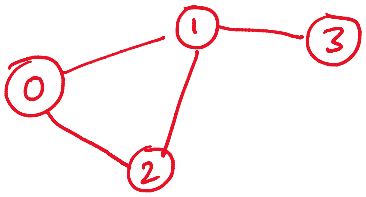
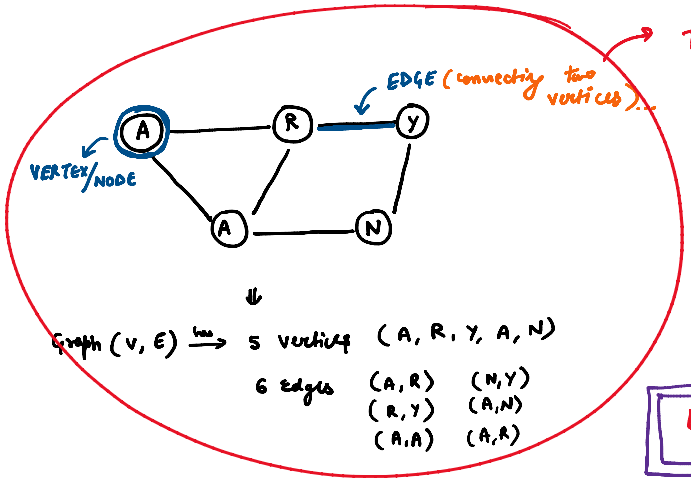
G-03 Graph Representation || Adjacency Matrix || Adjacency List || C++/Java

31 March 2023 07:01 PM



← we have a Graph

How to store this graph ?

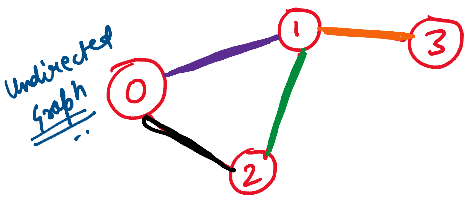


To represent a graph, we just need ↓ Vertices & Edges ....

↓

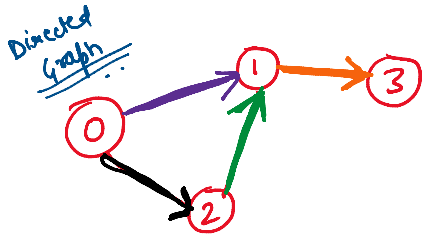
4 WAYS

* Method 1 : Create a 2-D Array (Adjacency Matrix)



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

→ represent an edge from Node 0 & 2



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

→ represent an edge from Node 0 to 2

Advantages

To know if we have an edge b/w node (u) & (v)

We can check in O(1)

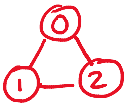
matrix[u][v] == 1 ⇒ Edge present ☺

!= 1 ⇒ Edge not present

Dis-advantages

⊙ To store all edges → O(n²) memory (Even when n nodes are there)

⊙ To find neighbours of a graph. → we have to iterate over entire row i → O(n) for 1 node even if neighbour was just 2 or 3.




```cpp
#include<bits/stdc++.h>
using namespace std;

int main()
{
    int n, m;
    cin >> n >> m;

    // Graph Nodes are from 0 to n-1
    int gr[n][n];
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        gr[u][v] = 1;
        gr[v][u] = 1;
    }

}
```

No. of Nodes

No. of edges → 0-1, 1-2, 0-2

n = 3, m = 3

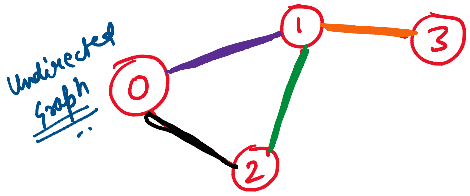
Make a 2D array

Just mark the cell u-v & v-u to 1

```java
import java.io.*;

public class AryanClass {
    Run | Debug
    public static void main (String[] args) {
        int n = 3, m = 3;
        int gr[][] = new int[n][n];

        // edge 0---1
        gr[0][1] = 1;
        gr[1][0] = 1;


        // edge 1---2
        gr[1][2] = 1;
        gr[2][1] = 1;


        // adge 0--2
        gr[0][2] = 1;
        gr[2][0] = 1;

    }
}
```

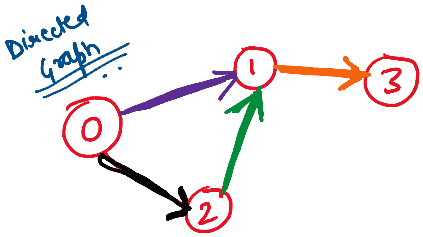
Method 2: Represent via list of Edges & Vertices → (Edge list Representation)

Method 2: Represent via list of Edges & Vertices → (Edge List Representation)



Vertices = { 0, 1, 2, 3 }

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

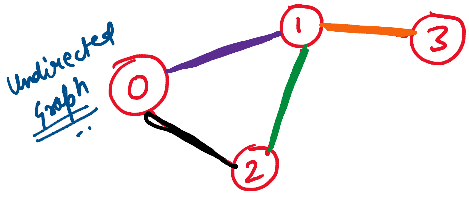


Vertices = { 0, 1, 2, 3 }

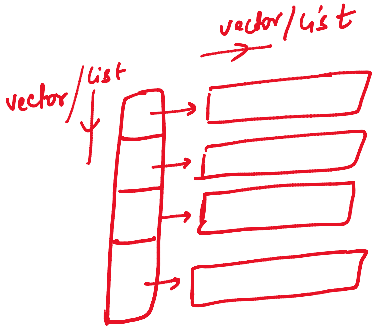
Edges = { (0,1) (1,3) (2,1) (0,2) }

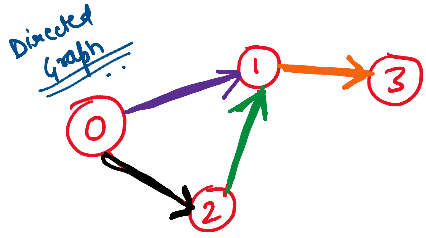
**

Method 4: Adjacency list (MOSTLY USED)



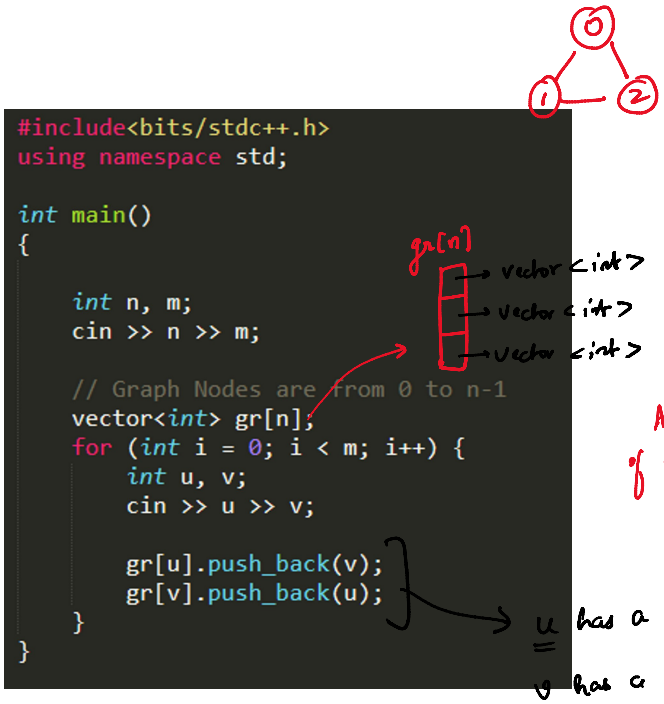
0 → { 1 , 2 }
1 → { 0 , 2 , 3 }
2 → { 0 , 1 }
3 → { 1 }

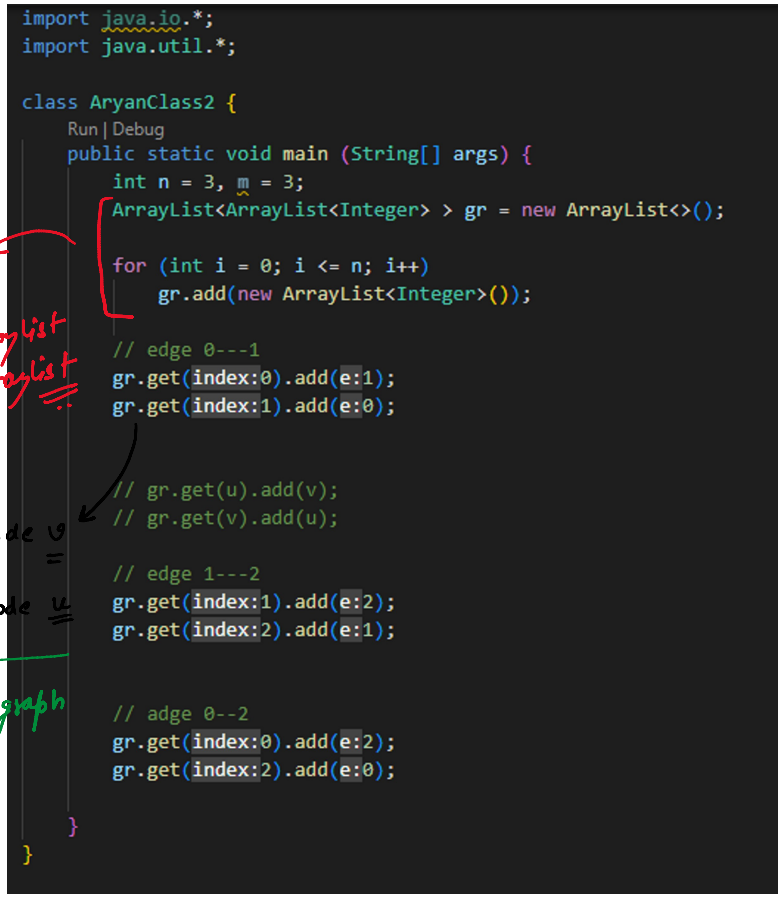




0 → { 1 , 2 }
1 → { 3 }
2 → { 1 }
3 → { }

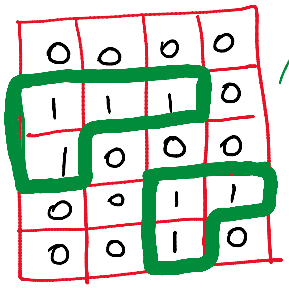

Advantages
① Find neighbours in O(neighbours) time
① Every edge twice
0 → { 1, 2 }
1 → { 0, 2, 3 }
② Memory efficient
Edges
Node
↳ O(2 * E + N) - Undirected Graph
↳ O(E + N) - Directed Graph
① When no edges at all
(still have to go through all the nodes)


0
1
2
#include<bits/stdc++.h>
using namespace std;
int main()
{
int n, m;
cin >> n >> m;
// Graph Nodes are from 0 to n-1
vector<int> gr[n];
for (int i = 0; i < m; i++) {
int u, v;
cin >> u >> v;
gr[u].push_back(v);
gr[v].push_back(u);
}
}
gr[n]
vector<int>
vector<int>
vector<int>
u has a node v
v has a node u
Undirected graph


import java.io.*;
import java.util.*;
class AryanClass2 {
Run | Debug
public static void main (String[] args) {
int n = 3, m = 3;
ArrayList<ArrayList<Integer> > gr = new ArrayList<>();
for (int i = 0; i <= n; i++)
gr.add(new ArrayList<Integer>());
// edge 0---1
gr.get(index:0).add(e:1);
gr.get(index:1).add(e:0);
// gr.get(u).add(v);
// gr.get(v).add(u);
// edge 1---2
gr.get(index:1).add(e:2);
gr.get(index:2).add(e:1);
// adge 0--2
gr.get(index:0).add(e:2);
gr.get(index:2).add(e:0);
}
}
Arraylist of Arraylist

Method - 4 : Implicit Graph

Given a 2-D array
Matrix
↓
Find Connected Component/
Groups in the
Graph.



You don't need to convert this to graph

It is already a graph where cell has neighbours