These solutions are being provided **for your personal use only**. They are not to be shared with, or used by, anyone outside this class (Spring 2014 section of Georgia Tech CS 3510A). Deviating from this policy will be considered a violation of the GT Honor Code.

1. Do CLRS Exercise 11.2-3. Give your answers in terms of the particular list lengths, not the worst case.

   Solution: Let ℓ be the length of the list for the relevant hash value. Both successful and unsuccessful searches still take $\Theta(\ell)$ time in the worst case, because we might need to search sequentially to the end of the list. However, in some cases unsuccessful searches could terminate more quickly, after the first element larger than the one being searched for. Insertions would actually be asymptotically *slower* (in the worst case), at $\Theta(\ell)$ time instead of $\Theta(1)$, since the element has to be added in sorted order instead of at the beginning of the linked list. Deletions would be unchanged, still taking $\Theta(\ell)$ time in the worst case to find and remove the element in the list.

2. In this problem you will demonstrate a close connection between QuickSort and binary search trees.

   (a) Precisely describe, with justification, how a full execution of QuickSort (not necessarily the randomized version) on an array $A[1 \ldots n]$ of distinct elements corresponds to a particular binary search tree on the keys $A[1], \ldots, A[n]$. (Make sure to address how the sequence of pivot elements used by QuickSort affects the layout of nodes in the BST, and how the partitioning relates to the BST property.)

   Solution: An execution of QuickSort on a given (sub)array corresponds to a BST as follows: the chosen pivot element is the root of the BST, and all elements smaller/larger than the pivot belong to the left/right subtree of this root. These left/right subtrees are simply the BSTs corresponding to the recursive executions of QuickSort on the left/right subarrays (of the elements smaller/larger than the pivot), in exactly the same way.

   Another way of describing the correspondence is that the BST is the one obtained by inserting the elements in the order they are chosen as pivots by QuickSort.

   (b) Suppose we "charge" each comparison (between a pivot element and another element) made by QuickSort's Partition subroutine to the non-pivot element. How does the total number of comparisons charged to a particular element x relate to the depth of x in the corresponding BST? Using your answer, give a precise formula relating the runtime of a particular QuickSort execution to the depths of the elements in the corresponding BST.

   Solution: Each array element x is compared to just those pivot elements in the recursive "stack" of QuickSort calls in which x is eventually chosen as a pivot. So x is charged exactly according to its depth in the BST corresponding to the QuickSort execution (where the root is at depth zero). Thus, the runtime of the QuickSort execution equals the sum of the depths of all the nodes in the corresponding BST, plus some $O(n)$ time associated with the overhead of recursion, array indexing, etc.

   (c) Now consider RandomizedQuickSort, which always chooses a uniformly random element from the (sub)array as the pivot. Explain why this corresponds to a BST in which the elements of A are

inserted (starting from an empty tree) in a uniformly random order. (Be careful: it is *not* true that the sequence of pivots chosen by RandomizedQuickSort forms a uniformly random permutation of A!)

> **Solution:** Inserting the elements of an array in random order to build a BST corresponds to an execution of RandomizedQuickSort in the following way: the first (randomly chosen) inserted element corresponds to the random choice of pivot. Now consider just the remaining elements *smaller* than the pivot: these are inserted in random order, which inductively corresponds to the first RandomizedQuickSort recursive call (on the left subarray following partition) in the same way. Similarly, the remaining elements *larger* than the original pivot are also inserted in random order, which inductively corresponds to the second RandomizedQuickSort recursive call (on the right subarray).
>
> In the above argument, it is important that if v_1, v_2 are respectively smaller and larger than the pivot, then it does not matter whether one is inserted before the other; the resulting BST is the same. This is what lets us restrict our attention to the elements smaller/larger than the pivot, effectively reordering the original random sequence of elements so that all those smaller than the pivot are handled before all those larger than the pivot (without changing the resulting tree).

(d) It is known that in a BST constructed by adding n distinct elements in a uniformly random order, the expected height is $O(\log n)$ (see CLRS Section 12.4 for details). Using this fact and the previous parts, give a brief and convincing proof that RandomizedQuickSort runs in expected time $O(n \log n)$.

> **Solution:** By part (c), a run of RandomizedQuickSort corresponds to a BST in which n elements are inserted in a uniformly random order. By CLRS Section 12.4, this BST will have expected depth of $O(\log n)$. Thus, by part (b), each element is charged only $O(\log n)$ comparisons, and the full RandomizedQuickSort runtime is the sum of these charges (depths), which is $O(n \log n)$.

3. Do CLRS Problem 12-2 ("Radix trees," also known as "tries").

> **Solution:** In the trie representing set S, each node stores a boolean value. A value of "true" indicates that the corresponding string (obtained by the path from the root to that node) is in the set, whereas "false" indicates that it is not. These boolean values are needed because not every *prefix* of a string in S is itself in S, e.g., consider the trie for the single string "aa."
>
> To create the trie for a set S, insert all of the strings in S into an initially empty trie, as follows: for each string, insert it just as we do for a BST, adding new nodes as necessary. Also, let all new nodes have value "false," and finally set the node value for the inserted string to "true." Constructing the trie takes $\Theta(n)$ time where n is the total length of all the strings, since each string is inserted in time proportional to its length. Also notice that there are only $O(n)$ nodes in the trie, because we create at most one new node per character in the strings.
>
> Once the tree is constructed, to output the strings in sorted order we use an appropriate type of tree walk to return *only* those strings corresponding to "true" node values. To do this in $\Theta(n)$ time, each time we follow a child pointer we add the appropriate character to the end of the current string in

$O(1)$ time. The pseudocode is as follows (we initially call TrieWalk with str $= \varepsilon$, the empty string):

```
TrieWalk(T, str):
    if T ≠ nil:
        if T.val = true output str;
        TrieWalk(T.left, str||0);
        TrieWalk(T.right, str||1);
```

Notice that we output the string corresponding to a node *before* recursing on either of its child subtrees; this is required by the rules for lexicographic (alphabetical) order.

4. You are using a balanced binary search tree T to represent a dynamic set of up to n real numbers, with $O(\log n)$-time insertion, deletion, search, etc. Your application also requires a fast operation SumLessThan(T, x), which should return the sum of all numbers currently in the (sub)tree rooted at T that are strictly less than a given number x. (The value x itself is not necessarily in the tree.)

   (a) To support the SumLessThan operation, you decide to augment each node in the tree with an additional $O(1)$-size attribute that captures some aggregate information about the subtree rooted at that node. What information should this attribute contain?

   Solution: We introduce an attribute T.sum that contains the sum of all values in the subtree rooted at T.

   (b) Using the new attribute, give *recursive* pseudocode for SumLessThan that runs in $O(\log n)$ time. (To simplify the code, you may assume that for a nil pointer, your attribute is some fixed value of your choice.)

   Solution: SumLessThan(T, x):
   if $T ==$ nil then return 0
   else if $x \leq T$.value then return SumLessThan$(T$.left$, x)$
   else return T.left.sum $+ T$.value $+$ SumLessThan$(T$.right$, x)$.

   The base case is obviously correct. The second line is correct because if $x \leq T$.value, then any element strictly less than x in the tree must be in the T.left subtree. The last line is correct because it is reached if and only if $x > T$.value, in which case every node in T.left and T itself are less than x, and potentially some nodes in T.right are also less than x.

   (c) Describe how to maintain the correct values of your attribute upon *insertion* of an element, and upon a *rotation* operation, without increasing their asymptotic $O(\log n)$ runtimes.

   Solution: When inserting a node v, add v.value to the sum attribute of every node in the path from the root to where v is eventually placed in the tree. This is only $O(1)$ extra work per node on the path, so $O(\log n)$ total because the tree is balanced.

   When rotating, we only need to update the sum attribute of the two nodes whose subtrees change. We can do this in $O(1)$ time because the correct value for any x.sum (assuming its children's sum attributes are correct) is x.value $+ x$.left.sum $+ x$.right.sum. So after rotating

we just set the sum attribute of the two relevant nodes (since one is a child of the other, we should set the child first).