CS61B Lecture 30

Wednesday, April 8, 2020

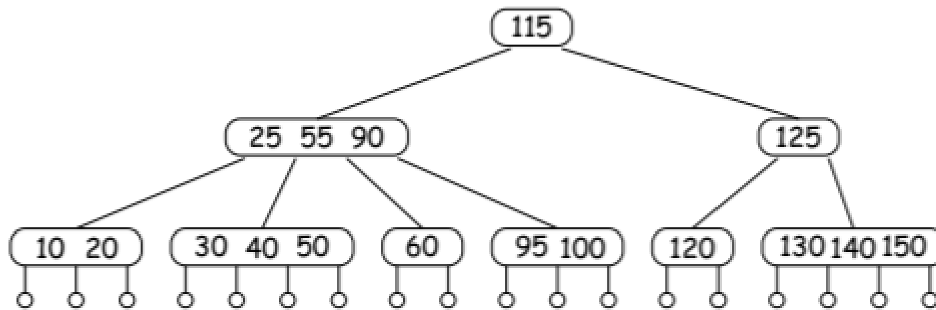
Balanced Search

Why are search trees important? They are data structures in which insertion and deletion is fast on every operation, and support both range queries and sorting, unlike the hash table.

However, the tree's logarithmic performances requires the tree to be "bushy", and "stringy" trees perform more like linked lists. It suffices that heights of any two subtrees of a node differ by no more than some constant factor `K`.

B-Trees

Here's an example of a search tree where we force a tree to be of a certain height:



The above is an order `M` B-Tree, an `M`-ary search tree, where `M` is some value greater than 2.

It obeys the search tree property, where keys are sorted in each node. All keys in subtrees to the left of a key, `K`, are less than `K`, and all to the right are greater than `K`. Children at the bottom of a tree are all empty and equidistant from root.

Searching is a simple generalization of a binary search.

How can we maintain this tree's balance. Here's an idea: if a tree grows/shrinks only at the root, then the two sides will always have the same height.

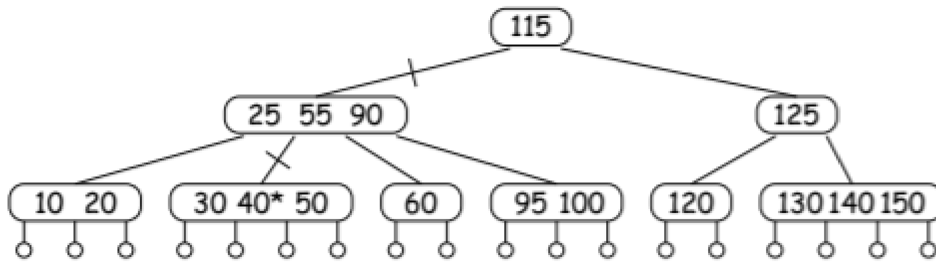
Each node, except the root, has a number of children `N`

$$\lceil \frac{M}{2} \rceil < N < M$$

and one key between each "two" children. The root has between 2 and `M` children in a non-empty tree.

To insert, we add just above the bottom, and split overfull nodes as needed by moving on key up to its parent.

To find, we use the search tree property. The diagram below shows us the path taken when finding 40 in this tree.

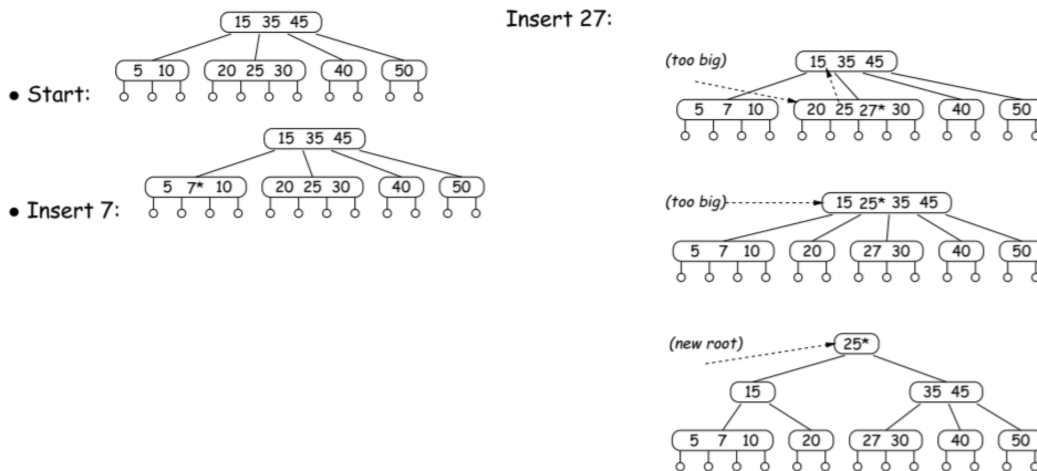


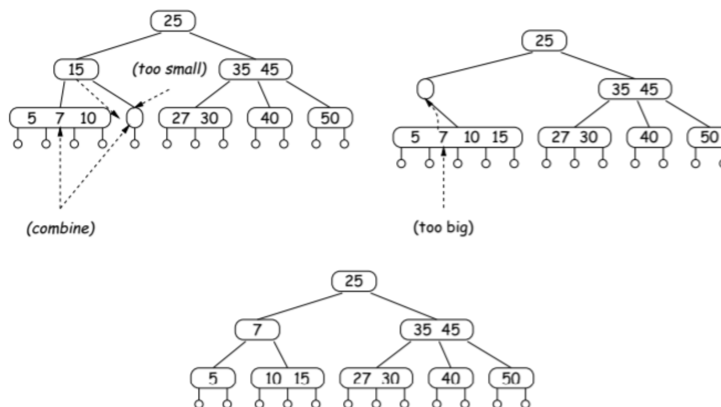
Keys on either side of each child pointer bracket 40, and this is how we find things.

Because each node has at least 2 children, and all leaves are at the bottom, the height **must** be

$$O(lgN)$$

In a real-life B-Tree, the order is typically much bigger, comparable to the size of a disk sector, page, or other convenient unit of input/output.



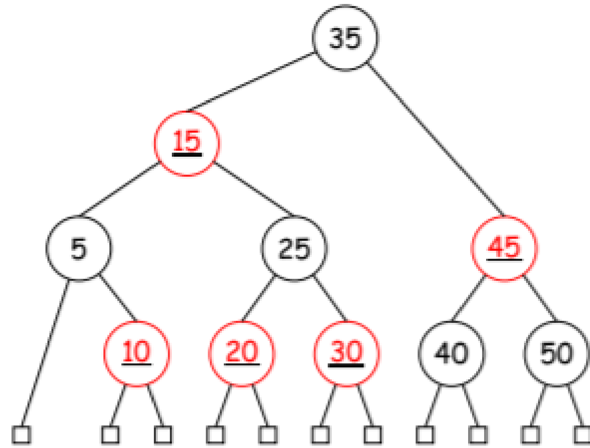


Red-Black Trees

Red-black trees are a type of binary search tree with constraints on how unbalanced it can be. Its nodes are colored red and black; the rules on how nodes are colored have the effect of keeping the tree balanced, such that searching is always

$$O(lgN)$$

Red-black trees are used for Java's `TreeSet` and `TreeMap` types. When items are inserted or deleted, the tree is rotated and recolored as needed to restore the balance.

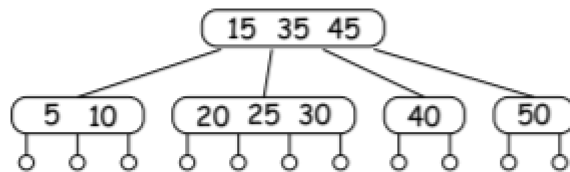


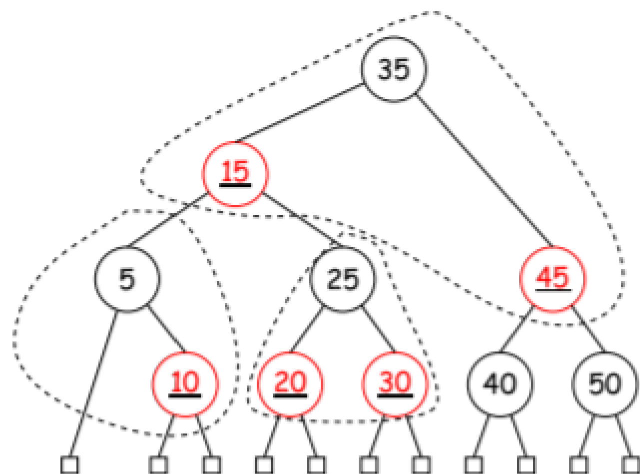
1. Each node is either (conceptually) colored red or black.
2. The root is always black.
3. Every leaf node contains no data, like a B-Tree, and is black.
4. Every leaf has the same number of black ancestors.
5. Every internal node has two children.
6. Every red node has two black children.

The last three conditions guarantee searching is always

$$O(lgN)$$

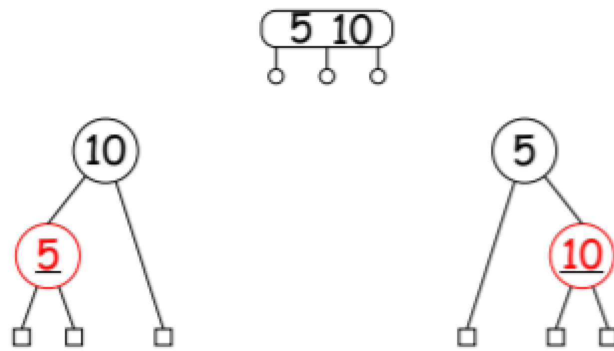
Every red-black tree corresponds to a 2-4 tree, and the operations on one can correspond to those on the other. Each node of the 2-4 tree corresponds to a cluster of 1-3 red-black nodes in which the top node is black and any others are red.





Left-Leaning Red-Black Search Trees (LLRBST)

A node in a 2-4 tree or a 2-3 tree with three children may be represented in two different ways in a red-black tree:

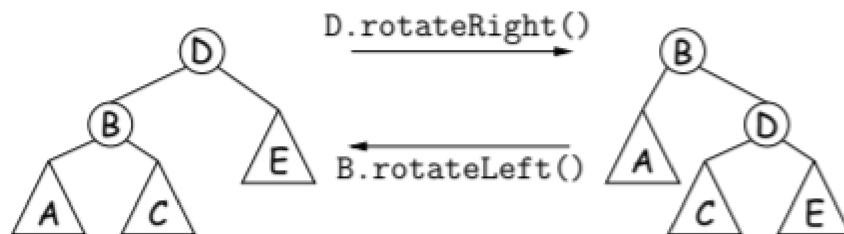


We can considerably simplify insertion and deletion in a red-black tree by always choosing the left option, ensuring a one-to-one relationship between a 2-4 tree and red-black trees. The resulting trees are called **left-leaning red-black search trees** (LLRBST).

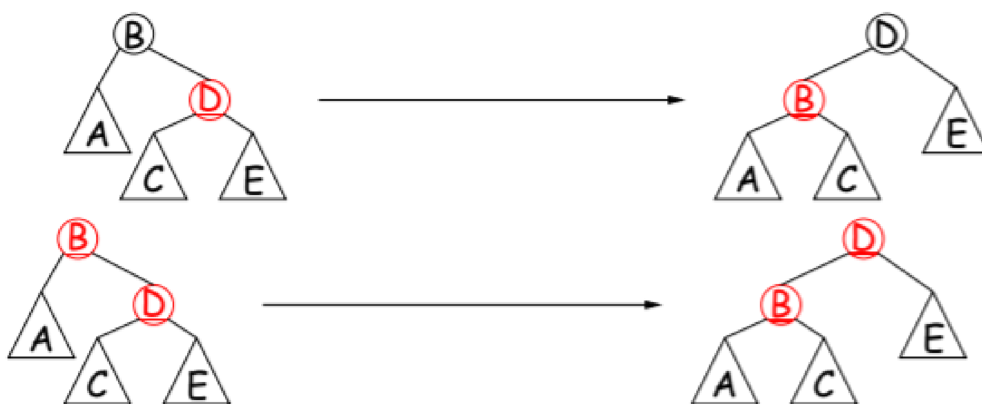
To further simplify this, let's restrict ourselves to red-black trees that correspond to 2-3 trees (whose nodes have no more than 3 children), such that no red-black node has two red children.

Red-Black Insertion and Rotations

Inserting at the bottom is just like a binary tree (color red except when tree initially empty). We then rotate (and recolor) to restore red-black property, and thus balance. The rotation of trees preserves binary tree property, but changes balance.

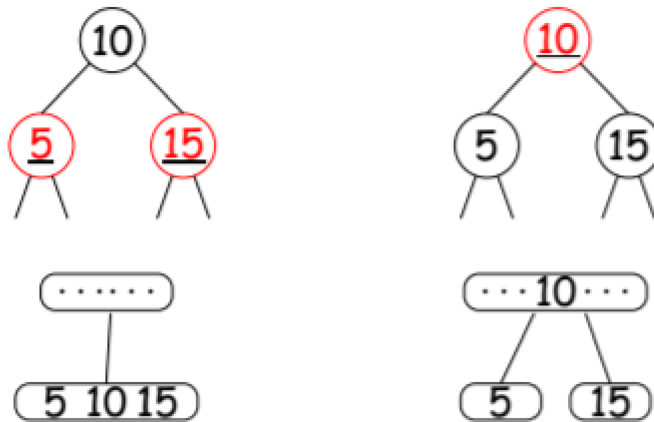


For our purposes, we'll augment the general rotation algorithms with some recoloring. Transfer the color from the original root to the new root, and color the original root red. Examples:



Neither of these changes the number of black nodes along any path between the root and the leaves.

Our algorithms temporarily create nodes with too many children, before splitting them up. A simple recoloring is analogous to splitting nodes, in a process we will call `colorFlip`.



Here, key 10 joins the parent node, splitting the original.

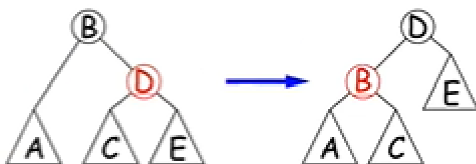
The Algorithm

The algorithm, as written by Robert Sedgewick of Princeton, is as follows, where `RBTree` is an ordinary BST with color:

```
RBTree insert(RBTree tree, KeyType key) {
    if (tree == null) { return new RBTree(key, null, null, RED); }
    int cmp = key.compareTo(tree.label());
    if (cmp < 0) {
        tree.setLeft(insert(tree.left(), key));
    } else {
        tree.setRight(insert(tree.right(), key));
    }
    return fixup(tree);
}
```

How does one `fixup` a tree? Well:

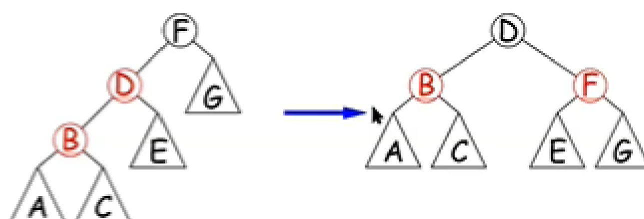
1. Convert right-leaning trees into left-leaning:


   ```
   if (tree.right().isRed() && tree.left().isBlack()) {
       tree.rotateLeft();
   }
   ```

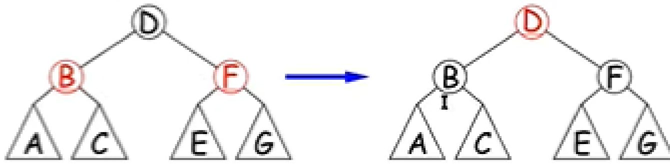
   Sometimes node `B` will be red, so both `B` and `D` end up red. This is fixed by...

2. Rotate linked red nodes into a normal 4-node, temporarily:



```
if (tree.left().isRed() && tree.left().left().isRed()) {
    tree.rotateRight();
}
```

3. Break up 4-nodes into 3-nodes or 2-nodes:

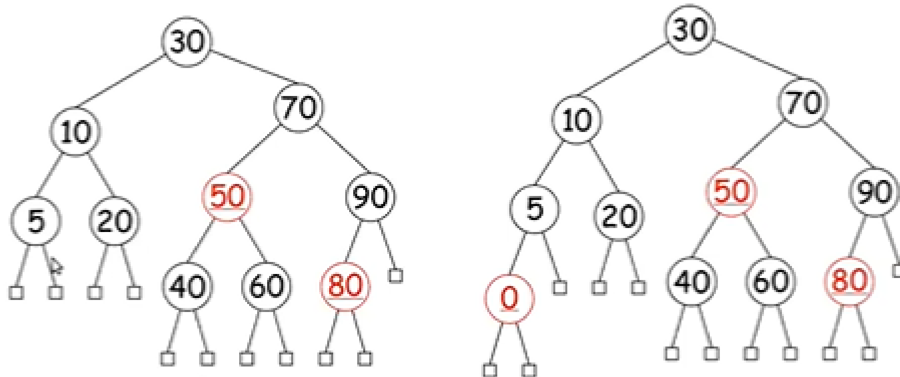



```
if (tree.left().isRed() && tree.right().isRed()) {
    colorFlip(tree);
}
```

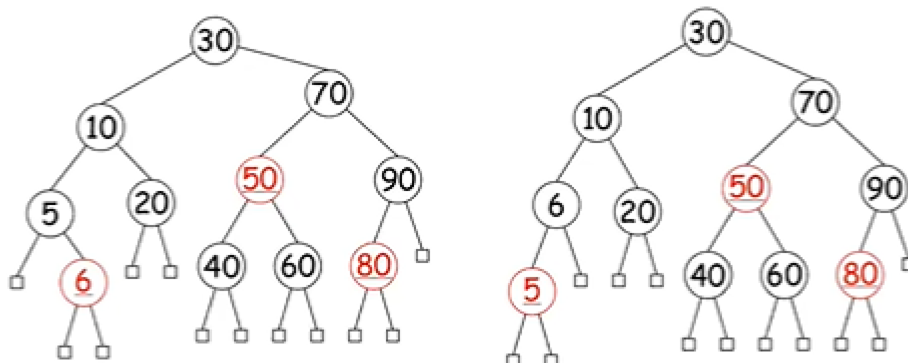
4. As a result of other fixups, or of insertion into the empty tree, the root may end up red, so color the root black after the rest of insertion and fixups are finished (this is not part of the `fixup` method). This is always okay at the root, without any side effects.

Insertion Examples

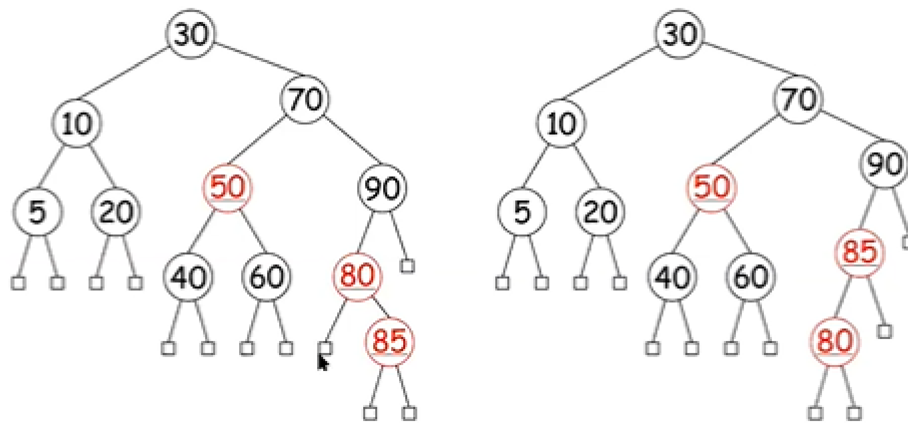
- Insert 0 into initial tree on left. No fixups needed.



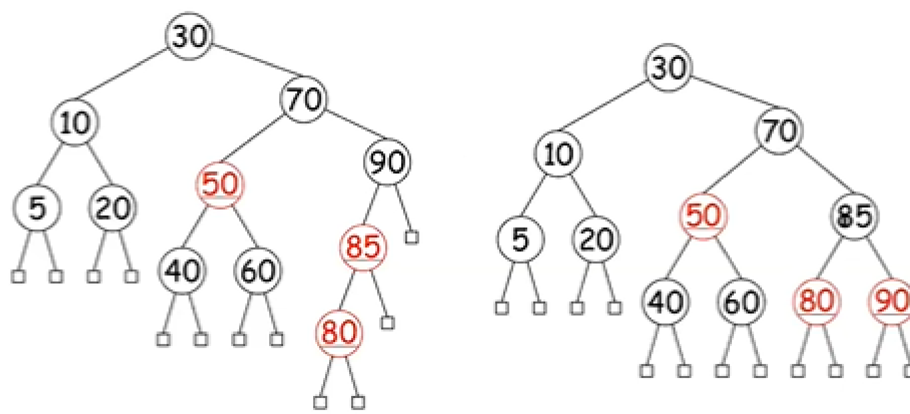
- Instead of 0, let's insert 6, leading to the tree on the left. This is right-leaning, so apply Fixup 1:



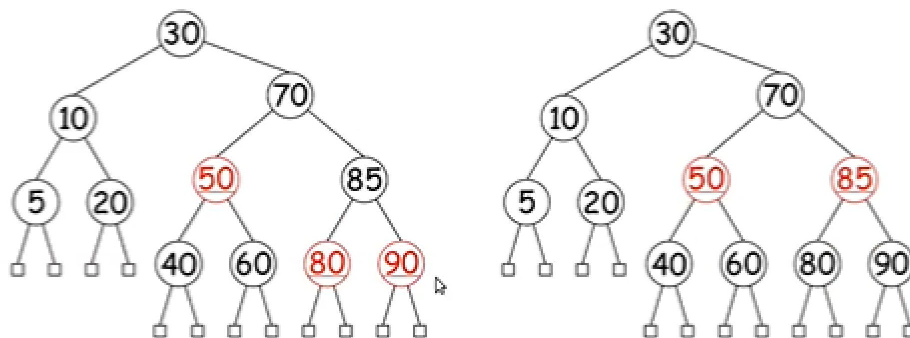
• Now consider inserting 85. We need fixup 1 first.



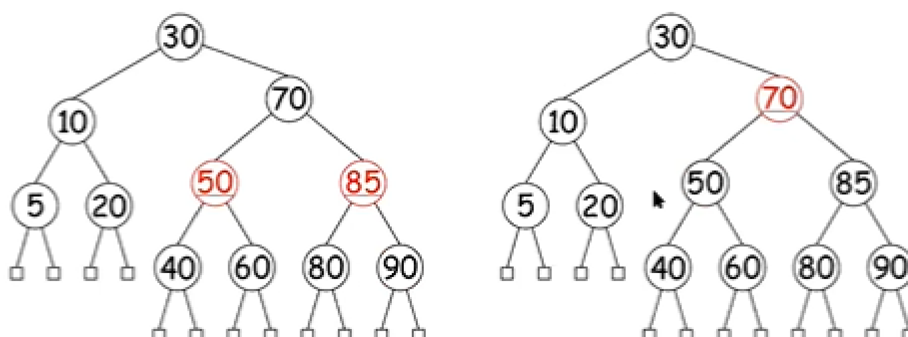
• Now apply fixup 2.



• This gives us a 4-node, so apply fixup 3.



• This gives us another 4-node, so apply fixup 3 again.



- This gives us a right-leaning tree, so apply fixup 1.