[redacted]

October 14, 2021

Problem Set 6

[redacted]

Due: 8 am on Thursday, October 21

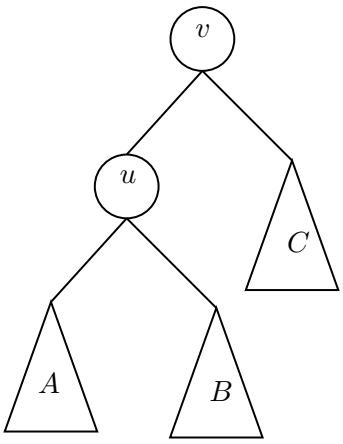
Problem 6-1 (Rotate that BST!) 15+6 points

A binary search tree T is defined by the field $T.root$ which is a pointer to the root node of the tree. Every node *node* in this tree contains the attributes: *key* which stores the value, $left$ which is a pointer to the left child, *right* which is a pointer to the right child, and p which is a pointer to its parent node. Further, recall that, in a binary tree for all node *node*, $node.key \geq node.left.key$ and $node.right.key > node.key$ provided $node.left \neq$ **nil** and $node.right \neq$ **nil**. If these sub-trees are empty, then it is set to **nil**.

A useful operation for binary trees is that of ROTATE. ROTATE takes as input a node u and produces a rotated tree where the parents of u and v are interchanged, while preserving the structure of a binary search tree. Further, based on whether u is the left child of its parent or its right child, one can define two auxiliary functions LEFTROTATE and RIGHTROTATE. In other words, ROTATE can be defined by the following pseudocode:

```
ROTATE(T, u)
    v = u.p
    if v = nil then return u
    if v.left = u then return LEFTROTATE(T, u)
    else return RIGHTROTATE(T, u)
```

(a) (2 points) For the input tree given below, draw the output of the invocation call LEFTROTATE(T, u):



(b) (5 points) Write a pseudocode LEFTROTATE(T, u) that takes a node u as input and outputs the rotated tree. State and justify the running time of your algorithm. Briefly argue its correctness.

A corresponding pseudocode for RIGHTROTATE can be written and this completes the algorithm for ROTATE. The reason why ROTATE operation is critical is because every other binary tree operations can be re-written to have ROTATE as its basis. We will investigate this claim for the case of DELETE, SUCCESSOR, and SEARCH. For all of these problems, you are given a binary search tree T with n nodes.

(c) (4 points) For this problem, you will present a pseudocode for ROTATEDELETE(T, z) that invokes only LEFTROTATE and TRANSPLANT where z is the node you wish to delete. Your algorithm will contain exactly three lines of code.

Do not forget to justify the correctness of your algorithm and state its running time. There will obviously be ways to optimize it but we are not looking for any optimization for this problem.

(d) (4 points) For this problem, you are asked to find the inorder successor of an input node u. For convenience you may assume that $u.right \neq \textbf{nil}$ (if $u.right = \textbf{nil}$, u does not have an inorder successor). As before, you will present a pseudocode for ROTATESUCCESSOR(T, u) but one that invokes only LEFTROTATE. Your algorithm will contain exactly four lines of code.

Do not forget to justify the correctness of your algorithm and state its running time. There will obviously be ways to optimize it but we are not looking for any optimization for this problem.

(e) (**Extra Credit**)(6 points) For this problem, you wish to search for the node that contains x as its key. More importantly, you wish to rotate the tree such that the target node (i.e., the node with x as its key) becomes the new root of the tree. You may assume that this target node exists.

Present a recursive algorithm ROTATESEARCH(T, x) that takes as input T and a value x and returns a tree T' such that $T'.root.key = x$. Your algorithm may only invoke SEARCH and ROTATE.

Do not forget to justify the correctness of your algorithm and state its running time. There will obviously be ways to optimize it but we are not looking for any optimization for this problem.

Problem 6-2 (Comparing Binary Search Trees) 13+5 points

(a) (5 points) Let T be a given binary tree with n nodes. T does not have any of the *.key* values initialized. You are also given an input array $A = [1, \ldots, n]$. Prove that there is a *unique* way to assign elements in $A[i]$ to *.key* fields of nodes in T such that T is now a binary search tree. You will prove this by presenting a recursive algorithm FILLTREE$(T, u, A, start)$ that fills the subtree rooted at u with values beginning from $A[start]$ in A. Further, your algorithm will return a value s such that $A[start, s-1]$ were used to completely fill the subtree rooted at u.

Do not forget to justify the correctness of your algorithm and state its running time. There will obviously be ways to optimize it but we are not looking for any optimization for this problem.

(b) (4 points) You are given a binary search tree T of n nodes (with $n = 2^m - 1$ for some integer $m \geq 1$). You need to transform T to a perfectly balanced search tree T'. Give an algorithm TransformBalanced(T, n) that returns a balanced search tree T' which is equivalent to T. Two search structures T, T' are said to be equivalent if they store exactly the same set of items. You may use additional storage of size n, along with the necessary storage needed to store T'.

Do not forget to justify the correctness of your algorithm and state its running time. There will obviously be ways to optimize it but we are not looking for any optimization for this problem.

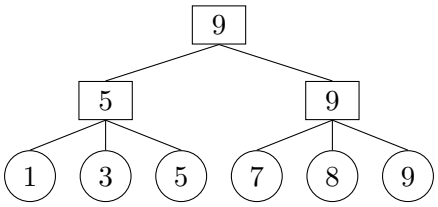
(c) (4 points) You are given two binary search trees T, T' of size n. The goal is to test if T and T' are equivalent or not. Design an algorithm IsEquivalent(T, T', n) which either returns **false** or **true**. Your algorithm cannot modify T or T' and may only use an additional storage of size n.

(d) (**Extra Credit**)(5 points) In part (b), you transformed a given binary search tree T into a complete binary search tree by using an additional $O(n)$ storage. Here, we want to transform T into a tree T_L which is a left-sided tree (which exists by part (a)). Recall that a left-sided tree is one where no node has a right child. However, you are only constrained to use Rotate (as defined in Problem 1) and use a constant amount of space. Design an algorithm TransformLeft(T, u) that transforms T into a left-sided tree T_L.

Do not forget to justify the correctness of your algorithm and state its running time. Present an upper-bound on the number of invocations of Rotate needed in the worst-case. There will obviously be ways to optimize it but we are not looking for any optimization for this problem.

Problem 6-3 (2-3 Trees) 18+2 points

Consider the following 2-3 tree T:



(a) (4 points) Show an element $x \notin T$ such that applying in sequence Insert(x) and Delete(x) will result with a tree T' that is different from T. Show the tree after the Insert operation and after the Delete operation.

(b) (4 points) Show an element $x \in T$ such that applying in sequence Delete(x) and Insert(x) will result with a tree T' that is different from T. Show the tree after the Delete operation and after the Insert operation.

Now, we will look at some applications of 2-3 Trees and augmented balanced search trees.

(c) (4 points) Let S be a set of n integers with many repeated values such that there are only $O(\log n)$ distinct values in S. Design an algorithm to sort this data in $O(n \log \log n)$ time. You may choose to describe the algorithm in words, as long as it is clear. Justify the correctness and running time of your algorithm.

(d) (6 points) Now, we will look at an application of using 2-3 Trees. You are asked to design a data structure for a book store and you would like to use a 2-3 Tree. The book store's requirements are as follows:

- The store wants to keep track of its books, indexed by distinct book id.
- The record contains the book number, book name, and a price field.
- Create a book in $O(\log n)$ time.
- Close a book in $O(\log n)$ time, i.e., if it is out of print, remove it from the system.
- Add (or subtract) from the price of given book number.
- Report the book name of the most expensive in $\Theta(1)$ time.
- Report the price of a given book number in $O(\log n)$ time.
- (**Extra Credit**)(2 points) Report the book name with the largest balance whose book id is in range $(low, high)$ in $O(\log n)$ time.

Here n is the total number of books. You can present your answer in words. Clearly describe the 2-3 Tree and the indexing operation. Further, map the operations to corresponding algorithm(s) of a 2-3 Tree, if such a mapping exists. You will use additional fields to solve the problem and do not forget to state how the correctness of the augmented fields are maintained. Justify the running time of your algorithm and also the correctness of the operations.

(**Hint**: You will need to augment the data structure where each internal node contains the largest balance in the subtree rooted there. Clearly describe how this augmenting is maintained - specifically when you update, delete, or insert.)