Lesson:

Graphs

Pre Requisites:

- Basic Java
- Trees in Java

Topics to be covered.

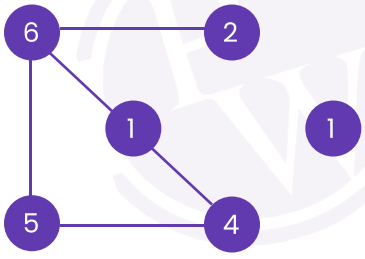
- Introduction to Graphs
- Types of Graph
- Terminologies of Graph
- Graph representation
- Traversal of Graph
- Depth first search
- Breadth First Search
- Topological Sorting

Introduction to Graphs

A graph is an ordered pair G = (V, E) comprising a set V of vertices or nodes and a collection of pairs of vertices from V, known as edges of a graph. For example, for the graph below.

V = { 1, 2, 3, 4, 5, 6 }

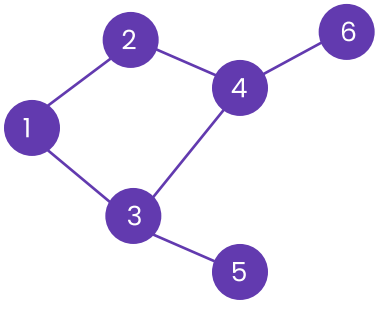
E = { (1, 4), (1, 6), (2, 6), (4, 5), (5, 6) }



Types of Graph

1. Undirected graph

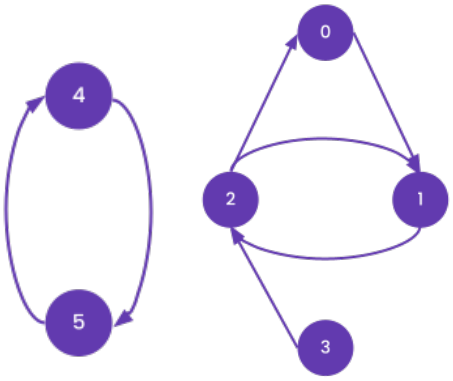
An undirected graph (graph) is a graph in which edges have no orientation. The edge (x, y) is identical to edge (y, x), i.e., they are not ordered pairs. The maximum number of edges possible in an undirected graph without a loop is n×(n-1)/2.



2. Directed graph

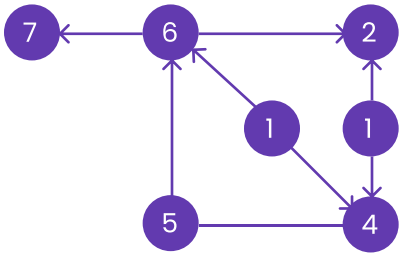
A Directed graph (digraph) is a graph in which edges have orientations, i.e., The edge (x, y) is not identical to

edge (y, x).



3. Directed Acyclic Graph (DAG)

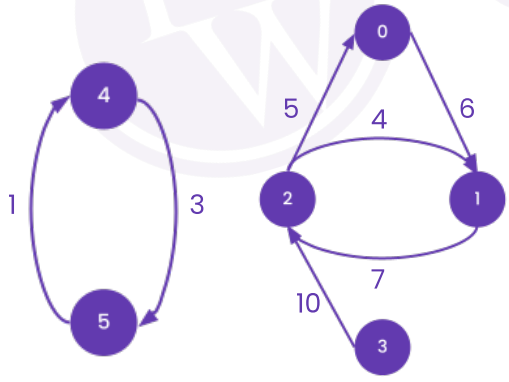
A Directed Acyclic Graph (DAG) is a directed graph that contains no cycles.



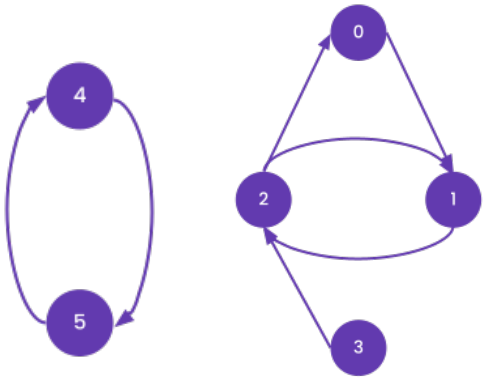
4. Weighted and Unweighted graph

A weighted graph associates a value (weight) with every edge in the graph. We can also use words cost or length instead of weight.

An unweighted graph does not have any value (weight) associated with every edge in the graph. In other words, an unweighted graph is a weighted graph with all edge weight as 1. Unless specified otherwise, all graphs are assumed to be unweighted by default.



Weighted Graph



Unweighted Graph

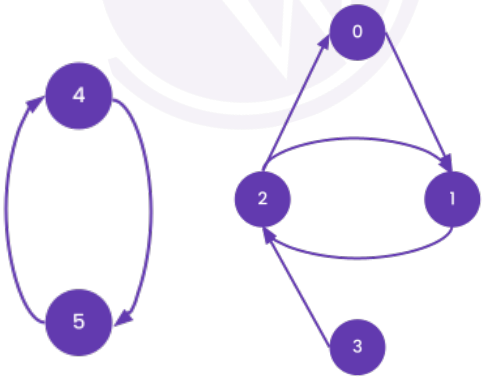
Most commonly used terms in Graphs

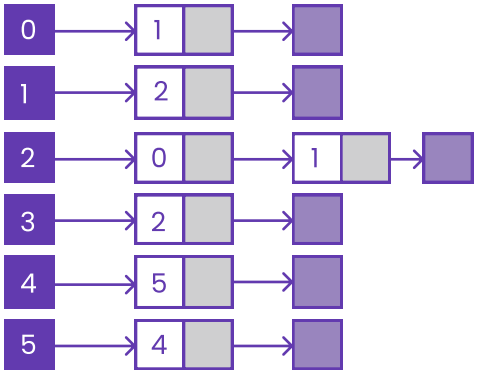
- An edge is (together with vertices) one of the two basic units out of which graphs are constructed. Each edge has two vertices to which it is attached, called its endpoints.
- Two vertices are called adjacent if they are endpoints of the same edge.
- Outgoing edges of a vertex are directed edges that the vertex is the origin.
- Incoming edges of a vertex are directed edges that the vertex is the destination.
- The degree of a vertex in a graph is the total number of edges incident to it.
- In a directed graph, the out-degree of a vertex is the total number of outgoing edges, and the in-degree is the total number of incoming edges.
- A vertex with in-degree zero is called a source vertex, while a vertex with out-degree zero is called a sink vertex.
- Cycle is a path that starts and ends at the same vertex.
- A bridge is an edge whose removal would disconnect the graph.
- Tree is a connected graph with no cycles. If we remove all the cycles from DAG (Directed Acyclic Graph), it becomes a tree, and if we remove any edge in a tree, it becomes a forest.
- Spanning tree of an undirected graph is a subgraph that is a tree that includes all the vertices of the graph.

Graph Representation

Adjacency List Representation:
An adjacency list representation for the graph associates each vertex in the graph with the collection of its neighboring vertices or edges, i.e every vertex stores a list of adjacent vertices.

There are many variations of adjacency list representation depending upon the implementation. This data structure allows the storage of additional data on the vertices but is practically very efficient when the graph contains only a few edges. i.e. the graph is sparse.





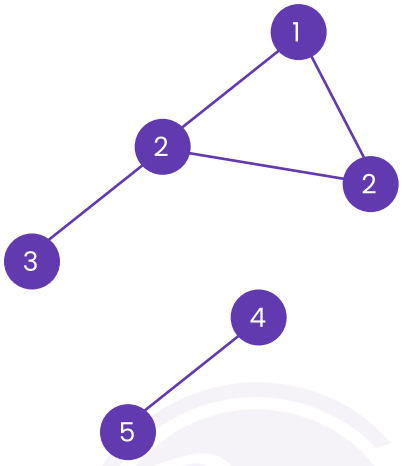
Adjacency List of above graph.

Now we will see how to store a graph in Java. Just like we used to store tree using Node class, here we will create a custom class name Edge which will contain two fields for now i.e. src, dest.

```
// A class to store a graph edge
class Edge
{
    int src, dest;

    Edge(int src, int dest)
    {
        this.src = src;
        this.dest = dest;
    }
}
```

For eg let's take a src graph as



LP_Code1.java

Output:

```
(0 --> 1)
(1 --> 2)
(2 --> 0)       (2 --> 1)
(3 --> 2)
(4 --> 5)
(5 --> 4)
```

In the above code we would create an object of Graph class which has a constructor which creates an adjacency list for us. The input for the constructor is List<Edge> edges;
We would iterate over each edge and in the List we would add src->dest mapping.

Traversal of Graph

Now we will see how to traverse a graph using the adjacency list created in above steps.

There are two algorithms to traverse a graph

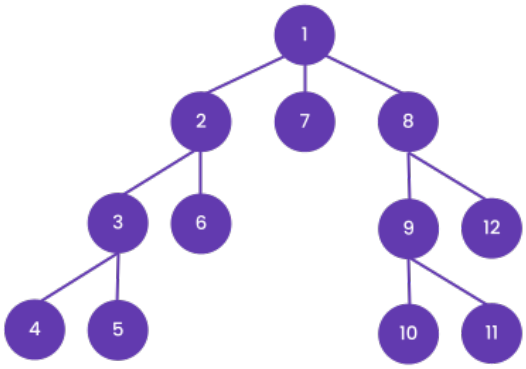
1. Depth first search (DFS)
2. Breadth first search (BFS)

We will learn both the methods in detail now.

Depth First Search (DFS)

Depth–first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root for a graph) and explores as far as possible along each branch before backtracking.

for eg.



For the above graph the ordering of nodes shows us how dfs will print nodes. Starting from number 1 it will go down to branch 4 then 5, then 6 and similarly the remaining graph.

Now let's say we have an adjacency list which contains various nodes. We term every node as a vertex and values present in its adjacency list as neighbors.

But to prevent infinite loops, keep track of the vertices that are already discovered and not revisit them.

```
function dfs(vertex v)
{
   visit(v);
   for each neighbor u of v
      if u is undiscovered
         call dfs(u);
}
```

Now we have a function dfs() which takes a vertex v. We first enter the function and visit the vertex "v". We will also mark it discovered (In a hashmap) so that if we reach vertex "v" by any other vertex we wouldn't revisit it.

After visiting "v", we would go to its neighbor. We will check for every undiscovered neighbor "u" and will call dfs(u) and this process will go on until we visit every node of graph

LP_Code2.java

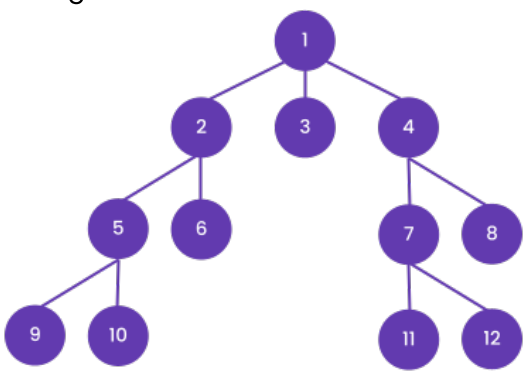
Output:

```
0 1 2 3 4 5 6 7 8 9 10 11 12
```

Breadth First Search

Breadth–first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key') and explores the neighbor nodes first before moving to the next-level neighbors.

For eg.



The above ordering of numbers shows how BFS will traverse the above graph. Starting from 1 it will first traverse 2,3,4 and then there next shortest distance nodes i.e. 5,6 and 7, 8. It will go on till the end of the graph.

```
function bfs(int x, Graph g) {
  Queue<integer> q = new ArrayDeque<>();
  // we will first visit x.
  visit(x);

  // Add "x" into the queue.
  q.add(x);

  while(!q.empty()) {
    // get the top element of the queue.

    v = q.poll();

    visit(v);

    // do for every edge (v, u)

    for (int u: g.adjList(v)) {

      if(!discovered (u)) {
        // mark it as discovered and enqueue it

        discovered[u] = true;
        q.add(u);
      }
    }

  }
}
```

Approach:

1. We would create a queue.
2. Add the starting node “x” into the queue and discover it.
3. Repeat steps 4 - 6 until the queue is empty.
4. Take the first element of the queue out and let it be v
5. Now for every neighbor of v i.e. u

6. If u is undiscovered, discover it and add it in the queue.

LP_Code3.java

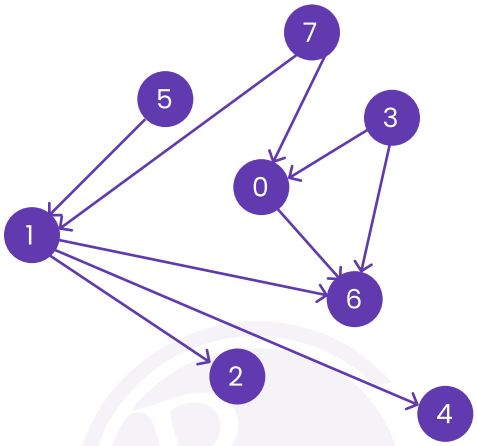
Output:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

Topological Sorting

A Topological sort or Topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering. Topological order is possible if and only if the graph has no directed cycles, i.e. if the graph is DAG.

For example, consider the following graph:



To Visit Node 1, we need to visit 5,7 first.
To Visit Node 0, we need to visit 7,3 first.
Some nodes are not dependent on any node to be visit for eg. 5, 7, 3

There can be many topological orders of this graph. One of the topological order is

7, 5, 3, 1, 4, 0, 2, 6

Indegree of every vertex is the number of incoming edges to it.

Approach to find topological order:

1. We will first find the indegree of every node.
2. We will add all the vertices to a Stack “S”, whose indegree == 0.
3. We will repeat steps 4-7 until the Stack is empty.
4. We will pop one element “v” from stack S, and add it to the end of List L.
5. For all the neighbors of v i.e. u we will reduce indegree by 1.
6. If indegree of any neighbor “u” becomes 0, we will add them to stack S.
7. At the end print all the values of List L.

Note: At the end of this algo we will check the indegree of every node once again, and if we found any node having indegree != 0, then it means the graph has a cycle in it.

LP_Code4.java

Output:

```
[7, 5, 1, 2, 3, 4, 0, 6]
```

Next Class Teasers:

- Cycle Detection
- Dijkstra's Algorithm
- Spanning trees.