Introduction to Graphs

A ***Graph*** is a data structure that consists of the following two components:

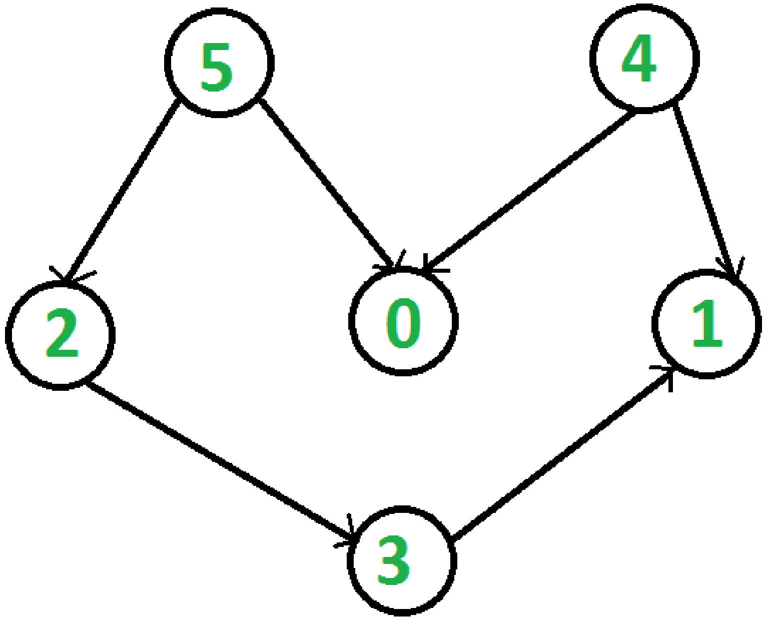
1. A finite set of vertices also called nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(digraph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

Graphs are used to represent many real-life applications:

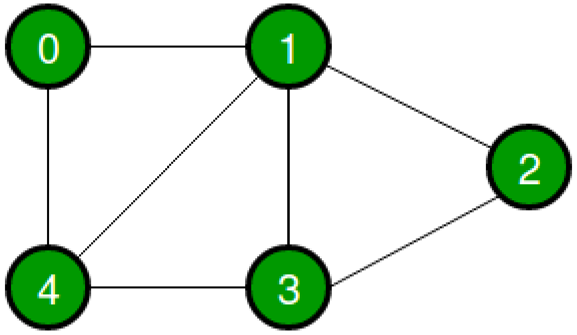
- Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. For example Google GPS
- Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender and locale.

Directed and Undirected Graphs

- **Directed Graphs**: The Directed graphs are such graphs in which edges are directed in a single direction. For Example, the below graph is a directed graph:



- **Undirected Graphs**: Undirected graphs are such graphs in which the edges are directionless or in other words bi-directional. That is, if there is an edge between vertices **u** and **v** then it means we can use the edge to go from both **u to v** and **v to u**. Following is an example of an undirected graph with 5 vertices:



Representing Graphs

Dash

All

Articles

Videos

Problems

Quiz

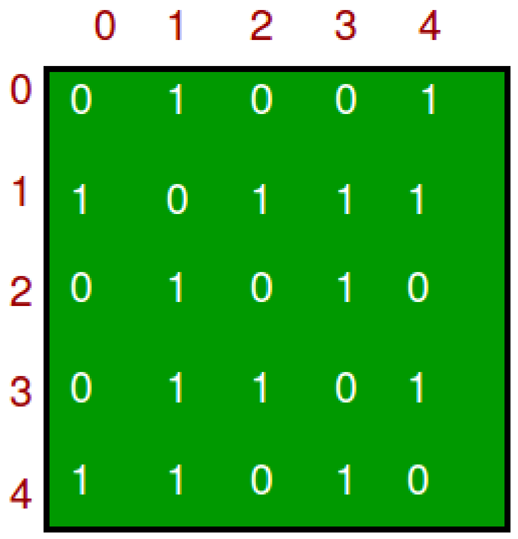
Contest

Following two are the most commonly used representations of a graph:

1. Adjacency Matrix.
2. Adjacency List.

Let us look at each one of the above two method in details:

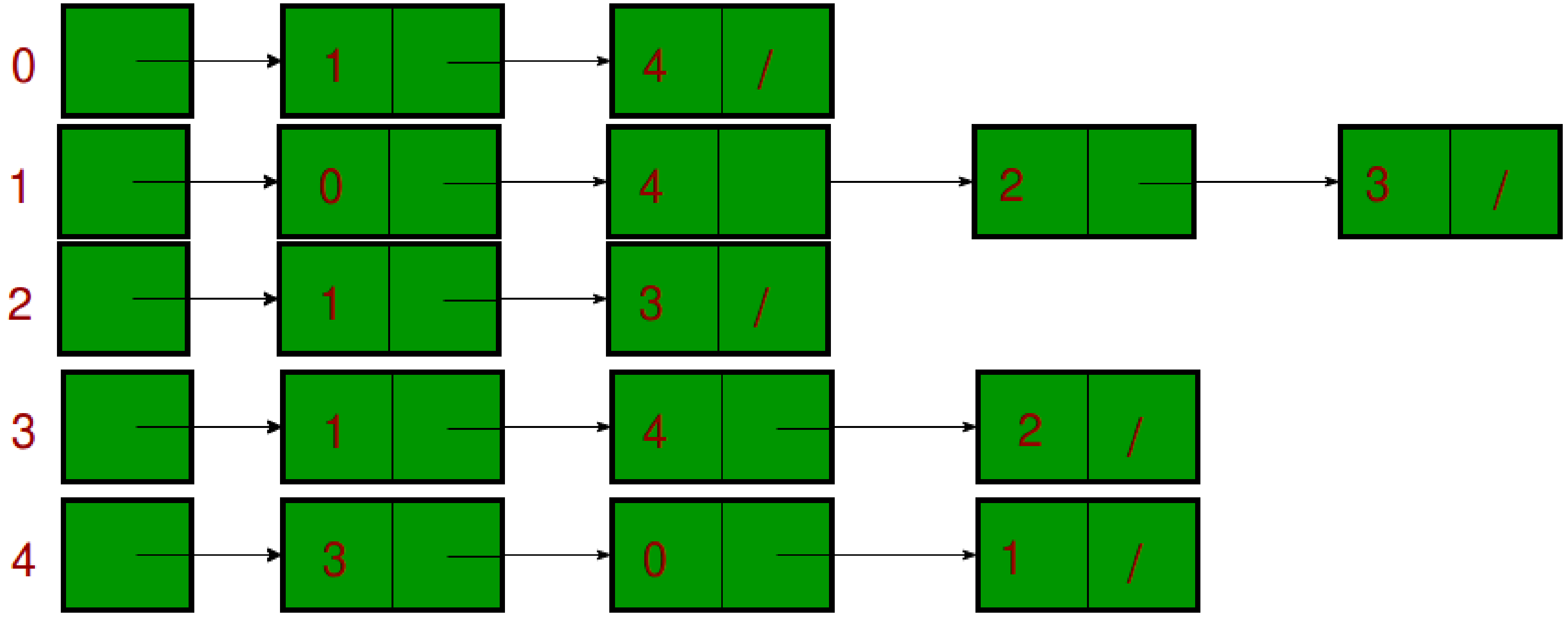
- **Adjacency Matrix:** The Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w. The adjacency matrix for the above example undirected graph is:



$$
\begin{array}{c|ccccc}
 & 0 & 1 & 2 & 3 & 4 \\
\hline
0 & 0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 \\
2 & 0 & 1 & 0 & 1 & 0 \\
3 & 0 & 1 & 1 & 0 & 1 \\
4 & 1 & 1 & 0 & 1 & 0
\end{array}
$$

Pros: Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done O(1). ***Cons:*** Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time. Please see this for a sample Python implementation of adjacency matrix.

- **Adjacency List:** Graph can also be implemented using an array of lists. That is every index of the array will contain a complete list. Size of the array is equal to the number of vertices and every index **i** in the array will store the list of vertices connected to the vertex numbered *i*. Let the array be array[]. An entry array[i] represents the list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above example undirected graph.



Below is the implementation of the adjacency list representation of Graphs: **Note**: In below implementation, we use dynamic arrays (vector in C++/ArrayList in Java) to represent adjacency lists instead of a linked list. The vector implementation has advantages of cache friendliness.

C++ Java

```
// Java code to demonstrate Graph representation
// using ArrayList in Java

import java.util.*;

class Graph {

    // A utility function to add an edge in an
    // undirected graph
    static void addEdge(ArrayList<ArrayList<Integer> > adj,
                        int u, int v)
    {
        adj.get(u).add(v);
        adj.get(v).add(u);
    }
```

```
    {
        for (int i = 0; i < adj.size(); i++) {
            System.out.println("\nAdjacency list of vertex" + i);
            for (int j = 0; j < adj.get(i).size(); j++) {
                System.out.print(" -> "+adj.get(i).get(j));
            }
            System.out.println();
        }
    }

    // Driver Code
    public static void main(String[] args)
    {
        // Creating a graph with 5 vertices
        int V = 5;
        ArrayList<ArrayList<Integer> > adj
                    = new ArrayList<ArrayList<Integer> >(V);

        for (int i = 0; i < V; i++)
            adj.add(new ArrayList<Integer>());

        // Adding edges one by one
        addEdge(adj, 0, 1);
        addEdge(adj, 0, 4);
        addEdge(adj, 1, 2);
        addEdge(adj, 1, 3);
        addEdge(adj, 1, 4);
        addEdge(adj, 2, 3);
        addEdge(adj, 3, 4);

        printGraph(adj);
    }
}
```

Output:

```
Adjacency list of vertex 0
head -> 1-> 4

Adjacency list of vertex 1
head -> 0-> 2-> 3-> 4

Adjacency list of vertex 2
head -> 1-> 3

Adjacency list of vertex 3
head -> 1-> 2-> 4

Adjacency list of vertex 4
head -> 0-> 1-> 3
```

Pros: Saves space O(|V|+|E|). In the worst case, there can be C(V, 2) number of edges in a graph thus consuming O(V^2) space. Adding a vertex is easier. ***Cons:*** Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done O(V).