Minimum Spanning Tree

Description In the Minimum Spanning Tree problem, we are given as input an undirected graph $G = (V, E)$ together with weight $w(u, v)$ on each edge $(u, v) \in E$. The goal is to find a minimum spanning tree for G. Recall that we learned two algorithms, Kruskal's and Prim's in class. In this assignment, you are asked to implement Prim's algorithm. The following is a pseudo-code of Prim's algorithm.

```
Initialize a min-priority queue $Q$.
for all $u \in V$ do
    $u.key = \infty$.
    $u.\pi = NIL$.
    Insert $(Q, u)$.
end for
Decrease-key$(Q, r, 0)$.
while $Q \neq \emptyset$ do
    $u =$ Extract-Min$(Q)$.
    for all $v \in Adj[u]$ do
        if $v \in Q$ and $w(u, v) < v.key$ then
            $v.\pi = u$.
            Decrease-Key$(Q, v, w(u, v))$.
        end if
    end for
end while
```

Input The input is G, w, and r, where r is an arbitrary vertex the user can specify as root. The input has the following format. There are two integers on the first line. The first integer represents the number of vertices, $|V|$. The second integer is the number of edges, $|E|$. Vertices are indexed by $0, 1, \ldots, |V| - 1$. Each of the following $|E|$ lines has three integers $u, v, w(u, v)$ representing an edge (u, v) with weight $w(u, v)$. Use vertex 0 as the root r.

Output The above pseudo-code stores the MST by π, where $v.\pi = u$ means that u is v's unique parent; here, $r.\pi =$ NIL since r has no parent. Output the MST by outputting the π value of a vertex in each line, in the order $1, 2, \ldots, |V| - 1$. (Do not output the root's parent.)

Implementation Issues Prim's algorithm requires a min-priority queue that supports the `DecreaseKey` operation which is not supported by the standard C++ priority queue. You are allowed to use an "inefficient" priority queue that supports each operation in $O(|V|)$ time. Such an inefficient priority queue can be easily implemented using an array. Then, the running time of your implementation is roughly $O(|E||V|)$. The given template code adopts that inefficient implementation.

However, you may still use the C++ priority queue with a simple "invalidation trick" and have your code to run in $O(|E| \log |V|)$. Instead of decreasing an element's key, just mark the element as invalid and push a new (valid) element with the new key value to the queue. Then you just have to be careful when extracting a minimum element because what you really want is a minimum element that is valid. So extracting a valid min element could need multiple iterations. However, at any point in time, the priority queue has at most $O(|E|)$ elements, so each `ExtractMin` operation takes $O(\log |E|) = O(\log |V|)$ time. Since you extract minimum elements at most $O(|E|)$ times, you only need $O(|E| \log |V|)$ time for extracting valid min elements. To use invalidation trick, you just need to maintain a boolean variable for each vertex to track if it is included in the current MST: when you extract a vertex from the min priority queue, it is valid only if it is not yet part of the current MST.

Example of input and output

Input

```
9
14
0 1 40
0 7 85
1 2 80
1 7 110
2 3 70
2 5 45
2 8 22
3 4 90
3 5 140
4 5 100
5 6 25
6 7 10
6 8 60
7 8 75
```

(The input is a graph with weights given per edge.)

Output

```
0
1
2
3
2
5
6
2
```

Here, the first number refers to the parent of vertex 1, and the second number to the parent of vertex 2, and so on. Note that every line is followed by an enter key.

See the lab guidelines for submission/grading, etc., which can be found in Files/Labs.