Problem Set 6

Data Structures and Algorithms, Fall 2021

Due: October 28, in class.

Problem 1

(a) Professor George O'Jungle has a 27-node binary tree, in which every node is labeled with a unique letter of the Roman alphabet or the character `&`. Preorder and postorder traversals of the tree visit the nodes in the following order:

- Preorder: `I Q J H L E M V O T S B R G Y Z K C A & F P N U D W X`
- Postorder: `H E M L J V Q S G Y R Z B T C P U D N F W & X A K O I`

Draw George's binary tree.

(b) We say a binary tree is *full* if every non-leaf node has exactly two children. Describe and analyze a recursive algorithm to reconstruct an arbitrary full binary tree, given its preorder and postorder node sequences as input. You may assume that all keys are distinct and that the input is consistent with at least one binary tree.

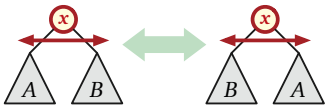
(c) Prove that there is no algorithm to reconstruct an *arbitrary* binary tree from its preorder and postorder node sequences.

Problem 2

Prove that any n-node binary search tree can be transformed into a height-balanced binary search tree with the same node values, using only $O(n)$ rotations. In a height-balanced binary search tree, the difference between the height of the left and right subtrees of every node is never more than 1. *(Hint: You might want to first transform the input binary tree into a chain.)*

Problem 3

Let T be a binary tree whose nodes store distinct numerical values. In class we have introduced the rotate operation. Here, we introduce another operation called *swap*, which swaps the left and right subtrees of an arbitrary node. See below for an example.



Describe an algorithm to transform an arbitrary n-node binary tree with distinct node values into a binary search tree, using at most $O(n^2)$ rotations and swaps.[1]

In this problem, your algorithm is not allowed to directly modify parent or child pointers, create new nodes, or delete existing nodes; the only way to modify the tree is through rotations and swaps.

[1] Actually, this can be done using only $O(n \log n)$ rotations and swaps. Here, devising an algorithm using $O(n^2)$ rotations and swaps suffices, but you are welcome to present better algorithms.

On the other hand, you may compute anything you like for free, as long as that computation does not modify the tree. In this problem, the running time of your algorithm is defined to be the number of rotations and swaps that it performs.

Problem 4

(a) Show the red-black trees that result after successively inserting the keys 99, 75, 62, 20, 38, 11 into an initially empty red-black tree. Note, you need to show the red-black tree after *each* insertion.

(b) Following part (a), now show the red-black trees that result from the successive deletion of the keys in the order 11, 20, 38, 62, 75, 99. Again, you need to show the red-black tree after *each* deletion.

Problem 5

An *AVL tree* (named after inventors Adelson-Velsky and Landis) is a binary search tree that is height balanced: for each node x, the heights of the left and right subtrees of x differ by at most 1. To implement an AVL tree, we maintain an extra attribute in each node: $x.h$ is the height of node x. As for any other binary search tree T, we assume that $T.root$ points to the root node.

(a) Prove that an AVL tree with n nodes has height $O(\log n)$. *(Hint: Prove that an AVL tree of height h has at least F_h nodes, where F_h is the h^{th} Fibonacci number.)*

(b) To insert into an AVL tree, first place a node into the appropriate place in binary search tree order. Now, the tree might no longer be height balanced: the heights of the left and right children of some node might differ by 2. Describe a procedure BALANCE(x), which takes a subtree rooted at x whose left and right children are height balanced and have heights that differ by at most 2, i.e., $|x.right.h - x.left.h| \leq 2$, and alters the subtree rooted at x to be height balanced. *(Hint: Use rotations.)*

(c) Using part (b), describe a recursive procedure AVLINSERT(x, z) that takes a node x within an AVL tree and a newly created node z (whose key has already been filled in), and adds z to the subtree rooted at x, maintaining the property that x is the root of an AVL tree. As in TREEINSERT from the textbook, assume that $z.key$ has already been filled in and that $z.left = NULL$ and $z.right = NULL$; also assume that $z.h = 0$. Thus, to insert the node z into the AVL tree T, we call AVLINSERT$(T.root, z)$.

(d) Argue that AVLINSERT, run on an n-node AVL tree, takes $O(\lg n)$ time and performs $O(1)$ rotations.

Problem 6

A meldable priority queue stores a set of keys from some totally-ordered universe (such as the integers) and supports the following operations:

- `MakeQueue`(): Return a new priority queue containing the empty set.
- `FindMin`(Q): Return the smallest element of Q (if any).
- `DeleteMin`(Q): Remove the smallest element in Q (if any).
- `Insert`(Q, x): Insert element x into Q, if it is not already there.
- `DecreaseKey`(Q, x, y): Replace an element $x \in Q$ with a smaller key y. (If $y > x$, the operation fails.) The input is a pointer directly to the node in Q containing x.
- `Delete`(Q, x): Delete the element $x \in Q$. The input is a pointer directly to the node in Q containing x.
- `Meld`(Q_1, Q_2): Return a new priority queue containing all the elements of Q_1 and $Q2$; this operation destroys Q_1 and Q_2.

A simple way to implement such a data structure is to use a heap-ordered binary tree, where each node stores a key, along with pointers to its parent and two children. (In this problem, a heap-ordered binary tree satisfies the min-heap property, but is not a binary min-heap. In particular, it is not stored in an array.) `Meld` can be implemented using the following randomized algorithm:

`Meld`(Q_1, Q_2)

1: **if** (Q_1 is empty) **then return** Q_2.
2: **if** (Q_2 is empty) **then return** Q_1.
3: **if** ($Q_1.key > Q_2.key$) **then** Swap Q_1 and Q_2.
4: **if** (toss a fair coin and head comes up) **then** ▷ This happens with probability $1/2$.
5: $\quad Q_1.left \leftarrow$ `Meld`$(Q_1.left, Q_2)$.
6: **else**
7: $\quad Q_1.right \leftarrow$ `Meld`$(Q_1.right, Q_2)$.

(a) Prove that for *any* heap-ordered binary trees Q_1 and Q_2 (particularly, not just those constructed by the operations listed above), the expected running time of `Meld`(Q_1, Q_2) is $O(\log n)$, where $n = |Q_1| + |Q_2|$. *(Hint: What is the expected length of a random root-to-leaf path in an n-node binary tree, where each left/right choice is made with equal probability?)*

(b) Show that each of the other meldable priority queue operations can be implemented with at most one call to `Meld` and $O(1)$ additional time.