Algorithms
Computer Science 140 & Mathematics 168
Spring 2009
Homework 6
Due Tuesday, March 3

- This assignment combines 6a with 6b.
- On this problem set, you are welcome to refer to anything that we covered in class by simply saying "in class we stated (or showed) that blah, blah, blah." This can save you some time and effort!

1. **[35 Points] Lazy Binomial Heaps!**

   Recall that a priority queue is an abstract data type which supports operations INSERT, FIND-MIN, and DELETE-MIN. The operation INSERT inserts an integer into the set, FIND-MIN returns the smallest integer in the set, and DELETE-MIN removes the smallest integer from the set. The classical data structure for this abstract data type is a heap. Recall that a heap is a height-balanced binary tree in which each node is strictly smaller than its descendants. (**Throughout this problem, we assume for simplicity that there are no duplicate values being stored.**) Recall that in a heap FIND-MIN takes $O(1)$ time since the minimum element is at the root, while INSERT and DELETE-MIN take $O(\log n)$ time.

   In some cases it is convenient to be able to take the union of two priority queues. Heaps are not a great data structure in this case, since taking two heaps of total size n and merging them into one heap takes $\Theta(n)$ time. In this problem we will investigate a very elegant data structure which supports operations INSERT, FIND-MIN, and UNION in $O(1)$ *actual time* (and *amortized time*) and DELETE-MIN in $O(\log n)$ *amortized time*. Thus, any sequence of n operations of which $n-k$ are INSERT, FIND-MIN, and UNION operations and k are DELETE-MIN operations will take a total of $O(n-k+k\log n)$ time. Notice that this ranges between $O(n)$ and $O(n\log n)$ depending on k.

   The data structure proposed here is called a *lazy binomial heap*. It is one of many excellent data structures with "lazy" properties. Here's

how it works. First, we define a *binomial tree* recursively as follows: A single vertex is a binomial tree of type 0. A binomial tree of type i is formed by taking two binomial trees of type $i-1$ and making one of the the two roots point to the other (the new root for the new binomial tree). For example, Figure 1 shows binomial trees of types 0, 1, 2, and 3, respectively:

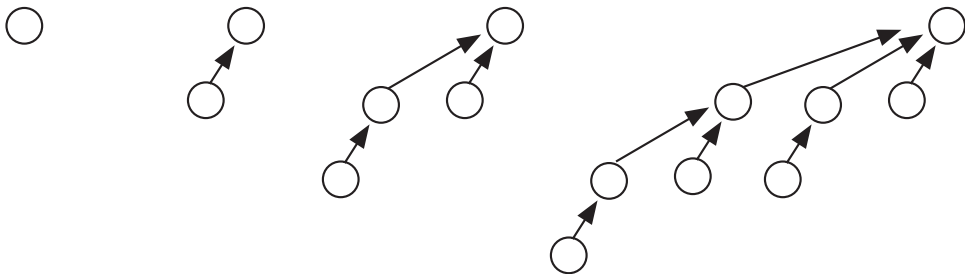

Figure 1: Four binomial trees. From left to right, binomial trees of types 0, 1, 2, and 3.

If you look at the number of nodes at each level of a binomial tree, you'll see where the name comes from! (Recall that a "binomial number" or "binomial coefficient" is a number which can be expressed as $\binom{n}{k}$.) Notice also that a binomial tree of type r has exactly 2^r nodes in it. In addition, the root of the binomial tree has r children. We assume that the root of each binomial tree stores the number of its children and the total number of nodes in the tree.

A lazy binomial heap is just a linked list (if you prefer to make it a doubly-linked list, that's OK too) in which each of the nodes is a binomial tree and the nodes in each binomial tree satisfy the heap property (that is, each node stores a number which is strictly smaller than its descendants). In addition, the lazy binomial heap maintains a pointer to the minimum element. Notice that this element is always a root node of one of the binomial trees in the linked list. We also keep track of the size of the lazy binomial heap (the total number of nodes it contains). Figure 2 shows an example of a lazy binomial heap. This heap happens to comprise 4 binomial trees.

Here is how each of the operations works:

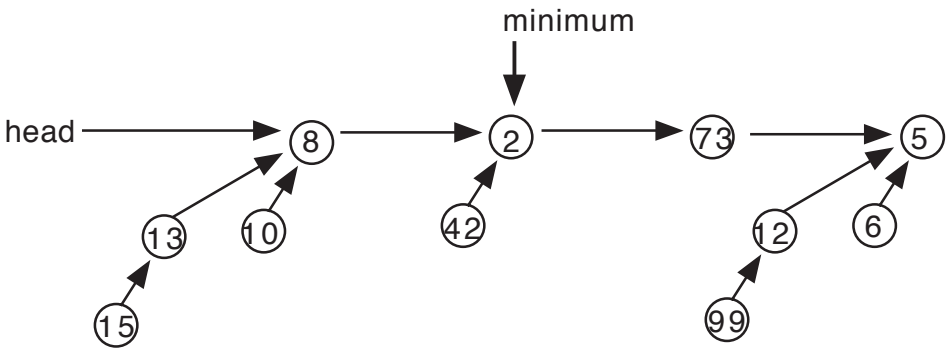

Figure 2: A binomial heap containing 4 binomial trees.

NEW-HEAP: This operation takes no arguments and returns a pointer to a new heap (just a null pointer). The size of the heap is set to 0.

INSERT: This operation takes the pointer to a particular lazy binomial heap and an integer value to insert in that heap. It constructs a new binomial tree of type 0 which, recall, is just a single node. This node stores the given integer value. The new binomial tree is inserted in some arbitrary place in the lazy binomial heap (either at the beginning or the end, for example), the pointer to the minimum element is updated if necessary, and the size of the binomial heap is incremented by 1.

FIND-MIN: This operation takes the name of the particular lazy binomial heap and returns the smallest element in that heap. Since we are keeping a pointer to that element, this is fun and easy!

UNION: This operation takes the names of two lazy binomial heaps and merges them into one lazy binomial heap. To do so, it appends one of the lazy binomial heaps onto the end of the other, updates the minimum pointer, and computes the size of the new structure by adding the sizes of the two original binomial heaps.

DELETE-MIN: This operation deletes the minimum element from the specified lazy binomial heap as follows:

(a) Use the pointer to the minimum element to find the minimum. Remove that element and return it. Reduce the size of the

binomial heap by 1.

(b) The removal of that element may cause its binomial tree to break up into a number of smaller binomial trees (if the minimum element was in a binomial tree of type $i > 0$). Add those trees to the lazy binomial heap. That is, each of those new trees is added to the list of roots which make up the lazy binomial tree.

(c) Clean up the lazy binomial heap by repeatedly linking two binomial trees of the same type into a larger binomial tree until all the binomial trees in the heap are of distinct types (recall the definition of "type" above). To facilitate this cleanup phase, we begin by allocating an array A with indices 0 through $\log s$ where s is the size of the lazy binomial heap. We initialize this array to be empty. Then, we traverse the linked list of binomial trees one-by-one. For each binomial tree, if the type of the tree is r we place the tree in location r of array A. If that location already contains a previously inserted binomial tree of type r, those two binomial trees are merged into a new binomial tree of type $r + 1$. (The binomial tree with the smaller root value becomes the root of the new binomial tree.) This new tree is then placed in location $r + 1$ of the array. If there is already a binomial tree of in-degree $r + 1$ in that location, the merging process may continue.

(d) After all of the binomial trees have been inserted into the array, the array contains at most one binomial tree of each type. These binomial trees are then threaded together via a new linked list which comprises the new clean lazy binomial heap.

(e) Finally, this new lazy binomial heap is traversed to set the new pointer to the minimum element.

This problem is broken up into a number of smaller parts:

(a) Briefly explain why each of the operations NEW-HEAP, INSERT, FIND-MIN, and UNION take $O(1)$ actual time.

(b) Briefly explain what is "binomial" about a binomial tree.

(c) Briefly explain why a binomial tree of rank r has 2^r nodes.

(d) In the DELETE-MIN operation, after the minimum element is removed, its binomial tree may become fragmented. Explain briefly why all of the fragments have sizes which are powers of 2 and can be considered binomial trees themselves.

(e) Now, let ℓ denote the total number of binomial trees in the linked list immediately after the minimum element was removed and the resulting fragments were added to the list. Show that the entire cleanup phase described in step (c) of the DELETE-MIN operation can be performed in $O(\ell)$ time. Notice that some binomial trees may merge several times while others might just get plopped in a location of the array and stay there. Therefore, your argument here will require an amortized analysis. Prove the $O(\ell)$ time **three different ways**: using aggregate analysis, the accounting method, and the potential method.

(f) Explain briefly why the resulting "cleaned up" lazy binomial heap contains at most $\log n$ binomial trees in its linked list (where n is the total number of operations performed and thus an upper bound on the number of nodes among all the heaps.)

(g) Now, define an appropriate potential function and show that the amortized cost of the entire DELETE-MIN operation is $O(\log n)$.

(h) Next, show that under this potential function, the amortized costs of all of the other operations are still $O(1)$.

(i) The famous Shmorbodian theoretical computer scientist, Professor Ima Lazeebeauns, is unhappy about the fact that Ran has asked you to show that the *amortized cost* of the operations (other than DELETE-MIN) is $O(1)$ when we already know that the *actual cost* of these operations is $O(1)$. Was there any good reason to look at the amortized cost of these operations or was this just an intellectual exercise? Explain.

(j) Conclude that this is a very slick data structure and an elegant piece of analysis. Eat chocolate and/or take a break to celebrate.

2. **[20 Points] Cycles in Directed Graphs.** The problem of detecting the existence of a cycle in a directed graph is a recurring problem in computer science.

(a) Describe a simple algorithm for determining if a directed graph contains a cycle. The graph is assumed to be represented with an adjacency list. Explain why your algorithm works and carefully derive its running time. (Your algorithm must be very fast because you'll soon be asked to prove that it's running time is asymptotically optimal.)

(b) Prove that the worst-case running time of your algorithm is asymptotically optimal. That is, show that there cannot exist an algorithm with better worst-case running time (up to constant factors).

3. **[25 Points] Graph Diameter!** In this problem we will consider connected undirected acyclic graphs (graphs that have no cycles). The graphs are assumed to be represented with adjacency lists. The distance between two vertices in such a graph is just the minimum number of edges on a path between the vertices. The *diameter* of the graph is the maximum distance between all pairs of vertices. Make sure that you understand this definition well before proceeding!

   (a) Give a precise description of an algorithm for computing the diameter of an undirected acyclic graph. The description can be in English or high-level pseudo-code. (Your algorithm must be very fast because you'll soon be asked to prove that it's running time is asymptotically optimal.)

   (b) Explain briefly but convincingly why your algorithm is correct.

   (c) Prove that the worst-case running time of your algorithm is asymptotically optimal.

4. **[20 Points] Napquest Revisited!** A few weeks ago, you devised an algorithm for Napquest that finds the shortest path from one vertex to another in a very special kind of graph.

   Napquest is ready for more! Now they want you to design an algorithm that finds the length of the shortest path between any pair of vertices in an arbitrary directed graph in which every edge has a positive weight (distance) associated with it. By "shortest path" we do not mean the number of edges, we mean the total sum of the weights on the edges in the path!

The graph is represented by an adjacency matrix. However, instead of having 0's and 1's as entries in the matrix, the entries of the matrix specify the weight on a given edge. In particular, if there is a directed edge from vertex i to vertex j with weight d, then the entry in row i and column j of the matrix will be d. If there is no edge from vertex i to vertex j, we put the value ∞ in that entry of the matrix. (You may assume that our programming language supports arithmetic with ∞ so, for example, $\infty + 42 = \infty$ and ∞ is greater than any integer when a comparison is performed.)

So, the program has access to a $n \times n$ adjacency matrix, A, indicating the directed edges and their weights. Assume that matrix A is global, so we don't have to pass it in explicitly to our COMPUTE-DISTANCES function. Instead, COMPUTE-DISTANCES takes just 3 arguments as input: The start vertex, s, the destination vertex, d, and the maximum number of permitted edges p, in the path. The function then returns the length of the shortest path (in terms of total weight) that uses p or fewer edges. (What's with the "p or fewer edges business?" Read on!)

(a) Argue briefly that if there are n vertices in the graph, then the absolute shortest path between two vertices involves at most $n-1$ edges. Remember, all of the edge weights are positive.

(b) Now, give a recursive algorithm for COMPUTE-DISTANCES(s, d, p) where s is the start vertex, d is the destination vertex, and p is the maximum number of edges permitted. Note that when we invoked this algorithm, we will use $p = n - 1$, but your algorithm should work for any p. *Hint: Use it or lose it!*

(c) Now describe the dynamic programming formulation for making this recursive algorithm efficient. In particular, describe what the table looks like, what the cells represent, and the order in which you fill them in. Notice that your table should actually allow you to see the shortest path between every pair of vertices in the graph.

(d) What is the running time of your algorithm? Explain how you derived this.

(e) Napquest would actually like to know the distances between every pair of the n vertices. Notice that you've already done this! There's nothing to write here.