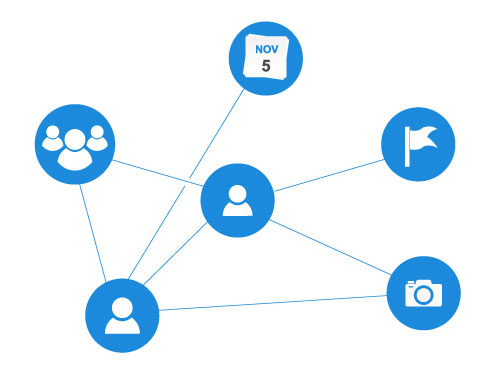
A graph data structure is a collection of nodes that have data and are connected to other nodes.

Let's try to understand this by means of an example. On facebook, everything is a node. That includes User, Photo, Album, Event, Group, Page, Comment, Story, Video, Link, Note...anything that has data is a node.

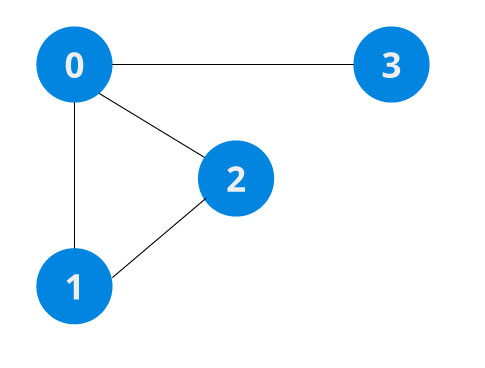
Every relationship is an edge from one node to another. Whether you post a photo, join a group, like a page etc., a new edge is created for that relationship.



All of facebook is then, a collection of these nodes and edges. This is because facebook uses a graph data structure to store its data.

More precisely, a graph is a data structure (V,E) that consists of

* A collection of vertices V
* A collection of edges E, represented as ordered pairs of vertices (u,v)



In the graph,

V = {0, 1, 2, 3}

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

G = {V, E}

## Graph Terminology

* **Adjacency**: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them.
* **Path**: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.
* **Directed Graph**: A graph in which an edge (u,v) doesn't necessary mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge.

## Graph Representation

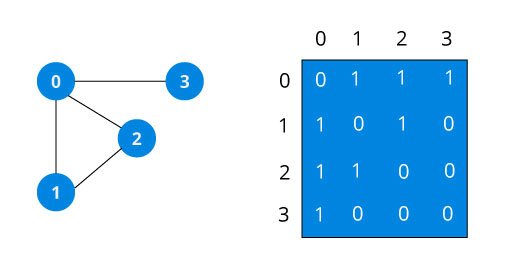
Graphs are commonly represented in two ways:

### 1. Adjacency Matrix

An adjacency matrix is 2D array of V x V vertices. Each row and column represent a vertex.

If the value of any element a[i][j] is 1, it represents that there is an edge connecting vertex i and vertex j.

The adjacency matrix for the graph we created above is



Since it is an undirected graph, for edge (0,2), we also need to mark edge (2,0); making the adjacency matrix symmetric about the diagonal.

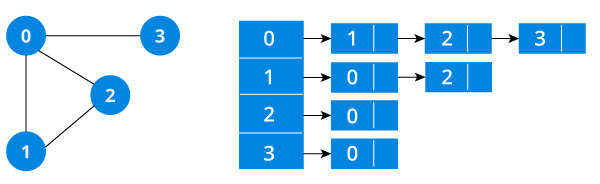
Edge lookup(checking if an edge exists between vertex A and vertex B) is extremely fast in adjacency matrix representation but we have to reserve space for every possible link between all vertices(V x V), so it requires more space.

### 2. Adjacency List

An adjacency list represents a graph as an array of linked list.

The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

The adjacency list for the graph we made in the first example is as follows:



An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a graph with millions of vertices, this can mean a lot of saved space.

## Graph Operations

The most common graph operations are:

* Check if element is present in graph
* Graph Traversal
* Add elements(vertex, edges) to graph
* Finding path from one vertex to another
* Java

// Java Program to demonstrate adjacency list

// representation of graphs

import java.util.LinkedList;

public class GFG

{

// A user define class to represent a graph.

// A graph is an array of adjacency lists.

// Size of array will be V (number of vertices

// in graph)

static class Graph

{

int V;

LinkedList<Integer> adjListArray[];

// constructor

Graph(int V)

{

this.V = V;

// define the size of array as

// number of vertices

adjListArray = new LinkedList[V];

// Create a new list for each vertex

// such that adjacent nodes can be stored

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

adjListArray[i] = new LinkedList<>();

}

}

}

// Adds an edge to an undirected graph

static void addEdge(Graph graph, int src, int dest)

{

// Add an edge from src to dest.

graph.adjListArray[src].addFirst(dest);

// Since graph is undirected, add an edge from dest

// to src also

graph.adjListArray[dest].addFirst(src);

}

// A utility function to print the adjacency list

// representation of graph

static void printGraph(Graph graph)

{

for(int v = 0; v < graph.V; v++)

{

System.out.println("Adjacency list of vertex "+ v);

System.out.print("head");

for(Integer pCrawl: graph.adjListArray[v]){

System.out.print(" -> "+pCrawl);

}

System.out.println("\n");

}

}

// Driver program to test above functions

public static void main(String args[])

{

// create the graph given in above figure

int V = 5;

Graph graph = new Graph(V);

addEdge(graph, 0, 1);

addEdge(graph, 0, 4);

addEdge(graph, 1, 2);

addEdge(graph, 1, 3);

addEdge(graph, 1, 4);

addEdge(graph, 2, 3);

addEdge(graph, 3, 4);

// print the adjacency list representation of

// the above graph

printGraph(graph);

}

}

// This code is contributed by Sumit Ghosh

Output:

Adjacency list of vertex 0

head -> 4-> 1

Adjacency list of vertex 1

head -> 4-> 3-> 2-> 0

Adjacency list of vertex 2

head -> 3-> 1

Adjacency list of vertex 3

head -> 4-> 2-> 1

Adjacency list of vertex 4

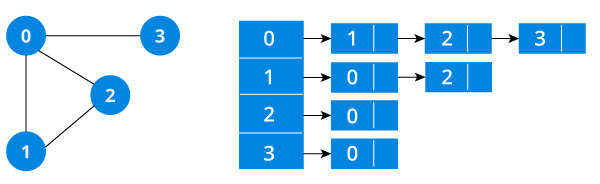
head -> 3-> 1-> 0

An adjacency list represents a graph as an array of linked list.

The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

## Adjacency List representation

A graph and its equivalent adjacency list representation is shown below.



An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a sparse graph with millions of vertices and edges, this can mean a lot of saved space.

## Adjacency List Structure

The simplest adjacency list needs a node data structure to store a vertex and a graph data structure to organize the nodes.

We stay close to the basic definition of graph - a collection of vertices and edges {V, E}. For simplicity we use an unlabeled graph as opposed to a labeled one i.e. the vertexes are identified by their indices 0,1,2,3.

## Adjacency List Java

We use Java Collections to store the Array of Linked Lists.

class Graph

{

private int numVertices;

private LinkedList<integer> adjLists[];

}

The type of LinkedList is determined what data you want to store in it. For a labeled graph, you could store a dictionary instead of an Integer

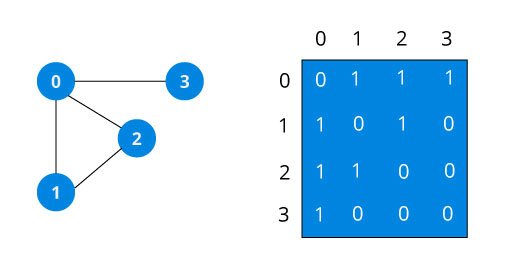
An adjacency matrix is a way of representing a graph G = {V, E} as a matrix of booleans.

## Adjacency matrix representation

The size of the matrix is VxV where V is the number of vertices in the graph and the value of an entry Aij is either 1 or 0 depending on whether there is an edge from vertex i to vertex j.

## Adjacency Matrix Example

The image below shows a graph and its equivalent adjacency matrix.



In case of undirected graph, the matrix is symmetric about the diagonal because of every edge (i,j), there is also an edge (j,i).

## Pros of adjacency matrix

The basic operations like adding an edge, removing an edge and checking whether there is an edge from vertex i to vertex j are extremely time efficient, constant time operations.

If the graph is dense and the number of edges is large, adjacency matrix should be the first choice. Even if the graph and the adjacency matrix is sparse, we can represent it using data structures for sparse matrix.

The biggest advantage however, comes from the use of matrices. The recent advances in hardware enable us to perform even expensive matrix operations on the GPU.

By performing operations on the adjacent matrix, we can get important insights into the nature of the graph and the relationship between its vertices.

## Cons of adjacency matrix

The VxV space requirement of adjacency matrix makes it a memory hog. Graphs out in the wild usually don't have too many connections and this is the major reason why [adjacency lists](https://www.programiz.com/dsa/graph-adjacency-list) are the better choice for most tasks.

While basic operations are easy, operations like inEdges and outEdges are expensive when using the adjacency matrix representation.

## Adjacency Matrix code

If you know how to create two dimensional arrays, you also know how to create adjacency matrix.

public class Graph {

private boolean adjMatrix[][];

private int numVertices;

public Graph(int numVertices) {

this.numVertices = numVertices;

adjMatrix = new boolean[numVertices][numVertices];

}

public void addEdge(int i, int j) {

adjMatrix[i][j] = true;

adjMatrix[j][i] = true;

}

public void removeEdge(int i, int j) {

adjMatrix[i][j] = false;

adjMatrix[j][i] = false;

}

public boolean isEdge(int i, int j) {

return adjMatrix[i][j];

}

public String toString() {

StringBuilder s = new StringBuilder();

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

s.append(i + ": ");

for (boolean j : adjMatrix[i]) {

s.append((j?1:0) + " ");

}

s.append("\n");

}

return s.toString();

}

public static void main(String args[])

{

Graph g = new Graph(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

System.out.print(g.toString());

/\* Outputs

0: 0 1 1 0

1: 1 0 1 0

2: 1 1 0 1

3: 0 0 1 0

\*/

}

}

Traversal means visiting all the nodes of a graph. Breadth first traversal or Breadth first Search is a recursive algorithm for searching all the vertices of a graph or tree data structure. In this article, you will learn with the help of examples the BFS algorithm, BFS pseudocode and the code of the breadth first search algorithm with implementation in C++, C, Java and Python programs.

## BFS algorithm

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

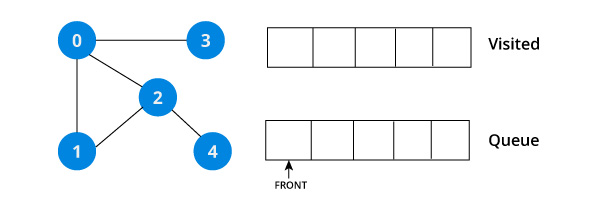
The algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

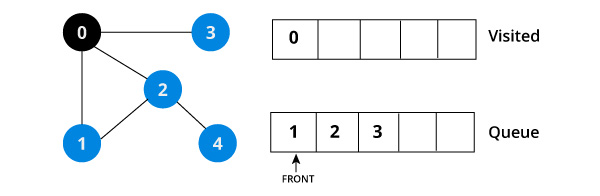
The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

## BFS example

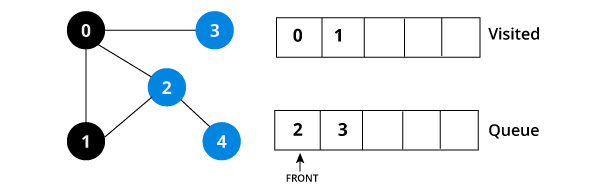
Let's see how the Breadth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



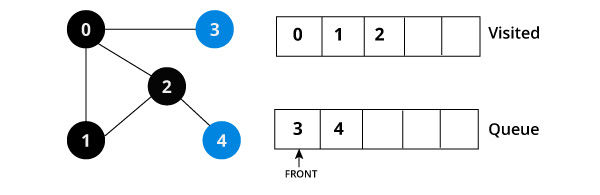
We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.

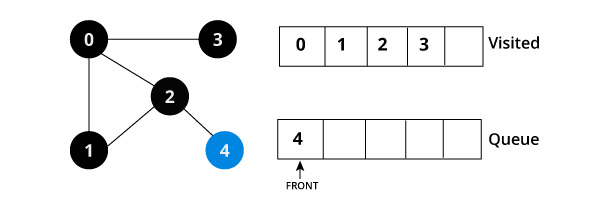


Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

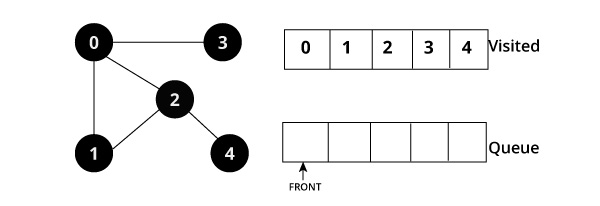


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.





Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited. We visit it.



Since the queue is empty, we have completed the Depth First Traversal of the graph.

## BFS pseudocode

create a queue Q

mark v as visited and put v into Q

while Q is non-empty

remove the head u of Q

mark and enqueue all (unvisited) neighbours of u

## BFS code

The code for the Breadth First Search Algorithm with an example is shown below. The code has been simplified so that we can focus on the algorithm rather than other details.

import java.io.\*;

import java.util.\*;

class Graph

{

private int numVertices;

private LinkedList<Integer> adjLists[];

private boolean visited[];

Graph(int v)

{

numVertices = v;

visited = new boolean[numVertices];

adjLists = new LinkedList[numVertices];

for (int i=0; i i = adjLists[currVertex].listIterator();

while (i.hasNext())

{

int adjVertex = i.next();

if (!visited[adjVertex])

{

visited[adjVertex] = true;

queue.add(adjVertex);

}

}

}

}

public static void main(String args[])

{

Graph g = new Graph(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

System.out.println("Following is Breadth First Traversal "+

"(starting from vertex 2)");

g.BFS(2);

}

}

# Breadth First Traversal or BFS for a Graph

[Breadth First Traversal (or Search)](http://en.wikipedia.org/wiki/Breadth-first_search) for a graph is similar to Breadth First Traversal of a tree (See method 2 of [this post](http://www.geeksforgeeks.org/archives/2686)). The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.  
For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don’t mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.

Note that the above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex (example Disconnected graph). To print all the vertices, we can modify the BFS function to do traversal starting from all nodes one by one (Like the [DFS modified version](http://www.geeksforgeeks.org/archives/18212)) .

Time Complexity: O(V+E) where V is number of vertices in the graph and E is number of edges in the graph.

**package** com.test.graph.bfs;

**import** java.util.Iterator;

**import** java.util.LinkedList;

// This class represents a directed graph using adjacency list

// representation

**class** Graph

{

**private** **int** V; // No. of vertices

**private** LinkedList<Integer> adj[]; //Adjacency Lists

// Constructor

@SuppressWarnings("unchecked")

Graph(**int** v)

{

V = v;

adj = **new** LinkedList[v];

**for** (**int** i=0; i<v; ++i)

adj[i] = **new** LinkedList<>();

}

// Function to add an edge into the graph

**void** addEdge(**int** v,**int** w)

{

adj[v].add(w);

}

// prints BFS traversal from a given source s

**void** BFS(**int** s)

{

// Mark all the vertices as not visited(By default

// set as false)

**boolean** visited[] = **new** **boolean**[V];

// Create a queue for BFS

LinkedList<Integer> queue = **new** LinkedList<Integer>();

// Mark the current node as visited and enqueue it

visited[s]=**true**;

queue.add(s);

**while** (queue.size() != 0)

{

// Dequeue a vertex from queue and print it

s = queue.poll();

System.***out***.print(s+" ");

// Get all adjacent vertices of the dequeued vertex s

// If a adjacent has not been visited, then mark it

// visited and enqueue it

Iterator<Integer> i = adj[s].listIterator();

**while** (i.hasNext())

{

**int** n = i.next();

**if** (!visited[n])

{

visited[n] = **true**;

queue.add(n);

}

}

}

}

// Driver method to

**public** **static** **void** main(String args[])

{

Graph g = **new** Graph(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

System.***out***.println("Following is Breadth First Traversal "+

"(starting from vertex 2)");

g.BFS(2);

}

}

Traversal means visiting all the nodes of a [graph](https://www.programiz.com/dsa/graph). Depth first traversal or Depth first Search is a recursive algorithm for searching all the vertices of a graph or tree data structure. In this article, you will learn with the help of examples the DFS algorithm, DFS pseudocode and the code of the depth first search algorithm with implementation in C++, C, Java and Python programs.

## DFS algorithm

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

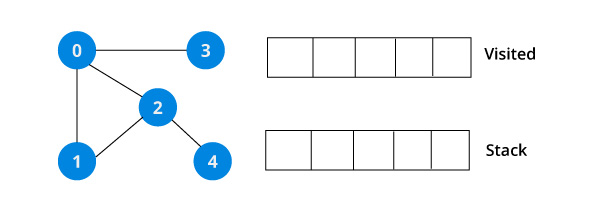
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

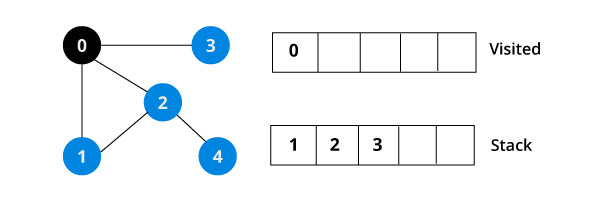
1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

## DFS example

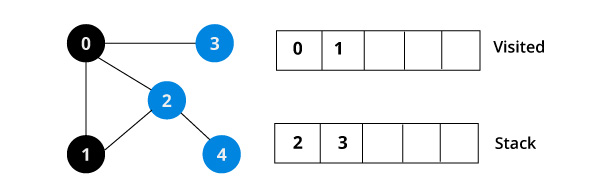
Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



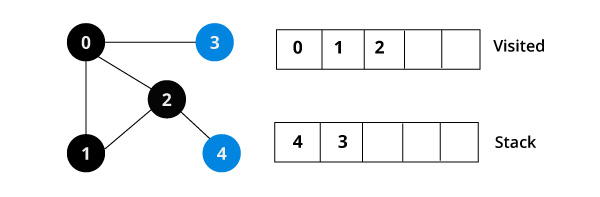
We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.

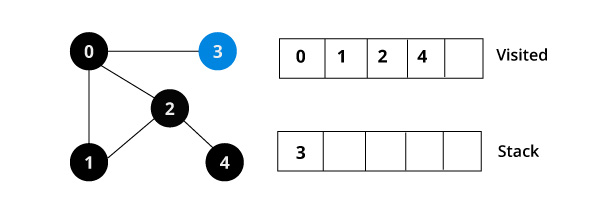


Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

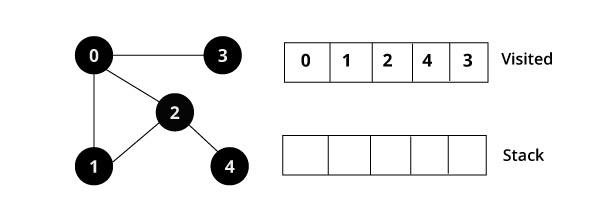


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.





After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



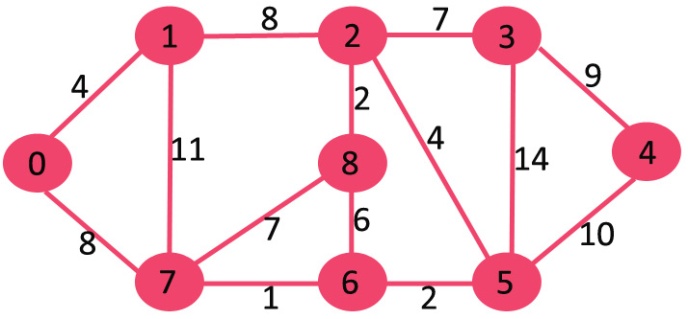
# Greedy Algorithms | Set 5 (Prim’s Minimum Spanning Tree (MST))

We have discussed [Kruskal’s algorithm for Minimum Spanning Tree](http://www.geeksforgeeks.org/archives/26604). Like Kruskal’s algorithm, Prim’s algorithm is also a [Greedy algorithm](http://www.geeksforgeeks.org/archives/18528). It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.  
A group of edges that connects two set of vertices in a graph is called [cut in graph theory](http://en.wikipedia.org/wiki/Cut_%28graph_theory%29). So, at every step of Prim’s algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the verices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).

**How does Prim’s Algorithm Work?** The idea behind Prim’s algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a Spanning Tree. And they must be connected with the minimum weight edge to make it a Minimum Spanning Tree.

**Algorithm**  
**1)** Create a set mstSet that keeps track of vertices already included in MST.  
**2)** Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.  
**3)** While mstSet doesn’t include all vertices  
….**a)** Pick a vertex u which is not there in mstSet and has minimum key value.  
….**b)** Include u to mstSet.  
….**c)** Update key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v

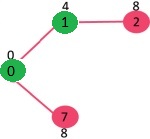
The idea of using key values is to pick the minimum weight edge from [cut](http://en.wikipedia.org/wiki/Cut_(graph_theory)). The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

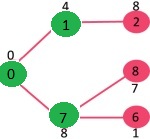
Let us understand with the following example:  
[](http://www.geeksforgeeks.org/wp-content/uploads/Fig-11.jpg)

The set mstSet is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum key value. The vertex 0 is picked, include it in mstSet. So mstSet becomes {0}. After including to mstSet, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.

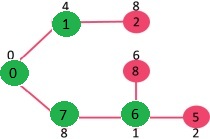
[](http://www.geeksforgeeks.org/wp-content/uploads/MST1.jpg)

Pick the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.

[](http://www.geeksforgeeks.org/wp-content/uploads/MST2.jpg)

Pick the vertex with minimum key value and not already included in MST (not in mstSET). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So mstSet now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (7 and 1 respectively).  
[](http://www.geeksforgeeks.org/wp-content/uploads/MST3.jpg)

Pick the vertex with minimum key value and not already included in MST (not in mstSET). Vertex 6 is picked. So mstSet now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.

[](http://www.geeksforgeeks.org/wp-content/uploads/MST4.jpg)

We repeat the above steps until mstSet includes all vertices of given graph. Finally, we get the following graph.

[](http://www.geeksforgeeks.org/wp-content/uploads/MST5.jpg)

## [Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/minimum-spanning-tree/1)

**How to implement the above algorithm?**  
We use a boolean array mstSet[] to represent the set of vertices included in MST. If a value mstSet[v] is true, then vertex v is included in MST, otherwise not. Array key[] is used to store key values of all vertices. Another array parent[] to store indexes of parent nodes in MST. The parent array is the output array which is used to show the constructed MST.