CSE 326 – Data Structures
Winter 2004
Wet assignment #3

Due dates.

Team formation: by 5pm on Monday, March 8, 2004
Full assignment: by class time on Friday, March 12 (electronic)
in class on Friday, March 12, 2004 (on paper; *see instructions*)

General.

In this assignment you will implement Kruskal's algorithms for minimum spanning tree. You have to use Union-Find (UF) as in the implementation of Kruskal explained in class. However, you need to support some dynamic changes in the graph – that will require an extension of the UF ADT.

The following operations need to be supported:

Init(int n) – Initialize the graph to consist n vertices and no edges. The vertices are labeled $0,..n-1$. You can assume that $n>1$. Required time complexity: $O(n)$.

New_edge (int i,int j,int c) – a new edge $e=(i,j)$ is added to the graph. The cost of e is c. You can assume that c is a positive integer, and that i,j are integers between 0 and $n-1$ (corresponding to vertices in the graph). You can also assume that $i\neq j$. If following this operation the graph becomes connected (it was not connected without the new edge), output the cost of a minimum spanning tree.
Required time complexity: $O(\log n)$.
Important: The edges are added to the graph in a sorted order from the cheapest to the most expensive one. That is, all edges that will be added to the graph after this operation will have cost at least c.

Remove_expensive_ST_edges (int k) – Remove from the graph (and therefore also from the minimum spanning tree) the k most expensive edges that are part of the minimum spanning tree. If the resulting graph is connected, output the cost of a minimum spanning tree. Otherwise, output that no ST exists. You can assume that $k<n$, and that whenever *Remove_expensive_ST_edges(k)* is performed, the graph is connected.
Let e_1 be the cheapest edge removed in this operation, let n_1 denote the number of non-removed edges having cost at least $c(e_1)$.
Required time complexity: $O(k+n_1 \log n)$.

Remarks.

1. You can assume that a single *init* operation will be performed. It is possible to have a sequence of *New_edge* operations, then *Remove_expensive_ST_edges()*, then another sequence of *New_edge()* and *Remove_expensive_ST_edges()* [and this can be repeated again and again].
2. An edge $e=(i,j)$ might be added and deleted more than once (see e=(3,4) in the example below).
3. There is only one graph, the number of vertices, n, does not change.

Example.

Init(6) – A graph with the 6 nodes {0,1,2,3,4,5} is created

New_edge (0,1,1)

New_edge (1,2,2)

New_edge (2,3,3)

New_edge (2,0,4)

New_edge (4,3,5)

New_edge (0,4,6)

New_edge (5,4,7) – The graph is connected now and the program should output that the MST has weight 18. [The MST consists of the edges having costs {1,2,3,5,7}, this is not part of the program output, only for you to follow the example]

New_edge (0,3,8)

New_edge (5,3,9)

New_edge (2,4,10)

Remove_expensive_ST_edges(2) – The MST consists of the edges having costs {1,2,3,5,7}, the edges {5,7} are removed from the graph. The edges {6,9} are added to the MST. The program should output that the MST has weight 21. [In this case k=2 and n_1=4. The MST consists of the edges having costs {1,2,3,6,9}, again, no need to output this information].

New_edge (3,4,11)

Remove_expensive_ST_edges(4) – The MST consists of the edges having costs {1,2,3,6,9}, the edges {2,3,6,9} are removed from the graph. The remaining graph is not connected; the program should output that no ST exists. [In this case k=4 and n_1=4].

New_edge (1,5,12) - The graph is connected now and the program should output that the MST has weight 35. [The MST consists of the edges having costs {1,4,8,10,12}]

Remark: for simplicity, in the above example edges have distinct costs. In general, this is not the case, and edges added consequently might have the same cost.

Directions.

1. Solve the following 'dry' question (include the answer in your submission):
Consider the disjoint union-find ADT. Suppose we want to add an extra operation, **deunion**, which undoes the last union operation that was not already undone.

a. Show that if we do union by weight and finds without path-compression, then each of union and deunion can be performed in *O(1),* and find in *O(log n).*

b. Why does path compression make deunion hard?

2. Implement the extended disjoint union-find ADT (union-find-deunion).

3. Use the extended ADT and additional data structures of your choice to implement the above operations. In your documentation you should explain how Kruskal's algorithm is adjusted to support the *Remove_expensive_ST_edges()* operation.

I/O.

Your program should take a single command-line argument which will be a file name. **If your program does not run properly with a single command-line argument which is a filename, points may be deducted.**

That file will be a series of commands, one command per line, corresponding to each of the three function calls defined above. Your program should correctly perform each command listed in the file. Each command takes the form of a single character followed by a series of numbers, all separated by spaces. You may assume that all commands are properly formatted.

1. "I <#graph nodes>". Perform the *Init(int n)* function. For example, I 6 will initialize a graph with 6 nodes which is indexed as {0,1,2,3,4,5}. You can assume there will be exactly one init command, and that it will be the first line of the file.
2. "N <node1> <node2> <edge cost>". Perform *New_edge (int i,int j,int c)*. For example, N 1 2 8 will add the edge {1,2} into the graph. The weight of the new edge is 8.
3. "R <#edges>". Perform *Remove_expensive_ST_edges (int k)*.

All outputs should go to standard output (e.g. System.out.println(), or printf(), depending if you are using Java or C++).

You should make your own files for testing, taking care of all possible cases.

Submission.

The submission is electronic and part of it is also paper-based, as specified below.
We need *only one* electronic submission and *only one* printed submission per team.

What to submit. All of the following (1-6) need to be submitted *electronically* by class time on the due date:

1) Your code – the class `UFKruskal.java` or `UFKruskal.cpp` (depending on which language you use) with the code for all of the implemented operations. Also include any supporting headers, packages, and the test module(s) you wrote.
2) A description of all modules your code consists of (including module names and a one-paragraph description of what each module does), as well as *clear* instructions on how to compile and run your code. Put this in a plain text file, called `description.txt`.

A description of the data structures you chose to use and a *clear* explanation of the algorithms for each of the three required operations (above). Put this in a plain text file, called `algorithms.txt (this part should also include` the explanation of how Kruskal's algorithm is adjusted to support *Remove_expensive_ST_edges().*

3) Your brief argument why the time and space complexities of your algorithms are what the specification asked for (and not worse). Put this in a plain text file, called `complexities.txt`.
4) Answering the dry problems 1.a and 1.b and put them into the file, `analysis.txt`.

5) A brief description of who (of the two partners in the team) did what part of the assignment: the different parts of the code, the complexity arguments, the analysis questions, and anything else you did. Be sure to include your names and emails here. You are encouraged to work on all parts together, and if so, indicate that this is indeed the case. If, however, you decided to split the work, we also need to know that, as well as how you did it. (Note that both partners are responsible for all parts of the assignments regardless of whether and how you may decide to split the work among yourselves.) Put this description in a plain text file, called `contributions.txt`.

In addition, *print* and bring to class on the due date:

1) The code of the `UFKruskal` class *only*. (Please, *do not* print header files, packages, or test modules/classes!)
2) The description text files 2-6 (described above). To save paper and trees, please merge them together into one file and then print.

Do not forget to write the names of both partners on the printed sheet you submit.

Printing guidelines. Double-sided printing and condensed code printing (i.e., double-sided at 2 pages of code per side) is highly encouraged! (This shows that you care to present yourselves as professionals.) Modern printers, including many of those available at UW facilities, can do that. Ask the lab person for assistance if you are unsure how to do it.

Where to submit. Instructions on where to electronically submit your assignment will appear on the course web page and will be announced on the mailing list.

Quality criteria.
For full credit, you need to meet the following quality criteria:

1) Your submission needs to contain all parts described in the previous section.
2) Your code needs to:
   - Compile and run without errors or warnings. (We will follow the instructions you will have provided us with, so those need to be accurate.)
   - Contain comments (this is really important!), making it easier for a human to understand what you have done.
3) Your argumentation about the time and space complexities needs to be sound. Brevity is encouraged but not at the expense of clarity or soundness.
4) Your printed materials need to look professional (i.e., be printed and stapled together rather than hand-written and loose).

Good Luck!