

Homework 7, CS420, Fall 2011

Due 11/11 at the beginning of class

1. Complete the proof of the $O(\log n)$ amortized bound of a splay as follows. Consider the second rotation that decreases the depth of x by two, for the case where y is a left child of z and x is a right child of y . (The analysis of the left-right mirror image of this case is identical, so we don't need to analyze it.)
 - (a) Show that if x 's rank increases by k during the operation, you can pay for the rotation and maintain the credit invariant (the savings plan) using $3k$ credits out-of-pocket. The credit invariant requires that each node have a number of credits equal to its rank.
 - (b) Show that if x 's rank doesn't increase during the operation, the rank of y or z decreases, freeing a credit to pay for the rotation.
2. Data structures books often compare the relative advantages and disadvantages of arrays and linked lists. Let us create an abstract data type `Darray` that has the best of both worlds, except for the expense of requiring $O(\log n)$ amortized time for an access instead of the $O(1)$ required by arrays. We want the following:

- `int retrieve(int i)` returns the value at position i of `this`.
- `void store(int i, int j)` replaces the value at position i with j
- `int remove(int i)` removes and returns the value at position i , decrementing the position numbers of all elements previously at positions greater than i
- `void add(int i, int j)` inserts j to position i , incrementing the position numbers of all elements previously at positions i or greater
- `Darray split(int i)` splits off the elements at positions $i + 1$ through the end, returning them as a new `Darray` object with their positions numbers reduced by $i + 1$.
- `Darray concatenate(Darray B)` concatenates `B` to the end of `this`, destroying `B`.
- `print()` Prints out the contents of the `Darray` object in ascending order of position number.

All of these operations must take $O(\log n)$ amortized time, except for `print()`, which can be carried out in $O(n)$ time. The key to this is making a `Darray` object point to the root of a splay tree, and maintaining `size` labels on all nodes on the splay tree. Maintain the invariant that an inorder traversal gives the items in ascending order of position number. Since the root of the tree changes with each splay, the root pointer must be updated after each splay.

- (a) If you can show how to update the `size` labels in $O(1)$ time after each rotation, it will follow that a splay operation still takes $O(\log n)$ amortized time. Draw pictures of each of the rotations. Let a, b, c , and d be the size labels of subtrees A, B, C, D . Label each of x, y, z with expressions for their new sizes. Be sure to write expressions that take $O(1)$ time to evaluate.

- (b) Write pseudocode for a recursive method that takes the root of a subtree and finds the i^{th} node of an inorder traversal of the subtree. The strategy of the method should resemble that of `Select`. Use the `size` labels l and r of the left and right children. Make sure that if the length of the search path is k , your method takes $O(k)$ time.
- (c) Why will this be $O(\log n)$ amortized time if we splay the node at position i once we've found it?
- (d) In English, describe how to perform each of the operations other than `print()` in $O(\log n)$ amortized time. (The `print()` operation can be carried out in $O(n)$ time by printing the node labels in an inorder traversal.) I will do the first one for you:

For `retrieve(i)`, find and splay the node at position i , and return the value in it.

Notice that I didn't explain why it took $O(\log n)$ amortized time, since this is immediate from the $O(\log n)$ amortized time bound for a splay.

3. Do problem 19-2 from the binomial-heap handout. Here is what they are hinting at: you need a way to figure out quickly which sets u and v belong to. Here's a nice trick: have each set V_i be labeled the number of vertices in it, and have each vertex in it labeled with i . When you want to merge V_i and V_j , if $|V_i| < |V_j|$ swap their roles.

That way, the vertices in the smaller of the two sets get relabeled. Each time a vertex is relabeled, it finds itself in a set that's at least twice as large as the one it was in before the merge that caused it to get relabeled. It never finds itself in a set larger than n , so the total number of times a vertex can be relabeled is $O(\log n)$. The total time spent relabeling the n vertices is therefore $O(n \log n)$.

Make sure you understand this trick. Then derive the time bound for the algorithm. You don't need to repeat the explanation of the trick; you can just reference the result.

4. Not just any greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities.
 - (a) Draw an example that shows that greedily selecting the activity of least duration doesn't work.
 - (b) Draw an example that shows that greedily selecting the activity that overlaps with the fewest other remaining activities doesn't work.

5. Here is an algorithm for minimum spanning tree. *While there exists a cycle in the graph, find a largest-weight edge on the cycle and remove it from the graph.*

The algorithm terminates when the graph is a tree. The claim is that it's a minimum spanning tree. There is a way to get this to run in $O(m \log n)$ time using a data structure that has splay trees as an important component. (If you are curious about this, sign up for my CS520 class in the spring.)

The main ingredient of the proof is the following claim:

- Let C be a cycle of G and let e be a maximum-weight edge of G . There exists a spanning tree of G that does not contain e .

Come up with a greedy proof of this. *Let T be a minimum spanning tree of G . If T doesn't contain e then it illustrates the claim. So consider the case where T does contain e , and explain why there exists another minimum spanning tree T' that does not contain e .*