CS2020: Data Structures and Algorithms (Accelerated)

Discussion Group Problems for Week 3

For: January 26/27, 2011

Problem 1. (Binary Search Tree Basics)

Assume we start with an empty tree. For each of the following sequence of operations, draw the resulting binary search tree.

a. `Insert`(1), `Insert`(2), `Insert`(3), `Insert`(4), `Insert`(5)

b. `Insert`(5), `Insert`(4), `Insert`(3), `Insert`(2), `Insert`(1)

c. `Insert`(6), `Insert`(9), `Insert`(15), `Insert`(4), `Insert`(22), `Insert`(5)

d. `Insert`(6), `Insert`(9), `Insert`(15), `Insert`(4), `Insert`(22), `Insert`(5), `Delete`(9)

e. `Insert`(6), `Insert`(9), `Insert`(15), `Insert`(4), `Insert`(5), `Insert`(7), `Insert`(8), `Delete`(6)

What would each of the above trees look like if the underlying data structure were an AVL tree?

Problem 2. (In-Order Tree Walk)

As we discussed in class, an in-order tree walk is one that starts at the root and outputs each key in the tree in order. In particular, at a node v, it first recursively performs an in-order walk on the left child of v (if any), it then outputs the key at v, and it then recursively performs an in-order walk on the right child of v (if any).

While it is quite easy to implement an in-order walk recursively, it is much more difficult to implement it iteratively. In this question, you will implement a non-recursive solution in Java. Assume that you have access to an object that implements a stack (i.e., supports push and pop in a manner typical of a stack). Assume that each node in your binary tree implements an interface that includes: `getLeft`, `getRight`, and `getKey`. Give a non-recursive algorithm for performing an in-order tree walk (using the stack as needed) that prints out every element in the tree in order to the standard output.

Problem 3. (Fast Successors and Predecessors) Some applications make heave use of the successor and predecessor functionality in BSTs, and thus we would like to support successor and predecessor in constant time. Describe how `insert` and `delete` have to be modified in order to support this faster functionality, without changing the asymptotic performance. (*Hint:* add additional pointers to the nodes.)

Problem 4. (Splitting and Joining Binary Search Trees)

For this problem, we consider how to merge two binary search trees, and how to split a binary search tree into two pieces.

Problem 4.a. Give an algorithm that takes as input two binary search trees $T1$ and $T2$ and returns a single tree $Tout$ that contains a merged version of the two input trees. That is, $Tout$ should contain exactly the same set of keys as $T1$ and $T2$, with no extra nodes in the tree. Assume that the largest element in tree $T1$ is smaller than the smallest element in tree $T2$. What is the running time of your algorithm? Can you devise a version in which the resulting tree $Tout$ has height at most $\max(h(T1), h(T2)) + 1$?

Problem 4.b. Give an algorithm that takes as input one binary tree T and an integer key k. It should return two trees $T1$ and $T2$ where every key in $T1$ is $< k$, and every key in $T2$ is $\geq k$. (Every key contained in T should be in either tree $T1$ or $T2$, and there should be no additional keys in $T1$ or $T2$.) What is the running time of your algorithm?