CMPT 307 — Data Structures and Algorithms

Exercises on Hash Tables and Binary Trees. Due: Friday, November 1st)

Reminder: the work you submit must be your own. Any collaboration and consulting outside resourses must be explicitely mentioned on your submission.

1. Suppose that in a hash table collisions are resolved by chaining. However, instead of using a list of keys that hash into the same hash value we use binary search trees. Assuming simple uniform hashing and that these trees are reasonably balanced (does not matter how), find the expected time of successful and unsuccessful searches.

2. Suppose that we are given a key k to search for in a hash table with positions $0, 1, \ldots, m-1$, and suppose that we have a hash function h that maps the key space into the set $\{0, 1, \ldots, m-1\}$. The search scheme is as follows.

   (a) Compute the value $i := h(k)$, and set $j := 0$

   (b) Probe in position i for the desired key k. If you find it, or if this position is empty, terminate the search

   (c) Set $j := (j+1) \pmod{m}$ and $i := (i+j) \pmod{m}$, and return to the previous step

   Assume that m is a power of 2.

   (a) Show that this scheme is an instance of the general "quadratic probing" scheme by exhibiting the appropriate constants c, d for the corresponding equation

   (b) Prove that this algorithm examines every table position in the worst case.

3. Prove that no matter what node we start at in a height h binary search tree, k successive calls to Tree-Successor take $O(k+h)$ time.

4. Is the operation of Deletion "commutative" in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y and then x? Argue why or give a counterexample.

5. Show that any arbitrary n-node binary search tree can be transformed into any other arbitrary n-node binary search tree using $O(n)$ rotations.

6. Suppose that a node x is inserted into a red-black tree with RB-Insert and then immediately deleted with RB-Delete. Is the resulting RB-tree the same as the initial RB-tree? Justify your answer.

7. During the course of an algorithm, we sometimes find that we need to maintain past versions of a dynamic set as it is updated. Such a set is called *persistent*. One way to implement a persistent set is to copy the entire set whenever it is modified, but this approach can slow down a program and also consumes much space. Sometimes, we can do better.

   Consider a persistent set S with the operations Insert, Delete, and Search, which we implement using binary search trees as shown in the picture. We maintain a separate root for every version of the set. In order to insert the key 5 into the set, we create a new node with key 5. This node becomes the left child of a new node with key 7, since we cannot modify the existing node with key 7. Similarly, the new node with key 7 becomes the left child of a new node with key 8 whose right child is the existing node with key 10. The new node with key 8 becomes, in turn, the right child of a new root r' with key 4 whose left child is the existing node with key 3. We thus copy only part of the tree and share some of the nodes with the original tree, as shown in the picture.

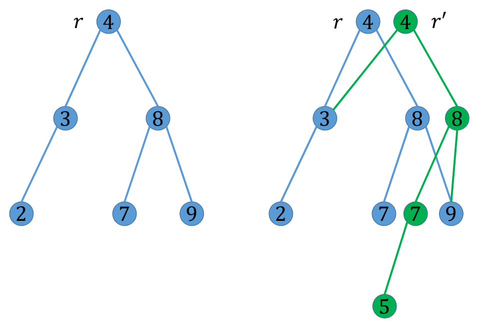

Figure 1: The persistent binary tree that results from the insertion of key 5. The most recent version of the set consists of the nodes reachable from the root r', and the previous version consists of the nodes reachable from r. Green nodes are added when key 5 is inserted.

Assume that each tree node has the fields *key*, *left*, and *right*, but no parent field.

(a) For a general persistent binary search tree, identify the nodes that need to be changed to insert a key k or delete a node y.

(b) Write a procedure Persistent-Tree-Insert that, given a persistent tree T and a key k to insert, returns a new persistent tree T' that is the result of inserting k into T. What is its running time? (c) Suppose that we had included the parent field in each node. In this case, Persistent- Tree-Insert would need to perform additional copying. Prove that Persistent-Tree-Insert would then require $\Omega(n)$ time and space, where n is the number of nodes in the tree.

8. Show the result of inserting keys

$$6, 19, 17, 11, 3, 12, 8, 20, 22, 23, 13, 18, 14, 16, 1, 2, 24, 25, 4, 26, 5$$

in order into an empty B-tree with $t = 2$. Only draw the final configuration.