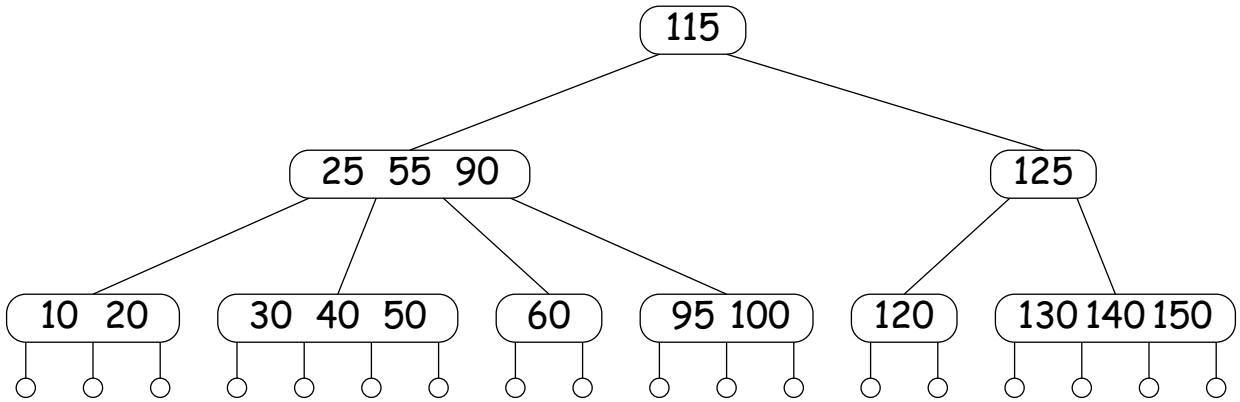
CS61B Lecture #29

Balanced Search: The Problem

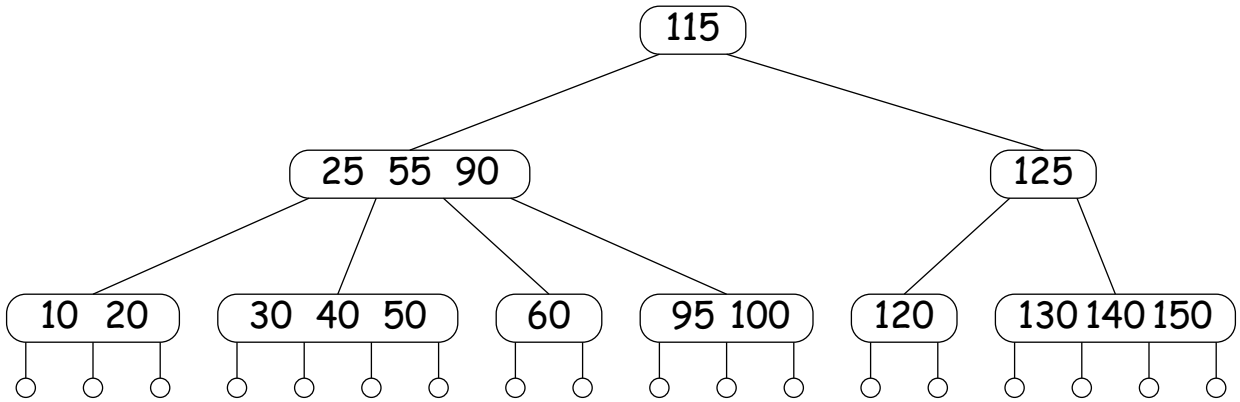
- Why are search trees important?
  - Insertion/deletion is fast (on every operation, unlike hash table, which has to expand from time to time).
  - They support range queries and sorting (unlike hash tables).
- But $O(\lg N)$ performance from binary search tree requires keys to be roughly divided by some some constant > 1 at each inner node.
- In other words, the tree must be "bushy"
- "Stringy" trees (where most inner nodes have one child) perform like linked lists.
- It suffices that the heights of the subtrees of nodes that are not near the bottom always differ in height by no more than some constant factor $C > 0$.

Example of Direct Approach: B-Trees



- An *order M B-tree* is an M-ary search tree, $M > 2$, where non-root nodes have at least $\lceil M/2 \rceil$ children.
- For example, if $M = 4$, non-root nodes have at least 2 nodes, so we also call these (2, 4) (or 2-4) trees.
- Obeys the search-tree property:
  - Keys are sorted in each node.
  - Each key separates two subtrees with all keys in the left subtree of a key, K, being $< K$, and all to the right being $> K$.
- Children at the bottom of tree are all empty (don't really exist) and equidistant from the root.
- Searching is a simple generalization of binary search.

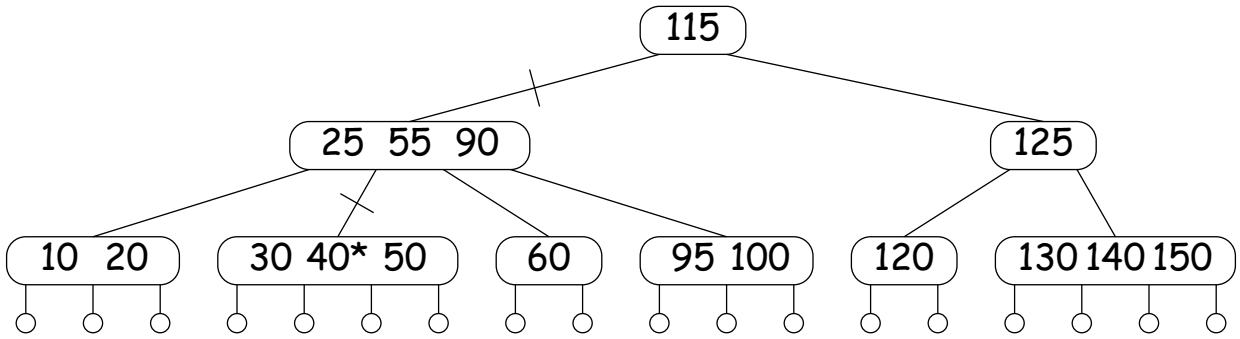
Example of Direct Approach: B-Trees



Idea: If the tree grows/shrinks only at the root, then the two sides always have same height.

- Each node, except the root, has from $\lceil M/2 \rceil$ to M children, and one key "between" each two children.
- The root has from 2 to M children (in a non-empty tree).
- Insertion: add to nodes just above the (empty) leaves; split overfull nodes as needed, moving one key up to its parent.

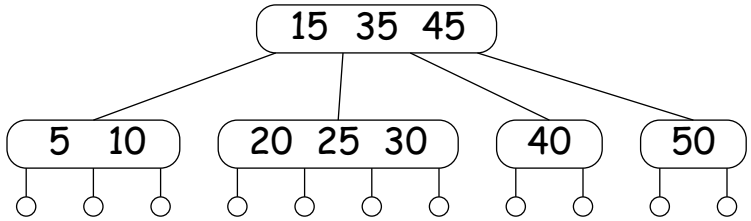
Sample Order 4 B-tree ((2,4) Tree)



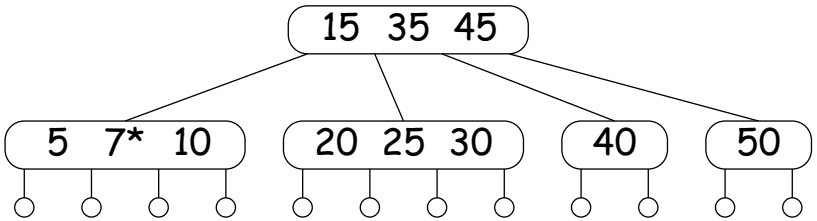
- Crossed lines show the path for finding 40.
- Keys on either side of each child pointer in the path bracket 40.
- Each node has at least 2 children, and all leaves (little circles) are at the bottom, so the height must be $O(\lg N)$.
- In real life, B-trees are stored on secondary storage, and the order is typically much bigger: comparable to the size of a disk sector, page, or other convenient unit of I/O.

Inserting in (2, 4) tree (Simple Case)

- Start:

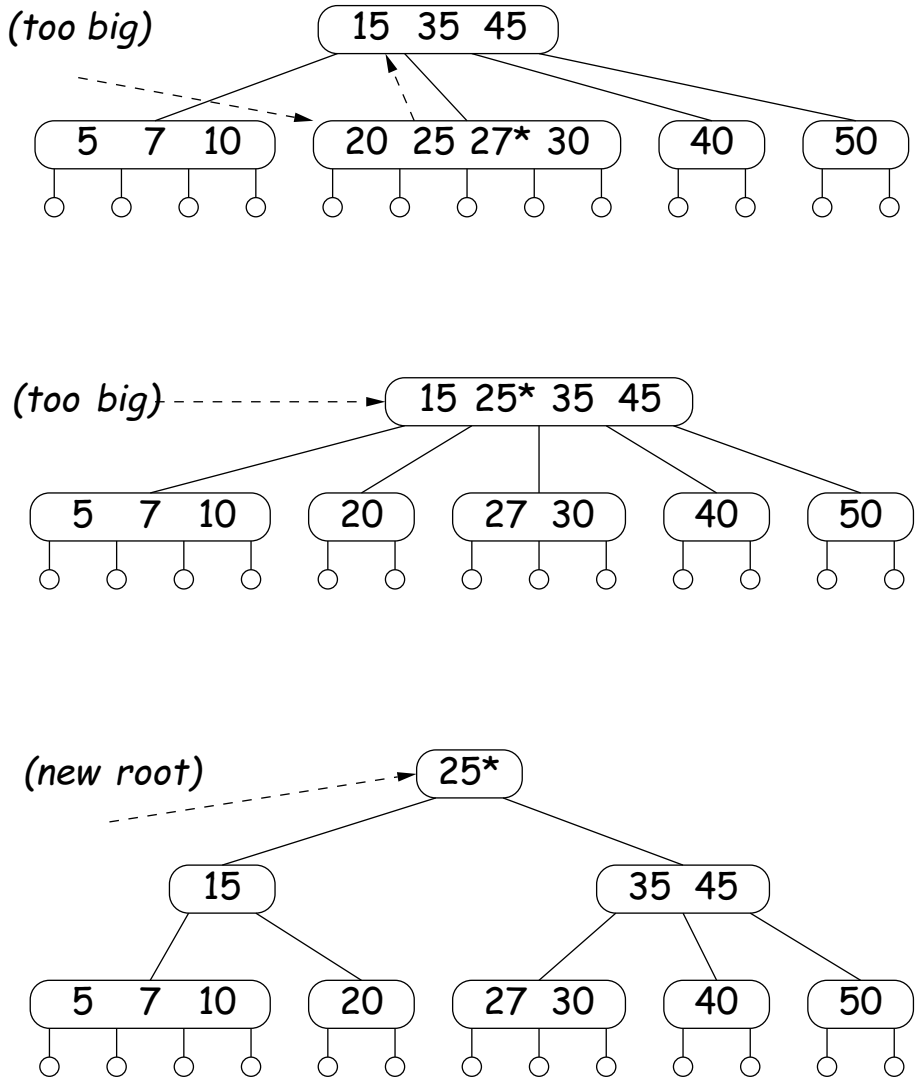


- Insert 7:



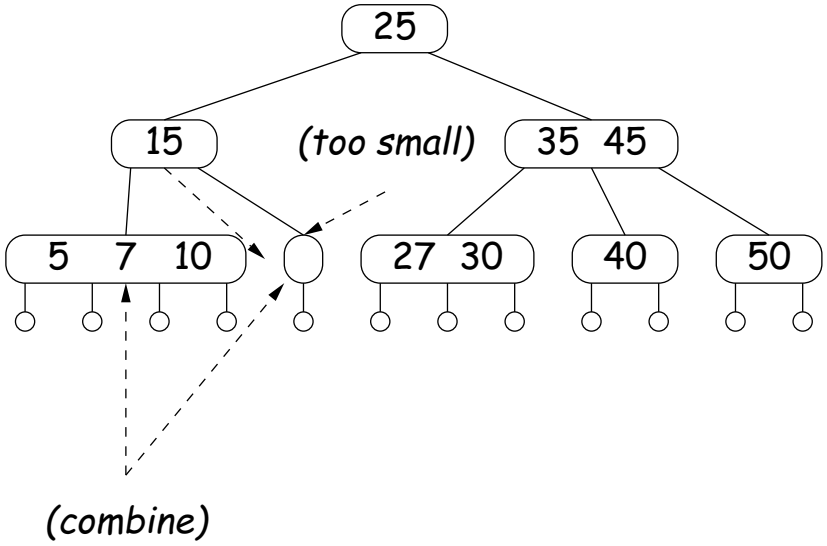
Inserting in B-Tree (Splitting)

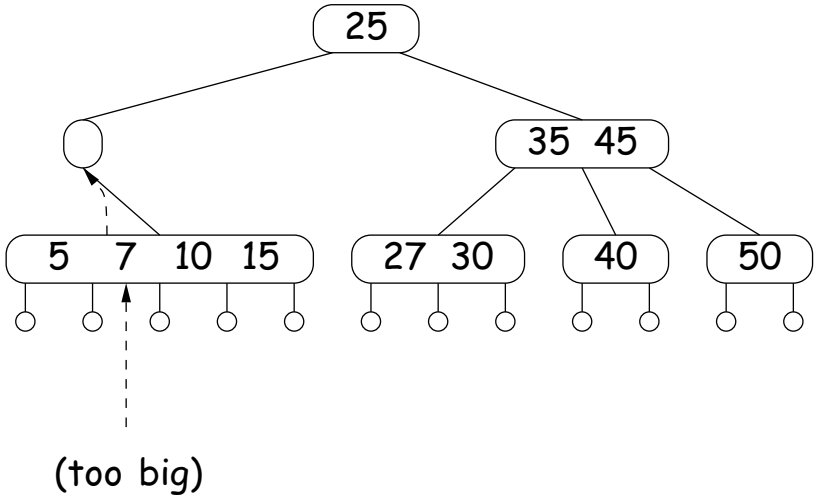
- Insert 27:

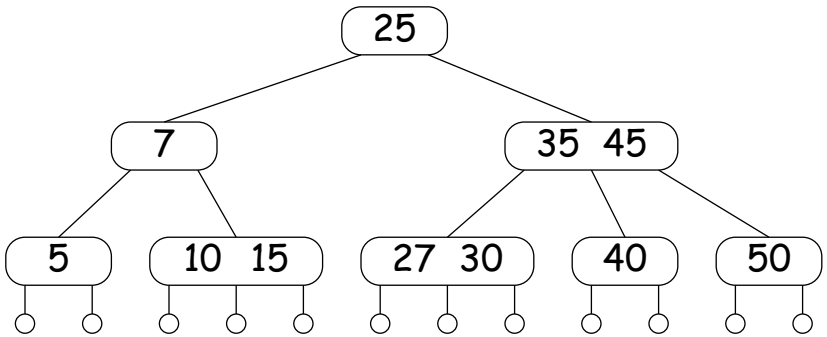


Deleting Keys from B-tree

- Remove 20 from last tree.



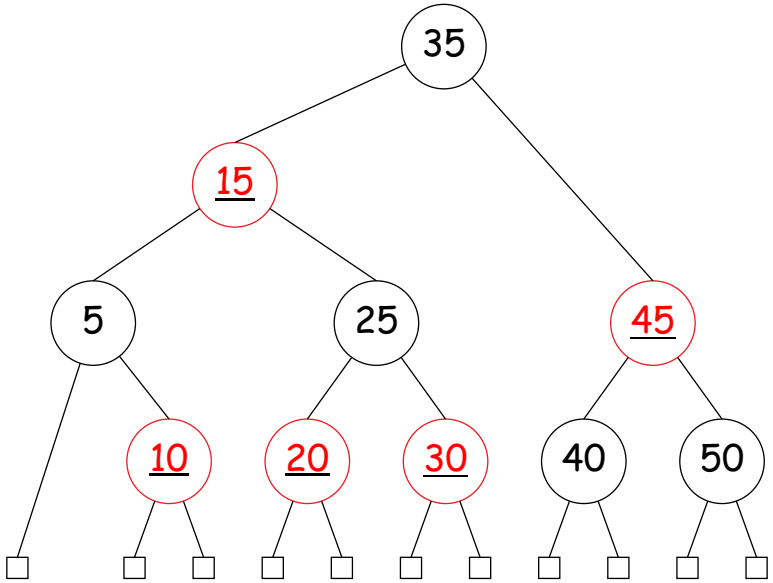




Red-Black Trees

- A *Red-black tree* is a binary search tree with additional constraints that limit how unbalanced it can be.
- Thus, searching is always $O(\lg N)$.
- Used for Java's `TreeSet` and `TreeMap` types.
- When items are inserted or deleted, the tree is *rotated* and *recolored* as needed to restore balance.

Red-Black Tree Constraints

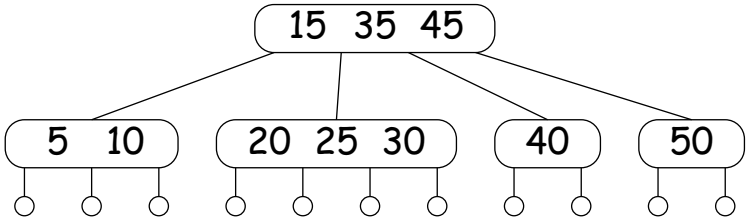


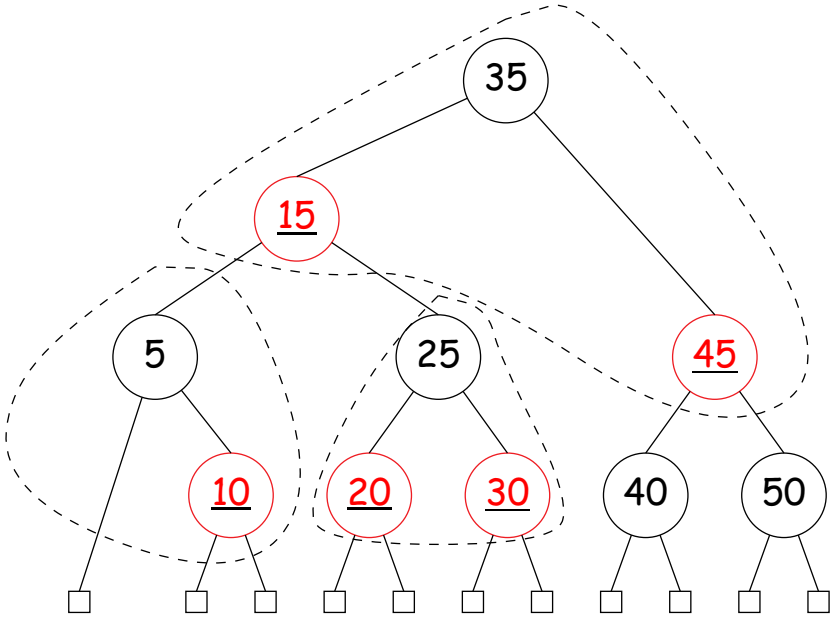
1. Each node is (conceptually) colored red or black.
2. Root is black.
3. Every leaf node contains no data (as for B-trees) and is black.
4. Every leaf has same number of black ancestors.
5. Every internal node has two children.
6. Every red node has two black children.

- Conditions 4, 5, and 6 guarantee $O(\lg N)$ searches.

Red-Black Trees and (2,4) Trees

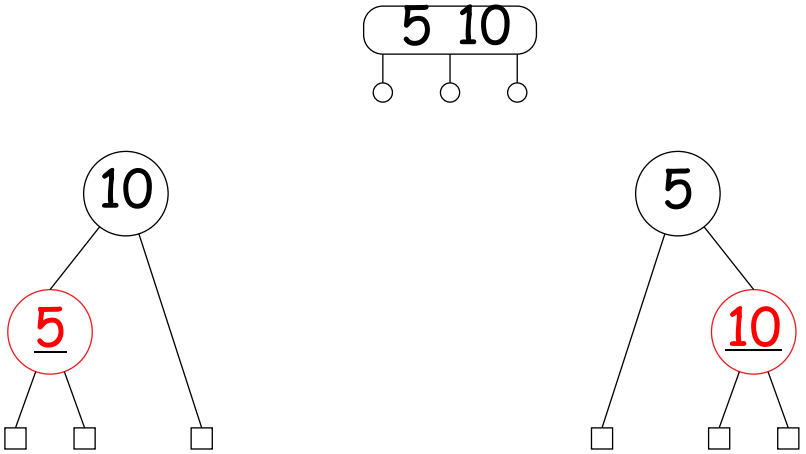
- Every red-black tree corresponds to a (2,4) tree, and the operations on one correspond to those on the other.
- Each node of (2,4) tree corresponds to a cluster of 1–3 red-black nodes in which the top node is black and any others are red.





Additional Constraints: Left-Leaning Red-Black Trees

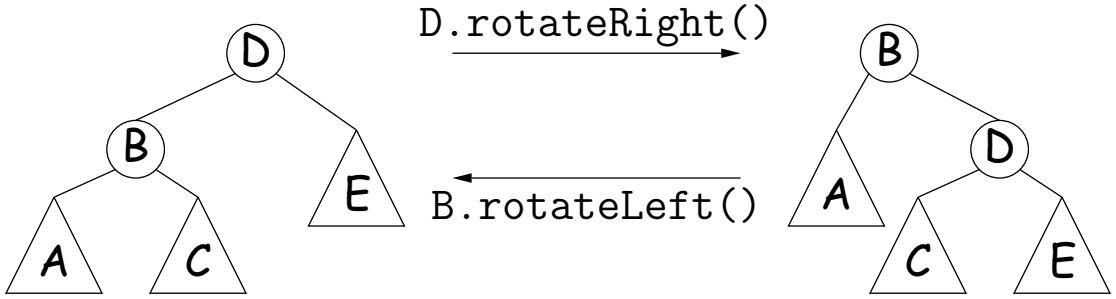
- A node in a (2,4) or (2,3) tree with three children may be represented in two different ways in a red-black tree:



- We can considerably simplify insertion and deletion in a red-black tree by always choosing the option on the left.
- With this constraint, there is a one-to-one relationship between (2,4) trees and red-black trees.
- The resulting trees are called *left-leaning red-black trees.*
- As a further simplification, let's restrict ourselves to red-black trees that correspond to (2,3) trees (whose nodes have no more than 3 children), so that no red-black node has two red children.

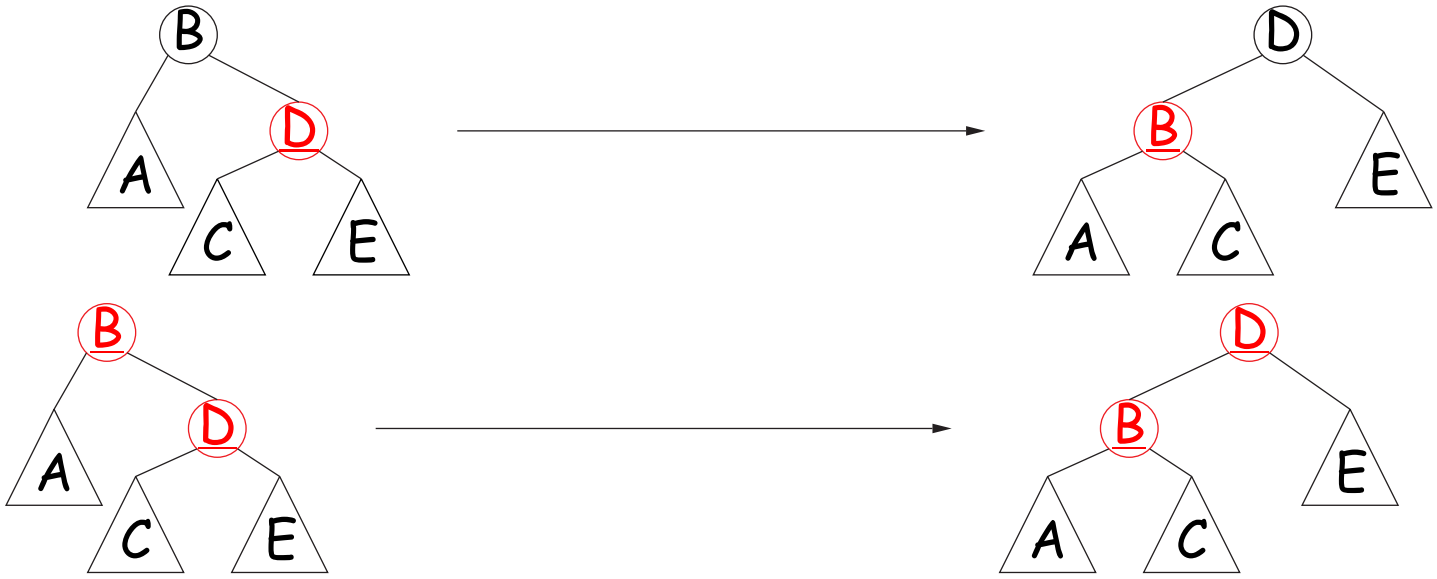
Red-Black Insertion and Rotations

- Insert at the bottom just as for binary trees (color the insertion red except when the tree is initially empty).
- Then rotate (and recolor) to restore the red-black property, and thus rebalance.
- Rotation of trees *preserves* the binary tree property, but changes balance.



Rotations and Recolorings

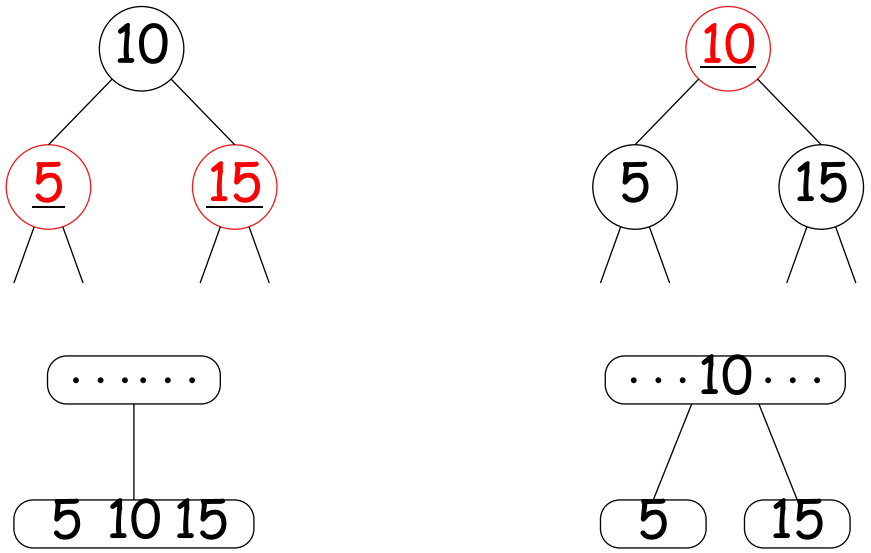
- For our purposes, we'll augment the general rotation algorithms with some recoloring.
- Transfer the color from the original root to the new root, and color the original root red. Examples:



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

Splitting by Recoloring

- Our algorithms will temporarily create nodes with too many children, and then split them up.
- A simple recoloring allows us to split nodes. We'll call it `colorFlip`:



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

The Algorithm (Sedgewick)

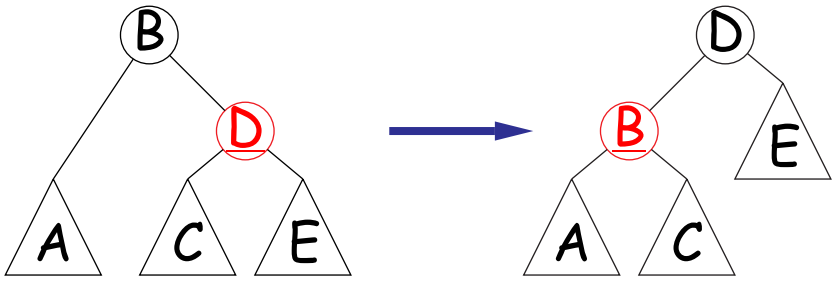
- We posit a binary-tree type `RBTree`: basically ordinary BST nodes plus color.
- Insertion is the same as for ordinary BSTs, but we add some fixups to restore the red-black properties.

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

    return fixup(tree);     // Only line that's all new!
}
```

Fixing Up the Tree

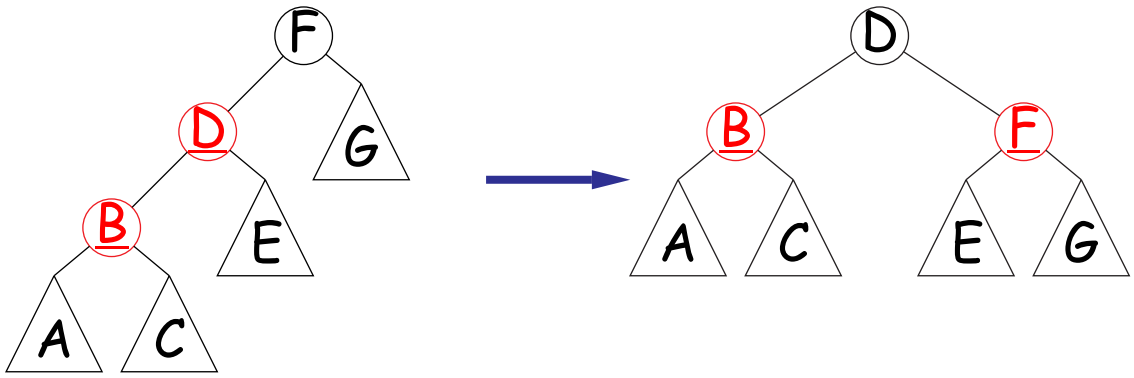
- As we return back up the BST, we restore the left-leaning red-black properties, and limit ourselves to red-black trees that correspond to (2,3) trees by applying the following (in order) to each node:
- Fixup 1: Convert right-leaning trees to left-leaning:




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

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

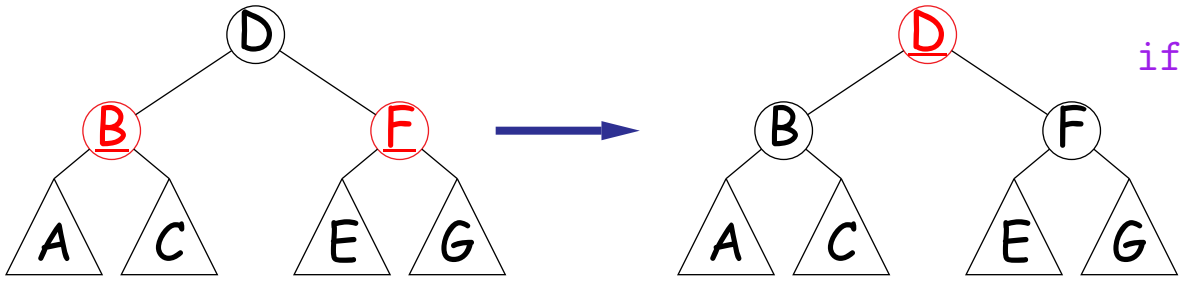
- Fixup 2: Rotate linked red nodes into a normal 4-node (temporarily).




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

Fixing Up the Tree (II)

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

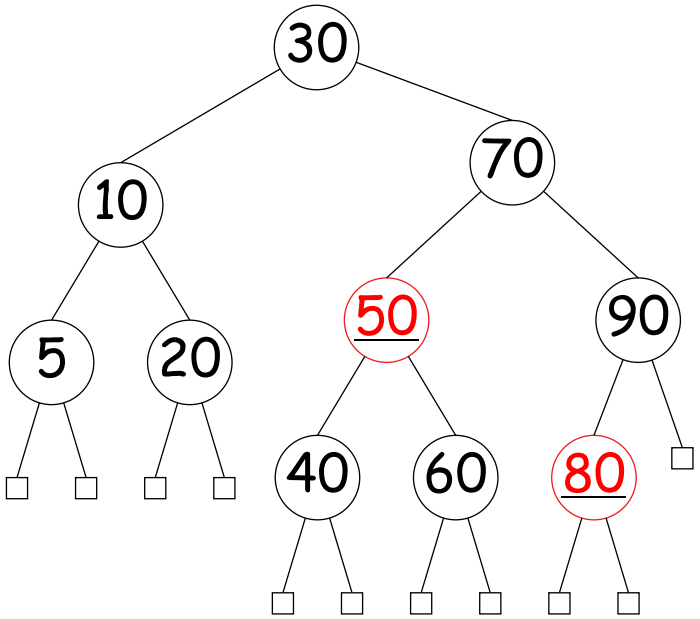


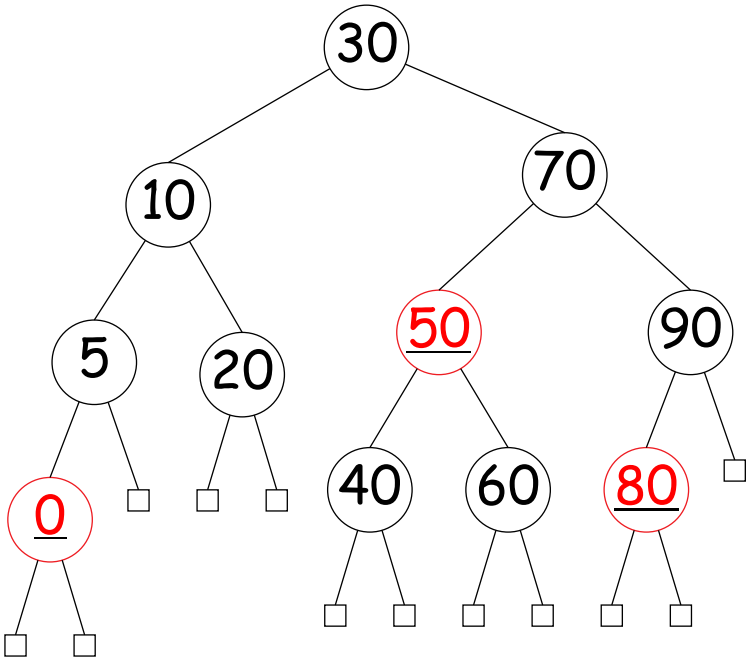

```
if (tree.left().isRed() &&
    tree.right().isRed())
        colorFlip(tree);
```

- Fixup 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. (Not part of the fixup function; just done at the end).

Example of Left-Leaning 2-3 Red-Black Insertion

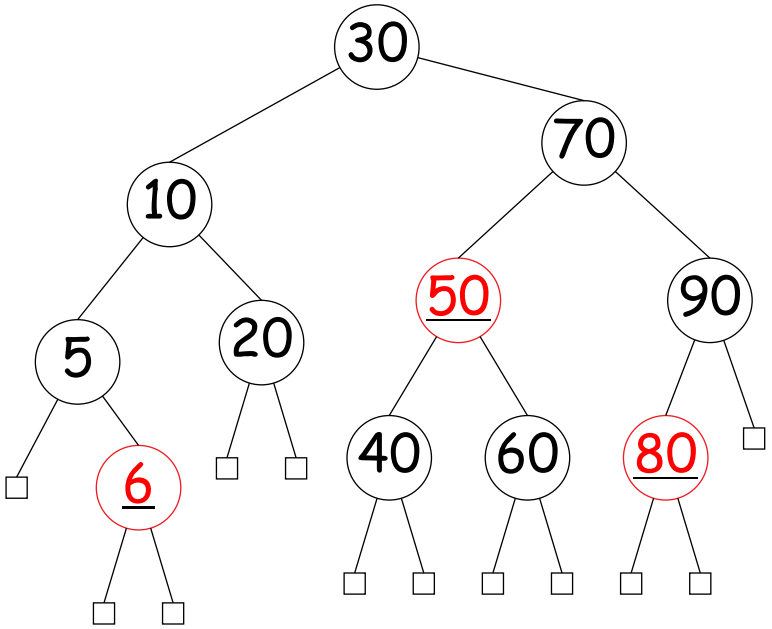
- Insert 0 into initial tree on the left. No fixups needed.

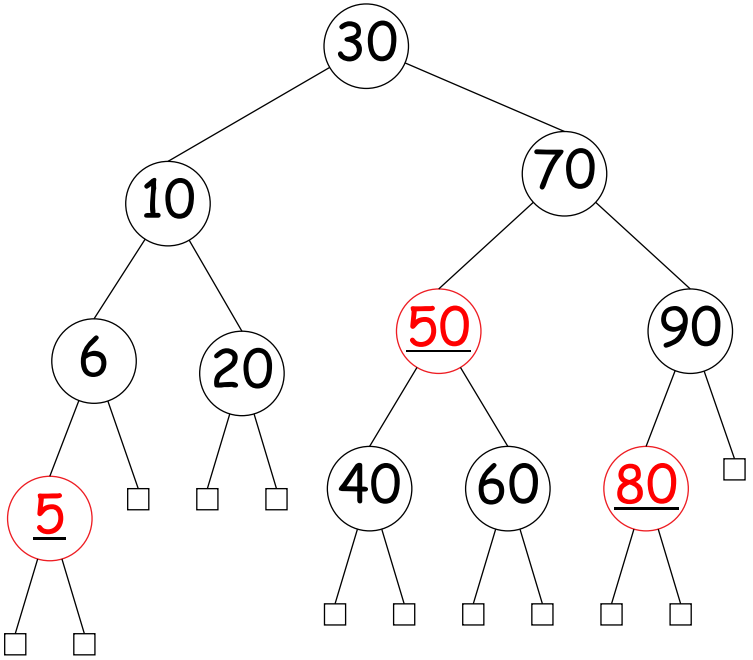




Insertion Example (II)

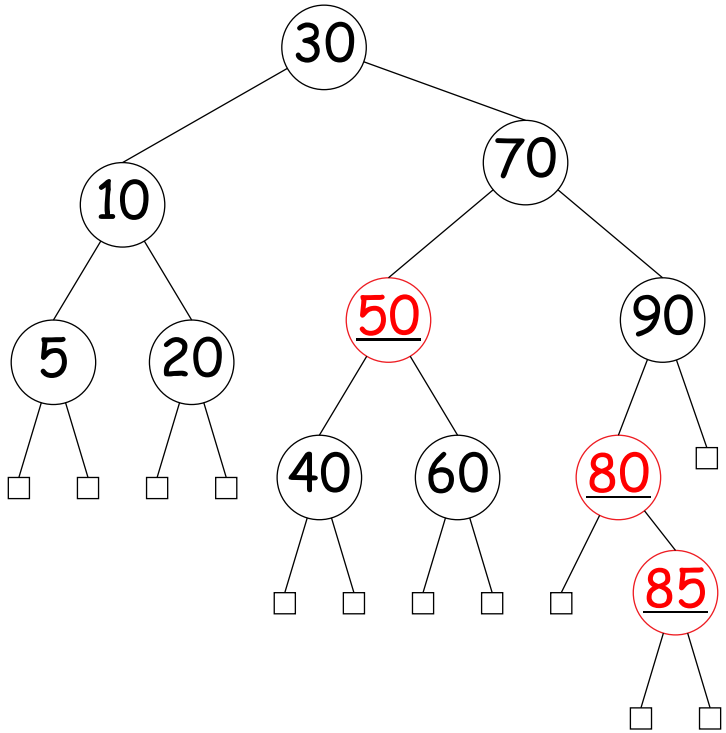
- Instead of 0, let's insert 6, leading to the tree on the left. This is right-leaning, so apply Fixup 1:

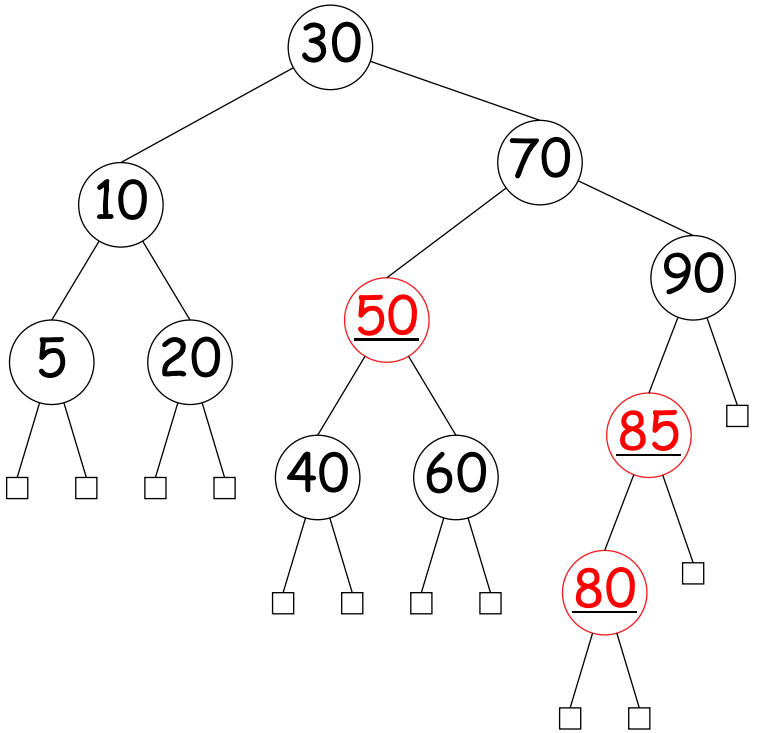




Insertion Example (III)

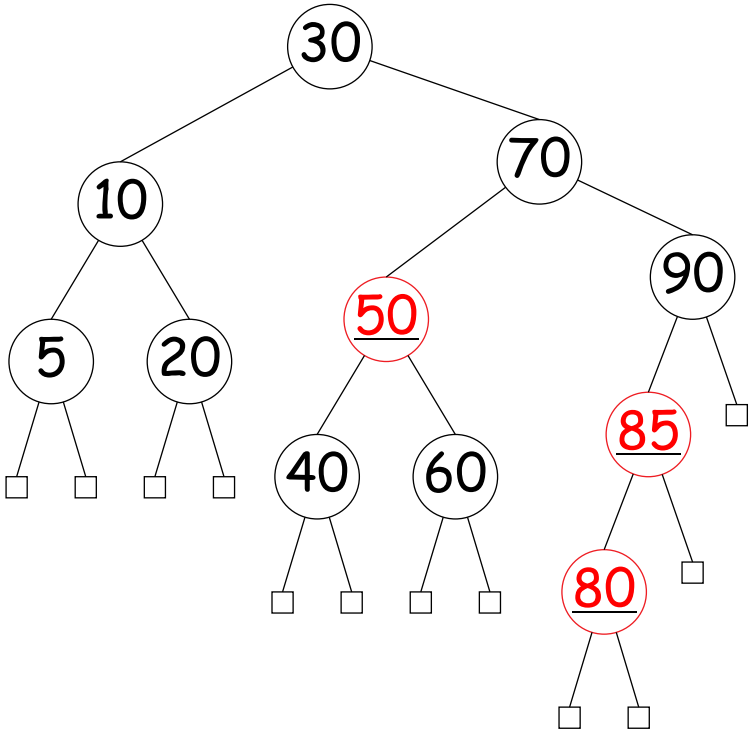
- Now consider inserting 85. We need fixup 1 first.

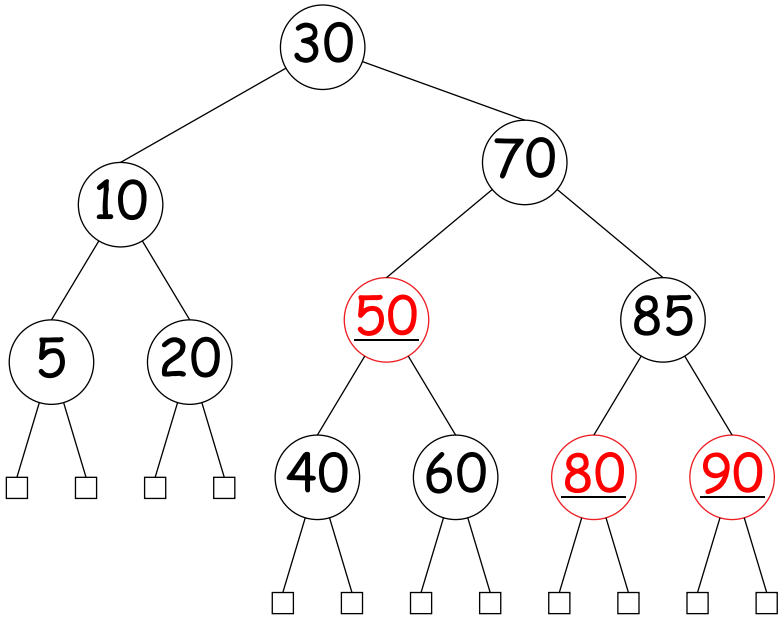




Insertion Example (IIIa)

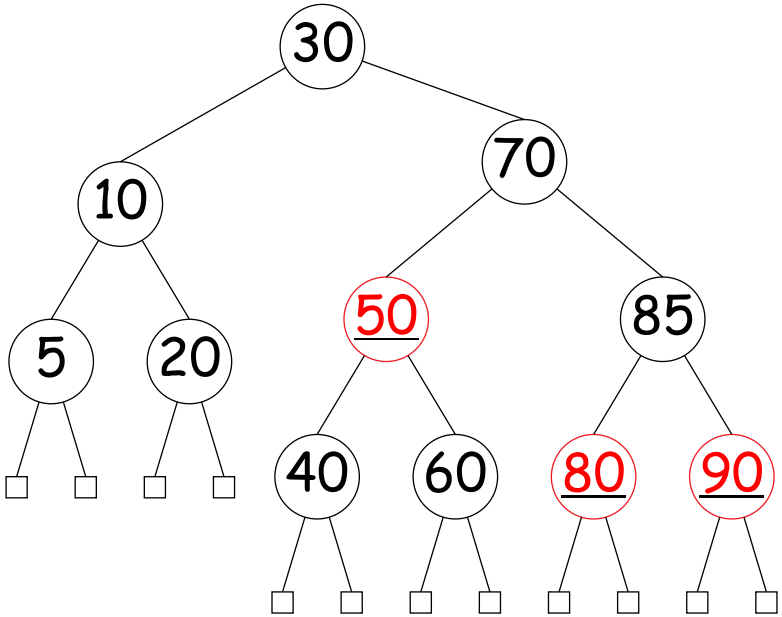
- Now apply fixup 2.

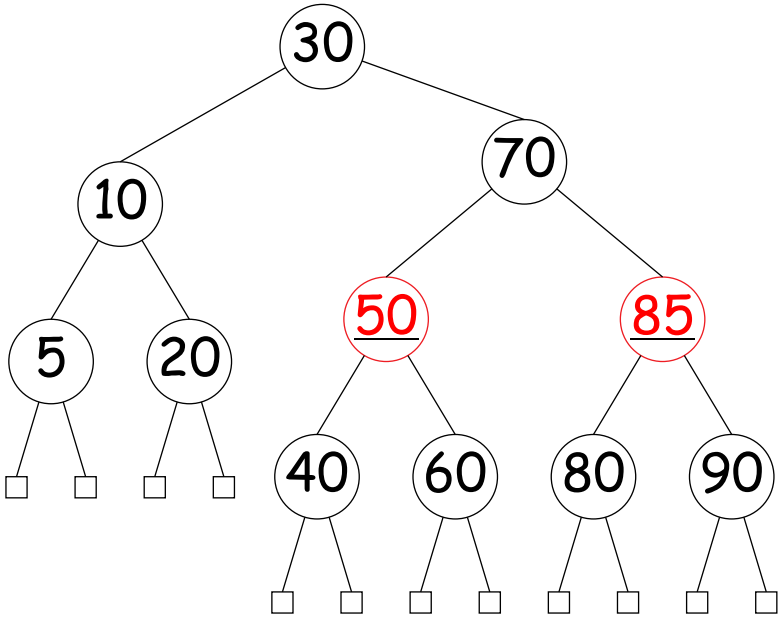




Insertion Example (IIIb)

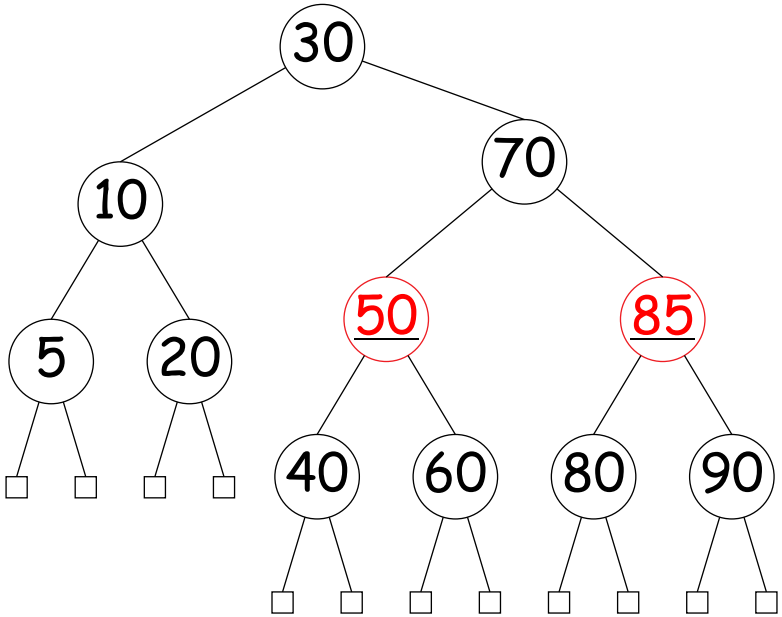
- This gives us a 4-node, so apply fixup 3.

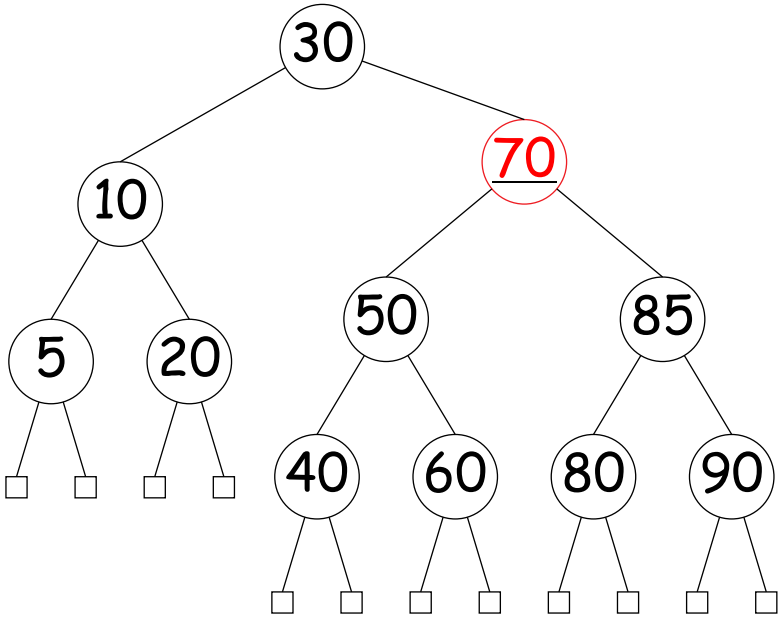




Insertion Example (IIIc)

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





Insertion Example (IIId)

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