Homework 3
Fundamental Algorithms, Fall 2004, Professor Yap

Due: Thu Oct 21, in class

INSTRUCTIONS:

- Please read carefully.
- General remark: we do encourage students to work in groups for discussion. However, when you finally write up the solutions, they *must* represent your individual work.

---

1. (15 Points) Rotation

   Figure 2 in Lecture III suggests that a rotation can be viewed as transforming a doubly-linked list of the form (x, u, v, w) to (x, v, u, w). This view of `rotate`(u) is illustrated by Figure 1.

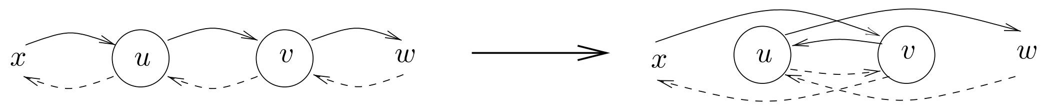

Figure 1: Viewing `rotate`(u) as interchanging two consecutive nodes in a doubly-linked list (x, u, v, w).

   There are 6 links (or pointers) to be re-assigned. Such re-assignments must be done in the correct order. Using the terminology of doubly-linked list, so that u`.next` and u`.prev` are the forward and backward pointers of the node u, here is the sequence of assignments:

```
"Rotate"(u):
    Fix the forward pointers
        1.  u.prev.next ← u.next
        2.  u.next ← u.next.next
        3.  u.prev.next.next ← u
    Fix the backward pointers
        4.  u.next.prev.prev ← u.prev
        5.  u.next.prev ← u
        6.  u.prev ← u.prev.next
```

   (a) Now translate the above sequence of 6 assignments into the corresponding assignments for rotation in a binary search tree: the u`.next` pointer may be identified with u`.Parent` pointer. However, u`.prev` would be u`.Left` or u`.Right`, depending on whether x is a left child or right child of u. You need tests to determine the correct interpretation for the `prev` link of any node. A further complication is that x or/and w may not exist. E.g., x does not exist iff u`.prev` = Nil. Keeping these all in mind, please give the correct code for performing rotation where the input is a node u. HINT: how are the interpretations of u`.prev` and v`.prev` correlated?

   (b) The result of part (a) tells us that double-rotation can be implemented with 12 assignments. Show that 10 assignments are necessary. Then try to give solution using as few assignments as possible. Be sure to first give a "high level description" of your solution. NOTE: one can always find a solution that uses assignments to extra temporary variables, but we want to avoid this.

2. (30 Points) Insertion and Deletion on AVL Trees
   (a) Insert the following sequence of keys, in the given order, into an initially empty AVL tree:

$$4, 1, 2, 3, 6, 8, 9, 11, 10, 12$$

You must draw the AVL tree at the end of each insertion. You may indicate any intermediate trees. Let the final tree be T_1.
(b) Now repeat part (a), but now inserting the keys in reverse order (starting with key 7 and ending with key 4). Again, draw the AVL tree at the end of each insertion. Call the resulting AVL tree T_2.
(c) Determine the value of $d(T_1, T_2)$, defined to be the minimum number of rotations necessary to convert T_1 to T_2. Draw the AVL tree at the end of each rotation. REMARK: Exercise 3.5 in Lecture II – which was a problem in Spring 2003 – shows that $d(T, T')$ is always a finite number.

3. (20 Points) Relaxed AVL Trees
Let us defined the **relaxed AVL-balance condition** to mean that at each node u in the binary tree, $|balance(u)| \leq 2$.
(a) Derive an upper bound on the height of a relaxed AVL tree on n nodes.
(b) Give an insertion algorithm that preserves relaxed AVL trees. Try to follow the original AVL insertion as much as possible; but point out diferences from the original insertion.

4. (20 Points) $(2,3)$-trees
Start with the $(2,3)$-tree in Figure 12 of Lecture III. Show the tree after each insertion or deletion:
(a) Insert the following keys

$$10.5, 11.5, 12.5, 13.5$$

into the tree.
(b) Delete the keys 8 and 10 from the tree of Figure 12.

5. (15 Points) (a,b)-trees
Spell out in detail the algorithms for inserting into an (a,b)-search tree that achieves a space-utilization factor of 3/4. You must provide enough details so that a competent programmer can implement your algorithm. This includes any data structure organization.