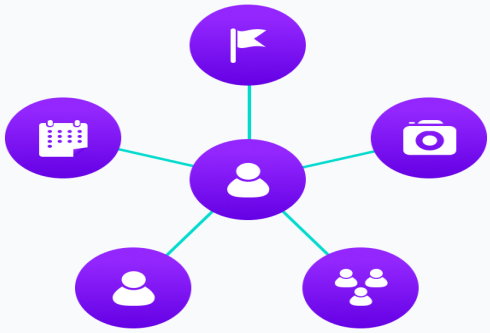
GRAPH

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

Let's try to understand this through 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.

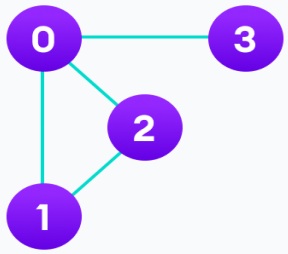


Example of graph data structure

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)



Vertices and edges

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 necessarily 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

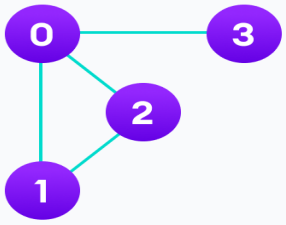
Graphs are commonly represented in two ways:

1. Adjacency Matrix

An adjacency matrix is a 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



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

Graph adjacency matrix

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.

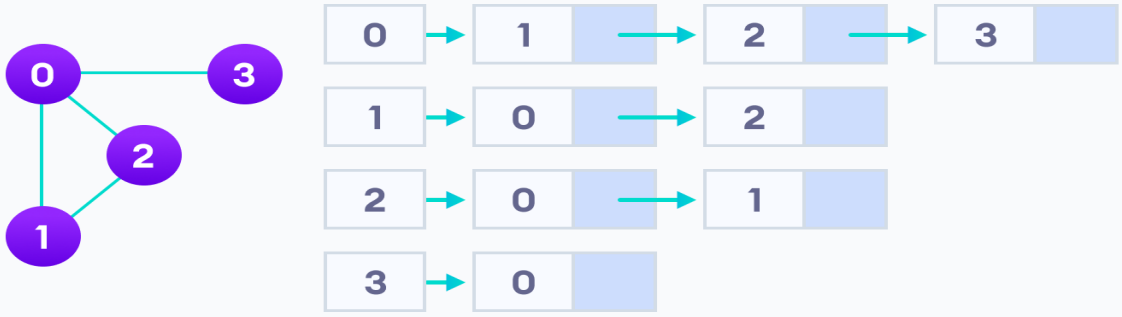
Edge lookup(checking if an edge exists between vertex A and vertex B) is extremely fast in adjacency matrix representation but we have to reserve space for every possible link between all vertices(V x V), so it requires more space.

2. Adjacency List

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

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:



Adjacency list representation

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 the element is present in the graph
- Graph Traversal
- Add elements(vertex, edges) to graph
- Finding the path from one vertex to another

MINIMUM SPANNING TREE

A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree.

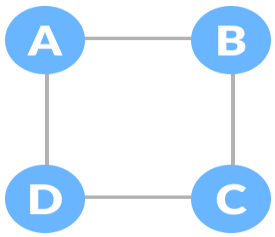
The edges may or may not have weights assigned to them.

> **The total number of spanning trees with `n` vertices that can be created from a complete graph is equal to `n Power (n-2)`.**

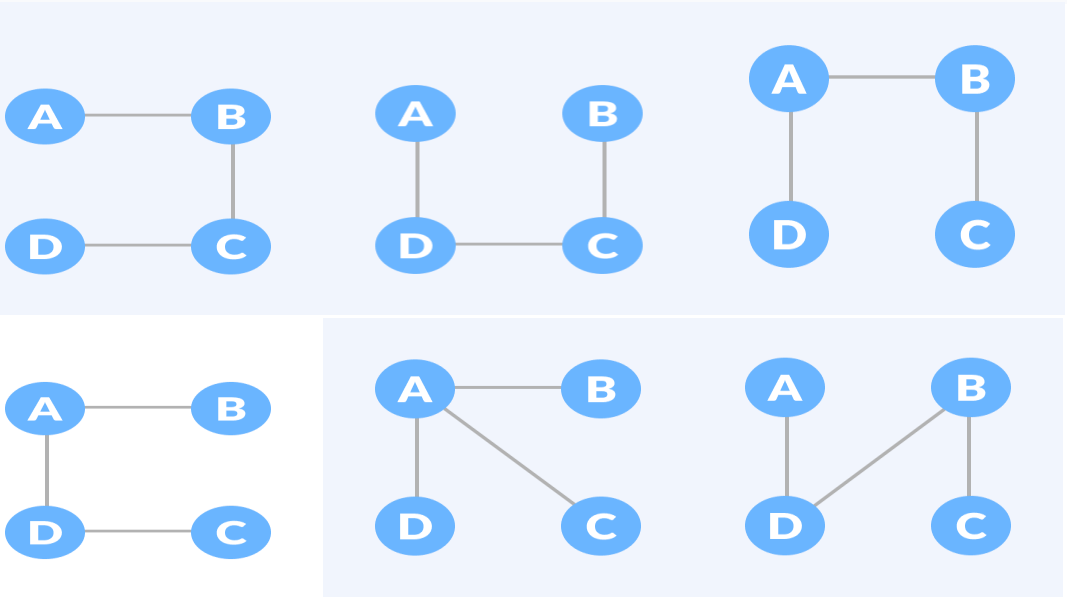
Example of a Spanning Tree

Let's understand the spanning tree with examples below:

Let the original graph be:



Some of the possible spanning trees that can be created from the above graph are:



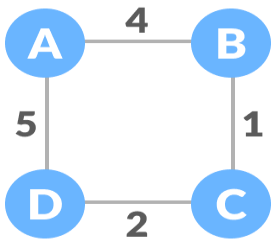
Minimum Spanning Tree

A minimum spanning tree is a spanning tree in which the sum of the weight of the edges is as minimum as possible.

Example of a Spanning Tree

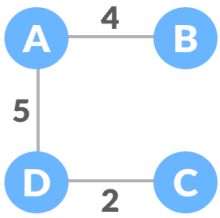
Let's understand the above definition with the help of the example below.

The initial graph is:

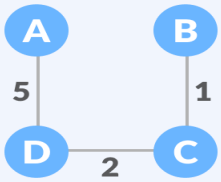


Weighted graph

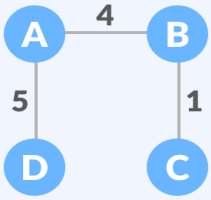
The possible spanning trees from the above graph are:



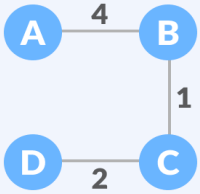
sum = 11



sum = 8

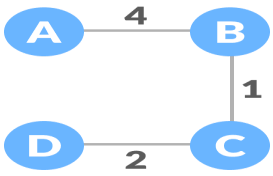


sum = 10



sum = 7

The minimum spanning tree from the above spanning trees is:



sum = 7

The minimum spanning tree from a graph is found using the following algorithms:

1. **Prim's Algorithm**
2. **Kruskal's Algorithm**

Spanning Tree Applications

- **Computer Network Routing Protocol**
- **Cluster Analysis**
- **Civil Network Planning**

Minimum Spanning tree Applications

- **To find paths in the map**
- **To design networks like telecommunication networks, water supply networks, and electrical grids.**

PRIMS ALGORITHM

Prim's Algorithm is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

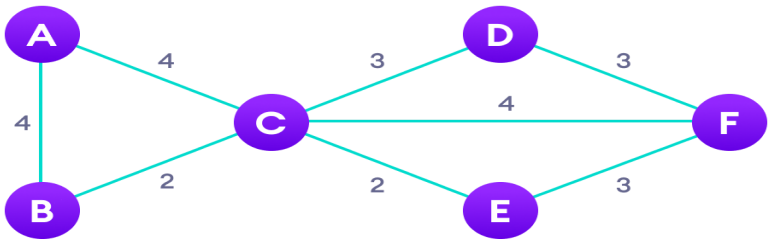
Prim's algorithm starts with the single node and explore all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

The steps for implementing Prim's algorithm are as follows:

1. **Initialize the minimum spanning tree with a vertex chosen at random.**

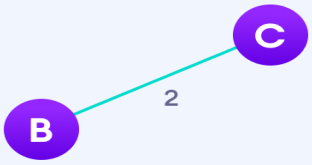
2. **Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree**
3. **Keep repeating step 2 until we get a minimum spanning tree**

Example of Prim's algorithm :

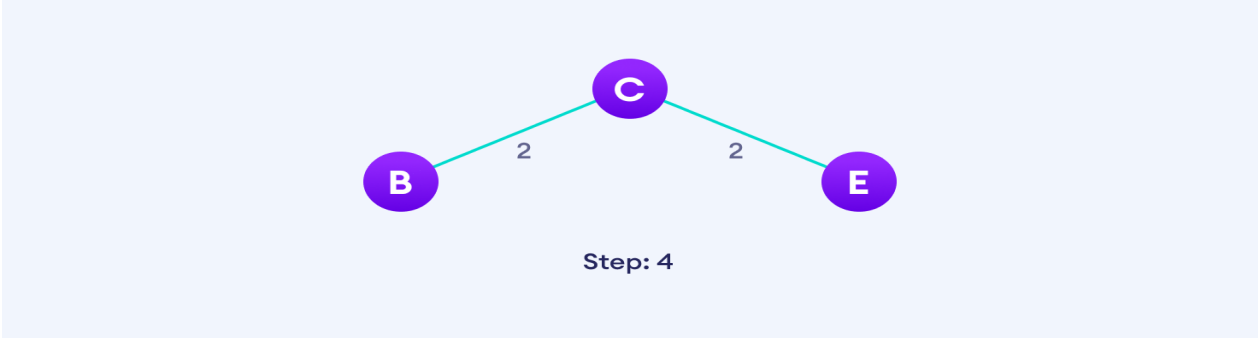


Step: 1

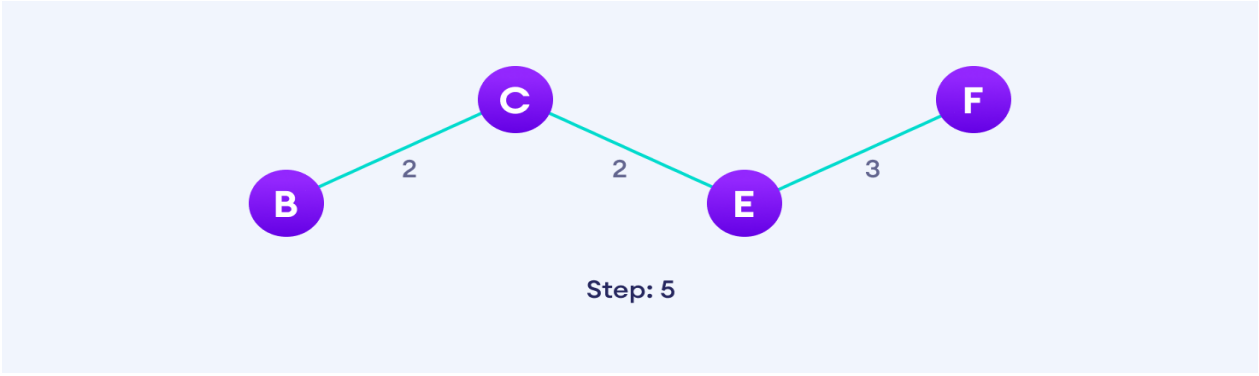
Step: 2



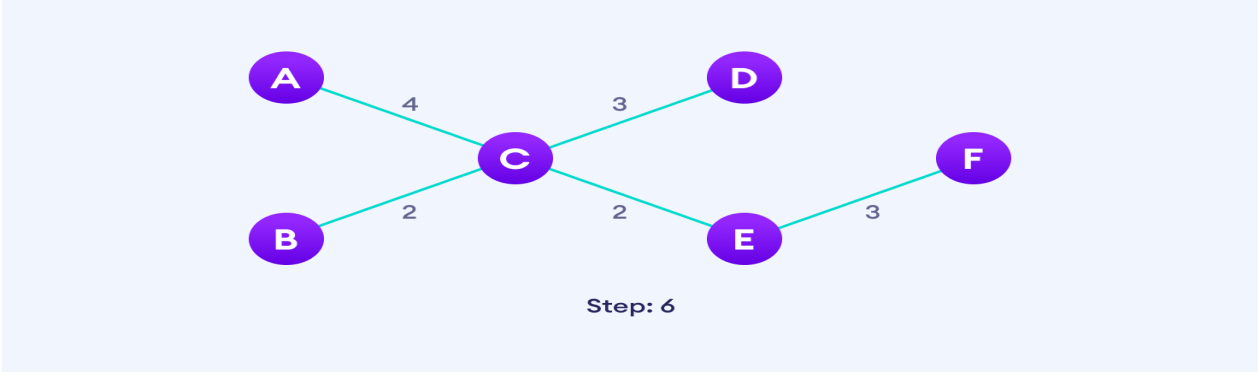
Step: 3



Step: 4



Step: 5



Step: 6

```
#include<bits/stdc++.h>
using namespace std;
vector<pair<int,int>>adj[20000007];
void addEdge(int u, int v, int w)
{
      adj[u].push_back({v, w});
      adj[v].push_back({u, w});
}

void primMST(int V)
```

```
{
priority_queue<pair<int, int>, vector <pair<int, int>> ,
greater<pair<int, int>> > pq;
     int src = 0;
     vector<int>dis(V, INT_MAX);
     vector<int> parent(V, -1);
     vector<bool> inMST(V, false);
     pq.push({0, src});
     dis[src] = 0;
          while (!pq.empty())
     {
          int u = pq.top().second;
          pq.pop();

          if(inMST[u] == true)
          continue;
          inMST[u] = true;

          for(auto it:adj[u])
          {
               int v=it.first;
               int weight=it.second;

               if (inMST[v] == false && dis[v] > weight)
               {
                    dis[v] = weight;
                    pq.push({dis[v], v});
                    parent[v] = u;
               }
          }
     }
     for(int i=1;i<V;i++)
     cout<<parent[i]<<"  ";
}
```

```
int main()
{

      int V = 9;
      addEdge(0, 1, 4);
      addEdge(0, 7, 8);
      addEdge(1, 2, 8);
      addEdge(1, 7, 11);
      addEdge(2, 3, 7);
      addEdge(2, 8, 2);
      addEdge(2, 5, 4);
      addEdge(3, 4, 9);
      addEdge(3, 5, 14);
      addEdge(4, 5, 10);
      addEdge(5, 6, 2);
      addEdge(6, 7, 1);
      addEdge(6, 8, 6);
      addEdge(7, 8, 7);

    primMST(V);
    return 0;
}
```

The time complexity of Prim's algorithm is `O(ElogV)`.

KRUSKAL'S ALGORITHM

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

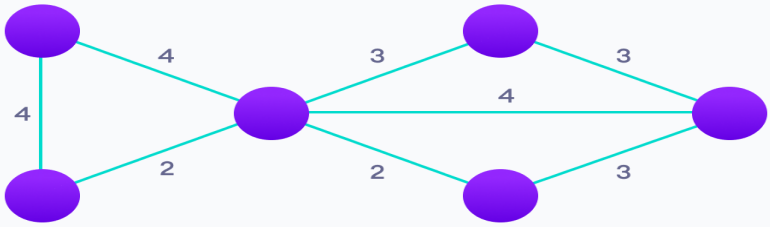
- **form a tree that includes every vertex**

- has the minimum sum of weights among all the trees that can be formed from the graph.

The steps for implementing Kruskal's algorithm are as follows:

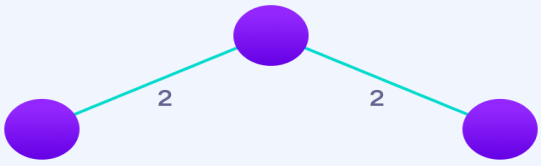
1. **Sort all the edges from low weight to high**
2. **Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.**
3. **Keep adding edges until we reach all vertices.**

Example of Kruskal's algorithm :

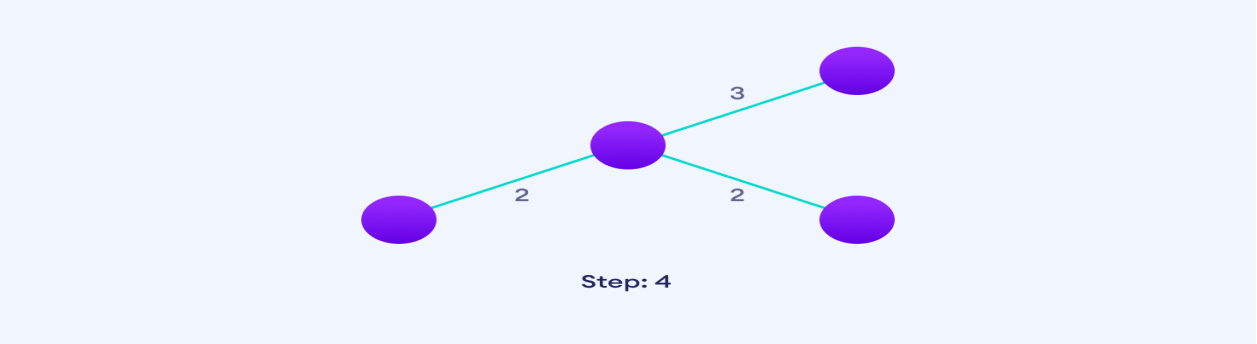


Step: 1

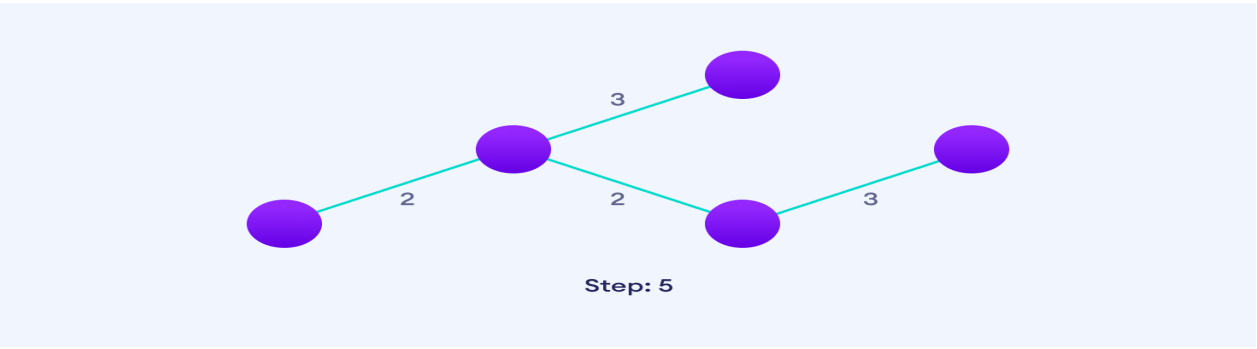
Step: 2



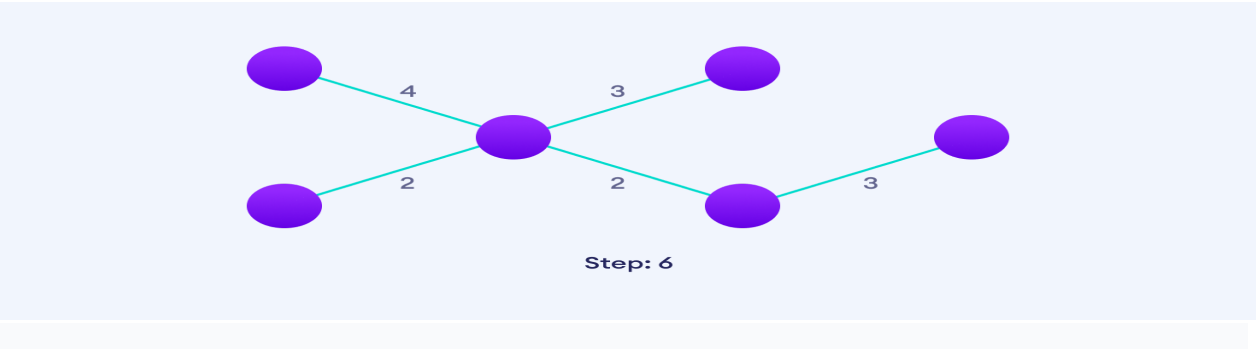
Step: 3



Step: 4



Step: 5



Step: 6

```
#include <bits/stdc++.h>
using namespace std;
vector<int>parent(2000007);
vector<int>rankk(2000007);
vector<vector<int>>edgelist;
vector<pair<int,int>>res;

    int find(int i)
     {
          if(parent[i] == -1)
          return i;
        return parent[i] = find(parent[i]);
     }
```

```
    void unite(int x, int y)
     {
          int s1 = find(x);
          int s2 = find(y);
          if (s1 != s2)
          {
               if (rankk[s1] < rankk[s2])
               {
                    parent[s1] = s2;
                    rankk[s2] += rankk[s1];
               }
               else
               {
                    parent[s2] = s1;
                    rankk[s1] += rankk[s2];
               }
          }
    }
    void addEdge(int x, int y, int w)
     {
          edgelist.push_back({w, x, y});
     }

     void kruskals_mst()
     {
          sort(edgelist.begin(), edgelist.end());
          int ans = 0;
          for (auto edge : edgelist)
          {
               int w = edge[0];
               int x = edge[1];
               int y = edge[2];
```

```
                if (find(x) != find(y))
                {
                     unite(x, y);
                     res.push_back({x,y});
                }
           }
        for(int i=0;i<res.size();i++)
        cout<<res[i].first<<" "<<res[i].second<<endl;
          return ;
     }

int main()
{
     for(int i=0;i<2000007;i++)
    {
        parent[i]=-1;
        rankk[i]=1;
    }
     addEdge(0, 1, 1);
     addEdge(1, 3, 3);
     addEdge(3, 2, 4);
     addEdge(2, 0, 2);
     addEdge(0, 3, 2);
     addEdge(1, 2, 2);
     kruskals_mst();
     return 0;
}
```