CSC265 Fall 2020 Homework Assignment 2

The list of people with whom I discussed this homework assignment: JiaWei Chen

A *leaf-oriented binary search tree* is an implementation of a dictionary in which the elements in the dictionary are stored at the leaves of a binary search tree. The keys of non-leaf nodes are only used for searching. They may or may not be the keys of elements in the dictionary. Every non-leaf node has exactly two children.

A LOBS is a leaf-oriented binary search tree with the following properties:

1. Every node is coloured red or black.
2. The root is black.
3. All leaves are black.
4. Every leaf has a key, a value, a colour, and a pointer to its parent.
5. Every non-leaf node has a key, a colour, and pointers to its left child, right child, parent, and the rightmost leaf in its left subtree.
6. The number of black nodes on all root-to-leaf paths is the same.
7. Every non-leaf node has a black right child.
8. Every non-leaf node has a left child, which is either black or red.
9. Any root-to-leaf path contains at most two consecutive red nodes.

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

If the tree was empty, put the element as root, colored black. Else, first, insert the node into the tree as if it is a normal BST, uncolored. Name its parent p. Then, create a new node with its value equal to the mean of the first new node and its parent, rounded up. Then, replace the value of p with the second new node, name this node p_2, colored red, and put the old value of p and the inserted element as its children, in the order maintaining BST property, both colored black. The parent of p_2 is the same as p before the insertion. At last, recolor and rotate the tree if some properties are violated, through methods in part c.

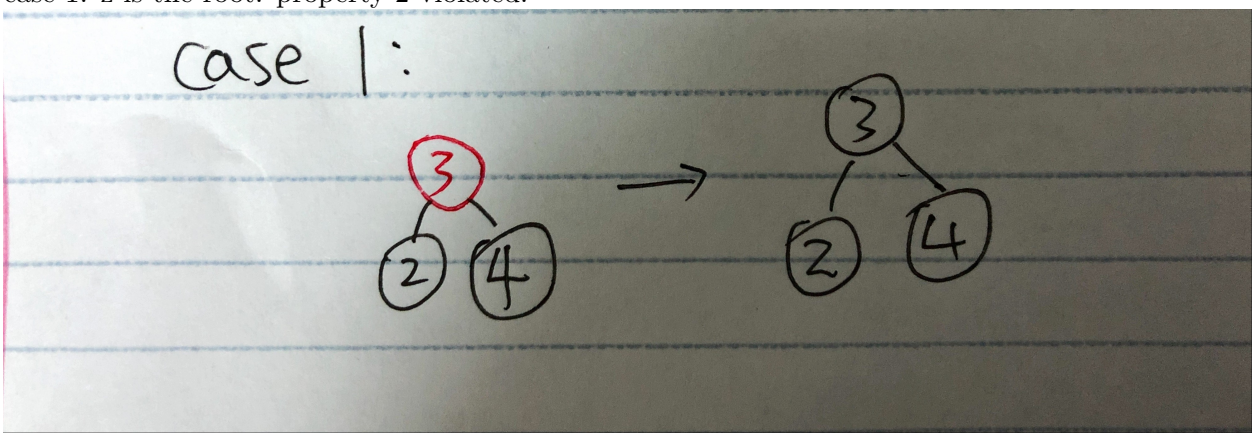
(b)f If L is a LOBS and you insert an element into L using your algorithm from (a), what properties can be violated?
for my algorithm in part a only:
(1) will not be violated. since the first added node and the original leaf are black, and the second added node is red.
(2) can be violated if the tree has only one or two nodes.
(3) will not be violated since the inserted element and the original leaf, which are the new leaves, are black
(4) will not be violated since the new leaves' parent is the second added node. Second added node's parent is same as the original p.

(5)the three changed nodes satisfy this. For the second added node's parent, it has a new left/right child as the node, and inorder predecessor (rightmost leaf in left subtree) pointing to one of the leaves of the three changed nodes. For all other nodes, Everything is unchanged except for maybe the inorder predecessor, which will be the same as the second added node's parent if it changed.
(6) is unchanged. The three new nodes has a black height of 1, same as the original black leaf, so the black height for simple paths the two new leaves does not change. For all other leaves, the path didn't change, so the black height is the same.
(7) can be violated if the second created node, which is colored red, is the right subtree.
(8) will not be violated since the second created node is either black or red if it is left subtree.
(9) can be violated if there were two consecutive red nodes for the path to second created node's parent, then there will be three consecutive red nodes.

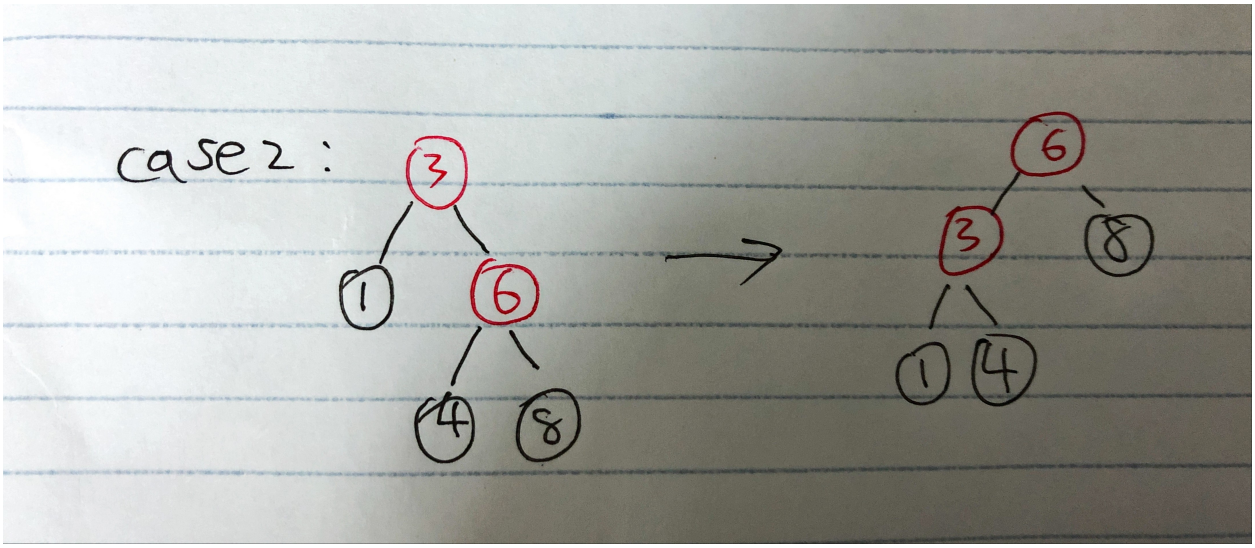
(c) Give an $O(\log n)$ time algorithm to perform INSERT(L, x), where L is a LOBS, n is the number of elements in the dictionary, and x is a pointer to a new node containing a key and a value, whose colour is black, and whose parent pointer is NIL.
Use diagrams and English explanations, as in class, 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.

First do the operations in part a before "at last". If the new tree doesn't satisfy the violations, then the insert is done. Now consider the new tree's violations. We denote the second created node node z, which is an internal node, its parent p, and its grandparent g. We have a special state "doubly red" after some of the cases, and all doubly red cases need to continue be handled. The diagrams only depict the nodes related to the cases, and all nodes maintain the same parameters unless they are changed in the diagrams. Since doubly red always happen when g is red, it is always the left subtree.

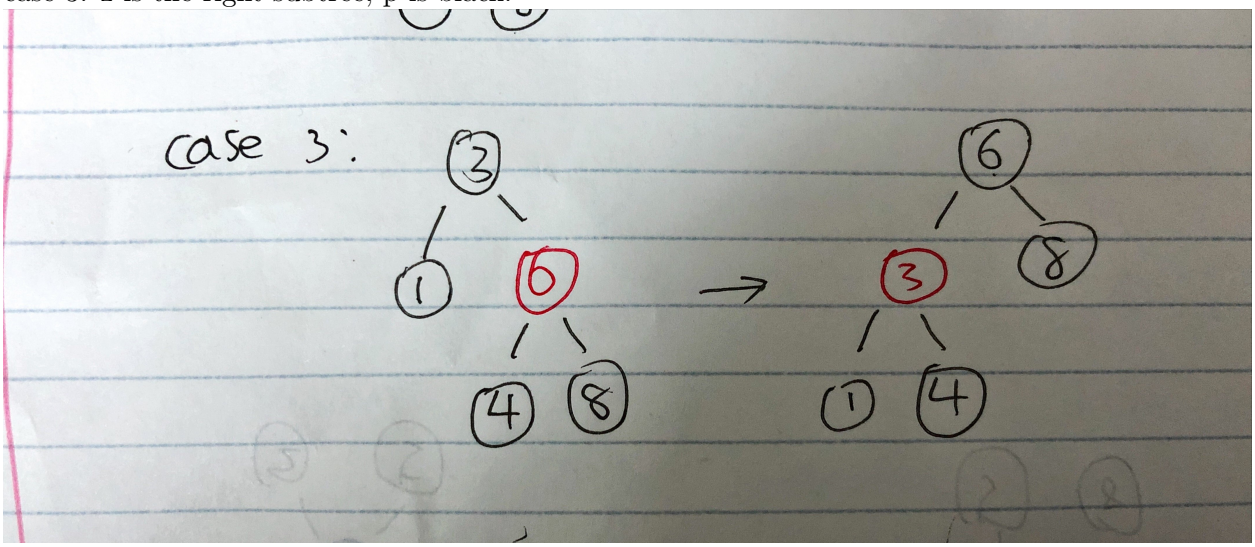
case 1: z is the root. property 2 violated.



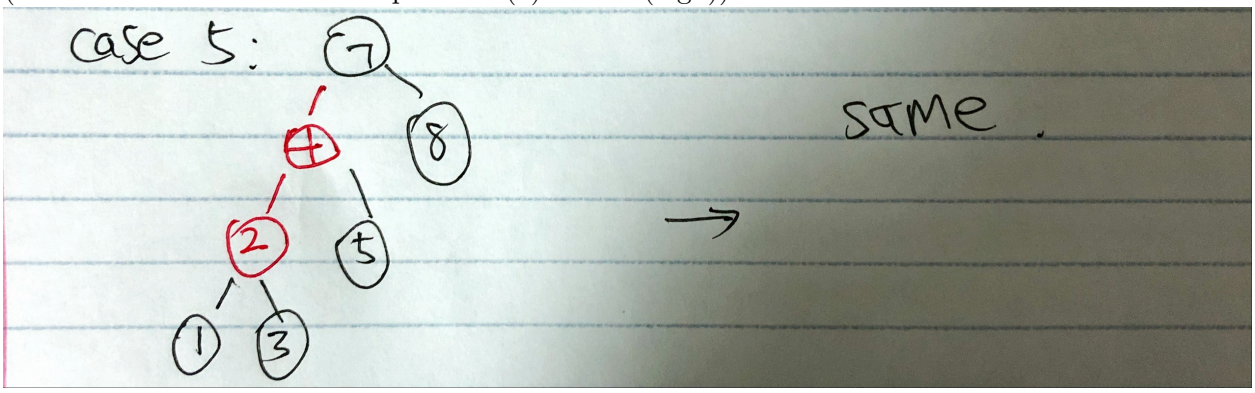
case 2: z is the right subtree, p is red. property 7 violated.



case 3: z is the right subtree, p is black.



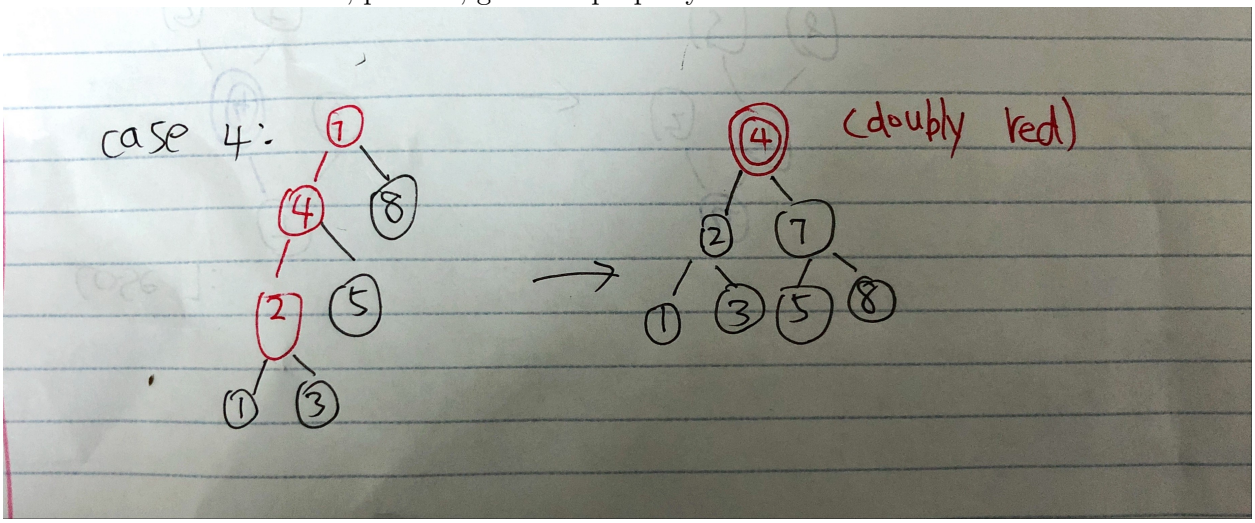
case 4: z is the left subtree, p is red, g is black. no property violated.
(I switched cases 4 and 5 to separate O(1) from O(logn))



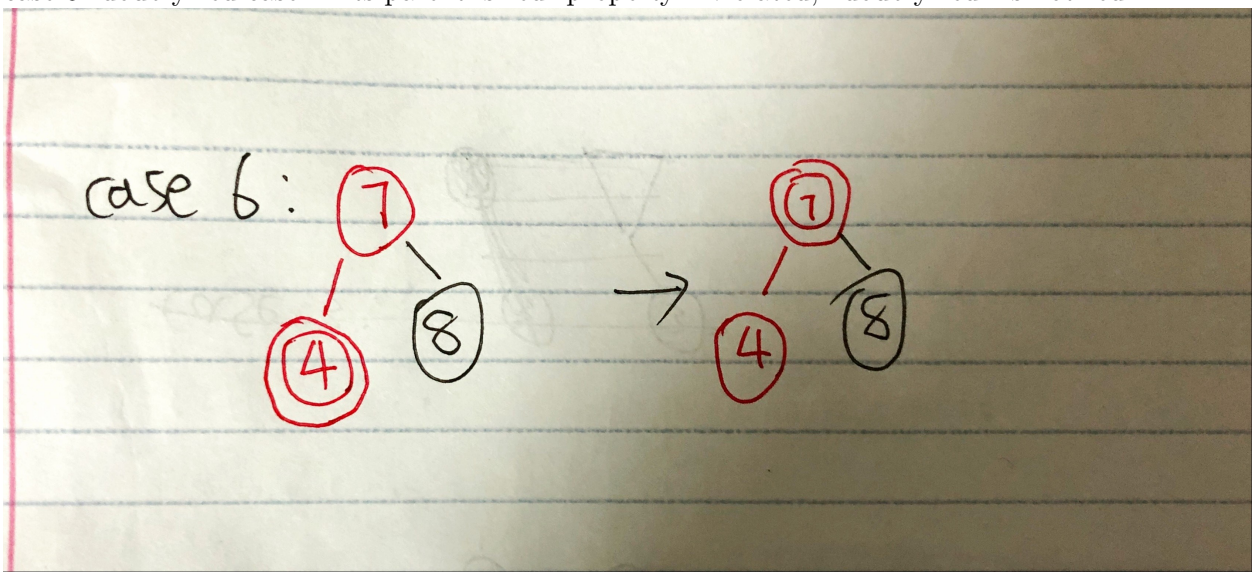
The above 4 cases changes neither the black height nor the color of p, so they are all O(1)

The 3 cases below are iterative, and each iteration they go up by 1 level.

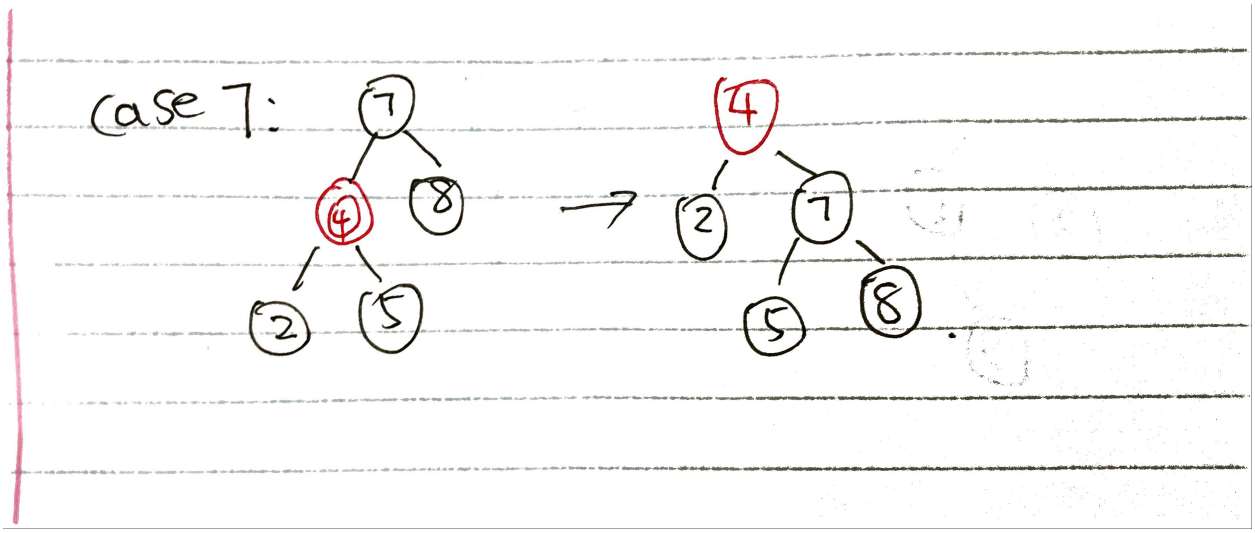
case 5: z is the left subtree, p is red, g is red. property 9 violated.



case 6: doubly red case 1: its parent is red. property 1 violated, "doubly red" is not red.



case 7: doubly red case 2: its parent is black. property 1 violated, "doubly red" is not red.



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

We first prove a lemma about a tree's height h with respect to n.

any sub-tree with root x has $2^{bh(x)}$ internal nodes. base case: a tree of height 0 has 0 internal nodes.

inductive step: every child of x has at least $2^{bh(x)-1}$ nodes, so by induction x has $2^{bh(x)}$ internal nodes.

For any path from root to leaf, since there can't be 3 consecutive red nodes, there are at least n/3 black nodes.

Therefore, $n \geq 2^{h/2} - 1$ or $h \leq 3lg(n+1)$

The operation in part a takes constant O(1) time. Now consider the operations in part c.

cases 1-4 are changes neither the black height nor the color of p, so they are all O(1).

now consider cases 5-7. Case 5 is takes O(1) time itself, however, it creates "doubly red" on g, which is 1 levels above p, and needs to go to cases 6 and 7. Case 6 is O(1), and it shifts "doubly red" up by 1 level. Case 7 is O(1), and it resolves doubly red. However, it changes the color of g to from black to red, which might go to cases 1-5, but in case 7 , in the worst case, it still shifts the node that needs resolve 1 level up.

If case 5 is not triggered, then cases 6/7 wont appear. If it is triggered, then for cases 5 and 6, each case take O(1) to handle, and after handling, the abnormal node shifts up by 1 level, and this can happen at most 3lg(n+1) times. In case 7, it might lead to cases 1-5, for 1-4 it has O(1) time, and even if it leads to case 5, the red node still shifts up by 1 level, and this can happen at most 3lg(n+1) times.

Therefore, each case takes O(1) to resolve, and this can happen at most h times, which is $O(\log n)$ time.