CSC265 Fall 20120 Homework Assignment 2

The list of people with whom I discussed this homework assignment: Kunal Chawla.

(a) Explain how to insert an element into a leaf-oriented binary search tree.

Solution

We can insert an element x into a leaf-oriented binary search tree L in the following way (for details see pseudocode):

1. If L is empty, then simply set x to be the root node.

2. If L consists of a single node y, then push $y.val$, together with x down to the children of node y. Put $y.val$ on the left children if y's key is less than that of x, otherwise put x on the left. Let the key of node y be the arithmetic average of x and y's keys.

3. If L's root has children (last condition guarantees that there are two), then recurse on the correct children depending $x.key$. Recurse on the left if $x.key \leq L.key$, otherwise do it on the right.

INSERT-BASIC(L, x)

```
1  if L.left == NIL
2      if L.key == NIL
3          L.key = x.key
4          L.val = x
5      else
6          L.key == (L.key + x.key)/2
7          if L.key ≤ x.key
8              L.left.key = L.key
9              L.left.val = L.val
10             L.right.key = x.key
11             L.right.val = x
12         else
13             L.left.key = x.key
14             L.left.val = x
15             L.right.key = L.key
16             L.right.val = L.val
17 else
18     if x.key ≤ L.key
19         INSERT-BASIC(L.left, x)
20     else
21         INSERT-BASIC(L.right, x)
```

Note that this version of insertion can be easily modified so that the parent of each node is updated, so assume this is done for the next question.

(b) If L is a LOBS and you insert an element into L using your algorithm from (a), what properties can be violated?

Solution

Since the notion of colour is nonexistent in (a), we shall consider two cases.

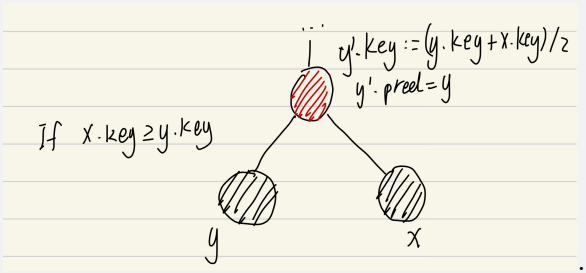
1. If we assume the inserted nodes are always coloured red, then we will violate properties 2 (root is red), 3 (leaves are red), 5 (predecessor is not stored), 7 (every node is red), and 9 (every node is red).

2. If we assume the inserted nodes are always coloured black, then we will violate properties 5 (predecessor is not stored) and 6 (tree might not be balanced).

(c) Give an $O(\log n)$ time algorithm to perform INSERT(L, x), where L is a LOBS, n is the number of elements in the dictionary, and x is a pointer to a new node containing a key and a value, whose colour is black, and whose parent pointer is NIL.
Use diagrams and English explanations, as in class, rather than pseudo-code.
In a clear, organized way, describe all the cases that can arise, describe what transformations can be performed so that the resulting tree is a LOBS, and explain why the tree resulting from your transformations satisfies all of the required properties of a LOBS.

Solution

Suppose we insert x with the method outlined in (a). It is clear that we should make the newly created nodes black, since leafs need to be black by property 3. If T is empty, then there is nothing left to do. If not, then by the base case of INSERT-BASIC (lines 1 to 16), we know that there must be a node y' under which we created two children nodes. One of them contains x and the other contains the content formerly in node y'



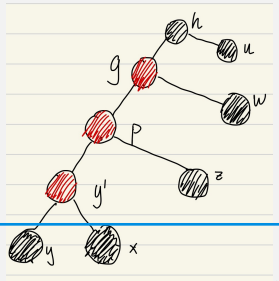
Note that we "must" colour y' red since we do not want to increase the number of

black nodes from the root to the newly added leaves while leaving the other paths unchanged. We consider cases.

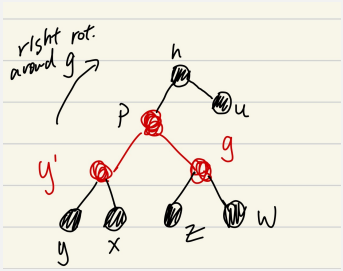
1. If $y'.par ==$ NIL, then this means y' is the root of T. So simply color y' black.

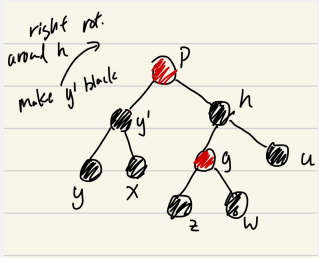
2. If y' is a child, there are a few cases.

   (a) $y'.par$ **and** $y'.par.par$ **are both red**. Then we must have the following:



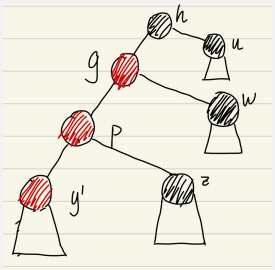
   Note that h must be black since h being red contradicts property 9. Also, z, w, u are necessarily single black nodes (with no children) by property 6 and the fact that there was only two black nodes from h to y'. In this case, we apply a rotation around g, a right rotation around h, and make y' black.





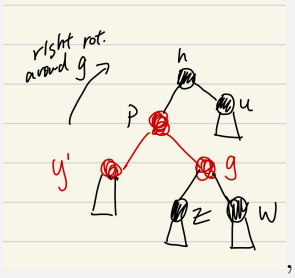
   This solves all violations except of property 9, since the parent of this subtree might be red. To solve this, we "recurse" using 2b.

   (b) **Previous case extended**. The tree would look like the following:



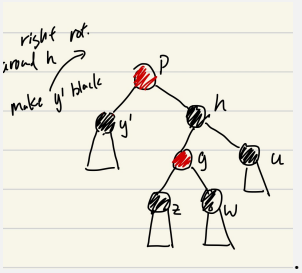
Denote the number of black nodes from the root r to the leaves as $d(r)$. We can assume that $d(y') = d(z) = d(w) = d(u)$ in this tree, since coloring at insertion as defined in the first paragraph does not change $d(y')$. It is clear that $d(h) = d(y') + 1$ since h is black.

To handle this, we perform operation similar to the one defined in 2a: a right rotation around g



,

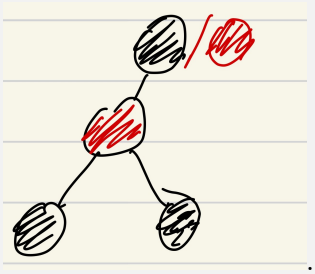
followed by right rotation around h and colouring node y' black



.

Note that $d(p)$ now is the same as $d(h)$ was previously. The only possible violation now is property 9, which can be solved by applying this procedure until it is resolved.

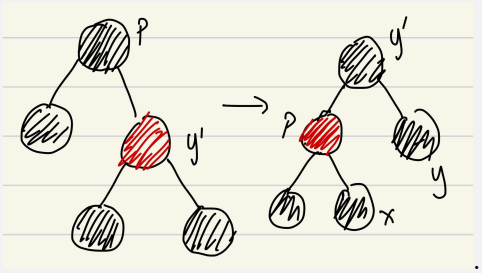
(c) **y' is a left child and ($y'.par$ is black or $y'.par.par$ is black)**


.

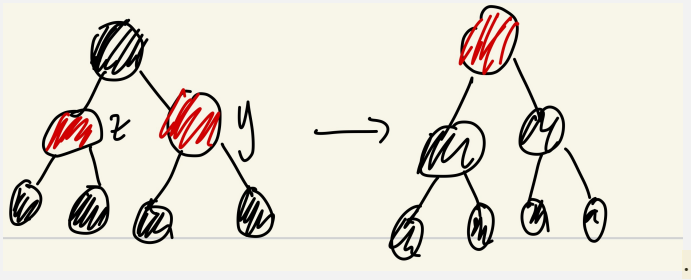
This case does not violate any property.

Note that we have already covered all cases where y' is the left child.

(d) y' **is a right child of a black node** p. One case here is that p must has a black left child

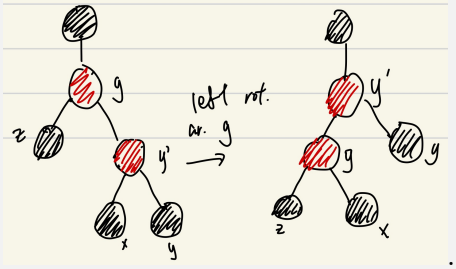


This child must not have children since y' used to be a black child and all root-to-leaf paths have the same number of black nodes. In this case, we rotate right around p and swap the colours of p and y'. Another case is



In this case, we simply make y, z black and p red. To resolve the problem of three red consecutive nodes, we could proceed to 2b.

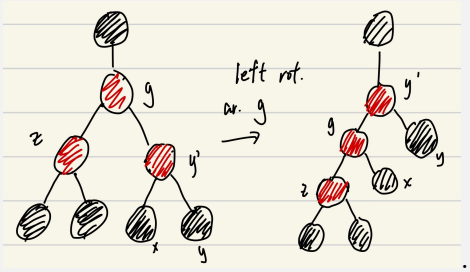
(e) y' **is a right child of a red node** g **and** g **has a black left child and black parent**



In this case, we simply rotated left around g.

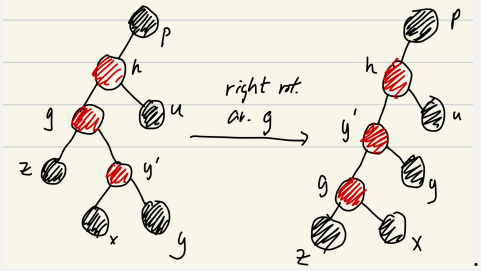
(f) y' **is a right child of a red node** g **and** g **has a red left child and a**

black parent



In this case, we simply rotated left around g to get 2a.

(g) **y' is a right child of a red node g and g has a red parent h**



We know that h must have a black right child u as a leaf by properties 6 and 7. And the grandparent of g must be black by property 9. To resolve this, rotate left around g to get 2a.

Finally, we update the predecessors. Notice that rotations do not change the predecessors, since the tree can be viewed as a set of totally ordered elements and tree rotation preserves that order. We set the predecessor of y' to the smaller node of the two newly created ones. Then, we iterate up the tree until the current node is the left child of some node p, we then set p's predecessor to the smaller node of the two.

(d) Explain why your algorithm runs in $O(\log n)$ time.

Solution

Our goal is to show that the height of a LOBS is in $O(\log n)$, where n is the total number of nodes.

We prove that a LOBS of height h has at least $2^{\lfloor \frac{1}{3}(h+1) \rfloor} - 1$ nodes. For a subtree x of a LOBS, define the **black height**, $\text{bh}(x)$, as the number of black nodes from an x-to-leaf path, excluding x. First, we show that any subtree x has $2^{\text{bh}(x)} - 1$ nodes by induction

on $\text{bh}(x)$. Clearly, for $\text{bh}(x) = 0, 1$, this holds. Let $\text{bh}(x) > 1$. By property of LOBS (prop. 6), the left and right subtrees of x must have at least black height $\text{bh}(x) - 1$. By induction hypothesis, we know x has at least $2(2^{\text{bh}(x)-1} - 1) + 1 = 2^{\text{bh}(x)} - 1$ nodes. This completes the induction.

We also know that for a tree x of height h, there exists a path of length $h + 1$. By property 9, at most $\lfloor \frac{2}{3}(h+1) \rfloor$ nodes are red. So we know there are at least $\lfloor \frac{1}{3}(h+1) \rfloor$ black nodes. This means $\text{bh}(x) \geq \lfloor \frac{1}{3}(h+1) \rfloor$. Hence, by the result above, we know that a LOBS x has $n \geq 2^{\lfloor \frac{1}{3}(h+1) \rfloor} - 1$ nodes.

Solving for n gives $h \leq 3(\log_2(n+1)) + 1 \in O(\log n)$.

Knowing this, it is almost immediate that our algorithm is $O(\log n)$. Inserting the element involves binary search, which is done in $O(\log n)$ given our height. Fixing colours and predecessor involves going up the tree, which is also $O(\log n)$.