Homework 3 Solutions
Fundamental Algorithms, Spring 2008, Professor Yap

Due: Wed Mar 5, in class.
HOMEWORK with SOLUTION, prepared by Instructor and T.A.s

INSTRUCTIONS:

- Please read questions carefully. When in doubt, please ask.
- Please write succinctly, to the point.
- You may post general questions to the homework discussion forum in class website. Also, bring your questions to recitation on Monday.

---

1. (16 Points)
   This is Exercise 2.1 (p.6) in Lecture III (please download the version dated Feb 27). Consider the dictionary ADT.
   (a) Describe algorithms to implement this ADT when the concrete data structures are linked lists. Try to be brief and give algorithms at the high level that we use in our lecture notes.
   (b) Analyze the worst complexity of your algorithms in (a). NOTE: complexity is be analyzed to Θ-order.
   (a') Describe algorithms to implement this ADT when the concrete data structures are arrays instead of linked lists.
   (b') Analyze the complexity of your algorithms in (a').

SOLUTION:

- (a) We implement the dictionary ADT with linked lists as follows. First we describe how a linked list L represents a dictionary. The linked list L has a node called the **head**, and it is denoted by $L.\texttt{head}$. Every node u in the list has two fields, $u.\texttt{key}$ and $u.\texttt{data}$. Thus $(u.\texttt{key}, u.\texttt{data})$ represents an item stored in L. Also, u has a pointer $u.\texttt{next}$ to the next node (unless u is the last node in which case $u.\texttt{next} = \mathsf{nil}$). From the node $L.\texttt{head}$, we can reach every node in the list using the `next` pointer. For simplicity, we assume there is a field called $L.\texttt{tail}$ that gives us the last node in the list.

  We use "shell programming" based on the following list traversal algorithm:

```
ListTraversal(L):
    u ← L.head
    while (u ≠ nil)
        VISIT(u)
        u ← u.next
    CLEANUP()
```

  The macros VISIT and CLEANUP can be empty initially. To implement the various other operations, we can redefine VISIT and CLEANUP appropriately. We now implement the 3 dictionary operations:

  (a) $\texttt{lookUp}(K)$: We use the ListTraversal shell, but define VISIT(u) as the following macro:

$$VISIT(u): \quad \mathsf{if}\ (u.\texttt{key} = K)\ \ \mathsf{return}(u)$$

  Thus, if the key K is found in u, we exit from the while-loop and return u. The CLEANUP() macro is:

$$\text{CLEANUP}(): \quad \mathsf{return}(\mathsf{nil})$$

  (b) $\texttt{insert}(K, D)$: We use the ListTraversal shell, but define VISIT(u) as the following macro:

$$VISIT(u): \quad \mathsf{if}\ (u.\texttt{key} = K)\mathsf{return}(\mathsf{nil})$$

  Thus, if the key K is found in u, we return nil, indicating failure (duplicate key). The CLEANUP() macro is:

```
CLEANUP():
    u ← new(Node)
    u.key := K, u.data := D
    u.next := L.head
    L.head := u
    return(u)
```

  where $new(Node)$ returns a pointer to space on the heap for a node.

  (c) $\texttt{delete}(u)$: Since u is a pointer to the node to be deleted, this can be done in $O(1)$ in the standard way.

SOLUTION:

- (b) For complexity analysis, let n be the current size of the linked list. The complexity of `lookUp`(K) is $\Theta(n)$ in the worst case because we have to traverse the entire list in the worst case. Both `insert`(K, D) and `delete`(u) are preceded by `lookUp`'s, and so these are $\Theta(n)$. The remaining operations of insert/delete are $\mathcal{O}(1)$. Hence all three operations are $\Theta(n)$.
- (a') We must implement `lookUp`, `insert`, and `delete` using arrays.

  Suppose A is our array of size N, and it currently stores n items. We maintain these two integer variables, N and n. It is assumed that the items are stored in the first n entries of A. If the first item is $A[0]$, then $A[n-1]$ is the last item, so $A[n]$ is the first free entry. We have the invariant $0 \leq n < N$.

  To `lookUp` a key K, we use a for-loop from 0 to $n-1$, looking the K in the array. If found, we return the index i, and otherwise we return -1. (NOTE: thus, the for-loop is the equivalent of the "array traversal routine", and we can write the loop like a shell program as in the case of list traversal.)

  To `insert` an item (K, D), we call `lookUp`(K), and either return -1 if key K was found, or we put the item into $A[n]$ and increment n.

  We must then check if the invariant $n < N$ is violated. If so, we have two options (both are reasonable): either throw an error (in case N is not automatically increase) or we double the size of the array. The latter strategy involves getting new space for the array A, copying the contents from the old location to the new one, deleting the old space. Finally, we update N by doubling it.

  To `delete` an item, we assume we are given an index i to the item $A[i]$ to be deleted. In this case, we just move item $A[n-1]$ into $A[i]$ and decrement n.
- (b') `lookUp` is $\Theta(n)$ in the worst case. `insert`, since it is preceded by a `lookUp`, the complexity is at least $\Omega(n)$. In case the array is full, and we need to double its size, the work to be done is still $\mathcal{O}(n)$. Hence, `insert` is $\Theta(n)$. For `delete`, we may move up to n items, and hence the complexit is $\Theta(n)$.

2. (12 Points)
   Consider the priority queue ADT.
   (a) Describe algorithms to implement this ADT when the concrete data structures are AVL trees. You may assume the standard AVL algorithms.
   (b) Analyze the complexity of your algorithms in (a).

SOLUTION: To support the operations for a Priority Queue, we must implement `deleteMin`(), `insert`(v,k).
`deleteMin`() is implemented immediately. Starting at the root, move to the left child repeatedly, until a node is found which has no left children. Utilize the AVL `delete` on this node and output its key value. `insert`(v, k) is simply the AVL insert operation, using key k as the value.
There is another operation called `update`$(v, newK)$, which we did not explicitly describe in the ADT section in Lecture III (§2). This opeation is sometimes called `decreaseKey`$(v, newK)$. For this operation, assume we have a pointer to node v in the AVL tree, and v.`key` $> newK$. We are going to replace the key of v with $newK$ (and thus increase its priority). To implement this operation, a simple solution is to just save v.`data`, then `delete`(v), and finally `insert`$(newK, v$.`data`$)$. All running times are $\mathcal{O}(\log n)$ on AVL tree.

3. (0 Points – do not hand in!)
   Do Exercise 3.1 in Lecture III (delete key 10 from the BST of Figure 3(a), using rotation-based and standard deletion algorithms).

4. (10 Points)
This is Exercise 3.11 (page 17) in Lecture III. Prove Lemma 2 (p. 14)that there is a unique way to order the nodes of a binary tree T that is consistent with any binary search tree based on T. HINT: remember the fundamental rule about binary trees!

SOLUTION: We prove this by contradiction. Let T be a binary tree with root r and n nodes and M_1 and M_2 denote two orderings. Assume $M_i(v)$ $(i = 1, 2)$ denote the value of node v under M_i, and the values lie in the set $\{1, 2, \ldots, n\}$. First observe that $M_1(r) = M_2(r)$. To see why, assume without loss of generality that $k_1 = M_1(r) < M_2(r) = k_2$. Regardless of the ordering, all nodes in the right subtree must satisfy $M_1(u) > k_1$. The first ordering then must have $n - k_1$ nodes in the right subtree while latter ordering must have $n - k_2$ nodes in the right subtree. As both M_1 and M_2 are orderings of the same tree, we see that $k_1 = k_2$. Now the rest of the result follows by structural induction on the left- and right-subtrees of T.
SUBTLE POINT: actually, we need to strengthen our induction hypothesis to allow any set of n distinct keys, not just $\{1, \ldots, n\}$.

5. (9 Points)
Exercise 3.5 (p.16) in Lecture III. Let T be the binary search tree in Figure 3(a) in Lecture III. HINT: although we only require that you show the trees at the end of the operations, we recommend that you show selected intermediate stages. This way, we can give you partial credits in case you make mistakes!
(a) Perform the operation Display T and T' after the split.
(b) Now perform tree after the operation in (a). Display the tree after insertion.
(c) Finally, perform tree after the insert in (b) and T' is the tree after the split in (a).

SOLUTION: Most people had no problems with this. Omitted.

6. (12 Points)
This is Exercise 3.19 (page 18) in Lecture III. Suppose we allow allow duplicate keys. Under the binary search tree property (equation (1) in p.7), we see that all the keys with the same value must lie in consecutive nodes of some "right-path chain".
(a) Show how to modify lookup on key K so that we list all the items whose key is K.
(b) Discuss how this property can be preserved during rotation, insertion, deletion.
(c) Discuss the effect of duplicate keys on the complexity of rotation, insertion, deletion. Suggest ways to improve the complexity.

7. (12 Points)
This is Exercise 4.2 (p. 21) in Lecture III on tree traversals.
(a) Let the in-order and pre-order traversal of a binary tree T with 10 nodes be $(a, b, c, d, e, f, g, h, i, j)$ and $(f, d, b, a, c, e, h, g, j, i)$, respectively. Draw the tree T.
(b) Prove that if we have the pre-order and in-order listing of the nodes in a binary tree, we can reconstruct the tree.
(c) Consider the other two possibilities: (c.1) pre-order and post-order, and (c.2) in-order and post-order. State in each case whether or not they have the same reconstruction property as in (b). If so, prove it. If not, show a counter example.
(d) Redo part(c) for full binary trees. Recall that in a full binary tree, each node either has no children or 2 children.

SOLUTION:

(c.1) does not have reconstruction property. A simple counter example are the two binary search trees with 2 nodes. Their pre-order and post-order are identical but they are different trees.

(c.2) does have reconstruction property, by the same argument as (b).

(d) Now, even (c.1) has reconstruction property.

8. (5 Points)
This is Exercise 6.3, Lecture III. What is the minimum number of nodes in an AVL tree of height 10?

SOLUTION: Minimum is 232. Let $\mu(h)$ be the minimum number of nodes for height h. The text shows that $\mu(h+1) = 1 + \mu(h) + \mu(h-1)$ and

$\mu(2) = 4, \mu(3) = 7, \mu(4) = 12, \mu(5) = 20, \mu(6) = 33, \mu(7) = 54, \mu(8) = 88, \mu(9) = 143, \mu(10) = 232.$

9. (0 Points, do not hand in!)
Exercise 6.6, p.32. Referring to Figure 17:
(a) Find all the keys that we can delete so that the rebalancing phase requires two rebalancing acts.
(b) Among the keys in part (a), which deletion has a double rotation among its rebalancing acts?
(c) Please delete one such key, and draw the AVL tree after each of the rebalancing acts.

10. (6 Points)
Exercise 6.8, p.32. Draw the AVL trees after you insert each of the following keys into an initially empty tree: $1, 2, 3, 4, 5, 6, 7, 8, 9$ and then $19, 18, 17, 16, 15, 14, 13, 12, 11$.

11. (10 Points)
For the midterm, you only need to understand 2-3 trees and not the general (a, b)-trees.
(a) Show the result of inserting the keys $1, 3, 4, 20, 22, 24$ (in this order) into the tree in Figure 21 (p. 36). As usual, show intermediate results.
(b) Show the result of deleting the keys 8 and then 10 from tree in Figure 21.