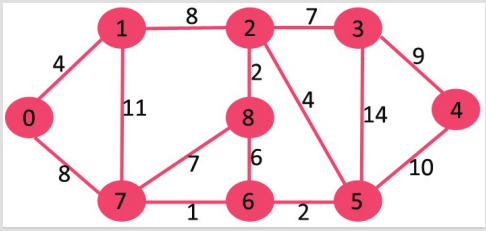
Prim's algorithm using priority_queue in STL

Given an undirected, connected and weighted graph, find Minimum Spanning Tree (MST) of the graph using Prim's algorithm.




```
Input : Adjacency List representation
        of above graph
Output :  Edges in MST
          0 - 1
          1 - 2
          2 - 3
          3 - 4
          2 - 5
          5 - 6
          6 - 7
          2 - 8


Note :  There are two possible MSTs, the other
        MST includes edge 0-7 in place of 1-2.
```

We have discussed below Prim's MST implementations.

- Prim's Algorithm for Adjacency Matrix Representation (In C/C++ with time complexity $O(v^2)$)
- Prim's Algorithm for Adjacency List Representation (In C with Time Complexity O(ELogV))

The second implementation is time complexity wise better, but is really complex as we have implemented our own priority queue. STL provides priority_queue, but the provided priority queue doesn't support decrease key operation. And in Prim's algorithm, we need a priority queue and below operations on priority queue :

- ExtractMin : from all those vertices which have not yet been included in MST, we need to get vertex with minimum key value.
- DecreaseKey : After extracting vertex we need to update keys of its adjacent vertices, and if new key is smaller, then update that in data structure.

The algorithm discussed here can be modified so that decrease key is never required. The idea is, not to insert all vertices in priority queue, but only those which are not MST and have been visited through a vertex that has included in MST. We keep track of vertices included in MST in a separate boolean array inMST[].

```
1) Initialize keys of all vertices as infinite and
   parent of every vertex as -1.

2) Create an empty priority_queue pq.  Every item
   of pq is a pair (weight, vertex). Weight (or
   key) is used used as first item  of pair
   as first item is by default used to compare
   two pairs.

3) Initialize all vertices as not part of MST yet.
   We use boolean array inMST[] for this purpose.
   This array is required to make sure that an already
   considered vertex is not included in pq again. This
   is where Ptim's implementation differs from Dijkstra.
   In Dijkstr's algorithm, we didn't need this array as
   distances always increase. We require this array here
   because key value of a processed vertex may decrease
   if not checked.

4) Insert source vertex into pq and make its key as 0.

5) While either pq doesn't become empty
    a) Extract minimum key vertex from pq.
       Let the extracted vertex be u.

    b) Include u in MST using inMST[u] = true.

    c) Loop through all adjacent of u and do
       following for every vertex v.

           // If weight of edge (u,v) is smaller than
           // key of v and v is not already in MST
           If inMST[v] = false && key[v] > weight(u, v)

               (i) Update key of v, i.e., do
                     key[v] = weight(u, v)
               (ii) Insert v into the pq
               (iv) parent[v] = u

6) Print MST edges using parent array.
```

Below is C++ implementation of above idea.

```
// STL implementation of Prim's algorithm for MST
#include<bits/stdc++.h>
using namespace std;
# define INF 0x3f3f3f3f

// iPair ==>  Integer Pair
typedef pair<int, int> iPair;

// This class represents a directed graph using
// adjacency list representation
class Graph
{
    int V;    // No. of vertices

    // In a weighted graph, we need to store vertex
    // and weight pair for every edge
    list< pair<int, int> > *adj;

public:
    Graph(int V);  // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v, int w);

    // Print MST using Prim's algorithm
    void primMST();
};

// Allocates memory for adjacency list
```

```
// Allocates memory for adjacency list
Graph::Graph(int V)
{
    this->V = V;
    adj = new list<iPair> [V];
}

void Graph::addEdge(int u, int v, int w)
{
    adj[u].push_back(make_pair(v, w));
    adj[v].push_back(make_pair(u, w));
}

// Prints shortest paths from src to all other vertices
void Graph::primMST()
{
    // Create a priority queue to store vertices that
    // are being preinMST. This is weird syntax in C++.
    // Refer below link for details of this syntax
    // http://geeksquiz.com/implement-min-heap-using-stl/
    priority_queue< iPair, vector <iPair> , greater<iPair> > pq;

    int src = 0; // Taking vertex 0 as source

    // Create a vector for keys and initialize all
    // keys as infinite (INF)
    vector<int> key(V, INF);

    // To store parent array which in turn store MST
    vector<int> parent(V, -1);

    // To keep track of vertices included in MST
    vector<bool> inMST(V, false);

    // Insert source itself in priority queue and initialize
    // its key as 0.
    pq.push(make_pair(0, src));
    key[src] = 0;

    /* Looping till priority queue becomes empty */
    while (!pq.empty())
    {
        // The first vertex in pair is the minimum key
        // vertex, extract it from priority queue.
        // vertex label is stored in second of pair (it
        // has to be done this way to keep the vertices
        // sorted key (key must be first item
        // in pair)
        int u = pq.top().second;
        pq.pop();

        inMST[u] = true;  // Include vertex in MST

        // 'i' is used to get all adjacent vertices of a vertex
        list< pair<int, int> >::iterator i;
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
        {
            // Get vertex label and weight of current adjacent
            // of u.
            int v = (*i).first;
            int weight = (*i).second;

            //  If v is not in MST and weight of (u,v) is smaller
            // than current key of v
            if (inMST[v] == false && key[v] > weight)
            {
                // Updating key of v
                key[v] = weight;
                pq.push(make_pair(key[v], v));
                parent[v] = u;
            }
        }
    }
```

```
    // Print edges of MST using parent array
    for (int i = 1; i < V; ++i)
        printf("%d - %d\n", parent[i], i);
}

// Driver program to test methods of graph class
int main()
{
    // create the graph given in above fugure
    int V = 9;
    Graph g(V);

    //  making above shown graph
    g.addEdge(0, 1, 4);
    g.addEdge(0, 7, 8);
    g.addEdge(1, 2, 8);
    g.addEdge(1, 7, 11);
    g.addEdge(2, 3, 7);
    g.addEdge(2, 8, 2);
    g.addEdge(2, 5, 4);
    g.addEdge(3, 4, 9);
    g.addEdge(3, 5, 14);
    g.addEdge(4, 5, 10);
    g.addEdge(5, 6, 2);
    g.addEdge(6, 7, 1);
    g.addEdge(6, 8, 6);
    g.addEdge(7, 8, 7);

    g.primMST();

    return 0;
}
```

Output :

```
0 - 1
1 - 2
2 - 3
3 - 4
2 - 5
5 - 6
6 - 7
2 - 8
```

Time complexity : O(E Log V))