Algorithms
Computer Science 140 & Mathematics 168
Spring 2012
Homework 5b
Due Tuesday, February 21

The first exam will be on Tuesday, February 21. Part 1 of the exam will be in-class and will comprise short questions that survey the topics and ideas that we've covered so far: Divide-and-conquer, dynamic programming, greed, and amortization. You may prepare an 8.5×11 sheet with notes on both sides to bring to the in-class part of the exam. You will then take Part 2 of the exam home with you. This part of the exam will comprise two homework-style problems. You can take that part of the exam in a 3-hour block of your choosing and those problems will be due at the beginning of class on Thursday, February 23. For the take-home part of the exam, you may use your 8.5×11 sheet, your own lecture notes, and your homework solutions and our sample solutions, but no other resources.

There is one problem on this assignment. It is a bit verbose, simply because I describe a new data structure. Don't be alarmed by the five pages! This problem is also good practice for the exam because it flexes your amortization muscles.

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 from class 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. (We won't consider the one other operation that we saw in class that changes the value of an element in the heap.)

   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 new heap takes $\Omega(n)$ time. Just to appeal to your intuition, if the two heaps each had size about $n/2$ and we unioned them by simply inserting the elements of one heap into the other, that would take something like $\Theta(n \log n)$ time because each insert of an element into the other heap would take $\Theta(\log n)$ time. There are slightly faster ways to merge two heaps, but none that is asymptotically faster than n.

   In this problem we will investigate a very elegant LAZY 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*. 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 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:

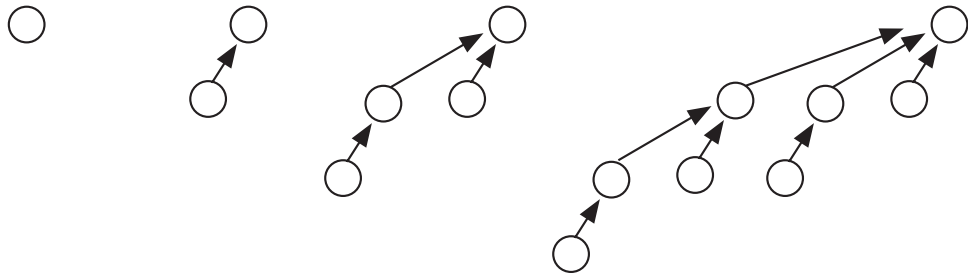

Figure 1: Four binomial trees. From left to right, binomial trees of types 0, 1, 2, and 3.

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:

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

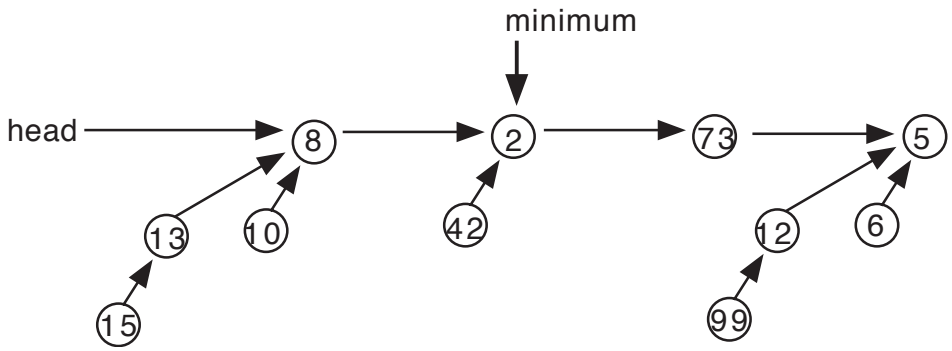

Figure 2: A binomial heap containing 4 binomial trees.

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.

Our data structure D is a collection of zero or more lazy binomial heaps. At first there are zero lazy binomial heaps. Over time, there may be multiple lazy binomial heaps created. As a result of unions, the number of lazy binomial heaps might be reduced. **However, the data structure should be viewed as the totality of all of the lazy binomial heaps that we are maintaining.**

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 why a binomial tree of rank r has 2^r nodes.

(c) 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.

(d) 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 using any one of the amortization methods that we've seen in class.

(e) 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.)

(f) Now, define an appropriate potential function and show that the amortized cost of the entire DELETE-MIN operation is $O(\log n)$. Keep in mind here that our data structure is a collection of multiple lazy binomial heaps. Any potential function that you define for this data structure at this point must be defined for

the entire data structure rather than for the individual heaps. (Our potential function method was not designed for use with multiple potential functions!)

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

(h) The famous Shmorbodian theoretical computer scientist, Professor Toomah Chwork, 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.

(i) Conclude that this is a very slick data structure and an elegant piece of analysis. Eat chocolate to celebrate. (If you are allergic to chocolate, a proxy treat is permitted for full credit. If you simply don't like chocolate, your answer is wrong and you may not get credit for this part.)