CS 61B Spring 2020 | Small Group Tutoring Section 10: Balanced Search Structures | Worksheet 13

1 Introduction

1. Why does a binary search tree have a worst case runtime of $O(n)$ for $find$?

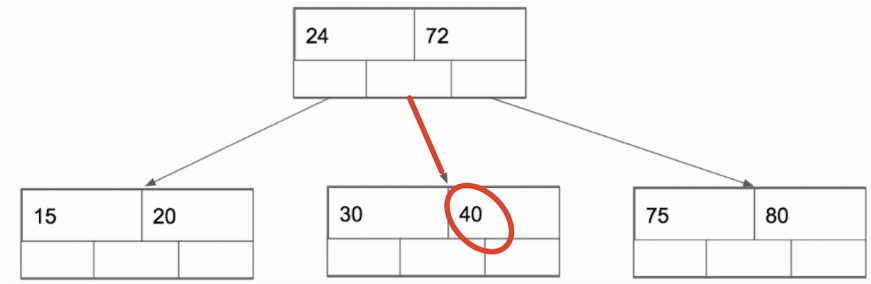
   If the search tree is not bushy, i.e. is just a line of nodes, we get very poor performance.

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.

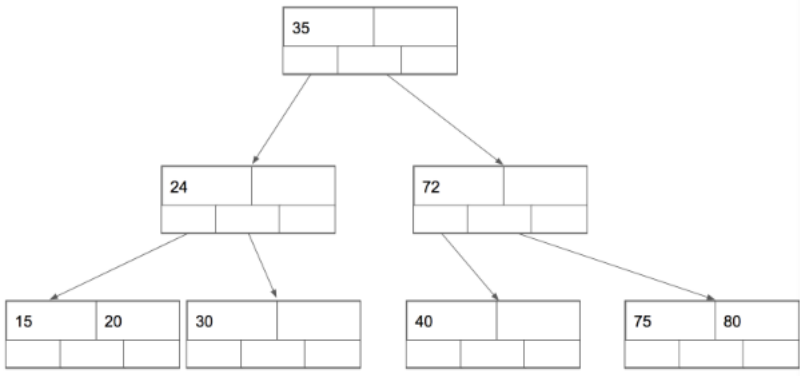
   Any increasing sequence will work. For example, [1, 2, 3, 4, 5].

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

   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

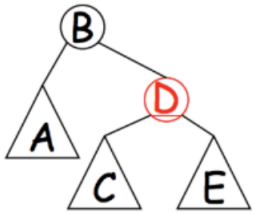
binary search trees. Red black trees provide a way for us to implement B-Trees in a simpler way, without losing the advantages of the B-Tree.

2 The Holy LLRB Invariant

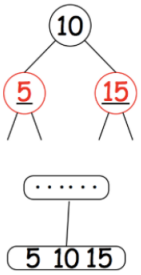
RB Tree Invariants: Node labels are in order from left to right. All paths through the tree contain the same number of black nodes. No red nodes have red parents. As a result, the height of a RB tree with n nodes is $O(logn)$.

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):

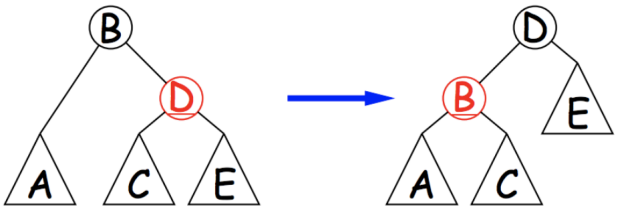


No “4-nodes” (black parent with two red children):

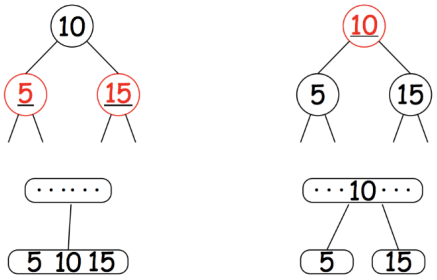


1. What are the ”fixups” for the two cases above in order to preserve the LLRB invariant (i.e. what operations do we perform on each tree to ensure it is a proper LLRB)?

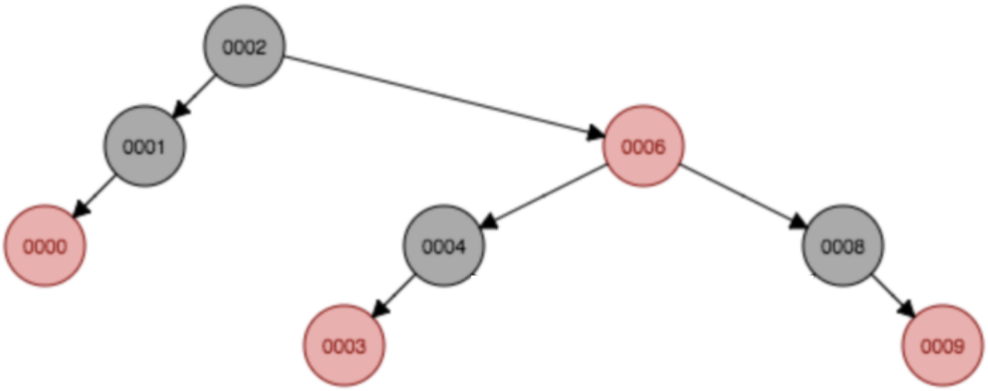
   Fixup 1 (for the left tree) is to rotate left on B and recolor, making our tree left leaning:



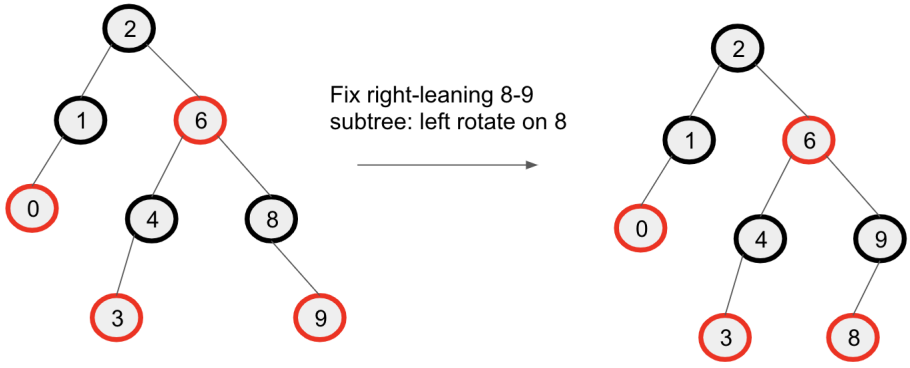
   Fixup 2 (for the right tree) is to recolor both children black and make the parent red (if the parent node is the root node of a tree, then simply color it black too):

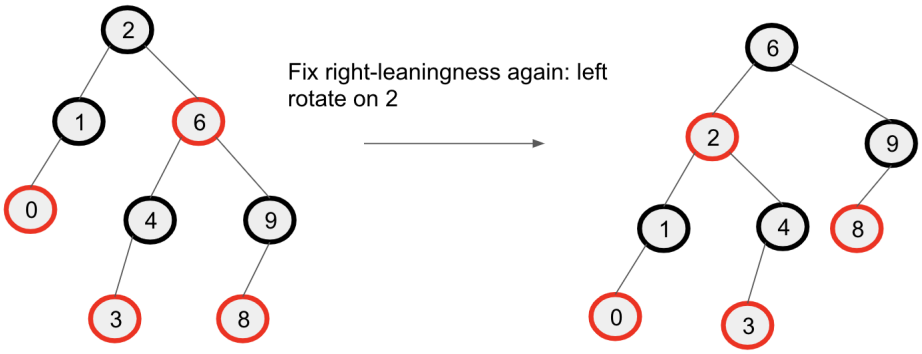


Consider the following RB tree:

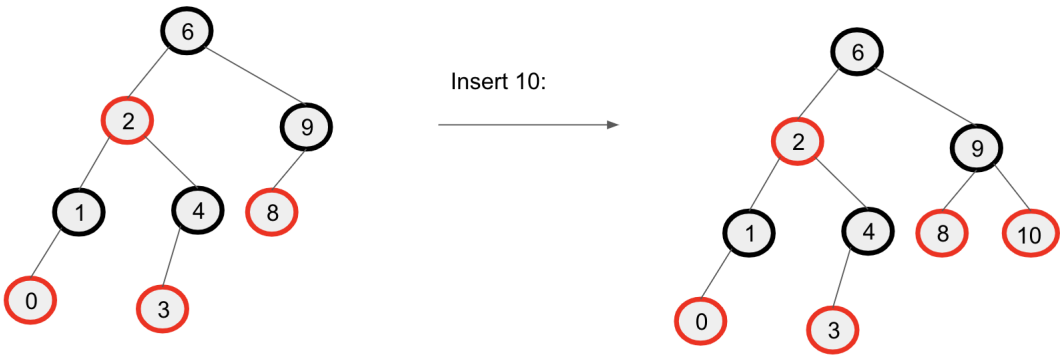


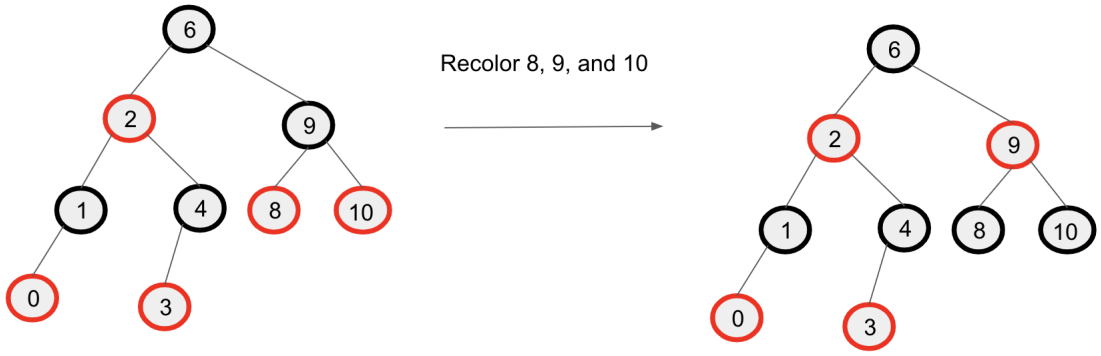
2. Draw the tree after applying all necessary fixups to make it a proper LLRB tree.

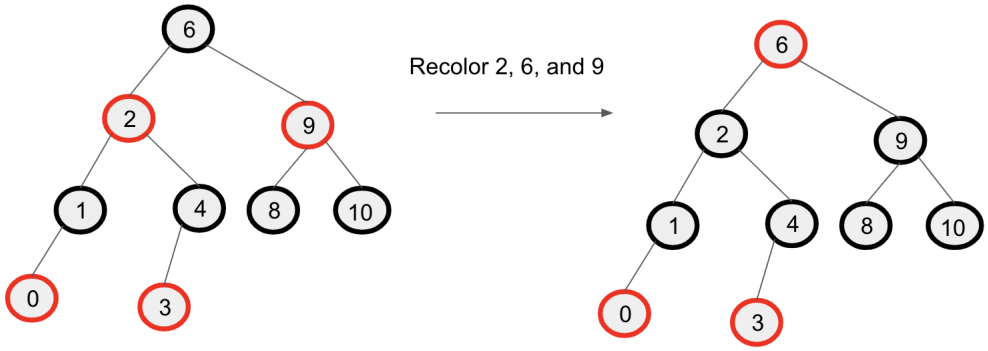


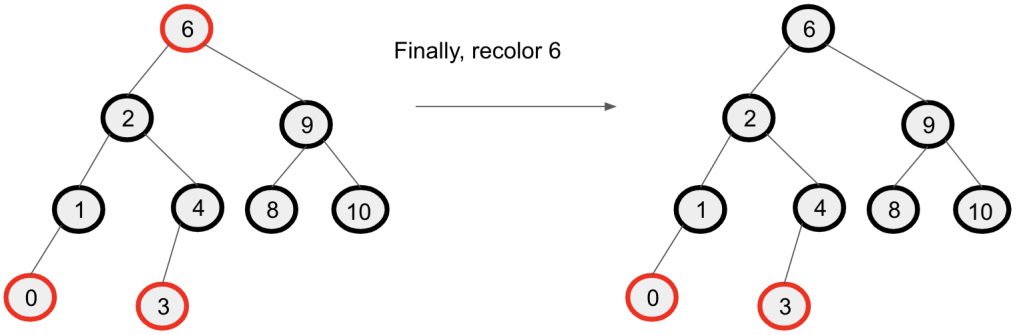


3. Next, insert 10 into the tree, and apply all fixups to preserve the LLRB invariant.

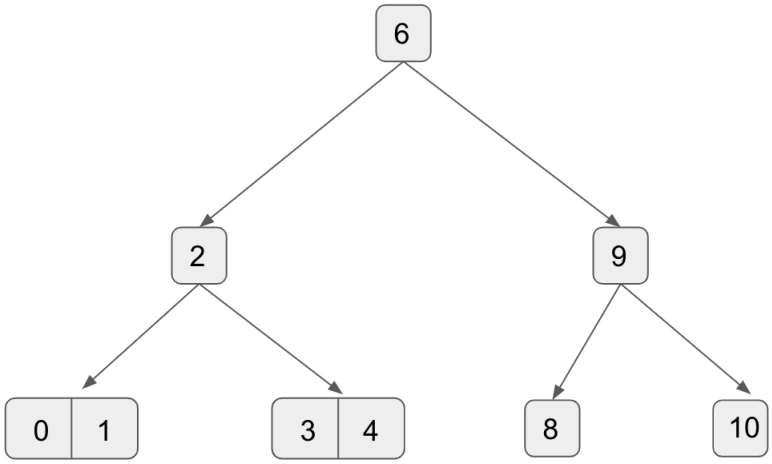








4. Now draw the corresponding 2-3 tree.


6
2
9
0
1
3
4
8
10

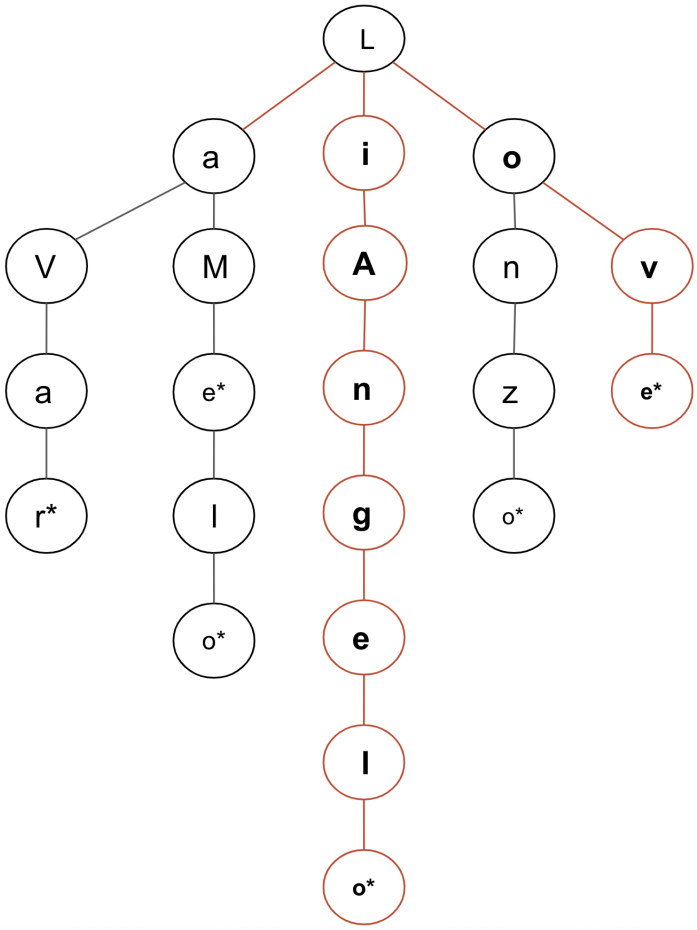
3 Trie me

1. The Biggest Baller of them all, CEO LaVar Ball, enjoys being reminded of who is a Big Baller. Remind him of who the Big Ballers are by finding all the words in the trie. Note: The nodes with an asterisk denote the end of a word.

   The words are: LaVar, Lame, LaMelo, Lonzo

2. Not again! LaVar Ball has forgotten about his son LiAngelo once again. Help LaVar by inserting "LiAngelo" and "Love" into the trie above so that no Big Baller is forgotten.

   The trie after inserting "LiAngelo" and "Love":



3. How long does it take to add n words, each of max length L?

   It takes $\theta(nL)$ to check whether a word of length L is in the trie.

4. Whats the best and worst case runtime to check whether a word of length L is in the trie?

   Best case $\theta(1)$: the first letter of the word is not in the trie.

Worst case: $\theta(L)$: the word is in the trie, and we have to traverse to the end of the word to confirm.