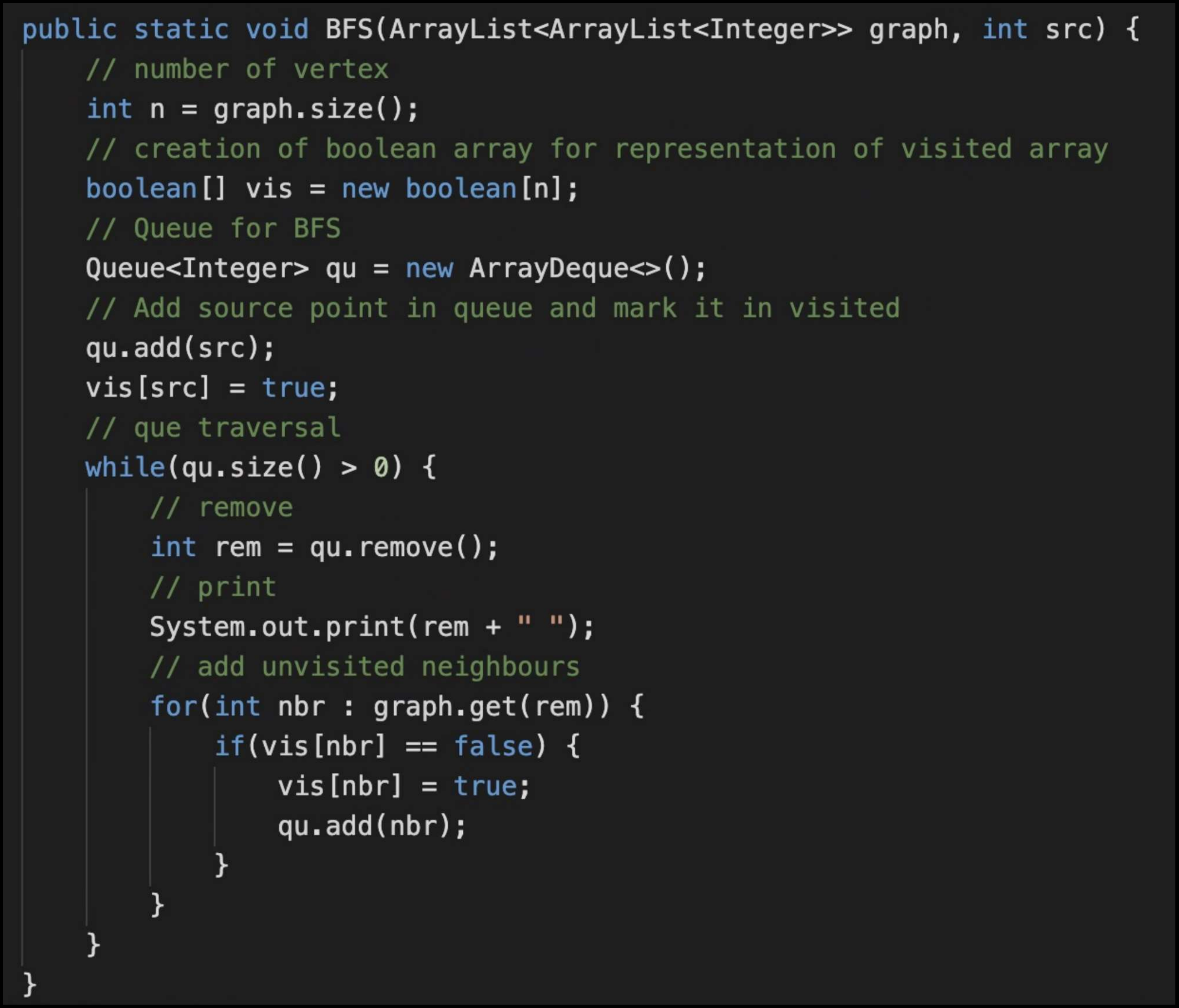
~~3~~, ~~0~~, ~~2~~, ~~4~~, ~~1~~, ~~5~~, ~~6~~

~~F~~ T	~~F~~ T	~~F~~ T	T	~~F~~ T	~~F~~ T	~~F~~ T
0	1	2	3	4	5	6

visited array

$\underbrace{3}, \underbrace{0, 2 \; 4}, \underbrace{1, 5, 6}$

NOTE: Either starting point is given otherwise you can start from any point.



```java
public static void BFS(ArrayList<ArrayList<Integer>> graph, int src) {
    // number of vertex
    int n = graph.size();
    // creation of boolean array for representation of visited array
    boolean[] vis = new boolean[n];
    // Queue for BFS
    Queue<Integer> qu = new ArrayDeque<>();
    // Add source point in queue and mark it in visited
    qu.add(src);
    vis[src] = true;
    // que traversal
    while(qu.size() > 0) {
        // remove
        int rem = qu.remove();
        // print
        System.out.print(rem + " ");
        // add unvisited neighbours
        for(int nbr : graph.get(rem)) {
            if(vis[nbr] == false) {
                vis[nbr] = true;
                qu.add(nbr);
            }
        }
    }
}
```

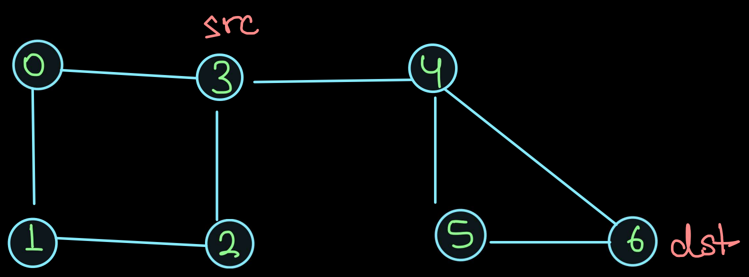
~~~~~~

Is Path Available from Source to Destination

~~~~~~

Given an undirected graph, source node and destination node.
Cheek if there is a path Available from source to destination or not.

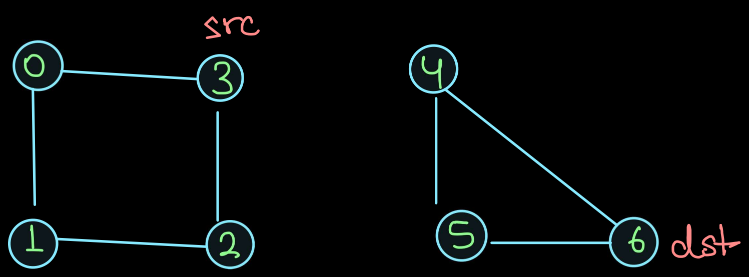
Ex 1:



Src = 3, dst = 6

path = true

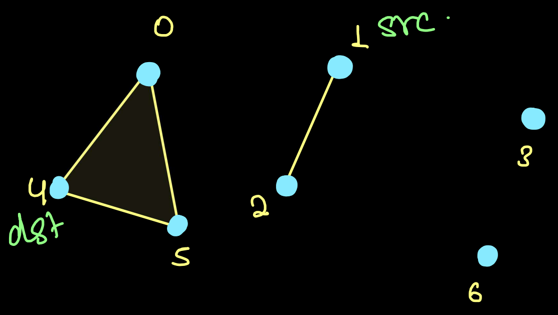
Ex 2:



Src = 3, dst : 6

path = false

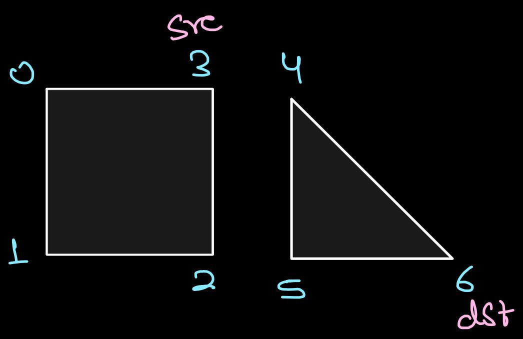
Ex 3:



Src = 1, dst = 4

path = false

Solution: Start BFS algorithm from source point, After completion of BFS Algo. check the status of vis[dst].



T~~F~~	~~F~~T	T~~F~~	T	F	F	F
0	1	2	3	4	5	6

~~3~~, ~~0~~, ~~2~~, ~~1~~