BİL 331/BİL 531:Spring 2025

Homework: Minimum Spanning Trees

Assigned Monday, February 24. You should upload your program to uzak.etu.edu.tr by Monday, March 17 (any time up to midnight). When uploading do not compress your source directory, upload each file in your source directory separately, and do NOT use packages.

Overview:

In this project you will implement a number of different data structures (graphs, multiway trees, and heaps) to compute and store a minimum spanning tree (MST) of a connected undirected graph with weighted edges. The implementation must be done in Java. Your program will input a connected undirected graph with weighted edges, and compute an initial MST for the graph. Then as changes occur in the graph structure, the MST is to be updated. These changes include the insertion of edges, and decreasing the weights of edges.

Graph Input:

Your program must read from an input file, for example text.txt, given as a command-line argument, and output to the standard output. In other words, your program will be invoked from the command line as:

```
java mstprogram test.txt
```

The input file begins with a description of the graph. This description starts with the number of vertices, followed by a list of vertex identifiers, one per line. Each vertex has an associated string *identifier*. You may assume that each string identifier consists of at most 20 characters. For example, here is the input for three vertices.

```
3                       (number of vertices)
V128.8.10.14            (vertex identifier)
V128.8.128.423
V128.8.10.268
```

To specify the edges, the number of edges is given followed by a list of edges, each consisting of the identifiers of the endpoints and the edge weight (float). You may assume that all weights are positive.

```
2                                       (number of edges)
V128.8.10.14   V128.8.10.268   43.56    (endpoints and edge weight)
V128.8.10.268  V128.8.128.423  170.36
```

You may assume that the first part of the input follows these specification (that is, you do not have to check for input format errors). You may assume that the vertex identifiers are unique, and that the edges are distinct (no multiple edges) and the endpoints are valid vertices.

MST by Prim's Algorithm:

You may assume that the input graph is connected. After inputting the graph, compute its MST using Prim's algorithm. For consistency, you should use the first vertex input (V128.8.10.14 in this case) as the starting vertex for the algorithm. Output the sequence of edges that have been added to the tree and their associated weights.

Directives:

The rest of the input consists of a sequence of directives, each indicating an operation to be performed on the graph. For those directives that alter the graph, you should incrementally update your MST after this operation.

Print the MST: Print the contents of the current MST, given a particular vertex v as the root. Details on the output format are provided later in the document, but basically involve printing the MST according to a preorder traversal of the tree.

Path: Given two vertex identifiers, compute and print the path between them in the MST.

Insert edge: Insert an edge between two existing vertices into the graph.

Decrease weight: Decrease the weight on an existing edge in the graph by the given amount.

Here is the format of the directives.

```
print-mst       u       //Print the MST using u as the root
path            u v     //Print the path from u to v in the MST
insert-edge     u v w   //Insert edge (u,v) with weight w in the graph
decrease-weight u v w   //Decrease the weight of edge (u,v) by w units
quit                    //This is the last directive.
```

You may assume that the input will adhere to these conventions. However if any operation is illegal (attempting to insert an edge that already exists or decreasing weight of an edge that does not exist), print "Invalid Operation".

Implementation Requirements:

You are required to implement at least three separate data structures for the assignment: (1) an undirected (weighted) graph, (2) a multiway tree, and (3) a heap priority queue. The undirected graph must be implemented using an adjacency list. The MST is to be stored in the multiway tree, using the firstChild and nextSibling representation, where each node in the tree stores a reference to its first child, its next sibling (and possibly previous sibling). (If you prefer another representation, that will be fine too, but in your documentation make sure to explain it well.) It will also be necessary to have a parent pointer for each node for path finding operations. The binary heap is used in Prim's algorithm.

The update operations must be performed incrementally, by making the fewest possible changes to the MST. In particular, it is illegal to completely rebuild the MST from scratch after each operation using Prim's algorithm.

You are not required to provide an efficient method for mapping a vertex identifier to a vertex object. You may use the built-in Hashtable class in java.util, but you can just do a linear search if you like.

I would suggest that you begin by implementing just Prim's algorithm and the procedure for printing the MST. Then go about adding the other operations. This way you can get partial credit if the other operations are not working.

The binary heap must support an operation `decreaseKey()`, which decreases the key value of an entry in the heap, and rearranges the heap contents accordingly. Note that the hard part of this operation is locating where the item is stored in the heap. There is no fast method for searching a heap for a given key other than checking every value in the heap (think about this). You should provide a cross-reference mechanism by which each vertex can find its corresponding node in the heap in $O(1)$ time.

Major Data Structures:

The main data structures consist of the graph, which is represented using an adjacency list representation and a multiway tree. The graph data structure will likely consist of at least two types of objects: *vertices* (shaded in the figure below) and *edges*. Each vertex object contains information such as the vertex identifier and a reference to the adjacency list. Each edge object contains information about the edge, such as the edge endpoints, (in the figure only destination vertex is shown), and the edge weight. In the figure the destination vertex of the edge is given as an identifier, but this would actually be a reference to the vertex object.

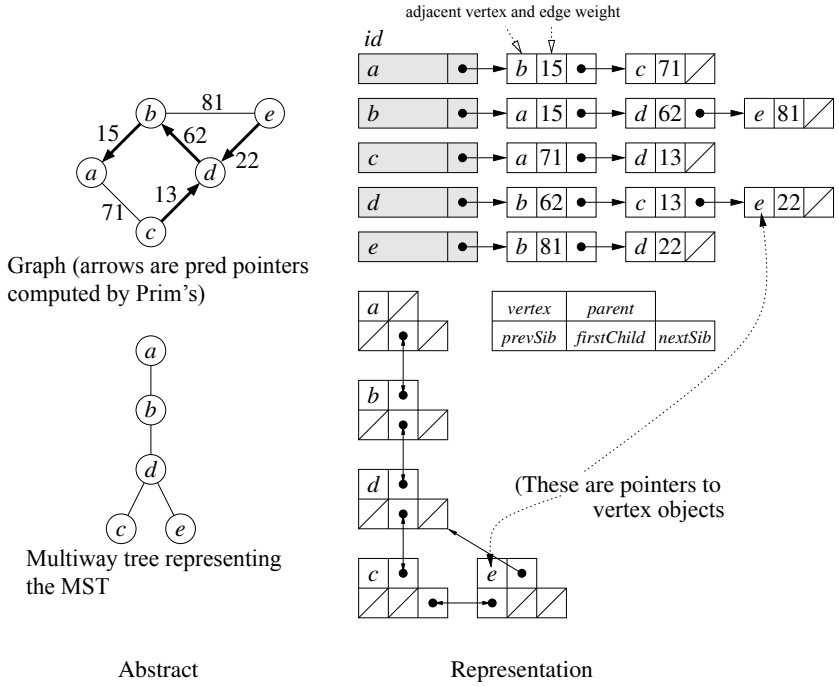

Figure 1: Overview of major data structures.

When Prim's algorithm runs, the link from each vertex u to its predecessor corresponds to a link in the MST. Each MST is represented as a multiway tree (a rooted tree where a node may have any

number of children). Each node of this tree contains a reference to the associated vertex in the graph. (In the figure this is given as the vertex identifier, but actually this would be a reference to the vertex object.) In addition the node will contain pointers to the first child, next sibling, previous sibling, and parent in the multiway tree.

The incremental MST update operations should be performed in a reasonably efficient manner. The insert edge operation may generally result in the addition of a new edge into the MST. To determine this, let u and v be the endpoints of this edge. Find the path from u to v in the current MST. Find the maximum weight edge (x, y) on the path between them. If $w(u, v) < w(x, y)$, then replace edge (x, y) with edge (u, v) in the MST. Otherwise there is no change. Decreasing the weight of an edge operates in much the same way, since the edge may now be added to the tree.

The multiway tree will need to provide a number of operations to support reorganization of the tree to make an arbitrary node the new root (which is required for the print-MST operation), or to compute the path between two vertices in the tree.

Evert:

The evert operation is a very useful operation, and it simplifies many aspects of the homework. Given an MST (represented as multiway rooted tree) and given any node u in this tree, the operation `evert(u)` produces a modified tree which has the same edge structure but in which u is made the root of the tree. The figure below shows an example of the evert operation, on the abstract multiway tree (left) and on the firstChild-nextSibling representation (right). We have shown parent links as dotted lines.

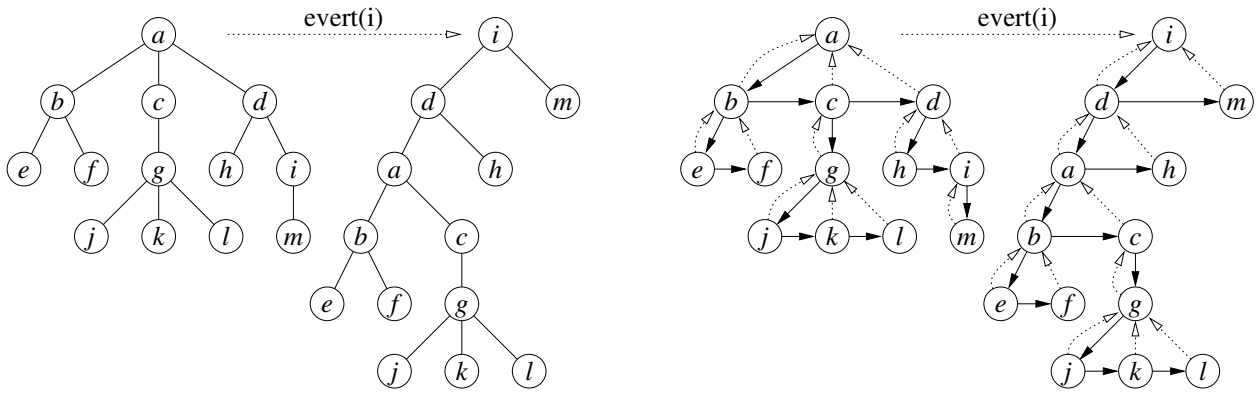

Figure 2: Evert operation on a multiway tree.

As a hint to implementing evert, you may want begin by implementing two utility operations, `cut(u)`, which cuts a node u and its associated subtree out of a tree by deleting the link to its parent in the multiway tree, and `link(u,v)`, which assumes that u is the root of a tree which has been cut, and links this subtree rooted at u as a new child of node v. The figure below show the result of applying `cut(c)` and `link(c,d)`. Evert can be implemented as a sequence of cuts and links.

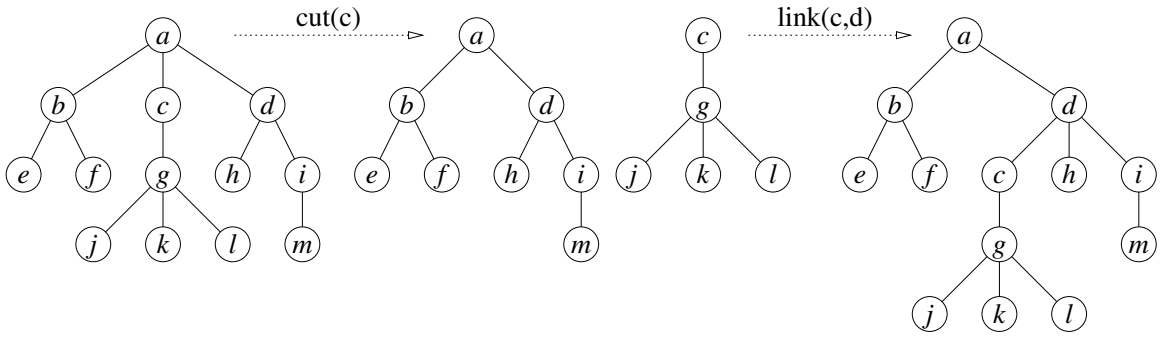

Figure 3: Application of `cut(c)` and `link(c,d)`.

Output Format: For each directive, first output the directive, for example, the operation "`insert-edge a b 2.0`" will print:

```
Directive-----------------> insert-edge a b 2.0
```

If the directive is illegal (attempting to insert an edge that already exists or decreasing the weight of an edge that does not exist), print the directive followed by a line "Invalid Operation". For example, if there is no edge (u, v) in the graph, the operation "`decrease-weight u v`" will print:

```
Directive-----------------> decrease-weight u v
Invalid Operation
```

For the path and print-mst directives print the associated output.
For the path operation the output will list the vertices on the path, separated by commas. For example for the tree shown in Figure 2, the operation "`path c i`" will print:

```
Directive-----------------> path c i
c, a, d, i
```

Each MST should be printed in preorder. Start each line with a string ". . . " to indicate depth. For consistency of output, the children of each node should be listed in increasing order by the string identifier. One easy way to implement this ordering is that whenever a node u is added as a child to a another node v, insert u in sorted order among v's children.
For example, for the tree shown in Figure 2, the operation "`print-mst i`" would produce the output shown on the right. This can be done by invoking `evert(i)`, and then applying a preorder traversal of the tree.

```
Directive-----------------> print-mst i
  i
  . d
  . . a
  . . . b
  . . . . e
  . . . . f
  . . . c
  . . . . g
  . . . . . j
  . . . . . k
  . . . . . l
  . . h
  . m
```

Submission Details: Your submission will consist of an encapsulation of files which must include the following items:

README: There must be a file called `README`, which contains:

Your name:

How to compile and run your program

Known Bugs and Limitations: List any known bugs, deficiencies, or limitations with respect to the project specifications.

File directory: If you have multiple source or data files please explain the purpose of each.

Source Files: All the source files.

DO NOT INCLUDE: Please delete all .class files prior to submission.