Red-Black Trees

Binary Search Trees were the first data structure we looked at for implementing Ordered Maps and Dictionaries. BSTs work well except:

- the height of the tree is dependant on the data and could be as bad as O(n) if the data is in sorted order when it is added to the tree.

2-4 Trees were an improvement over binary search trees for implementing Ordered Maps and Dictionaries because:

- we can guarantee the height of a 2-4 is O(log n) which means we can search, insert and delete in O(log n).

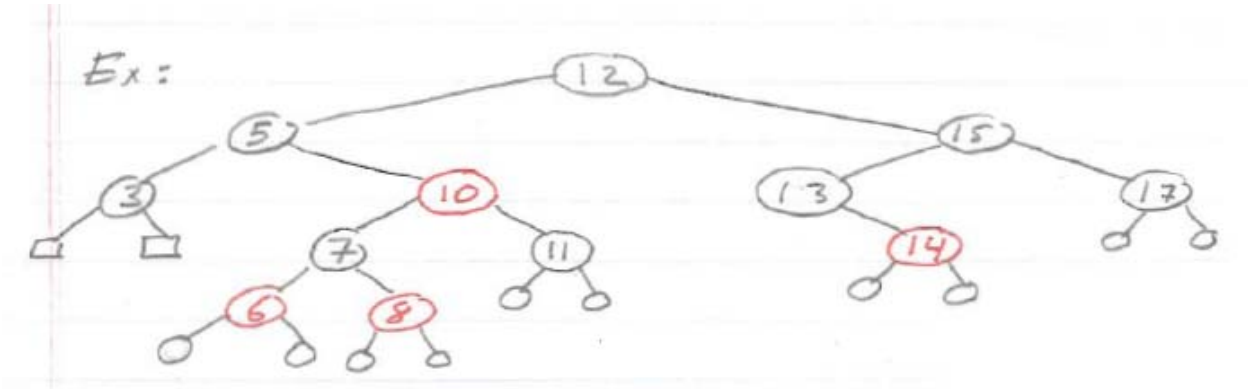
The drawback to 2-4 trees was:

- we had to deal with multi-entry nodes.

The next data structure we are going to look at is called a Red-Black Tree. They will give us the performance of a 2-4 tree with the simplified structure of a binary search tree. The cost of doing this will be in the complexity of the operations…not complexity in terms of running time, but complexity in terms of recognizing when the tree needs to be fixed and how to fix it.

A Red-Black tree is a binary search tree where each node has a color attribute (red or black). This attribute will be used to help manage the height of the tree.

Here's a sample tree:



Definition of a Red-Black Tree:

A Red-Black Tree is a Binary Search Tree where each node has an additional attribute, color which is red or black, and the tree satisfies the following properties:

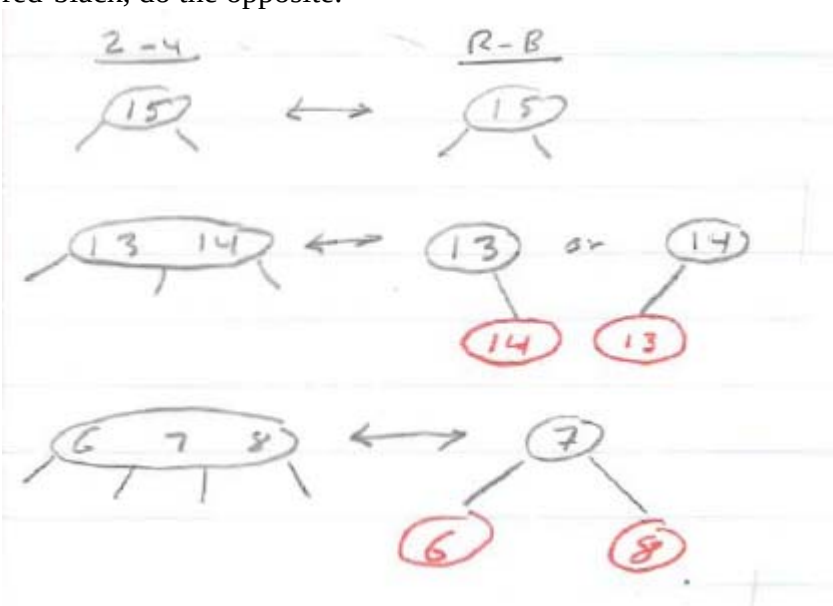
1. Root Property: the root is black
2. External Property: every leaf node is black (all leaves are place holders/dummy nodes)
3. Internal Property: children of a red node are black. (no red-red child-parent relationships)
4. Depth Property: all leaves have the same "black depth." Black depth is the number of ancestor nodes that are colored black.

With these properties, we can keep the height = O(log n). To be specific, height ≤ 2log(n+1).

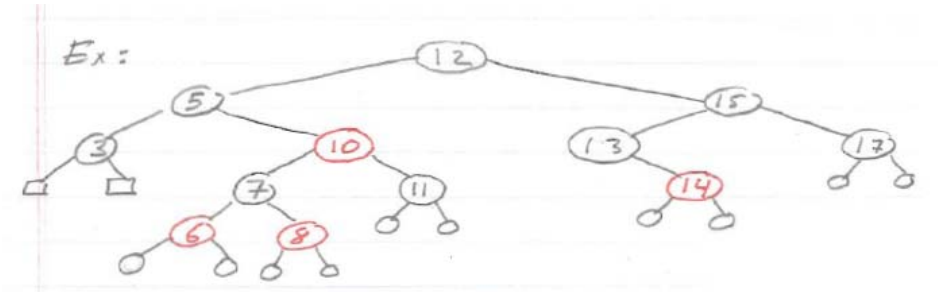
Throughout the discussion of red-black trees, we will fall back on a correspondence between red-black trees and 2-4 trees. It turns out that a red-black tree can be represented by 2-4 trees and vice versa. Since we've already studied 2-4 tree operations, so we'll use this correspondence to shed light on what's going on with red-black tree operations.

Red-Black Tree / 2-4 Tree Correspondence

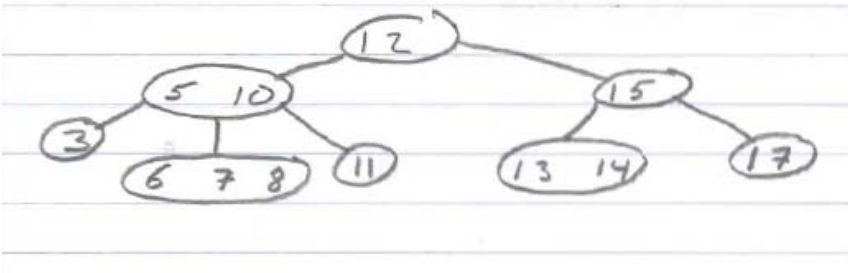
To go from a red-black tree to a 2-4 tree, merge red nodes into their parent. To go from 2-4 to red-black, do the opposite.



Let's look at an example. Given the following red- black…



its corresponding 2-4 tree would be:



Find

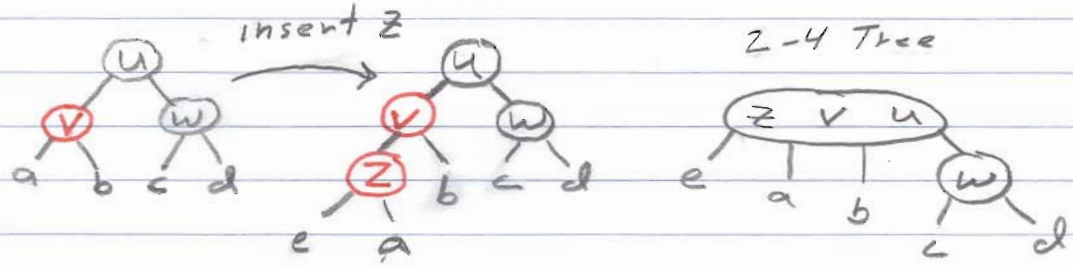
Works exactly the same as with a binary search tree

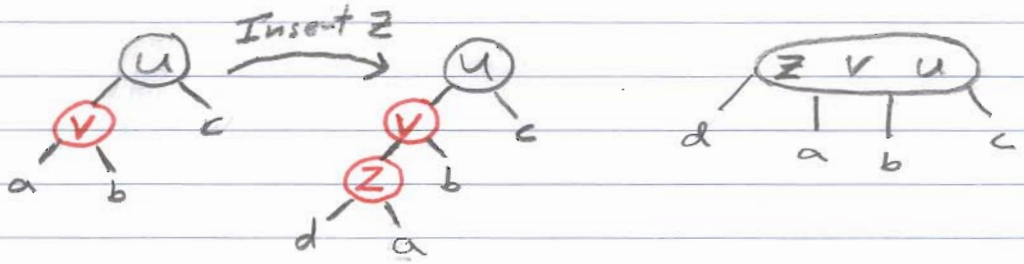
Insert

1. Find the location of a leaf node to add the entry. Exactly the same as adding to a binary search tree
2. Add the entry as a <u>red</u> node
3. Fix the tree if necessary. If adding the red node <u>does</u> <u>not</u> cause a red-red condition, then no fix required. If red-red occurs, the it must be fixed.

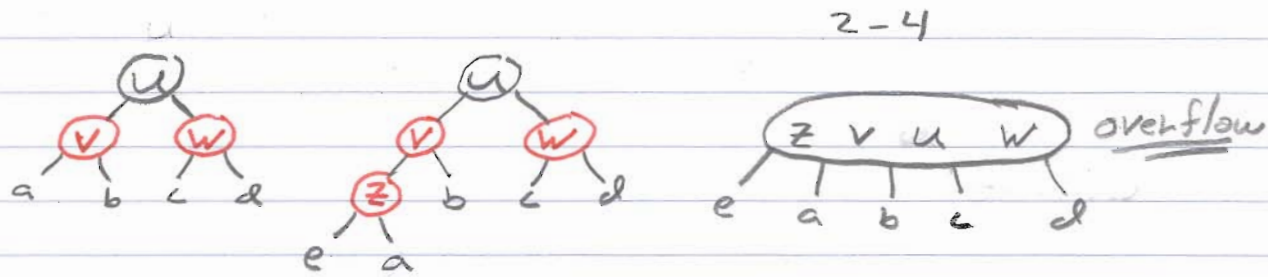
Two cases can arise:

I: Sibling of parent is black

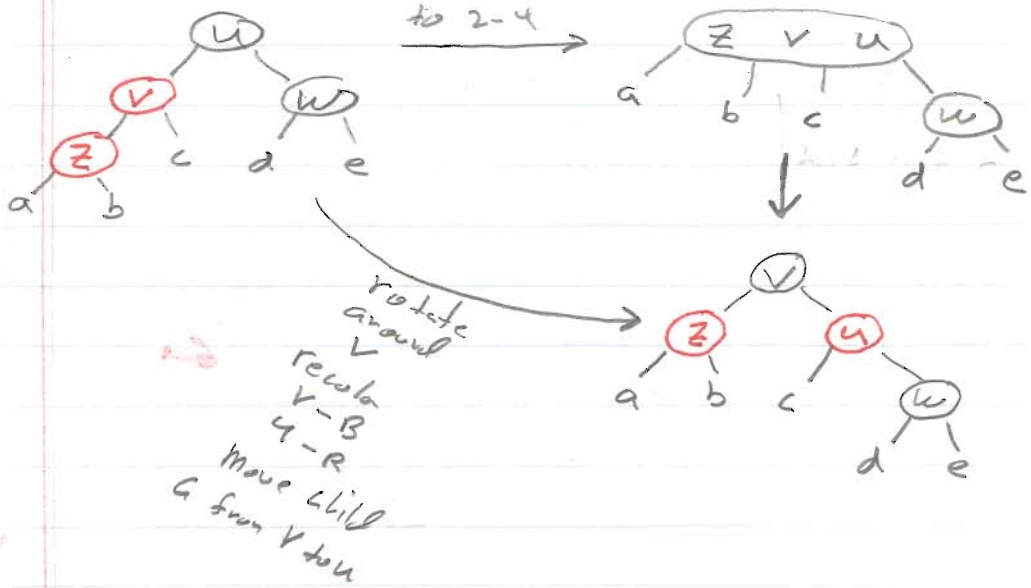




II: sibling of parent is red

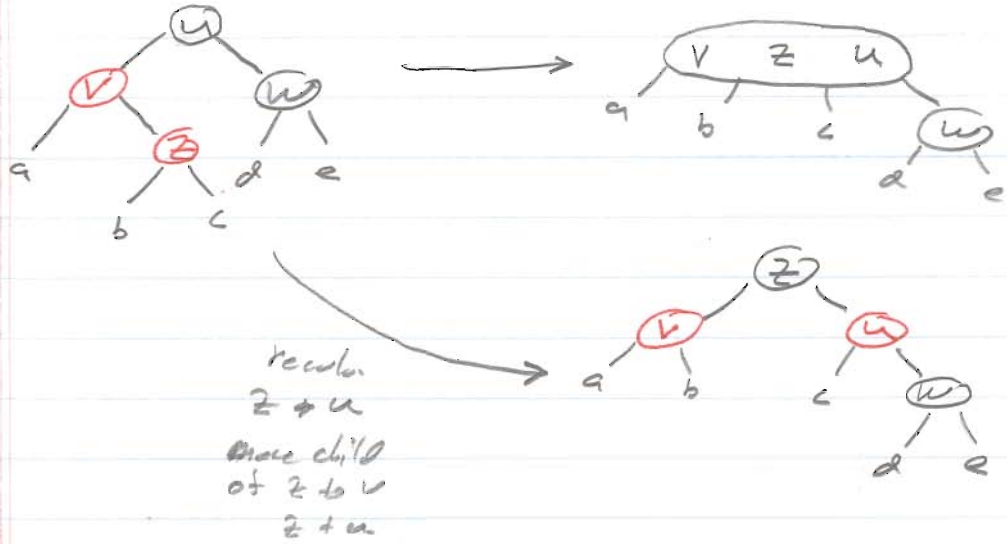


Case I - Sibling of parent is black



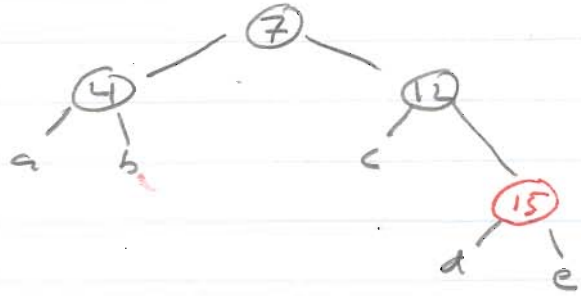
Result: Black depth unchanged
R-R removed

Another ex.

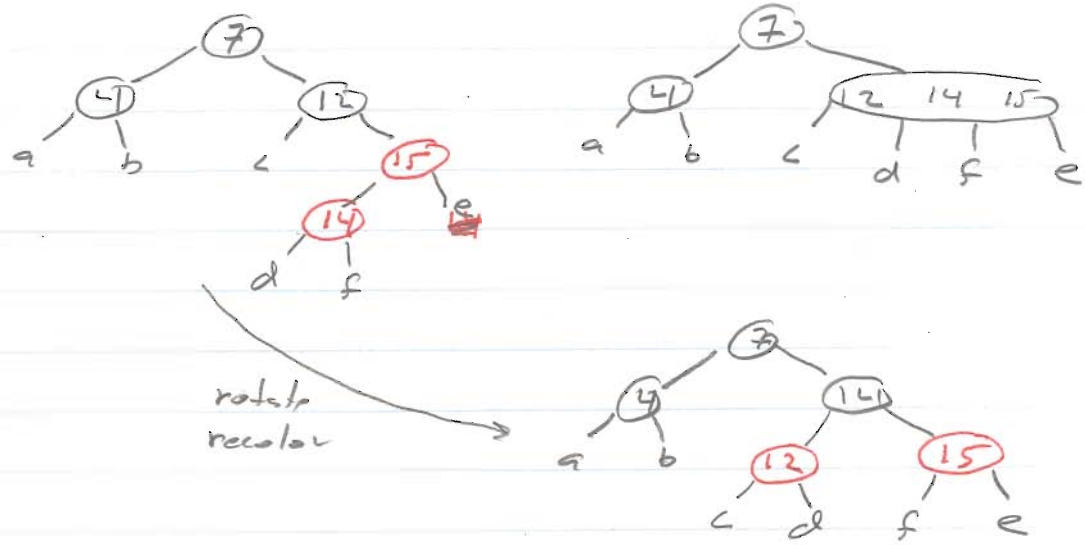


Example

Insert 15



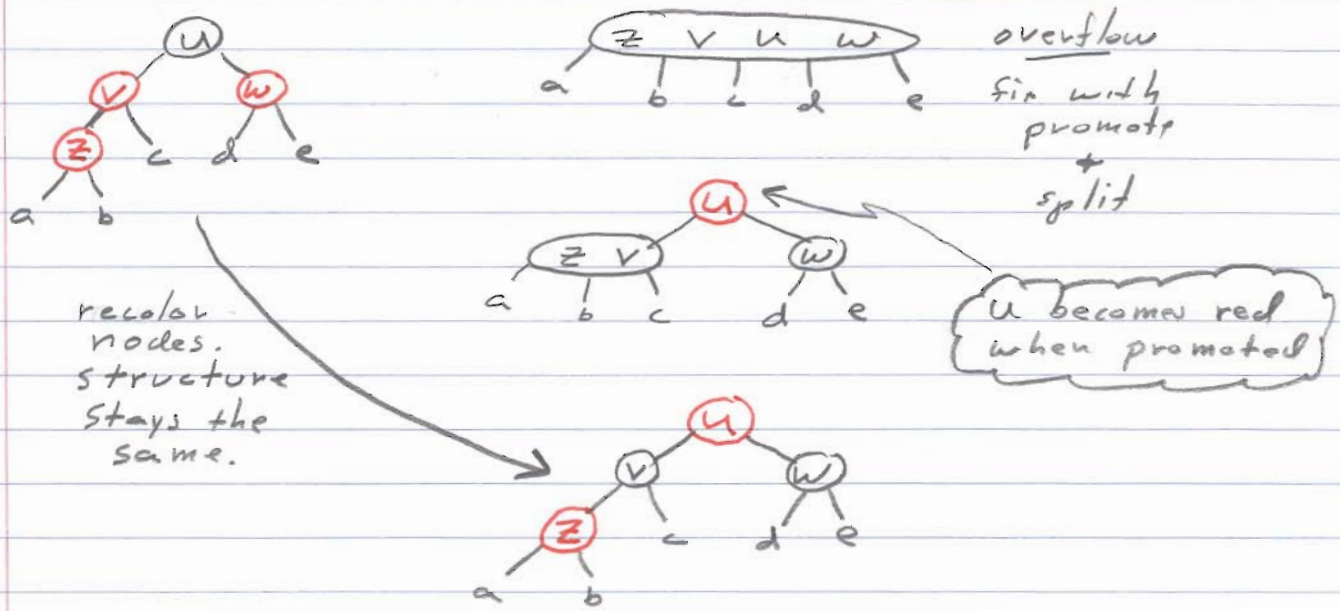
Insert 14



How long? $O(h)$ to find location, $O(1)$ to add.

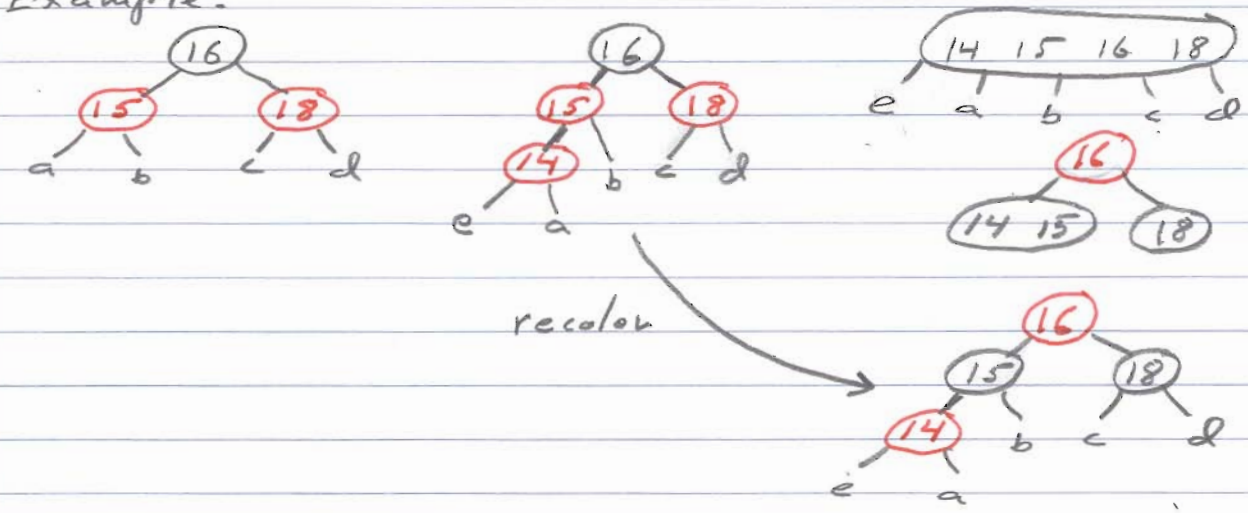
$\therefore O(h) \longrightarrow O(\log n)$

Case II: Sibling of parent is red *overflow* (2-4)

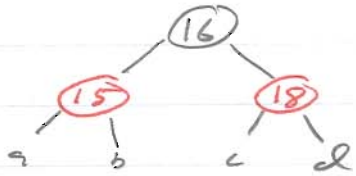


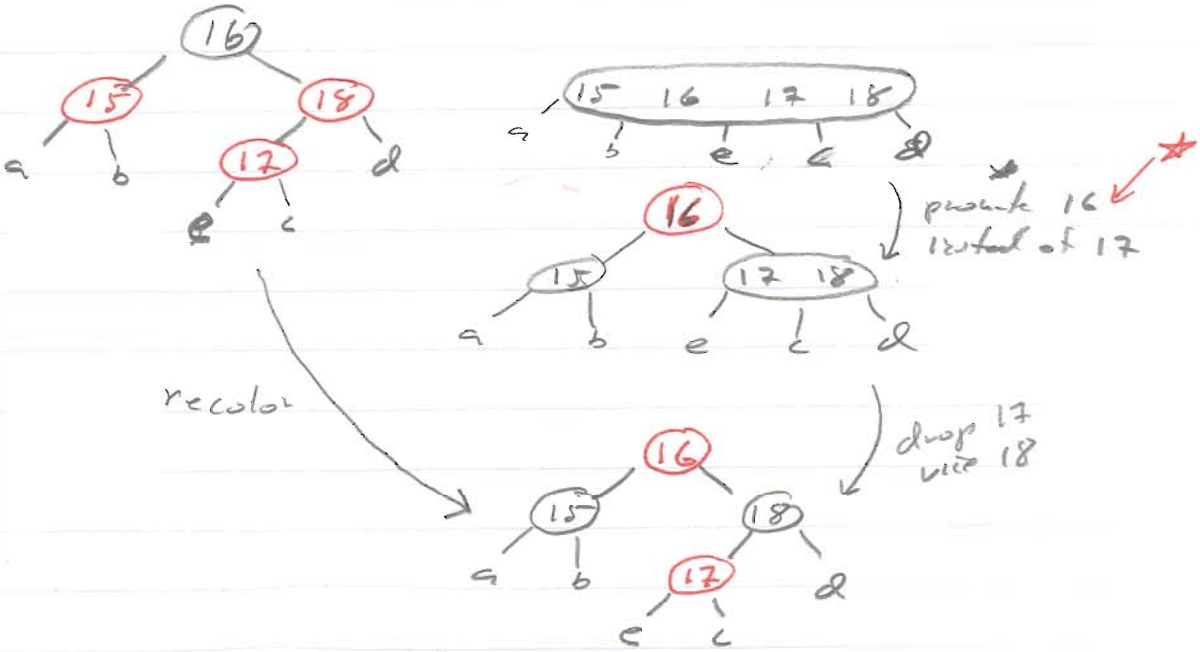
*Note: This caused a red to propogate up one level. Recoloring u red may cause another double red. If so, then fix.

Example:



Ex: Same tree, but insert 17





What if it colors root red?

simply color root black - B

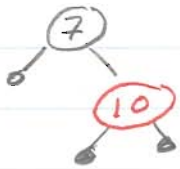
- B - B is ok
- Every black depth increased by one since it was the root.

How long? $O(1)$, but may propagate ... $O(h)$ ... $O(\log n)$

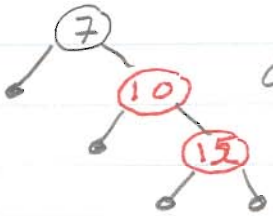
Ex: Insert {7, 10, 15, 18, 6, 8, 17}

Insert 7

Insert 10



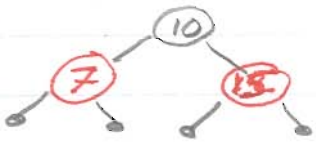
Insert 15



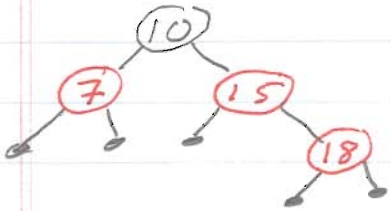
Case I
rotate
recolor

Fix

7 10 15



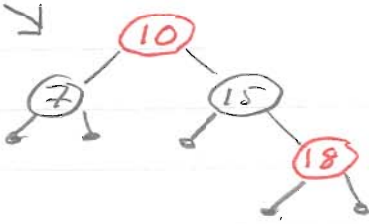
Insert 18



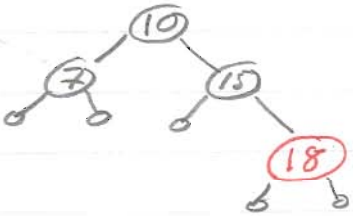
Case II
recolor

Fix

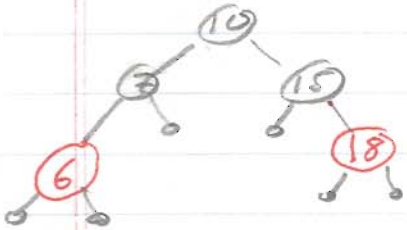
7 10 15 18



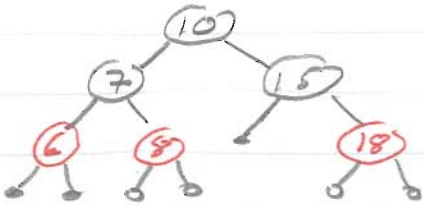
can't have
red root.



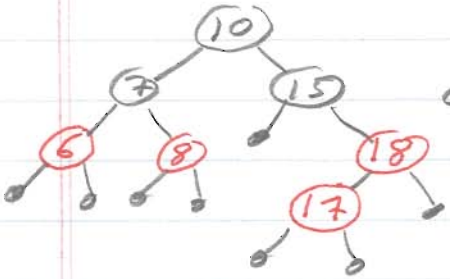
Insert 6



Insert 8



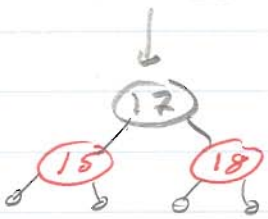
Insert 17

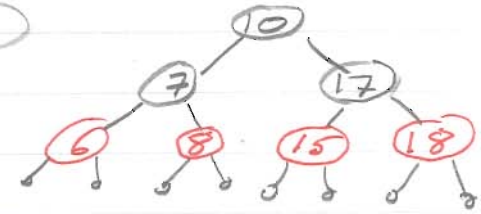


Case I,
rotate
recolor

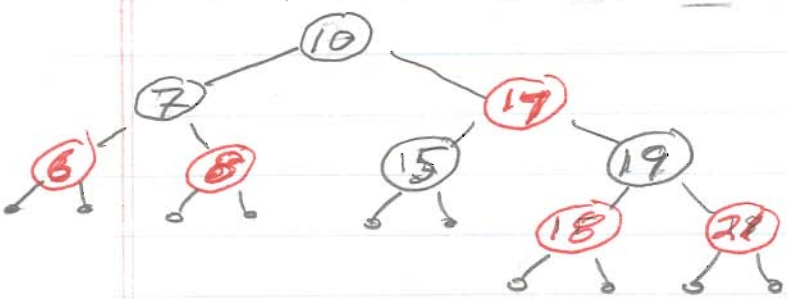
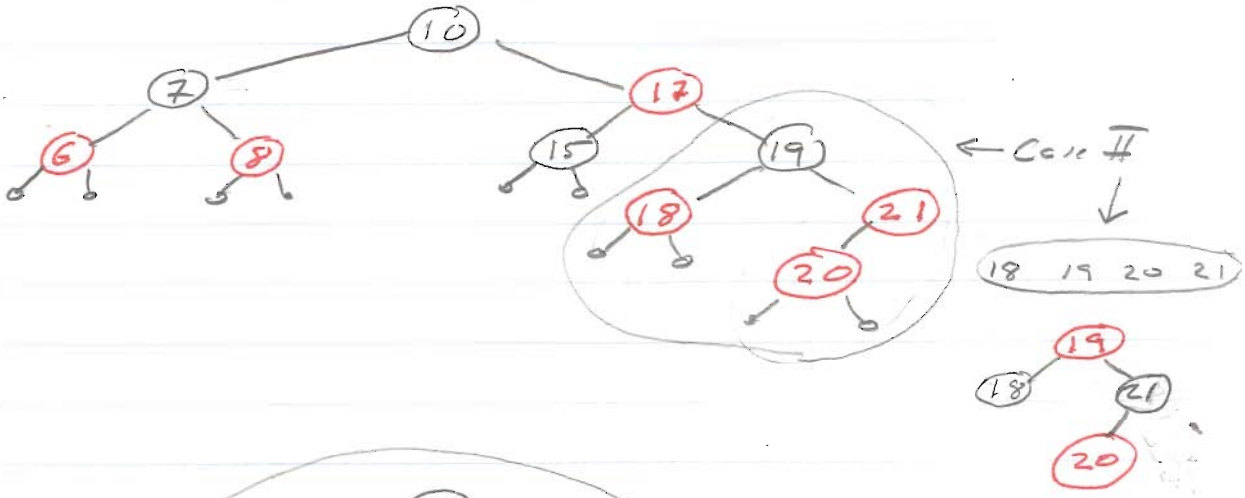
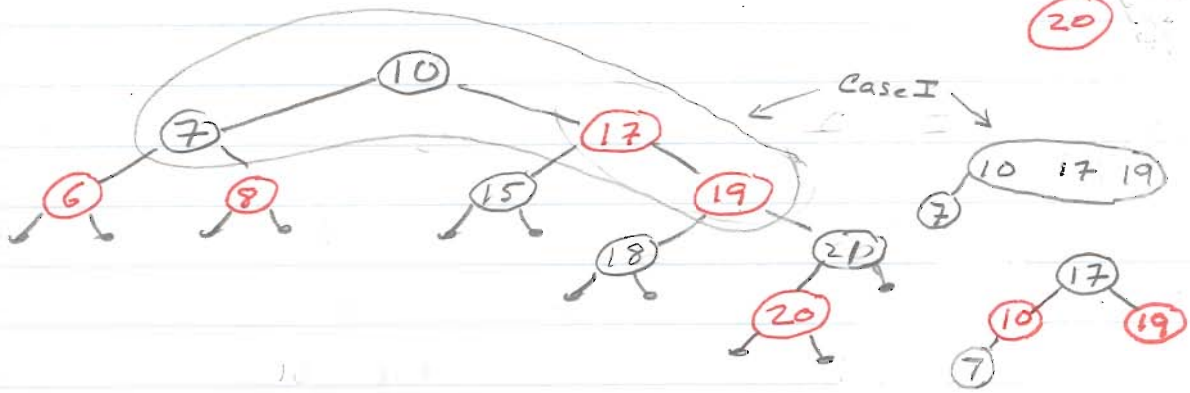
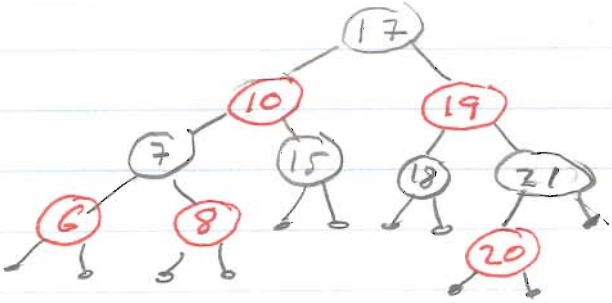
Fix

15 17 18





Example: Double red propagating up tree



Insert 20