Homework 2

ECE 1762 Algorithms and Data Structures
Fall Semester, 2016

Due: October 21, 4PM in box (near SF B560)

Unless otherwise stated, all page numbers are from 2001 edition of Cormen, Leiserson, Rivest and Stein (parentheses contain page numbers from the 1990 edition). **Unless otherwise stated, for each algorithm you design you should give a detailed description of the idea, proof of correctness, termination, analysis and proof 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.** When requested, do not give C code but explain your algorithm briefly with pseudocode!

1. **[Sorting, 20 points]**. We consider n elements in an array. The goal is to sort the array in increasing order (or rather, nondecreasing order, since equal elements are possible). The basic operation is to compare the elements in two cells and to exchange them if they are out of order. An *swap sorting* algorithm works by performing a sequence of basic comparisons. The algorithm is *oblivious* if the sequence of pairs considered is independent of the input. A comparison is *primitive* if it compares 2 adjacent cells. There are $n-1$ different primitive comparisons (compare cell i with cell $i+1$, with $i = 1, \ldots, n-1$). Consider the following algorithm:

```
for j=1 to n-1
   for i equal to 1,..., n-1, taken in any arbitrary order
      perform the i-th primitive comparison,
      swapping the elements if necessary
```

   (a) Show that this algorithm sorts the input array.

   You may use the following fact called the **0-1 Principle**: *If an oblivious sorting algorithm sorts all sequences of where each element is* 0 *or* 1*, then it sorts all sequences of arbitrary values.*

   Now suppose that the comparisons are performed on pairs taken at random. More precisely: while the array is not yet sorted, choose a pair of consecutive cells at random (each of the $n-1$ is equally likely) and compare and swap if necessary.

   (b) Give an upper bound on the expected number of comparisons performed by the algorithm until they array becomes sorted. *Hint:* Use (a).

2. **[Order Statistics, 15 points]** Let $A_i[1 \ldots n], i = 1, 2$, be two arrays, each containing n numbers in sorted order. Devise an $O(\lg n)$ algorithm that computes the kth largest number of the $2n$ numbers in the union of the two arrays. Do *not* just give pseudocode—explain your algorithm and analyze its running time.

3. **[Searching, 15 points]** Consider a sorted linked list that is stored in an array. The list can be traversed in increasing order by traversing the links (there is a pointer from each cell to the next

in the list, and the head and tail of the list are known). We consider the problem of searching for an element that is in the list. Note that since the list elements do not necessarily appear in sorted order acording to the linear ordering of the cells in the array, the array indices of the cells cannot be used to perform a fast $O(\log n)$ binary search in the list.

(a) Argue precisely that a deterministic algorithm cannot do better than $\Omega(n)$ running time to search in this list.

Consider now a deterministic algorithm that starting at each of the first $\sqrt{n}$ cells searches for the query element from there (following the list pointers). We are interested in the expected running time of this algorithm over all the different $n!$ permutations of the n elements in the array.

(b) Show that the *expected* running time of this algorithm is $O(\sqrt{n})$.

This expected running time is over all possible permutations. But introducing randomization, we can achieve the same running time for a fixed input. Consider the following algorithm: get a random sample of

$\sqrt{n}$ items, determine the interval in the sample that contains the query and then follow the pointe rs. The expected length of the latter search is $O(\sqrt{n})$. In this exercise you are asked to do t he precise calculations.

Let $L = \{x_1 < x_2 < \cdots < x_n\}$ be an ordered list of n keys, and let q be a fixed query key not in L. Let $R = \{x_{i_1} < \cdots < x_{i_r}\}$ be a random sublist taken from L and let x_{i_s} be the predecessor of q in R (for convenience, let $x_0 = x_{i_0} = -\infty$). Let X be the number of keys in L between x_{i_s} and q. Compute exactly the expected value of X in the following *sampling model* (try to simplify the expression if possible):

(c) R is formed by taking each element of L into R with probability p independently, with a fixed $0 < p < 1$.

4. **[Leftist Heaps, 25 points]**. Fill in the missing details of a heap based data structure known as *leftist heaps* or *mergeable heaps*.

The mathematical objects involved are multi-sets of items of type `ItemType`. (A *multi-set* is a collection of items in which there may be multiple copies of a single item.) Every item has a key, of type `KeyType`, and these keys are linearly ordered by the relation $\preceq$. We support the following operations.

- `MakeHeap(`h`)` returns a new empty heap.
- `Insert(`x`,`h`)` inserts the item `x` into the heap `h`.
- `FindMin (`h`)` returns the item in heap `h` with the $\preceq$-smallest key.
- `DeleteMin (`h`)` is like `FindMin`, but also deletes this item from the heap.
- `Merge(`h_1`,`h_2`)` returns a single heap containing all of the elements of heaps h_1 and h_2.

Each heap is a binary tree. The nodes of this tree are items of type `ItemType`. For any item x in such a tree, `left(`x`)` denotes its left child, `right(`x`)` its right child, and `key(`x`)` its key. In addition, we define the `rank` of a node of a tree to be the length of the shortest path from that node to a leaf. Equivalently, we define `rank` recursively as follows:

- If the node x is a leaf then $\texttt{rank}(x) = 0$.
- If the node x is not a leaf, then

$$\texttt{rank}(x) = 1 + \min\{\texttt{rank}(\texttt{left}(x)), \texttt{rank}(\texttt{right}(x))\}$$

This is useful in describing and maintaining the *balance* of leftist heaps.

We maintain two properties of these trees.

Order. The trees are *partially ordered* or *heap-ordered*. Recall this means that for every node x, $\texttt{key}(x) \preceq \texttt{key}(\texttt{left}(x))$ and $\texttt{key}(x) \preceq \texttt{key}(\texttt{right}(x))$

Balance. The trees are *leftist*. This means that for every node x, the shortest path from x to a leaf is the rightmost path (the path you get by following x, $\texttt{right}(x)$, $\texttt{right}(\texttt{right}(x))$, etc. to a leaf. This "leftist" bias can also be expressed in terms of the rank of a node. Now, for every node in a leftist tree either (1) the left and right children have the same rank, or (2) the right child has the smaller rank. In other words, $\texttt{rank}(x) = 1 + \texttt{rank}(\texttt{right}(x))$, for every node x.

We also assume that we have stored in some field of each node its current rank. We can refer to this field by writing $\texttt{rank}(x)$ for any item x in the heap.

The easiest operations to implement are `MakeHeap` and `FindMin`. `MakeHeap` requires only the construction of an empty tree. To do a `FindMin`, we just return the item at the root of the tree. This works because the trees are partially ordered, so the operation is essentially no different than a `FindMin` on the `Heaps` presented in class.

The most interesting operation is the `Merge`. Once the `Merge` is implemented, we can use it to define `Insert` and `DeleteMin` in a natural way. We merge two leftist heaps by first merging their rightmost paths. The rightmost path of a tree h is the path we follow when visiting the nodes h, right(h), right(right(h)), etc. Remember that *every path* in a partially ordered tree is sorted by key. So we can use the familiar algorithm for merging sorted lists to merge these paths as the MERGE procedure described in the textbook.

Call these two heaps we want to merge h_1 and h_2. First we compare the first elements of h_1 and h_2 (their roots). The first element (root) of the merged path is the least of these—that is, the one with the smallest `key`-value. Remove this element from the appropriate rightmost path (so we remove the root and its left subtree from the appropriate h_i), and then recursively merge the resulting rightmost paths. The left children of these nodes are unaltered; only the right children of nodes on the rightmost path are modified. Now the merged list is just the smallest element followed by the recursively merged path. For example, if $\texttt{key}(h_1) \preceq \texttt{key}(h_2)$ then the first element of the merged path is h_1 (the root of tree h_1) and the rest of the merged path is gotten by recursively merging the rightmost paths $\texttt{right}(h_1)$ and h_2. Merging of right paths in the manner described above guarantees that the resulting tree is partially ordered.

Unfortunately, after merging two trees, the resulting data structure may no longer be a leftist heap because the balance invariant may be violated. The balance invariant is the one which guarantees an expected $O(\log n)$ time for both `DeleteMin` and `Insert` operations, therefore we must rearrange the new tree in a way such that the resulting data structure is a leftist heap.

We recompute the ranks along the rightmost path of this new tree. We start at the bottom of the tree, let it be x. x has no right child, its `rank` should be 0. Let x `:= parent(`x`)` and check $\texttt{left}(x)$ and $\texttt{right}(x)$. If the `rank` of $\texttt{lef}(x)$ is smaller than the `rank` of $\texttt{right}(x)$ we swap left and right subtrees to guarantee that the child with the smallest rank is always to the right. Set the `rank` of x

to 1+`rank(right(`x`))`. We recursively continue rearranging the children/ranks of the nodes along the righmost path of the data structure until we reach the root of the tree.

(a) Show that the rank of the root is $O(\log n)$ (this is equivelant to saying that the length of the rightmost path (from the root) is $O(\log n)$).

(b) Prove that merging two partially ordered trees h_1 and h_2 by merging their rightmost paths takes $O(\log n)$ time and that it yields a partially ordered tree (that is, the order invariant is maintained).

(c) Show that when the rightmost paths of two trees are merged, the rank of a node x might change if and only if x is on the rightmost path.

(d) Prove that `Merge` of *leftist heaps* $h1$, $h2$ with the addition of the above `rank` update step, results to a new valid $leftistheap$ h (that is, prove that h maintains both invariants). Analyze its overall running time.

(e) Show how to implement `DeleteMin` and `Insert` for leftist heaps such that each of them runs in $O(logn)$ time. Explain your algorithms, analyze their correctness and asymptotic running times.