Graph Representations - Adjacency Matrix and List

There are two ways in which we represent graphs, these are:

- Adjacency Matrix
- Adjacency List

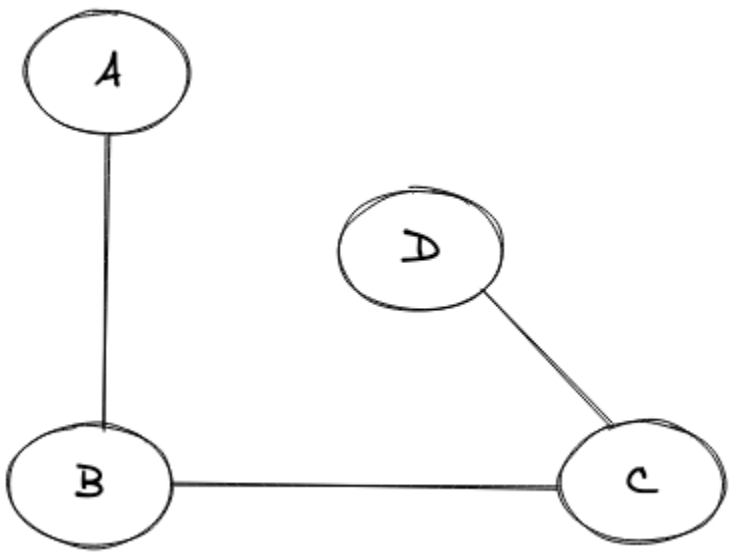
Both these have their advantages and disadvantages. In this tutorial, we will cover both of these graph representation along with how to implement them.

Adjacency Matrix

Adjacency matrix representation makes use of a matrix (table) where the **first row and first column of the matrix denote the nodes** (vertices) of the graph. The **rest of the cells contains either 0 or 1** (can contain an associated **weight w** if it is a weighted graph).

Each **row X column** intersection points to a cell and the value of that cell will help us in determining that whether the vertex denoted by the row and the vertex denoted by the column are connected or not. If the value of the cell for v1 X v2 is equal to 1, then we can conclude that these two vertices v1 and v2 are connected by an edge, else they aren't connected at all.

Consider the given graph below:

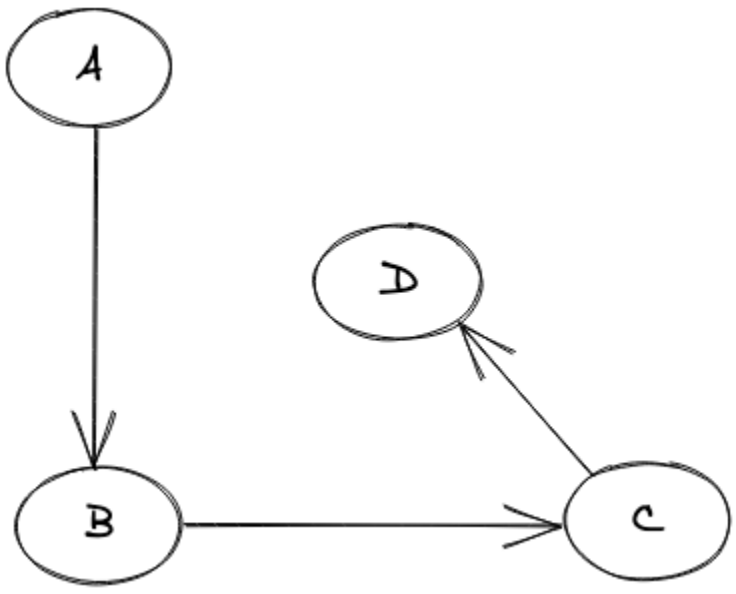


The graph shown above is an undirected one and the adjacency matrix for the same looks as:

-	A	B	C	D
A	0	1	0	0
B	1	0	1	0
C	0	1	0	1
D	0	0	1	0

The above matrix is the adjacency matrix representation of the graph shown above. If we look closely, we can see that the matrix is symmetric. Now let's see how the adjacency matrix changes for a directed graph.

Consider the given graph below:



For the directed graph shown above the adjacency matrix will look something like this:

-	A	B	C	D
A	0	1	0	0
B	0	0	1	0
C	0	0	0	1
D	0	0	0	0

Implementation of Adjacency Matrix

The structure (constructor in Java) for the adjacency matrix will look something like this:

```
public AdjacencyMatrix(int vertex){

        this.vertex = vertex;

        matrix = new int[vertex][vertex];

}
```

Copy

It should also be noted that we have two class-level variables, like:

```
int vertex;

int[][] matrix;
```

Copy

We have a constructor above named AdjacencyMatrix which takes the count of the number of the vertices that are present in the graph and then assigns our global vertex variable that value and also creates a 2D matrix of the same size. Now since our structure part is complete, we are simply left with adding the edges together, and the way we do that is:

```
public void addEdge(int start,int destination){

        matrix[start][destination] = 1;

        matrix[destination][start] = 1; // for un-directed graph

}
```

Copy

In the above `addEdge` function we also assigned 1 for the direction from the destination to the start node, as in this code we looked at the example of the undirected graph, in which the relationship is a two-way process. If it had been a directed graph, then we can simply make this value equal to 0, and we would have a valid adjacency matrix.

Now the only thing left is to print the graph.

```
public void printGraph(){

        System.out.println("Adjacency Matrix : ");

        for (int i = 0; i < vertex; i++) {

            for (int j = 0; j <vertex ; j++) {

                System.out.print(matrix[i][j]+ " ");

            }

            System.out.println();

        }

}
```

Copy

The entire code looks something like this:

```
public class AdjacencyMatrix {

    int vertex;

    int[][] matrix;


    // constructor

    public AdjacencyMatrix(int vertex){

        this.vertex = vertex;

        matrix = new int[vertex][vertex];

    }


    public void addEdge(int start,int destination){
```

```
        matrix[start][destination] = 1;

        matrix[destination][start] = 1;

    }


    public void printGraph(){

        System.out.println("Adjacency Matrix : ");

        for (int i = 0; i < vertex; i++) {

            for (int j = 0; j <vertex ; j++) {

                System.out.print(matrix[i][j]+ " ");

            }

            System.out.println();

        }

    }


    public static void main(String[] args) {

        AdjacencyMatrix adj = new AdjacencyMatrix(4);

        adj.addEdge(0,1); // 0 as the array is 0-indexed

        adj.addEdge(1,2);

        adj.addEdge(2,3);

        adj.printGraph();

    }

}
```

Copy

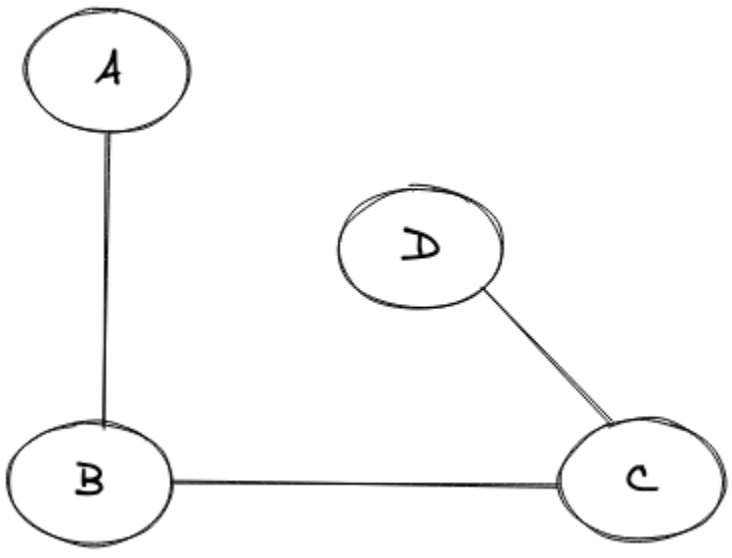
The output of the above looks like:

```
Adjacency Matrix :

0 1 0 0

1 0 1 0

0 1 0 1

0 0 1 0
```

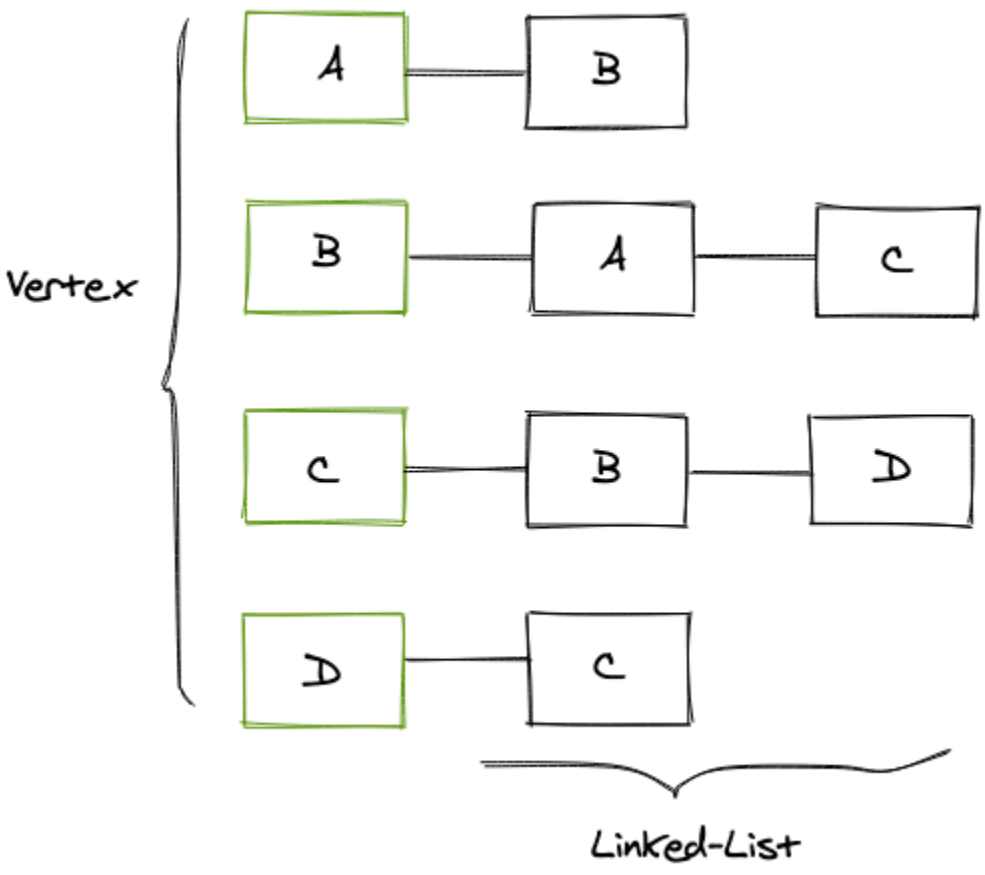
Adjacency List

In the adjacency list representation, we have an array of linked-list where the size of the array is the number of the vertex (nodes) present in the graph. Each vertex has its own linked-list that contains the nodes that it is connected to.

Consider the graph shown below:

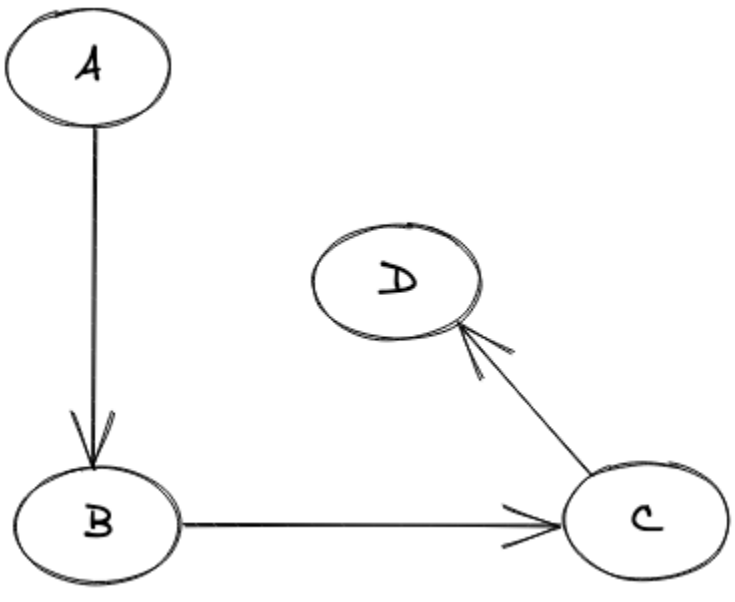


The above graph is an undirected one and the Adjacency list for it looks like:

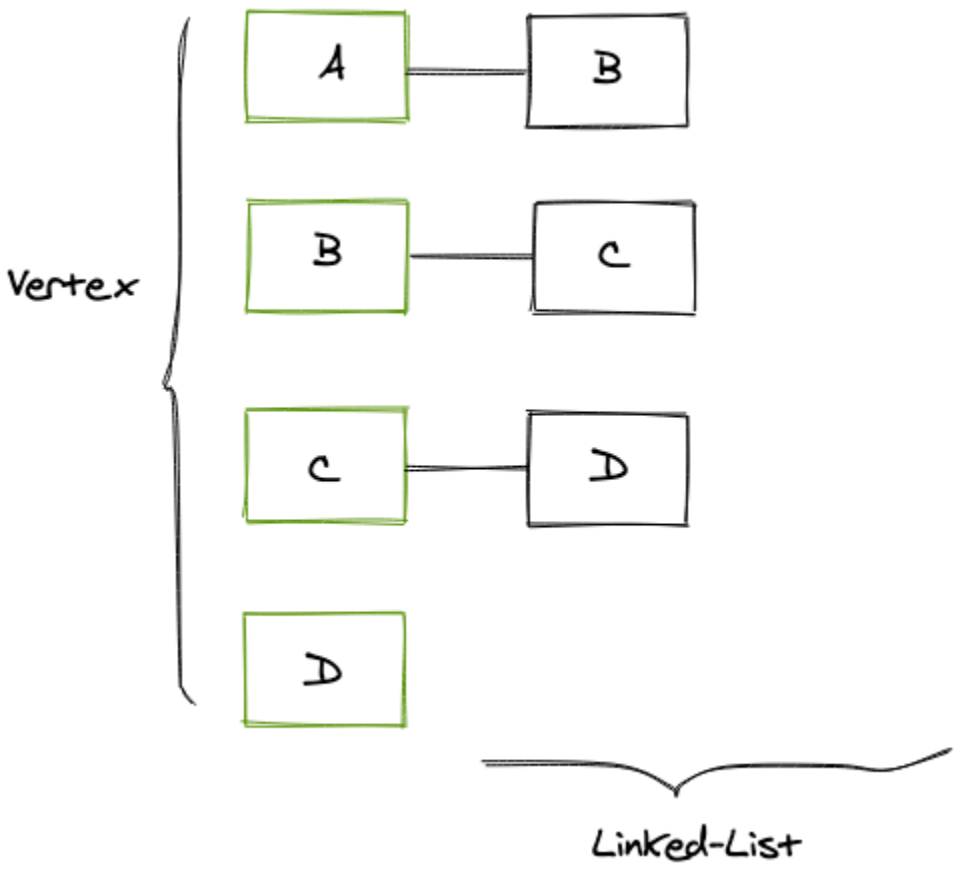


The first column contains all the vertices we have in the graph above and then each of these vertices contains a linked list that in turn contains the nodes that each vertex is connected to. For a directed graph the only change would be that the linked list will only contain the node on which the incident edge is present.

Consider the graph shown below:



The above graph is a directed one and the Adjacency list for this looks like:



Implementation of Adjacency List

The structure (constructor in Java) for the adjacency list will look something like this:

```
public AdjacencyList(int vertex){

        this.vertex = vertex;

        list = new LinkedList[vertex];

        for(int i=0;i<vertex;i++){

            list[i] = new LinkedList<Integer>();

        }

}
```

Copy

The above constructor takes the number of vertices as an argument and then assigns the class level variable this value, and then we create an array of LinkedList of the size of the vertices present in the graph. Finally, we create an empty LinkedList for each item of this array of LinkedList.

It should also be noted that we have two class-level variables, like:

```
int vertex;

LinkedList<Integer> []list;
```

Copy

Now we have laid the foundations and the only thing left is to add the edges together, we do that like this:

```
public void addEdge(int start,int destination){

        list[start].addFirst(destination);
```

```
        list[destination].addFirst(start); // un-directed graph

}
```

Copy

We are taking the vertices from which an edge starts and ends, and we are simply inserting the destination vertex in the LinkedList of the start vertex and vice-versa (as it is for the undirected graph).

Now the only thing left is to print the graph.

```
public void printGraph(){

        for (int i = 0; i < vertex ; i++) {

            if(list[i].size()>0) {

                System.out.print("Node " + i + " is connected to: ");

                for (int j = 0; j < list[i].size(); j++) {

                    System.out.print(list[i].get(j) + " ");

                }

                System.out.println();

            }

        }

}
```

Copy

The entire code looks something like this:

```
import java.util.LinkedList;


public class AdjacencyList {

    int vertex;

    LinkedList<Integer> []list;


    public AdjacencyList(int vertex){

        this.vertex = vertex;

        list = new LinkedList[vertex];

        for(int i=0;i<vertex;i++){

            list[i] = new LinkedList<Integer>();

        }
```

```
    }


    public void addEdge(int start,int destination){
        list[start].addFirst(destination);
        list[destination].addFirst(start); // un-directed graph
    }


    public void printGraph(){
        for (int i = 0; i < vertex ; i++) {
            if(list[i].size()>0) {
                System.out.print("Node " + i + " is connected to: ");
                for (int j = 0; j < list[i].size(); j++) {
                    System.out.print(list[i].get(j) + " ");
                }
                System.out.println();
            }
        }
    }


    public static void main(String[] args) {
        AdjacencyList adl = new AdjacencyList(4);
        adl.addEdge(0,1);
        adl.addEdge(1,2);
        adl.addEdge(2,3);
        adl.printGraph();
    }


}
```

Copy

The output of the above looks like:

```
Node 0 is connected to: 1

Node 1 is connected to: 2 0

Node 2 is connected to: 3 1

Node 3 is connected to: 2
```

Conclusion

- We learned how to represent the graphs in programming, via adjacency matrix and adjacency lists.