Algorithms
Computer Science 140 & Mathematics 168
Spring 2007
Homework 5b
Due Tuesday, February 20

- The material on Union-Find is in Section 5.1.4 of the book.

1. **[20 Points] Analyzing Build-Heap!**

   Recall from class that the Heapsort algorithm sorts an array (indexed from 1 to n) as follows: Consider the array as representing a binary tree where the children of the node at index i are found at indices $2i$ and $2i + 1$ (unless these indices are out of range).

   We begin by turning this array into a heap: a binary tree in which each node is less than or equal to its descendants. To do so, we invoke a process called BUILD-HEAP that starts at node $i = \lfloor \frac{n}{2} \rfloor$ and works down to 1. For each i in this range, BUILD-HEAP ensures that the tree rooted at i has the property that it is a heap itself! To this end, node i is compared with both of its children. If it is less than or equal to both of its children, we are done with node i and we continue on to node $i - 1$. Otherwise, we swap the value at node i with the smaller of its two children and then repeatedly "percolate" this value down until it reaches its correct location in the heap rooted at node i.

   In class, we argued that since the heap has height $O(\log n)$ and there are approximately $n/2$ nodes that are examined in the BUILD-HEAP process, the total running time of BUILD-HEAP is $O(n \log n)$.

   In this problem you will prove that the actual total running time of BUILD-HEAP is $O(n)$ by using an amortization argument. For simplicity, assume that $n = 2^k - 1$ for some $k > 1$. That is, the heap is a complete binary tree in which all of the leaves are at the same level.

   In this amortization scheme, each of the n nodes receives 3 rubles at the outset. We claim that this is enough to pay for all of the work incurred by BUILD-HEAP. To help see why, notice that we start the BUILD-HEAP process at index $\lfloor \frac{n}{2} \rfloor$, a node with two leaf children. *Let's call this node "Joe". This is not a proof yet, so giving names to nodes is fun and easy!* Joe does two comparisons and perhaps a swap, for

a total of 3 operations. (We are assuming that a comparison counts as one operation and a swap counts as one operation.) However, the tree rooted at Joe has 9 rubles at its disposal: 3 for each of the two leaves and 3 for Joe. We spend at most 3 rubles, leaving 6 rubles in Joe's tree for future work. Next, Joe's sibling, Josette, does the same thing, leaving her tree with 6 rubles for later work. Eventually, we get to Joe and Josette's parent, Pat. Pat has 3 rubles and each of its children, Joe and Josette, have 6 rubles remaining in their own trees. This is a total of 15 rubles. How much work does Pat incur? There are two comparisons (compare Pat to Joe and Josette) and perhaps a swap. Then, in the worst case that Pat is swapped, there are two more comparisons and perhaps a swap for a total of up to 6 operations. This uses 6 of the 15 rubles in this tree, leaving 9 rubles in Pat's tree for later work.

Based on the observations above, give a careful proof by induction that shows that the 3 rubles allocated at each node pays for the entire BUILD-HEAP process, resulting in $O(n)$ running time for BUILD-HEAP. *Hint: You will need to state a precise invariant that is preserved by this accounting scheme. Proving this invariant by induction will imply that the scheme pays for all of the work.*

2. [**25 Points**] **Delete-Less Dictionaries!** Recall that a dictionary is an ADT that supports operations INSERT, FIND, and DELETE. Usually, self-balancing search trees are used to implement dictionaries because they support all of these operations in $O(\log n)$ worst-case time. However, imagine that we don't care about the cost of any *one* operation, we only care that a sequence of n operations will cost no more than a total of $O(n \log n)$. That is, we only care that our data structure have *amortized* $O(\log n)$ running time. In this case, there are a number of clever data structures that we can use in lieu of self-balancing trees. This problem explores one such data structure. For simplicity though, we will assume that only INSERT and FIND are being supported.

   Our data structure works like this: Our data will be distributed over multiple arrays. Each array has length which is a power of 2, all the arrays have different lengths, all the arrays are completely full, and all the arrays are kept sorted. For example, if there are 9 elements in the dictionary, then since the binary representation of 9 is $2^3 + 2^0$, 8 of the

elements will be in a sorted array of length 8 and 1 of the elements will be in an array of length 1. If a new element is added now, we'll have 10 elements. Thus, we'll keep 8 of the elements in their own array but place the new element with the element previously in the array of length 1 into a new sorted array of length 2. There is no particular relationship between elements in different arrays.

(a) Carefully describe how the FIND operation works for this data structure and analyze its worst-case running time. *Hint: It will be much faster than* $\Theta(n)$ *but probably not quite as fast at* $\Theta(\log n)$.

(b) Carefully describe how to insert a new element in this array so that the amortized time of an INSERT operation is $O(\log n)$. (That is, the total cost of n INSERT operations is $O(n \log n)$. The cost of FIND operations may, in fact, be slightly higher.) **Give three different amortization arguments using the three different amortization techniques we've seen in class.**

3. **[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*. 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:

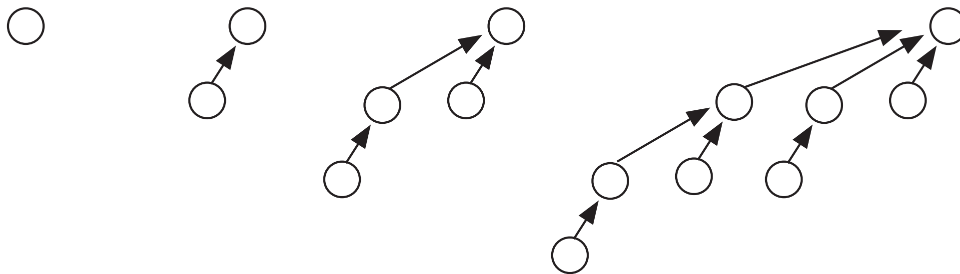

Figure 1: Four binomial trees. From left to right, binomial trees of types 0, 1, 2, and 3.

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.

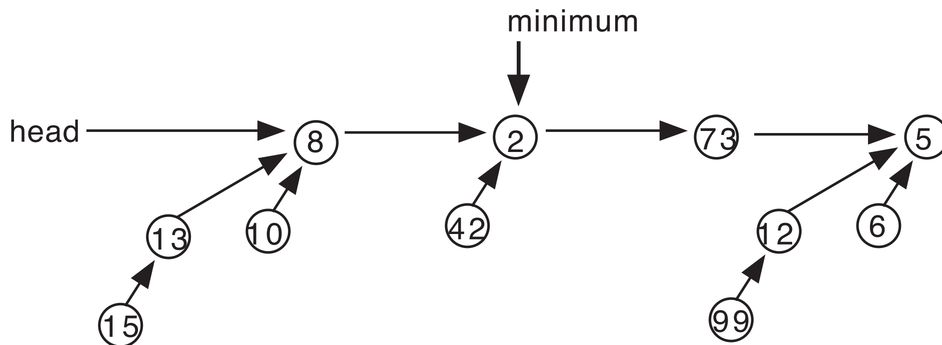

Figure 2: A binomial heap containing 4 binomial trees.

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 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 type 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.

4. [**15 Points OPTIONAL Bonus Problem**] **Union-Find with Partial Path Compression!** To do this problem, you will need to read Section 5.1.4 to understand how the $O(\log^* n)$ amortized cost is derived for the union-find data structure. The proof is not very hard and the result is shockingly amazing!

   Recall the union-find data structure using union-by-rank and path compression. One disadvantage of path compression is that it is a two-phase process: First we must "climb" the path to find the root. Then, we climb the path a second time and set the parent of each node on that path to be the root.

   The famous Shmorbodian theoretical computer scientist, Professor Y. Woerksohaard, proposes the following simpler approach called *partial path compression*: As we "climb" a path from a node to the root of its tree, each node on the path has its parent pointer changed to point to its grandparent. This process can be done in just one pass as we climb up the path. Professor Woerksohaard claims that this approach still allows us to perform any sequence of n MAKESET, UNION, and FIND operations in time $O(n \log^* n)$ time. Is this true or false? If true, give a short but precise explanation (you may cite any part of the proof from the book without replicating it). If false, give a short but precise counter-example.