Homework 3

ECE 345 Algorithms and Data Structures
Fall Semester, 2017

Due: Friday October 27, 2017, 2PM

- All page numbers are from 2009 edition of Cormen, Leiserson, Rivest and Stein.
- For each algorithm you asked to design you should give a detailed *description* of the idea, proof of algorithm correctness, termination, analysis of time and space complexity. If not, your answer will be incomplete and you will miss credit. You are allowed to refer to pages in the textbook.
- Do not write C code! When asked to describe an algorithm give analytical pseudocode.
- Staple your homework properly. Use a stapler; do not use glue or other weird material to put it together. If you are missing pages, we are not responsible for it but you are!
- Write *clearly*, if we cannot understand what you write you may not get credit for the question. Be as formal as possible in your answers. Don't forget to include your name(s) and student number(s) on the front page!

---

1. **[Search Trees, 15 points]**

   Let X be a set of n items, each with a *key* and a *priority*. For sake of simplicity, assume that no two keys or priorities are identical. A **PK-Tree** for X is a rooted binary search tree whose nodes are the items in X such that:

   (i) $key[x] < key[y]$ if x is a left descendant of y or y is a right descendant of x, and

   (ii) $priority[x] < priority[y]$ if x is a descendant of y.

   (a) Given a key value, how would you search for that value in a PK-Tree?

   (b) Give an algorithm to insert a new item into a PK-Tree.

   (c) Give an algorithm to delete an item from a PK-Tree.
   (*Hint*: Use rotations to restore the priority structure of the tree after an insertion or deletion.)

2. **[Search Trees, 20 points]**

   Suppose we have a list of n fixed length strings that are sorted lexicographically. This list is stored in a balanced binary search tree for easy access. Let k be the number of strings with prefix x. Devise an algorithm to output all strings with prefix x in $O(k + \lg(n))$ time. (*Hint*: First, find the first node in the tree with prefix x.)

   Example: A sorted list with strings of length three may look like: *ABC, ABD, BCC, BDC, CAB, CAD, DAB*. If $x = "CA"$, then your algorithm should print out CAB, CAD.

3. **[Hashing, 10 points]**

   Demonstrate the insertion of keys 7, 9, 88, 11, 25, 23, 22, 28, 14, and 21 into a doubly-hashed table where collisions are resolved with open addressing. Let the table have 11 slots, let the *primary* hash function be $h_p(key) = key \ mod \ 11$ and let the *secondary* hash function be $h_s(key) = (key * 3) \ mod \ 4$.

4. **[Greedy Algorithms, 15 points]**

   Consider an interval (job) scheduling problem with n jobs, all of which are available at the beginning. For $i = 1, \ldots, n$, job J_i has a duration d_i and an associated weight w_i. For a schedule S (ordering of the jobs such that they do not overlap and are executed without preemption), let f_i be the finish time of J_i, $j = 1, \ldots, n$. Then the total weighted waiting time of S is

   $$w(S) = \sum_{i=1}^{n} w_i f_i.$$

   We are interested in a schedule that minimizes the total weighted waiting time.

   Describe an efficient algorithm to solve this scheduling problem (prove correctness and analyze run-time).

5. **[Dynamic Programming, 20 points]**

   A pipeline is to be built in Siberia and Prof. Kostochka is to decide on the optimal location of the stations along the path that has already been decided. More precisely, the pipeline connects points x_1 and x_n and goes through locations $x_2, x_3, \ldots, x_{n-1}$. There are stations at x_1 and x_n, but part of the problem is to decide whether to built a station at point x_i, $i = 2, 3, \ldots, n-1$. If a station is built in x_i there is an associated cost b_i, and if a pipeline section is built between stations x_i and x_j—with no other station in between—there is an associated cost cost $c_{i,j}$. The total cost is the sum of the station costs plus the corresponding pipeline section costs. The goal is to decide the optimal location of stations so that the total cost is minimized. Suggest a general algorithm that Prof. Kostochka could use to make the decision (without any assumptions about the costs). As usual describe it clearly (pseudcode is not mandatory), argue that it is correct and analyze its running time. Try to make it as efficient as you can.

6. **[Skewed Heaps, 20 points]**

   In this problem you are required to develop a data structure similar to that of the leftist heap from Homework 2. In *leftist heaps*, the `Merge` operation preserved the heap ordering and the balance (leftist bias) of the underlying tree. *Skewed heaps* use the same idea for merging heap-ordered trees. `SkewHeapMerge` is performed by merging the rightmost paths of two trees without keeping any explicit balance conditions. This means that there's no need to store the rank of the nodes in the tree. This is similar to self-adjusting trees, since no information is kept or updated.

   Good performance of those data structures is guaranteed by a "rebalancing step"—like the splay in self-adjusting trees, only simpler. At each step of the merging along the rightmost paths of the two heaps, we swap *all* of the left and right children of nodes along this path, except for the last one. The modified procedure for merging two skewed heaps looks as follows:

```
function SkewedHeapMerge($h$,$h'$) : heap

    if $h$ is empty then return $h'$
    else if $h'$ is empty then return $h$

    if the root of $h'$ $\preceq$ the root of $h$ then
        exchange $h$ and $h'$ (* $h$ holds smallest root *)

    if right($h$) = nil then
        right($h$) := $h'$ (* last node, we don't swap *)
    else
        right($h$) := SkewedHeapMerge(right($h$), $h'$)
        swap left and right children of $h$

    return $h$
```

The above recursive routine can also be done iteratively. In fact, it can be done more efficiently than the leftist heap `Merge` (by a constant factor), because everything can be done in one pass, while moving down the rightmost path. In the case of leftist heaps, we go down the rightmost path, and then back up to recompute ranks. In leftist heaps, that also requires either a recursive algorithm or pointers from nodes to their parents [1].

Since there is no balance condition, there's no guarantee that these trees will have $O(\log n)$ worst-case performance for the merge (and hence all of the other operations). But they do have good *amortized performance.*

Here's the intuition for the above: In the previous merge algorithm we only had to swap children when the right one had a larger rank than the left. In this merge algorithm, we always swap children, so we might actually replace a right child of "small" rank with one of "large" rank. But the *next time* we come down this path, we will correct that error, because the right child will be swapped onto the left.

(a) Show that a `SkewedHeapMerge` of two skewed heaps uses amortized time $O(\log_2 n)$ by the use of the *accounting method.*

(b) Show that `Insert` and `DeleteMin` have the same amortized time bounds.

(*Hint*: Use weights instead of ranks. Define the *weight* of a node to be the number of nodes in the subtree rooted at that node (below that node, including the node itself). Let node x be a *heavy* node if its weight is more than half of the weight of its parent. Otherwise it is *light*

[1] Just a note on implementation here. Sometimes we may want to be able to move up a tree as easily as we move down; so every node will also include a pointer to its parent. That means that a node has a pointer to its left child, a pointer to its right child, and a pointer to its parent, increasing the amount of space needed to store trees. To decrease the *space* requirement, you can do this. Look at three nodes, a parent p and its two children r and l. Now p will have a pointer to l, but not to r; l has a pointer to r and r has a pointer back to p. So, all the left children in a tree have pointers to their right siblings (brothers) and to their own left children. All the right children have pointers to their parents and to their right children. If you draw a picture, this should make more sense. Now every node still has two pointers, and you can visit left nodes, right nodes and parent nodes easily.

one. What can we say about light and heavy nodes in any binary tree? Look at any root–leaf path in the tree. *How many* light nodes can you encounter in such path? Why? The best case is when the light nodes are also right children of their parents. For the accounting method, every time we swap, we must pay \$1. Moreover, if it happens to swap a heavy child to a right child position, then you must deposit a “penalty” amount.)