CPS 130 Homework 10 - Solutions

1. (CLRS 12.2-5) Show that if a node in a binary search tree has two children, then its successor has no left child and its predecessor has no right child.

   Solution: Denote for a node x having two children its predecessor by p and its successor by s. First we show by contradiction that the sucessor of x has no left child. Suppose s has a left child. Then the key of s is greater than that of $left[s]$. The key of s is also larger than the key of x, and since s has a left child the key of $left[s]$ is larger than that of x. Thus

   $$key[s] \geq key[left[s]] \geq key[x],$$

   which is a contradiction, since s is the successor of x. Hence the sucessor of x has no left child.

   Similarly, we show by contradiction that the predecessor of x has no right child. Suppose p has a right child. Then the key of p is less than that of $right[p]$. The key of p is also less than the key of x, and since p has a right child the key of $right[p]$ is less than that of x. Thus

   $$key[p] \leq key[right[p]] \leq key[x],$$

   which is a contradiction, since p is the predecessor of x. Hence the predecessor of x has no right child.

2. (CLRS 12-2 - Radix trees) Show how to use a radix tree to sort S lexicographically in $O(n)$ time.

   Solution: Insert all the strings into a radix tree. This is done in $\Theta(n)$ time since the cost of inserting a string is proportional to the length of the string and the lengths of all strings sum to n. Using a preorder traversal of the tree we can print the strings in sorted order in $\Theta(n)$ time since the number of nodes in the tree is at most $n + 1$ (a string of length i corresponds to i nodes plus the root, and lengths of all strings sum to n).

3. (CLRS 9-1) Given a set of n numbers, we wish to find the i largest in sorted order using a comparison-based algorithm. Find the algorithm that implements each of the following methods with the best asymptotic worst-case running time, and analyze the running times of the algorithms on terms of n and i.

   (a) Sort the numbers, and list the i largest.

   Solution: Sort using MERGE-SORT in $O(n \lg n)$ time and print the last i elements of the sorted list in $O(i)$ time. The worst-case running time is $O(n \lg n) + O(i) = O(n \lg n)$.

   (b) Build a max-priority queue from the numbers, and call EXTRACT-MAX i times.

   Solution: Building a max-priority queue takes $O(n)$ time and EXTRACT-MAX takes $O(\lg n)$ time. The worst-case running time is $O(n) + i\,O(\lg n) = O(n + i \lg n)$.

(c) Use an order-statistics algorithm to find the ith largest number, partition around the number, and sort the i largest numbers.

Solution: Use linear time selection to find the ith largest number. Partitioning takes $O(n)$ time, and sort i numbers using MERGE-SORT in $O(i \lg i)$ time. The worst-case running time is $O(n) + O(n) + O(i \lg i) = O(n + i \lg i)$.

4. (CLRS 8-2)

Solution:

(a) Allocate an output array of size n. Scan the input array to count the number of zeroes and ones as in COUNTING-SORT - this gives the range of indices for the sorted array. Scan the input array a second time, placing zeroes starting the the first index of the output array and ones starting at the index obtained from the count. This runs in $O(n)$ time and is stable.

(b) Add 'left' and 'right' pointers to the first and last elements of the input array. Compare the first and last element. While the left pointer is zero, increment it, and while the left pointer is one, decrement it. When the left pointer is one and the right is zero, swap the records. This runs in $O(n)$ time and sorts in place.

(c) INSERTION-SORT sorts in place and is stable, in worst-case $O(n^2)$ time.

(d) Radix sort is a stable sort. The algorithm in (a) can be used (b times) because the sort is stable and runs in linear time. The algorithm in (b) is not stable and cannot be used. The algorithm in (c) is not linear and cannot be used.

(e) Allocate a count array of size k and scan the input array to count the keys as in COUNTING-SORT, which gives the range of indices which hold the sorted keys. Now allocate a pointer array of size k, where pointer i points to the record of the input array indicated by index i of the count array. Begin with the first pointer. While the record is in the correct place, we increment the pointer. If the record is not correct, we go to the pointer indexed by the key of the incorrect record and increment it until it also points to an incorrect element, at which point we swap elements. At least one element is in the correct place and we increment that pointer. At each step we either increment a pointer (with or without swapping elements) or we stop at an incorrect record, in which case the following step will increment a pointer.

We allocate two arrays of size k in giving $O(k)$ additional storage. The arrays are populated in $O(k)$ time. The sorting is done in $O(n)$ time. Thus the records are sorted in place in $O(n + k)$ time. This algorithm is not stable.