CS 410 Summer 2000
Homework 4 Solutions

1. (a) *Write code for the `addRight` method and the `removeRight` method.*

```
void addRight (item i) {
  // requires: none
  // ensures: Left = #Left, Right = #i + #Right
  Node n = new Node();
  n.data = i;

  // If the right string is empty, then (current.next == NULL),
  // and the last pointer is set to current.  In this case, the
  // item we're adding is now the new last item, so we must
  // update last.
  if (last == current)
    last = n;

  // The added item is supposed to be the first item in the
  // right string, so we want current.next to point to the new
  // item.
  n.next = current.next;
  current.next = n;
  right_length++;
}

item removeRight () {
  // requires: Right != <>
  // ensures: Left = #Left, #Right = removeRight + Right
  Node temp = current.next;
  current.next = current.next.next;

  // If the item we're removing was the last item, that means
  // the right string is now empty, and we need to update the
  // last pointer.
  if (last == temp)
    last = current;
  return temp.data;
  right_length--;
}
```

(b) *Write a contract for a procedure `quickSort (OneWayList l)`, and implement it using only the provided OneWayList operations.*

One thing many of you missed on the requires and ensures clauses (which I admit you don't have too many different examples to work from) is that they only refer to mathematical objects. This is why we had a string representation of the OneWayList. In the case of quickSort, recall at the beginning of class when we defined what sorting is supposed to do: it permutes the elements so that they're ordered. So that's what we need to ensure. The sort procedure has no requirements – requires clauses don't have "convenience" requirements, but rather mathematical necessities. You can't remove an item from an empty string, which is why we require the right string to be non-empty in the `removeRight` method. But we can sort empty lists.

Another way of thinking about it is that requires and ensures specify the minimal functionality and requirements necessary to get the job done. It may be convenient that we only accept lists with all items on the right, but it's not necessary, nor is the division between left and right strings a requirement to correct sorting.

`void quickSort (OneWayList l)`
requires: none
ensures: l = a permutation of $\#l$ such that $l_{i-1} \leq l_i \;\; \forall\, i$ in $2..(|l.left| + |l.right|)$

```
void quickSort (OneWayList l) {

  l.moveToStart();

  // lists of length 0 and 1 are sorted.
  if (l.rightLength() < 2) return;

  // We remove the pivot...
  item pivot = l.removeRight();

  OneWayList before = new OneWayList();
  OneWayList after = new OneWayList();

  //... and partition the element into a set < pivot and >=
  // pivot.
  while (l.rightLength() > 0) {
    item temp = l.removeRight();
    if (temp < pivot)
      before.addRight(temp);
    else
      after.addRight(temp);
  }

  // Recursively sort...
  quickSort(before);
  quickSort(after);

  // And merge the sorted lists back into the result:
  // <before> < pivot <= <after>
  before.moveToStart();
  while (before.rightLength() > 0) {
    item temp = before.removeRight();
    l.addRight(temp);
    l.advance();
  }
  l.addRight(pivot);
  l.advance();
  after.moveToStart();
  while (after.rightLength() > 0) {
    item temp = after.removeRight();
    l.addRight(temp);
```

```
                l.advance();
            }
        }
```

(c) *Do a run-time analysis of your* quickSort.

Suppose $|l| = n$, and $|before| = b$. Then $|after| = n - b - 1$. Now the runtime recurrence is then $T(n) = T(b) + T(n - b - 1) + \Theta(n)$. Except for the -1, it is identical to the recurrence for the quicksort in CLR. Following the same steps, you get the same asymptotic runtime for all cases: best case, worst case, "balanced partitioning", and average case as in CLR. The runtime is again $\Theta(n^2)$ for the worst case, and $\Theta(n \lg n)$ for all other cases.

2. *The following figure depicts a linked list implemented with an array.*

list $\boxed{4}$ free $\boxed{3}$

Index	Key	Next
1	12	0
2	19	5
3	8	2
4	6	10
5	14	1
6	32	7
7	67	9
8	95	0
9	68	8
10	11	6

(a) *What elements are in the list?*

6, 11, 32, 67, 68, 95.

(b) *What array positions are part of the free-space list?*

3, 2, 5, 1

(c) *What would the array look like after deletion of 68?*

list $\boxed{4}$ free $\boxed{9}$

Index	Key	Next
1	12	0
2	19	5
3	8	2
4	6	10
5	14	1
6	32	7
7	67	8
8	95	0
9	68	3
10	11	6

(d) *What would the array look like after insertion of 17?*

list $\boxed{3}$ free $\boxed{2}$

Index	Key	Next
1	12	0
2	19	5
3	17	4
4	6	10
5	14	1
6	32	7
7	67	9
8	95	0
9	68	8
10	11	6

3. *Suppose you are given the following information about a hashtable.*

Space Available (in words)	*10000*
Words per Item	*7*
Words per Pointer	*1*
Number of Items	*1000*
Proportion Successful Searches	*1/3*

Which hashing method, chaining or double hashing, looks best (i.e., the one that we expect to use the least amount of time)?

Chaining:

Words per item stored	9
Space taken by items	9000
Space available for hashtable	1000
α	1
Time to compute hash	2
Time for successful search	1
Time for unsuccessful search	1
Average time for search	1
Total hash plus search	3

Double hashing:

Words per item stored	7
Space taken by items	7000
Space available for hashtable	3000
α	1/3
Time to compute hash	4
Time for successful search	4.2
Time for unsuccessful search	1.5
Average time for search	3.3
Total hash plus search	7.3

Average number of probes in a successful search is $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}+\frac{1}{\alpha}=4.2$. Average number of probes in an unsuccessful search is $\frac{1}{1-\alpha}=1.5$.

So it looks like chaining is better.

4. *How many 5-element binary search trees are there with the vowels "A", "E", "I", "O", and "U" as the keys (note ordering is by alphabetical order)? You do* not *have to draw them (but as always you must justify your answer).*

   The number of ways to create a binary tree can be computed by choosing a root, and calculating the number of ways to create children of that root, and summing over all possible choices of root. Since the letters are distinct, once we choose a root, there is only one set of letters that can be in the left or right children. Let $n(i)$ be the number of ways to create a binary tree with i items. Then we have:

$n(1)$	1
$n(2)$	$n(1) + n(1) = 2$
$n(3)$	$n(2) + n(1) \cdot n(1) + n(2) = 5$
$n(4)$	$n(3) + n(2) \cdot n(1) + n(1) \cdot n(2) + n(3) = 14$
$n(5)$	$n(4) + n(3) \cdot n(1) + n(2) \cdot n(2) + n(1) \cdot n(3) + n(4) = 42$

   For example, when creating a tree with all 5 elements, if we choose "I" as the root, we have two elements in the left and right subtrees each. The ways to make those subtrees are independent of each other, so we have $n(2) \cdot n(2)$ as the middle term.

5. *CLR 13.3-4: If a node in a binary tree has two children, then its successor has no left child and its predecessor has no right child.*

   First, we consider the successor. From the (correct) code in the book on p. 249, we see that we are looking for TREE-MINIMUM(*right*[x]). From the code for TREE-MINIMUM, we see that we have found the minimum only when there is no left child. From the definitions, we can also see that if the successor s had a left child, the child would be smaller than s but still greater than x, contradicting that s is the successor.

   Now consider the predecessor. The argument is symmetrical, with *left* for *right* and TREE-MAXIMUM for TREE-MINIMUM.

6. *Describe a general way to construct red-black trees that are as unbalanced as possible. By "most unbalanced," we mean that the height of a tree constructed this way should be as close as possible to the number of nodes in the tree. (You do not need to give pseudocode for your method, just describe your construction, and explain why it results in the most unbalanced red-black trees. You may use pictures, if you like.)*

   Consider the 2 element red-black tree with a black root and a single red left child. This will be the "base case" of our most unbalanced tree. As a general extension, we add another base case as the left child of the leftmost branch, and a single black node as a child everywhere there is no child. This clearly makes the black height of the tree one higher.

   Give a lower bound for h in terms of n (height of the tree and number of internal nodes, respectively) for your construction. Remember that in general the relation is $h \leq 2\log_2(n+1)$.

   Consider the "base case". It has 3 sites at which we can add a child, and under our construction, we will add a black child at 2 of them and a base case at 1 of them. In general, if we have k sites for children, we add $(k-1)$ black children and 1 base case, thus adding $k+1$ new nodes. Each of these black children has 2 sites available for children, and the one new base case has 3, thus giving us $2(k-1)+3$, or $2k+1$ sites at which we can attach children on the next iteration.

This gives us the following recursive definitions for height and number of nodes (with $h_0 = 2, k_0 = 3, n_0 = 2$):

$$\begin{aligned} h_{i+1} &= h_i + 2 \\ k_{i+1} &= 2k_i + 1 \\ n_{i+1} &= n_i + k_i + 1 \end{aligned}$$

Clearly $h(i) = 2i + 2$, $k(i) = 2^{i+2} - 1$. We wish to show that $n(i) = 2^{i+2} - 2$. We'll do this inductively.

Base case: $i = 0$. $n_0 = 2$; $2^{0+2} - 2 = 2$.
Inductively:

$$\begin{aligned} n(i+1) &= n(i) + k(i) + 1 \\ &= (2^{i+2} - 2) + (2^{i+2} - 1) + 1 && \text{by inductive hypothesis} \\ &= 2 \cdot 2^{i+2} - 2 \\ &= 2^{i+3} - 2 \end{aligned}$$

Now, the question becomes: how does $h(i)$ relate to $n(i)$? Well, $\log_2(n(i)+2) = i+2$, so $h(i) = 2\log_2(n(i)+2) - 2$.

7. *Draw all possible red-black trees containing keys 1, 2, and 3. Observe the convention that the root node should be colored black. How many different ways are there to do this?*

   2 ways: The tree must have 2 at the root, with 1 as its left child and 3 as its right child. We can color the left and right children both black or both red.

8. *CLR 14.2-3: Argue that rotation preserves the inorder key ordering of a binary tree.*

   We need to prove that the BST property (about ordering of keys) still holds.

   Consider figure 14.2 on p. 266. From the diagram on the left, we can see that $\{\alpha\} \le x \le \{\beta\}$, by the definition of a binary search tree. In addition, we can see that $x \le y$, $\{\alpha\} \le y$, $\{\beta\} \le y$, and $y \le \{\gamma\}$. Together, this implies $\{\alpha\} \le x \le \{\beta\} \le y \le \{\gamma\}$.

   After the rotation, the placement of x relative to y, and the placement of β relative to x and y are changed, but all other relative placements remain the same.
   y is now the right child of x, but since $x \le y$ this is OK.
   β is now the left child of y, which is now the right child of x, but this also works since $x \le \{\beta\} \le y$.

9. *CLR 19.1-5: Describe the data structure that would result if each black node in a red-black tree were to absorb its red children, incorporating their children with its own.*

   Under this change, a red-black tree becomes a B-tree with minimum degree 2.

10. *CLR 19.2-1*

    See hw4_fig1 link on homework page.

11. *CLR 19.2 on page 399.*

    Modern relational databases rely heavily on tree structures, used for indices, to allow fast lookups into the data based on key values. The `join` and `split` operations are basic operations for relational databases which are similar to the ones described in this problem.

    (a) Recall that the height is the distance from the leaves, not the root.
    We add a field height$[x]$ to each node. This is initialized to 0 for the initial root with no children. Whenever we split a node, we assign the new sibling node's height to the height of its sibling which was split. When we split the root, the new root node's height is the old root's height + 1. When we delete, it never changes the height of a node (we only delete the node with the greatest height, the root). If we don't create a new node, there's nothing to be done, so we only do a constant amount of additional work for nodes we are visiting anyway, and no changes are required for search or delete, so asymptotic running times of searching, insertion and deletion are unchanged.

    (b) Suppose $height(S') = height(S'')$. Then we simply make a new node for x with its height $height(S') + 1$ and S' and S'' as its left and right child, respectively. This is $O(1) = O(|h' - h''| + 1) = O(|h' - h''|)$. Note that since we are only considering 2-3-4 trees, the root of any non-empty tree has enough keys and children to be another node in the tree (otherwise it might be a concern to move a root node down into a tree, since there is an exemption to the minimum degree only for the root node) If the heights are not the same, we descend through the higher tree until we reach a node with height one greater than the height of the smaller tree, following a path as though we were looking for k in the larger tree. As we go down, we split up 4-nodes, just as in a regular B-tree insert. When we reach the node with the correct height, it cannot have 4 elements (or it would have been split.) So we add k and the shorter tree in the appropriate place (it will either be at the far left or the far right).

    (c) Let LoTrees be the set of trees $\{T'_0, T'_1, \ldots, T'_m\}$ as defined in this problem, and let HiTrees be the corresponding set of trees greater than k. Let LoKeys be $\{k'_1, k'_2, \ldots, k'_m\}$, and let HiKeys be the corresponding set of keys greater than k.
    Start at the root, with all 4 sets described above initially empty. For each node you visit:
    Case 1: key is not in this node.
    Locate which child pointer you will follow in your search for the key k. Append all the keys before this pointer to LoKeys, and all the children before this pointer to LoTrees. Similarly, append all the keys after this pointer to HiKeys, and all the children after this pointer to HiTrees. Note that we have put all the keys in this node into one of the two key lists, and all the children except the one we will follow into one of the two tree lists. Exactly as many keys went into LoKeys as trees were placed into LoTrees; likewise for HiKeys and HiTrees. Furthermore,

the desired relational properties hold.

Then traverse the path to the next node (the one child not in either list of trees).

Case 2: key is in this node.

Append all the keys before k into LoKeys, and all the children before k to LoTrees. Similarly, append all the keys after k to HiKeys, and all the children after k to HiTrees. Note that here we have put all the keys in this node except k into one of the two key lists, and all the children into one of the two tree lists. Exactly one more tree went into LoTrees as keys were placed into LoKeys ; likewise for HiKeys and HiTrees. Furthermore, the desired relational properties hold.

The heights between consecutive trees in each tree list is monotonically decreasing, and the difference cannot be greater than 1.

(d) Create the sets described in part (c). This can be done in a constant number of steps ($O(t) = O(2) = O(1)$) for each node visited, and the number of nodes visited is no more than the height of the tree, which is $O(\lg n)$, so creating these sets can be done in $O(\lg n)$.

Then merge these trees and keys as discussed in (b).

Merge all keys in LoKey with all trees in LoTrees. First, you can merge all trees of a given height into one tree with a new root node containing all the keys between (e.g. if trees T'_i through T'_j are the ones of height h, the keys between them will be k'_{i+1} through k'_j. There can't be more than 4 trees of this height ($2t$) since they all came from one node in the original 2-3-4 tree, so 3 keys (and 4 children) will fit into a root node. This can be done in constant time for each height, and there are $O(\lg n)$ different heights of trees to be merged. Then, pairwise merge the trees of different heights, from largest to smallest (or vice versa). Each merge is done in $O(|h' - h''|)$ time, and there are $O(\lg n)$ trees to merge. So the overall running time is $O(\lg n)$.

Similarly, merge all keys in HiKey with all trees in HiTrees in $O(\lg n)$ time.