Homework 2

ECE 345 Algorithms and Data Structures
Fall Semester, 2017

Due: Thursday, October 12 2017, 3PM

- 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, 15 points]**

   Suppose you are given a set S of n integers. You are also given a black box that returns three elements $a, b, c \in S$ such that $a + b + c = 0$ if any such elements exist. Devise an algorithm that uses only one call to the black box to determine if S contains any three integers a, b, c such that $a + b = c$.

   (*Hint*: Construct a set S' so that the following holds: there exist $a', b', c' \in S'$ such that $a' + b' + c' = 0$ if and only if there exist $a, b, c \in S$ such that $a + b = c$.)

2. **[Search, 15 points]**

   You are given a sorted array of distinct integers $a[0], \ldots, a[n-1]$, positive and negative, and you want to find out whether there is an index i such that $a[i] = i$. Devise an algorithm for this problem that runs in time $O(\log n)$.

3. **[Sorting, 5+5+5 points]**

   Suppose we are give two sorted n-element sequences A and B that may contain duplicates.

   (a) Describe an $O(n)$-time method for computing a sequence representing the set intersection $A \cap B$ (with no duplicates).

   (b) Describe an $O(n)$-time method for computing a sequence representing the set union: $A \cup B$ (with no duplicates).

(c) Describe an $O(n)$-time method for computing a sequence representing the set difference $A \setminus B$ (elements that are in A, but not in B with no duplicates).

4. **[Sorting, 10+10 points]**

   Consider the **sort** algorithm below, which takes as input an unsorted array A of n integers.

   (a) Determine the runtime of **sort**, assuming the **length** operation runs in $O(1)$ time and **maxindex**(A, i) runs in $O(i)$ time.

   (b) Prove the algorithm is correct using induction.
   Hint: Use the following induction hypothesis: after the iteration in which $i = k$, $A[k...(n-1)]$ contains the $n-k$ largest elements of A sorted in increasing order. Use $i = (n-1)$ for the basis step and work backwards (*i.e.,* show that if the hypothesis holds for k, this implies it holds for $(k-1)$, etc.).

Algorithm sort(A)

```
1: $n$ = length($A$)              # number of elements in $A$
2: for $i = (n-1)$ downto 1 do
3:     $j$ = maxindex($A$, $i$)     # index of the largest element in $A[0..i]$ (including index $i$)
4:     $tmp = A[j]$
5:     $A[j] = A[i]$
6:     $A[i] = tmp$
7: end for
```

5. **[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 `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, `key(`x`)` $\preceq$ `key(left(`x`))` and `key(`x`)` $\preceq$ `key(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, `right(`x`)`, `right(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 `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 `key(`h_1`)` $\preceq$ `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 `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 `left(`x`)` and `right(`x`)`. If the `rank` of `left(`x`)` is smaller than the `rank` of `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 rightmost 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.