CS 61B
Summer 2020

Small Group Tutoring
Section 7: BSTs and Balanced Trees

Worksheet 7

1 Runtime

Provide the best case and worst case runtimes in theta notation in terms of N, and a brief justification for the following operations on a binary search tree. Assume N to be the number of nodes in the tree. Additionally, each node correctly maintains the size of the subtree rooted at it. [Taken from Final Summer 2016]

boolean contains(T o); //Returns true if the object is in the tree

Solution: Best: $\Theta(1)$ Why: If the object is at the root.

Worst: $\Theta(N)$ Why: If the object is at the leaf of a spindly tree.

void insert(T o); //Inserts the given object.

Solution: Best: $\Theta(1)$ Why: One example may be inserting to the left child of the root of a right leaning spindly tree.

Worst: $\Theta(N)$ Why: One example may be inserting to the leaf node of a right leaning spindly tree.

T getElement(int i); //Returns the ith smallest object in the tree.

Solution: Best: $\Theta(1)$ Why: One example may be if i = 1 and the tree is a very spindly right leaning tree.

Worst: $\Theta(N)$ Why: One example may be if i = N and the tree is a very spindly right leaning tree.

2 Pruning Trees

Assume we have some binary search tree, and we want to prune it so that all values in the tree are between L and R, inclusive. Fill out the method below that takes in a BST, as well as L and R, and returns the pruned tree. Note that the root of the original tree might not be between L and R, so make sure you return the root of the new pruned tree.

```
class BST {
    int label;
    BST left; // null if no left child
    BST right; // null if no right child
}

public BST pruneBST(BST root, int L, int R) {
    if (root == null) {
        return null;
    } else if (root.label < L) {
        return pruneBST(root.right, L, R);
    } else if (root.label > R) {
        return pruneBST(root.left, L, R);
    }
    root.left = pruneBST(root.left, L, R);
    root.right = pruneBST(root.right, L, R);
    return root;
}
```

3 All About Trees

1. Why does a binary search tree have a worst case runtime of $\theta(n)$ for *contains*?

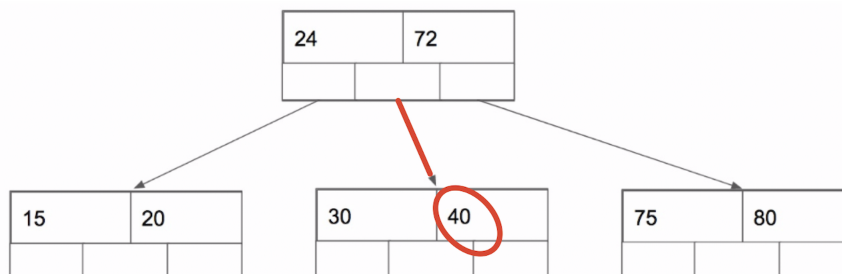
   If the search tree is not bushy, i.e. is just a line of nodes, we get very poor performance.

2. Give a sequence of operations, such that if they were inserted in the order they appear, would result in a "poor" binary search tree.

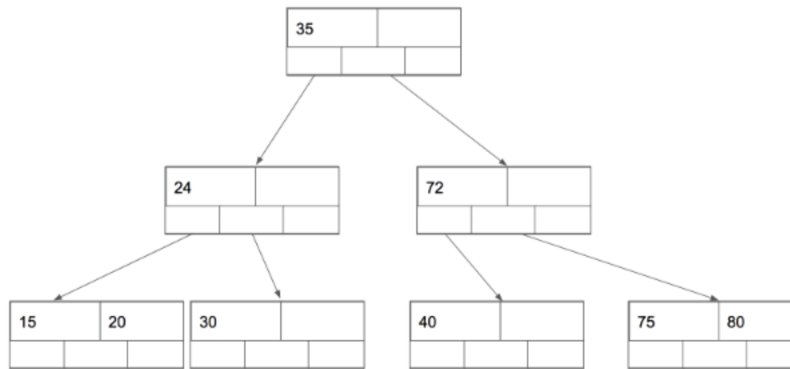
   Any increasing sequence will work. For example, [1, 2, 3, 4, 5].

3. Examine this B-tree with order 3. Mark the paths taken when the user calls $contains(40)$.

   We examine the root node and see that 40 is greater than 24 and less than 72, so we take the middle edge to the child node. We examine this node and find 40.



4. Now call *insert*(35), and draw the resulting tree.



5. What property of a B-tree rectifies problems of binary search trees, such as the one in 1.1? Why would you not use a B-tree?

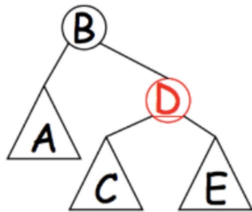
   B-trees are balanced - because we always split nodes on insertion and move keys upwards in the tree, we ensure we never get the ”long tail” of nodes that can occur in a normal binary search tree. That ensures we get $O(logn)$ performance. Another benefit is that because we store more elements at a node, we have to do fewer traversals in the tree. B-trees are significantly more complicated than binary search trees. Red black trees provide a way for us to implement B-Trees in a simpler way, without losing the advantages of the B-Tree.

4 The Holy LLRB Invariant

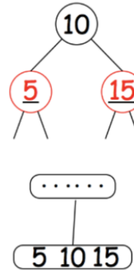
RB Tree Invariants: Node labels are in order from left to right. All paths through the tree contain the same number of black nodes. No red nodes have red parents. As a result, the height of a RB tree with n nodes is $O(logn)$.

LLRB trees must also maintain the following invariant (in addition to the regular red-black invariant):

No right-leaning trees (black parent with right red child):

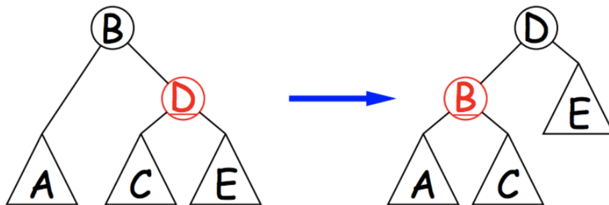


No "4-nodes" (black parent with two red children):

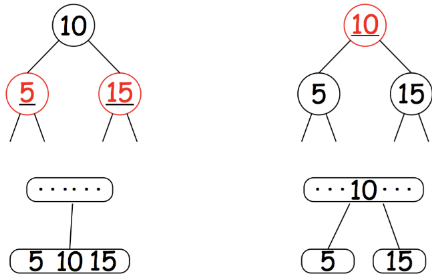


1. What are the "fixups" for the two cases above in order to preserve the LLRB invariant (i.e. what operations do we perform on each tree to ensure it is a proper LLRB)?

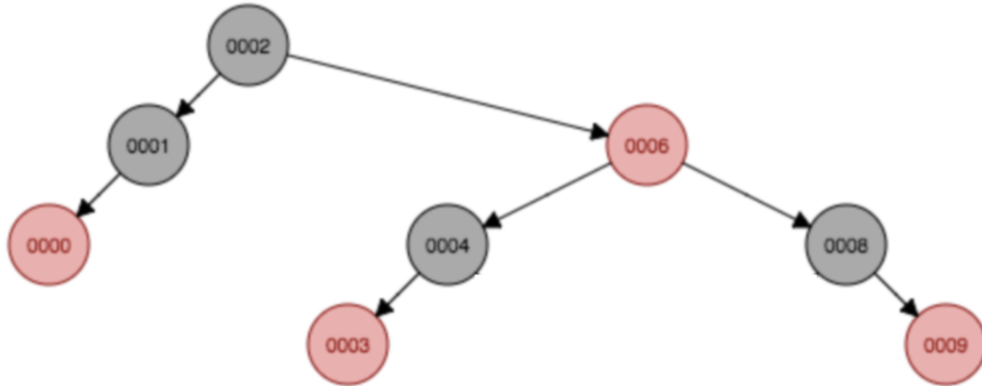
   Fixup 1 (for the left tree) is to rotate left on B and recolor, making our tree left leaning:

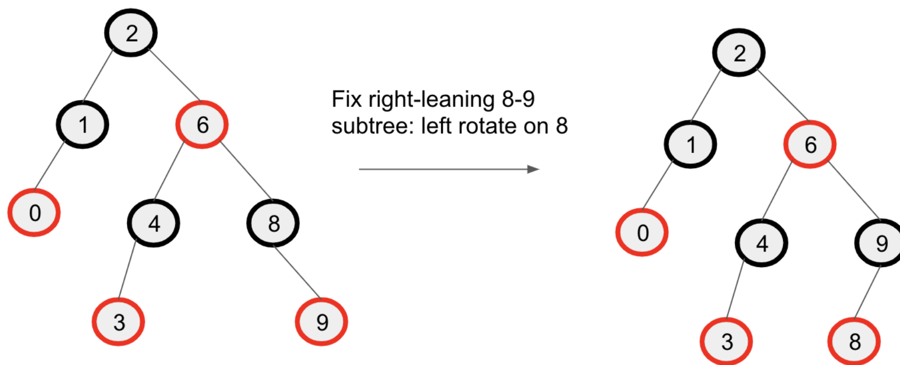
   Fixup 2 (for the right tree) is to recolor both children black and make the parent red (if the parent node is the root node of a tree, then simply color it black too):

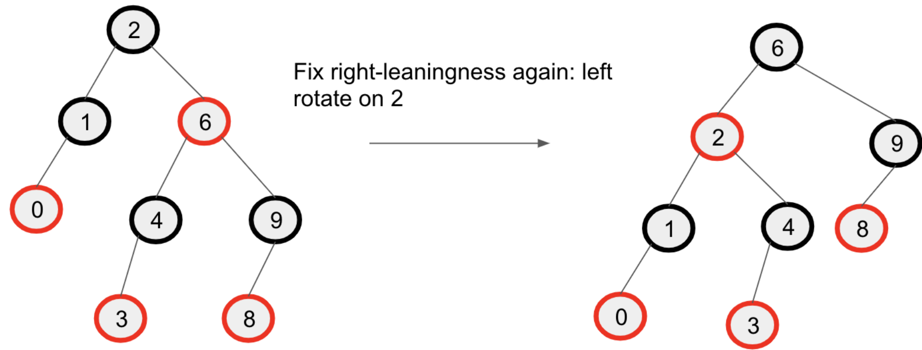


Consider the following RB tree:

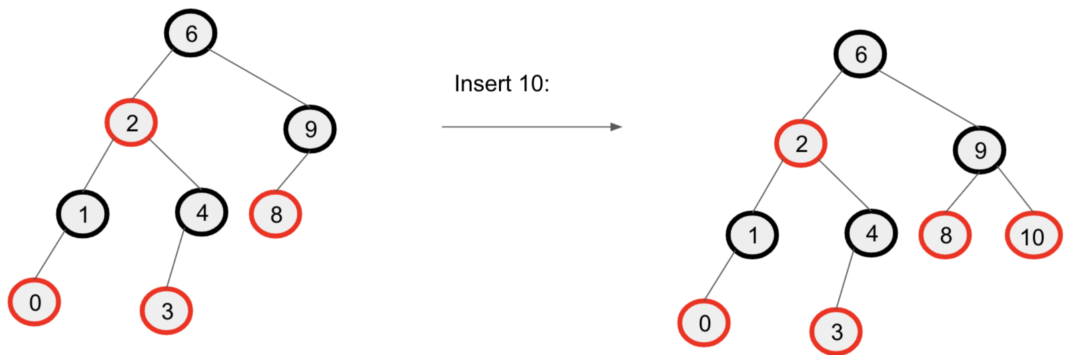


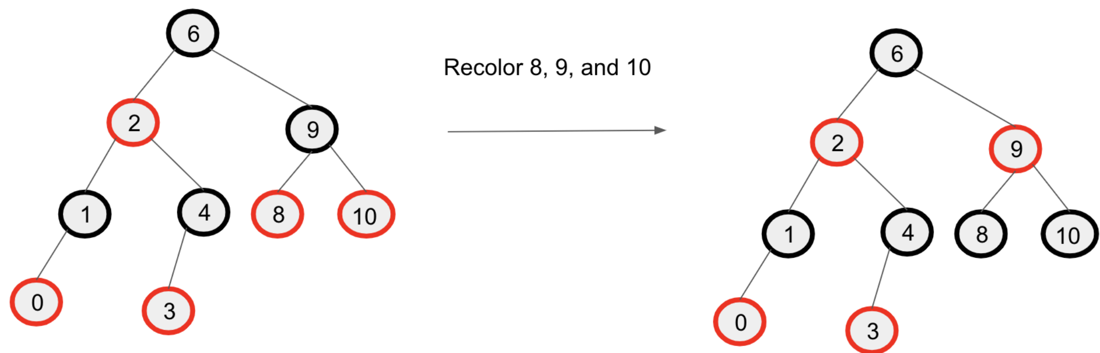
2. Draw the tree after applying all necessary fixups to make it a proper LLRB tree.

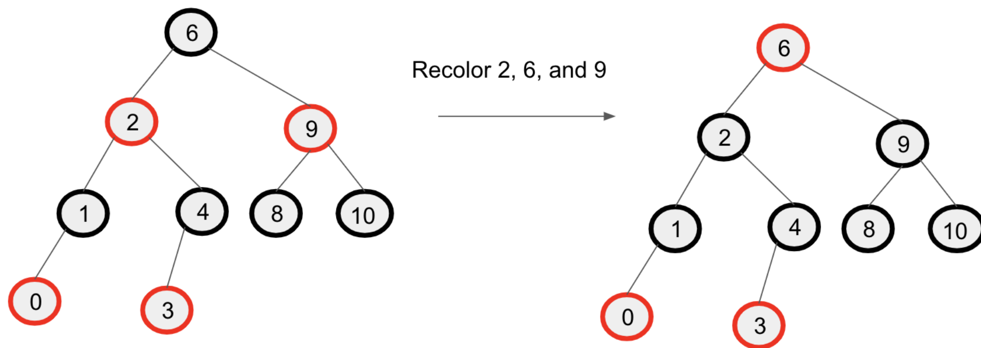


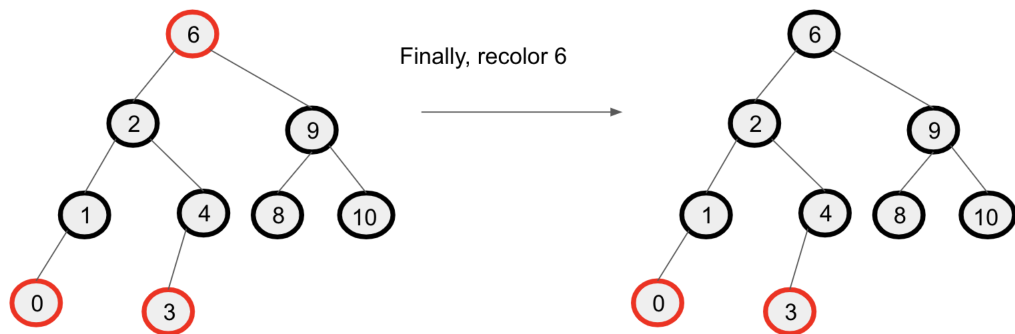


3. Next, insert 10 into the tree, and apply all fixups to preserve the LLRB invariant.

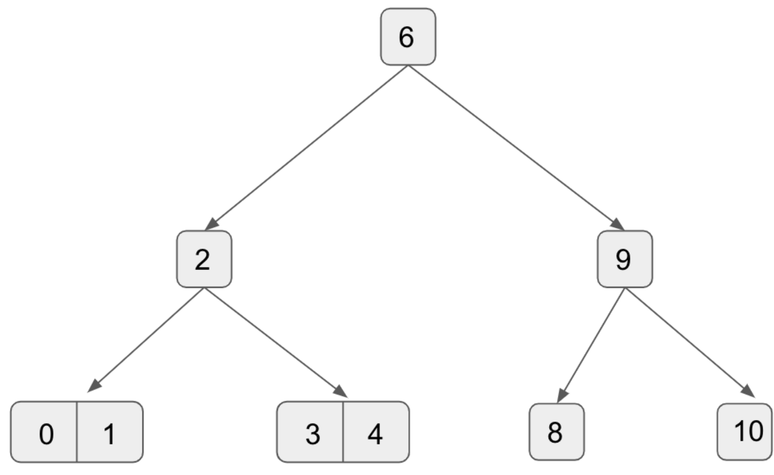








4. Now draw the corresponding 2-3 tree.


6
2
9
0
1
3
4
8
10