CS61B Lecture #29

Today:

- Balanced search structures (*DS(IJ)*, Chapter 9

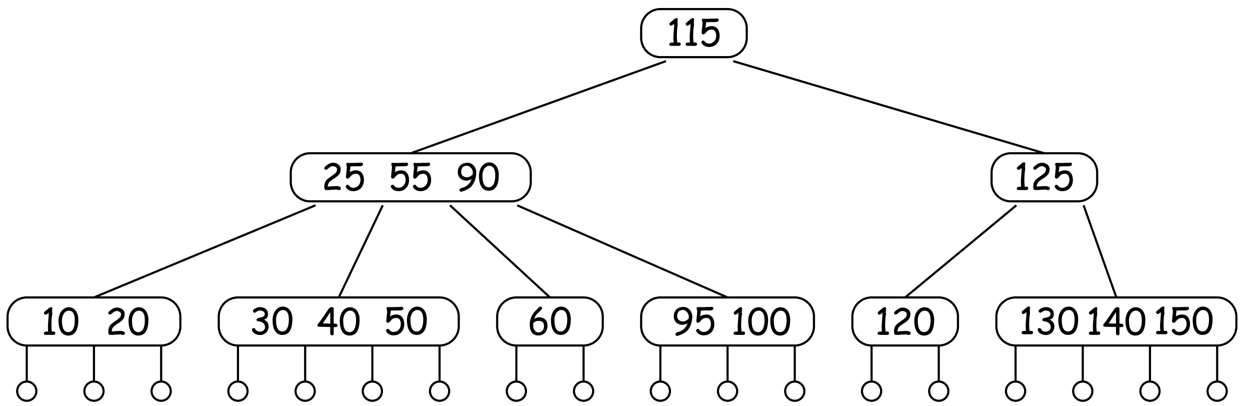
Coming Up:

- Pseudo-random Numbers (*DS(IJ)*, Chapter 11)

Balanced Search: The Problem

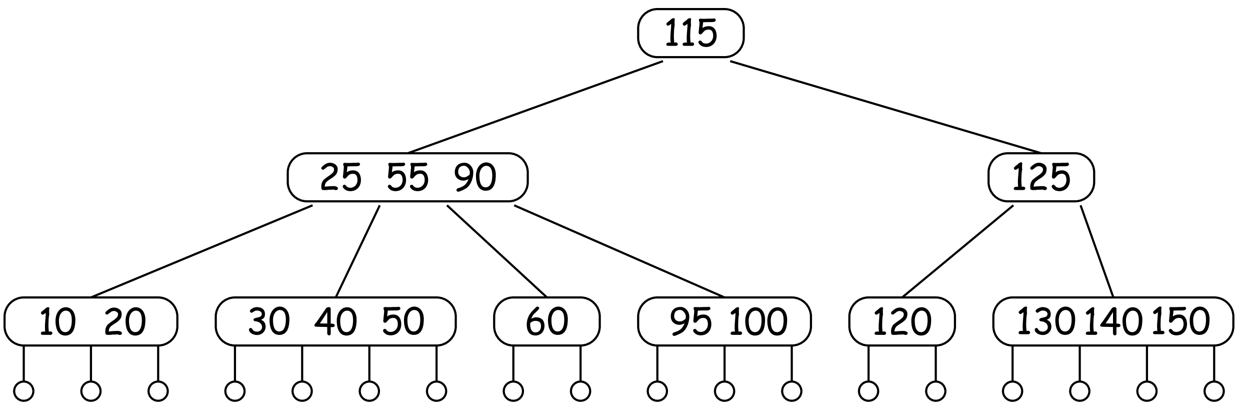
- Why are search trees important?
  - Insertion/deletion fast (on every operation, unlike has which has to expand from time to time).
  - Support range queries, sorting (unlike hash tables)
- But $O(\lg N)$ performance from binary search tree requires re keys be divided $\approx$ by some some constant > 1 at each node.
- In other words, that tree be "bushy"
- "Stringy" trees (most inner nodes with one child) perform lik lists.
- Suffices that heights of any two subtrees of a node alway by no more than constant factor K.

Example of Direct Approach: B-Trees



- *Order M B-tree* is an M-ary search tree, $M > 2$.
- Obeys search-tree property:
  - Keys are sorted in each node.
  - All keys in subtrees to left of a key, K, are $< K$, and all are $> K$.
- Children at bottom of tree are all empty (don't really ex equidistant from root.
- Searching is simple generalization of binary search.

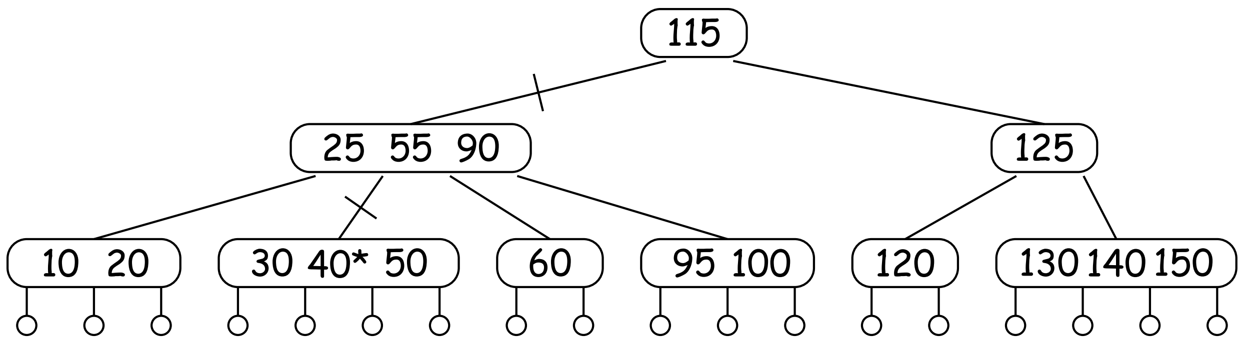
Example of Direct Approach: B-Trees



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

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

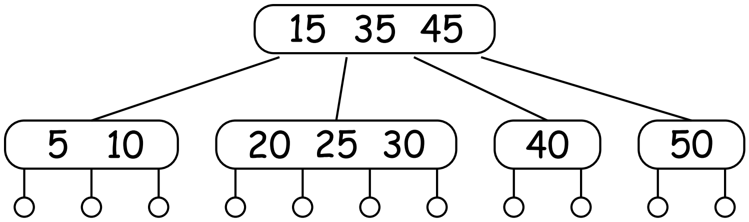
Sample Order 4 B-tree ((2,4) Tree)



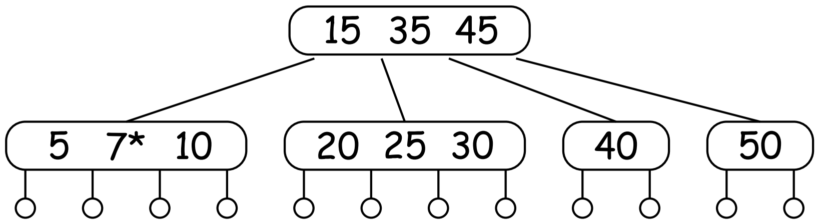
- Crossed lines show path when finding 40.
- Keys on either side of each child pointer in path bracket 40
- Each node has at least 2 children, and all leaves (little circ at the bottom, so height must be $O(\lg N)$.
- In real-life B-tree, order typically much bigger
  - comparable to size of disk sector, page, or other conveni of I/O

Inserting in B-tree (Simple Case)

- Start:

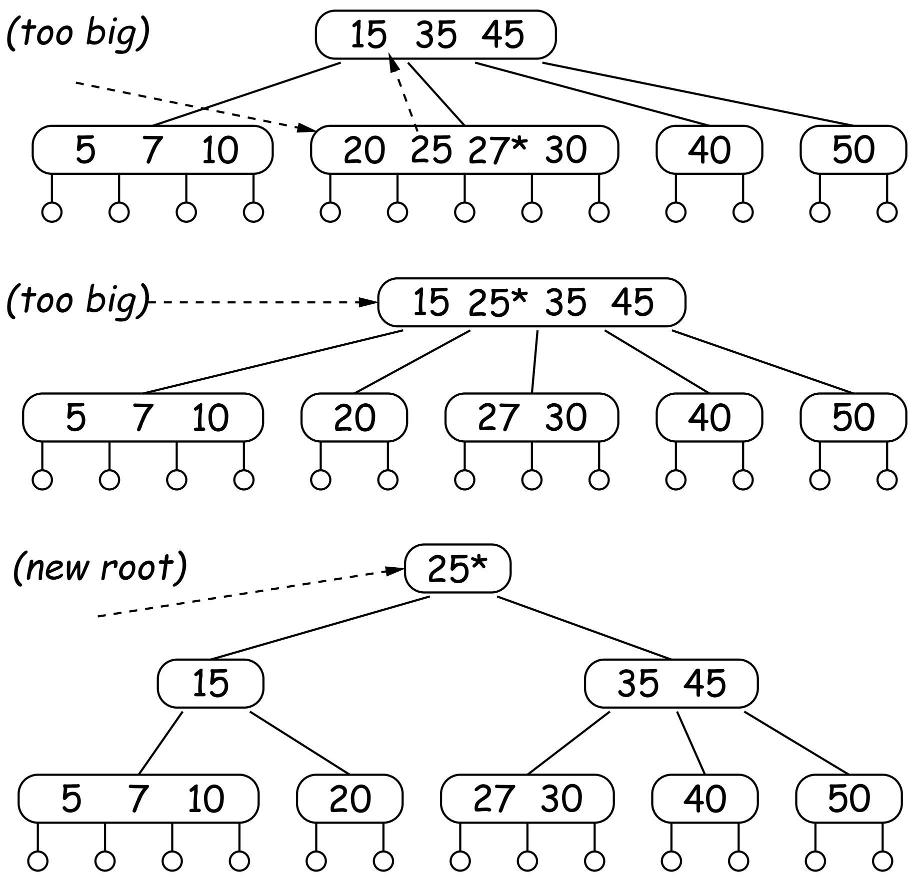


- Insert 7:



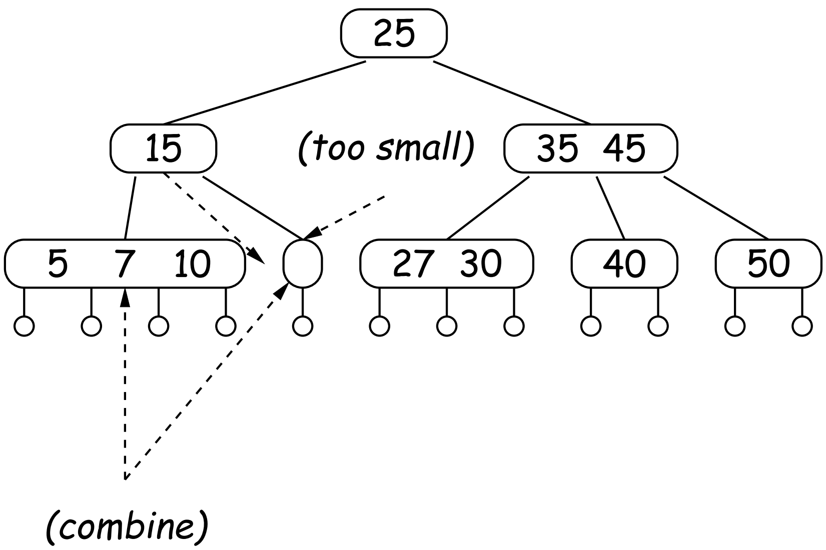
Inserting in B-Tree (Splitting)

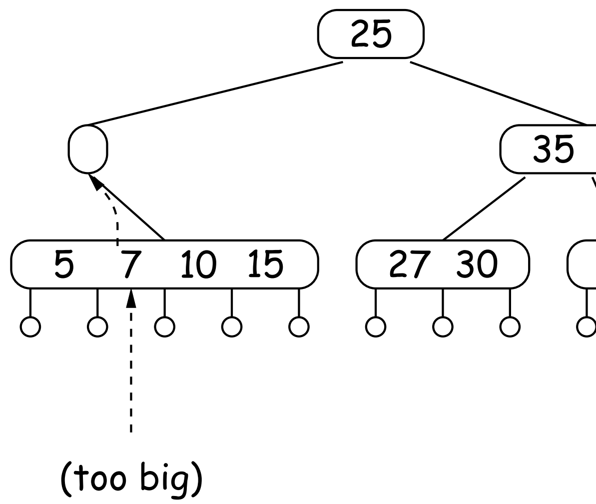
- Insert 27:

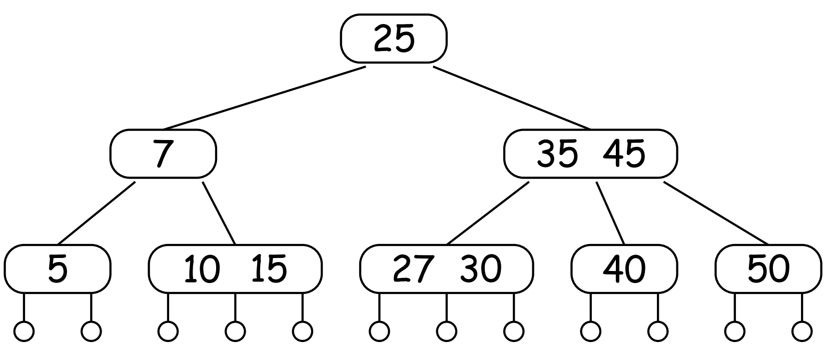


Deleting Keys from B-tree

- Remove 20 from last tree.



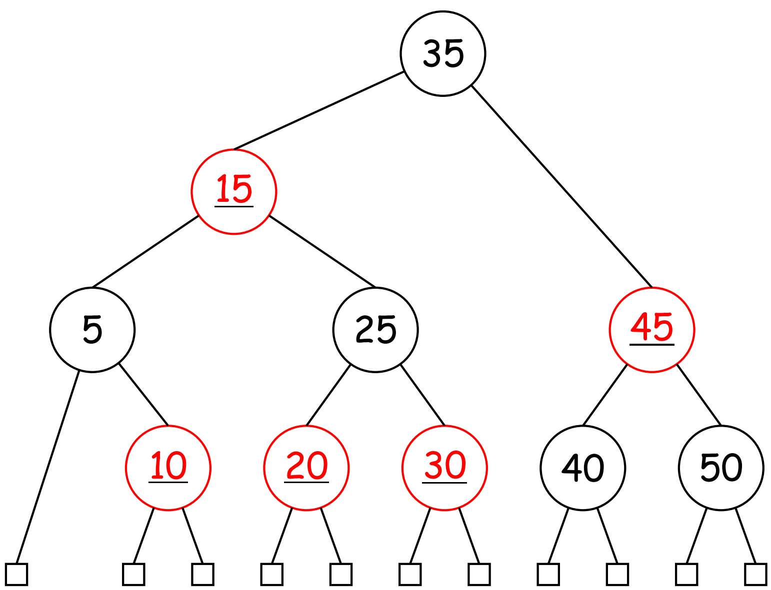




Red-Black Trees

- Red-black tree is a binary search tree with additional con
  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, tree is *rotated* and *re*
  as needed to restore balance.

Red-Black Tree Constraints

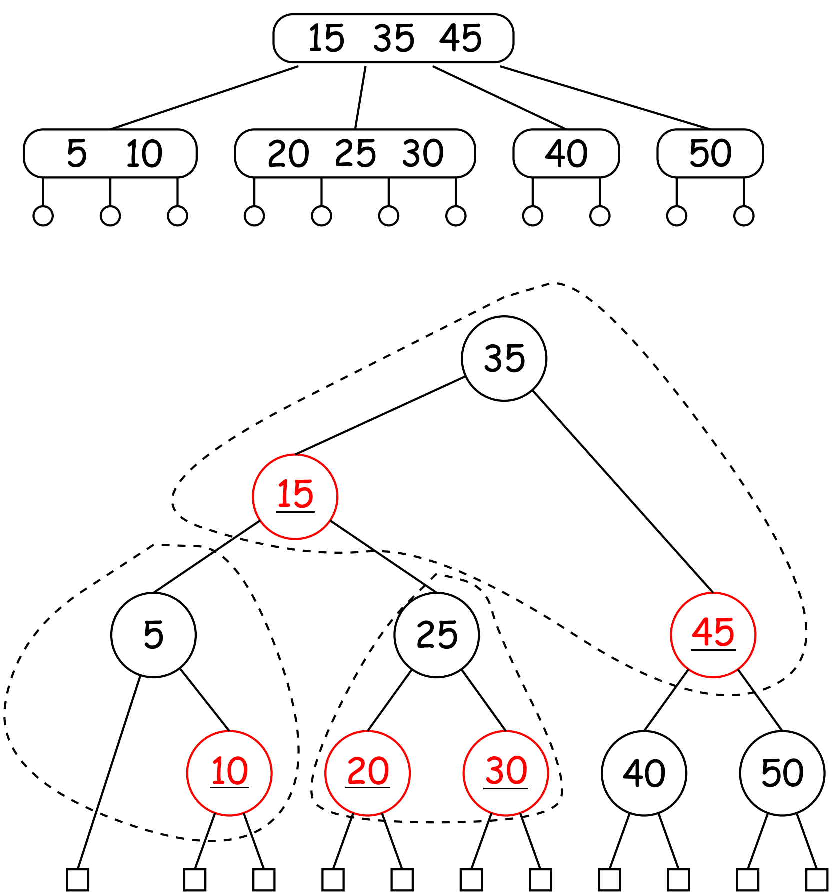


1. Each node is (conceptually) colored red or black.
2. Root is black.
3. Every leaf node contains no data (as for B-trees) and is blac
4. Every leaf has same number of black ancestors.
5. Every internal node has two children.
6. Every red node has two black children.

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

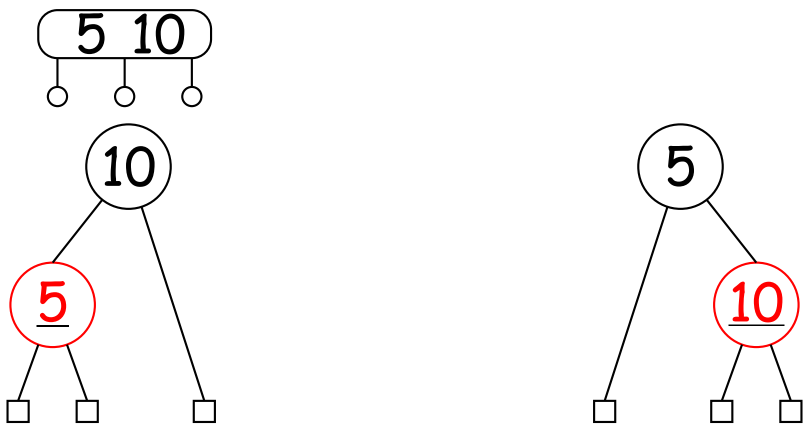
Red-Black Trees and (2,4) Trees

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



Additional Constraints: Left-Leaning Red-Black T

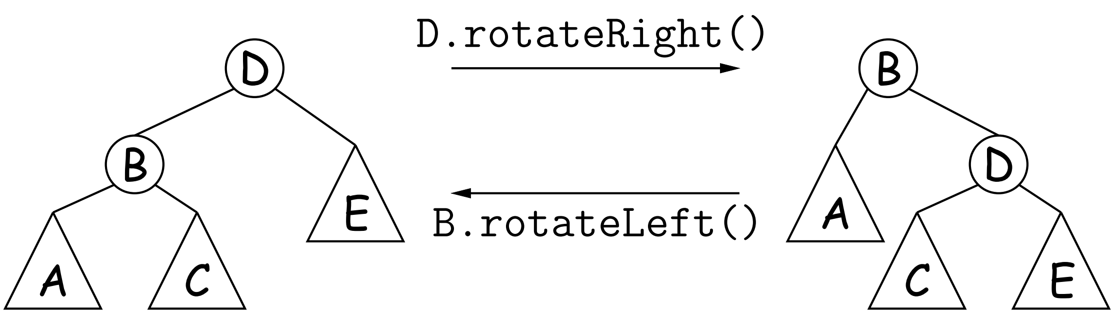
- A node in a (2,4) or (2,3) tree with three children may be sented in two different ways in a red-black tree:



- We can considerably simplify insertion and deletion in a re tree by always choosing the option on the left.
- With this constraint, there is a one-to-one relationship b (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 re trees that correspond to (2,3) trees (whose nodes have n than 3 children), so that no red-black node has two red chil

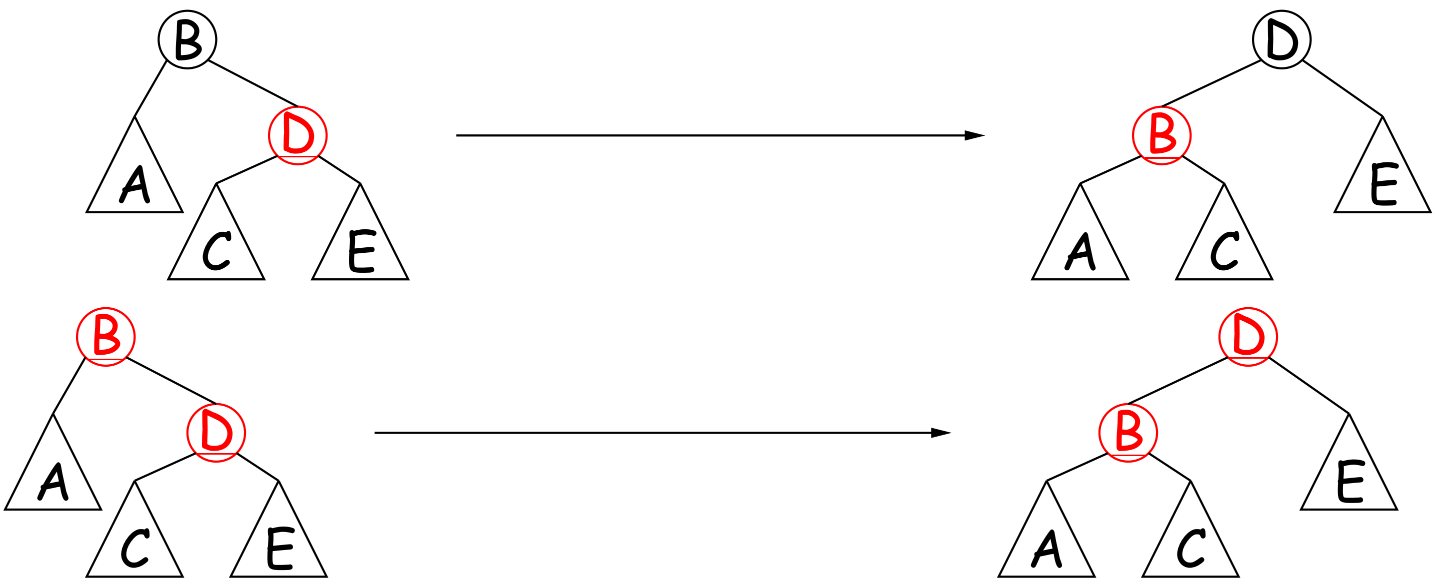
Red-Black Insertion and Rotations

- Insert at bottom just as for binary tree (color red except wh initially empty).
- Then rotate (and recolor) to restore red-black property, a balance.
- Rotation of trees *preserves* binary tree property, but chan ance.



Rotations and Recolorings

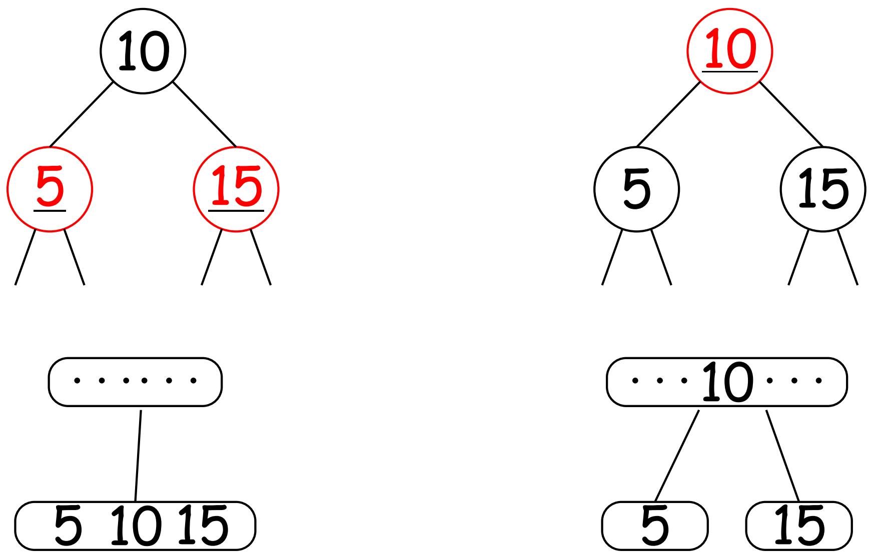
- For our purposes, we'll augment the general rotation algorith some recoloring.
- Transfer the color from the original root to the new root, a the original root red. Examples:



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

Splitting by Recoloring

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



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

The Algorithm (Sedgewick)

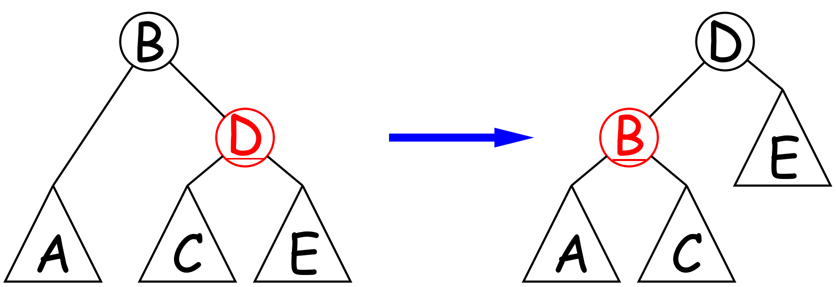
- We posit a binary-tree type `RBTree`: basically ordinary BS
plus color.
- Insertion is the same as for ordinary BSTs, but we add som
to restore the red-black properties.

```
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(
    else              tree.setRight(insert(tree.righ

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

Fixing Up the Tree

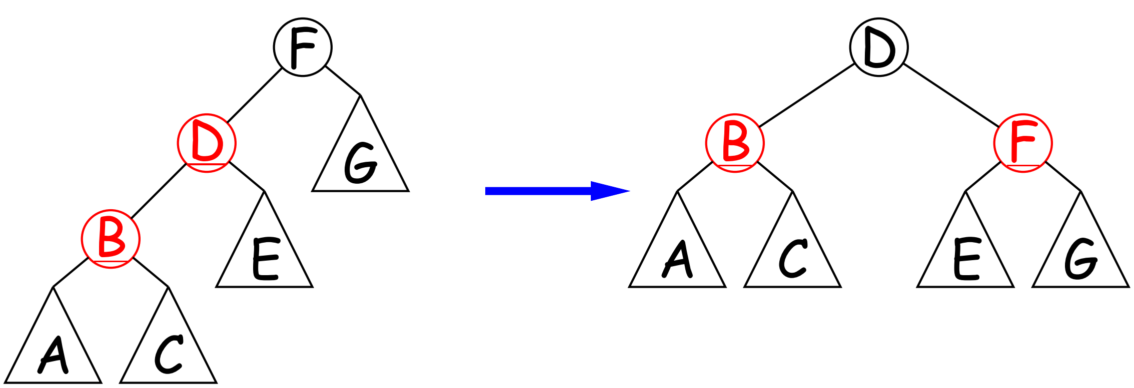
- As we return back up the BST, we restore the left-leaning re properties, and limit ourselves to red-black trees that cor to (2,3) trees by applying the following (in order) to each no
- Fixup 1: Convert right-leaning trees to left-leaning:




```
if (tree.right().i
    && tree.left()
       tree.rotate
}
```

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

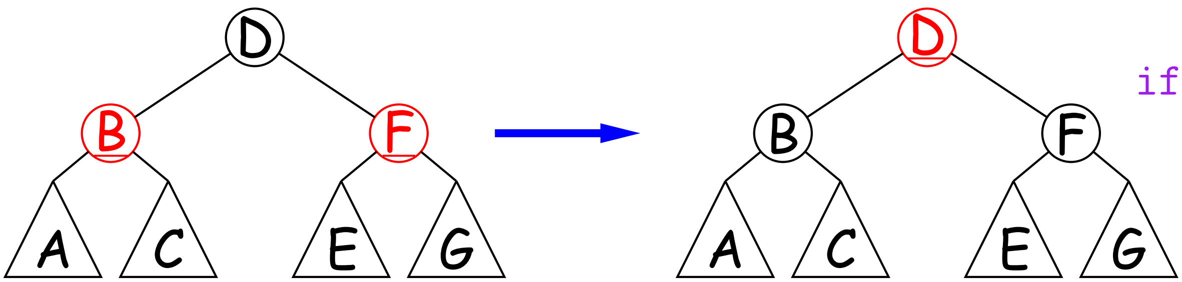
- Fixup 2: Rotate linked red nodes into a normal 4-node (temp




```
if (tree.left(
    tree.left(
       tree.r
```

Fixing Up the Tree (II)

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

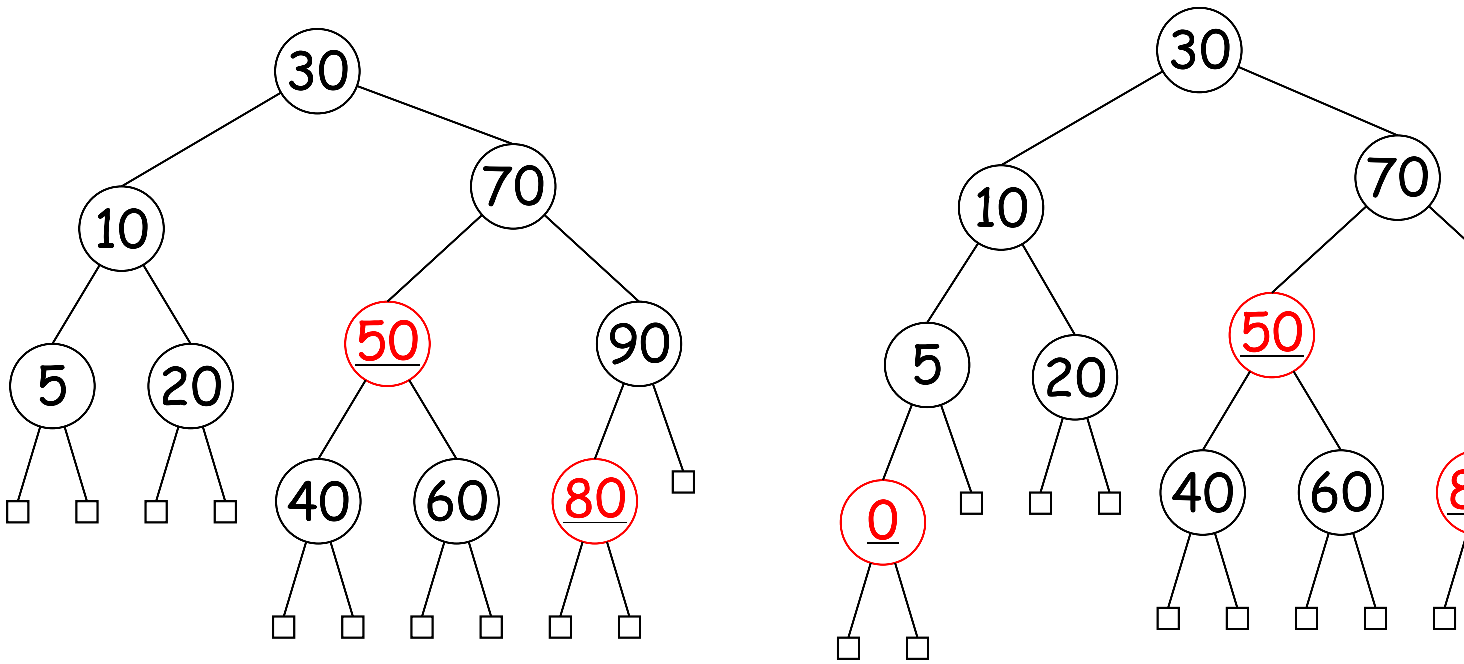



```
if (tree.left(
    tree.right
        colorF
```

- Fixup 4: As a result of other fixups, or of insertion into th tree, the root may end up red, so color the root black after of insertion and fixups are finished. (Not part of the fixup f just done at the end).

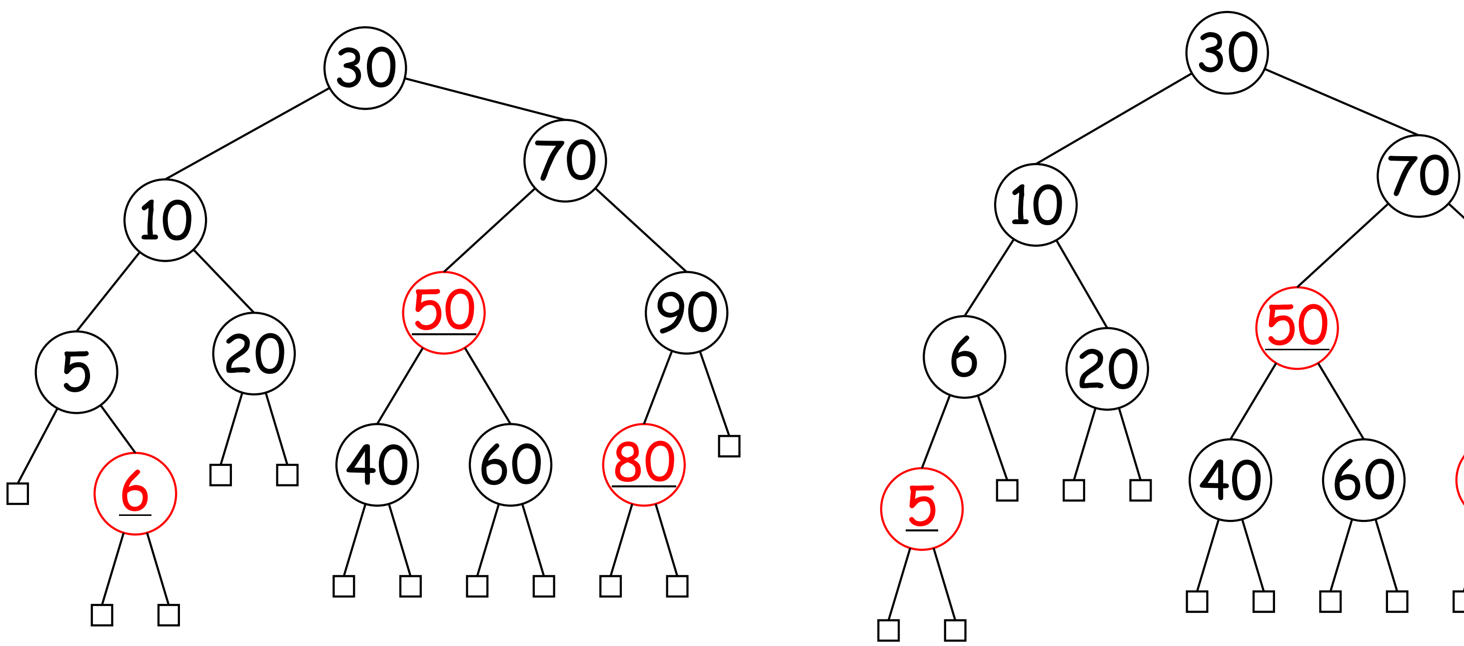
Example of Left-Leaning 2-3 Red-Black Insert

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



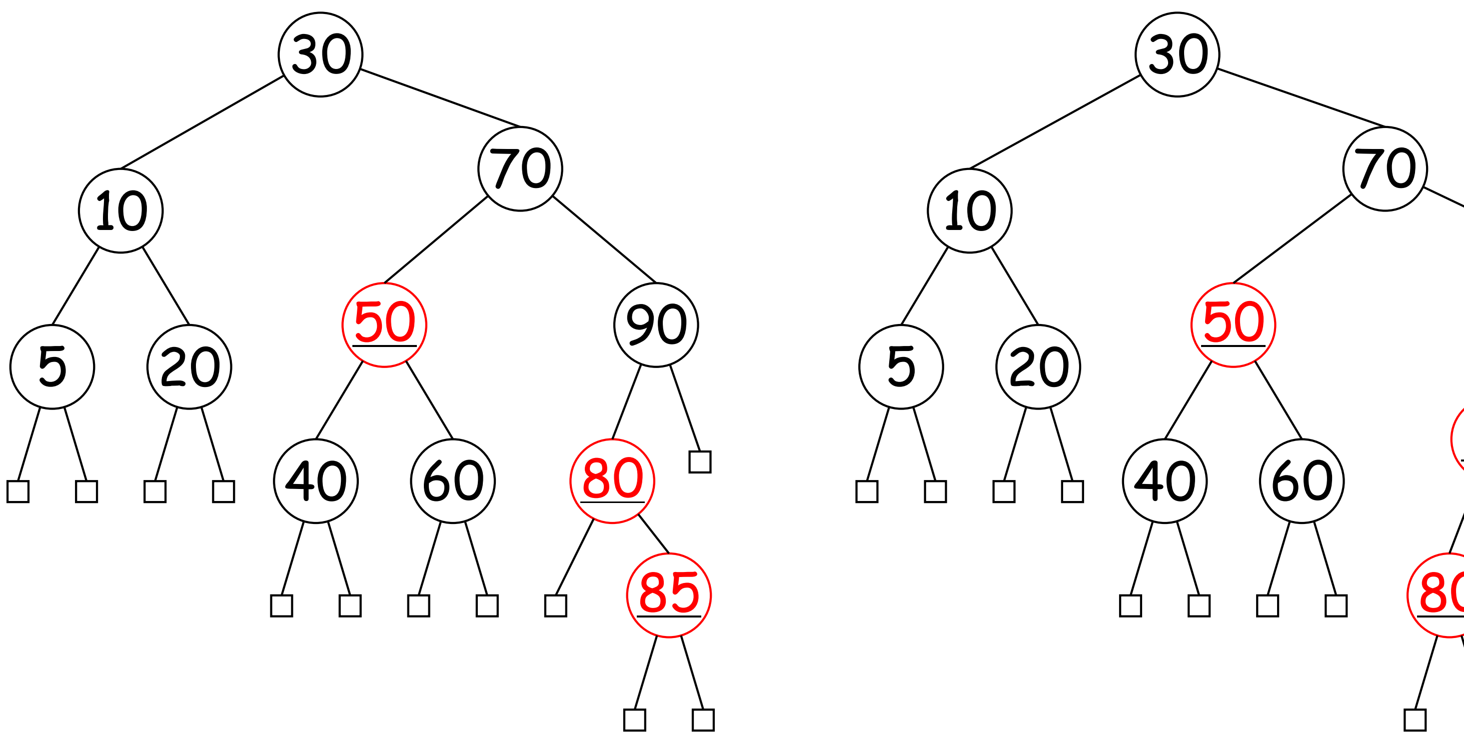
Insertion Example (II)

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



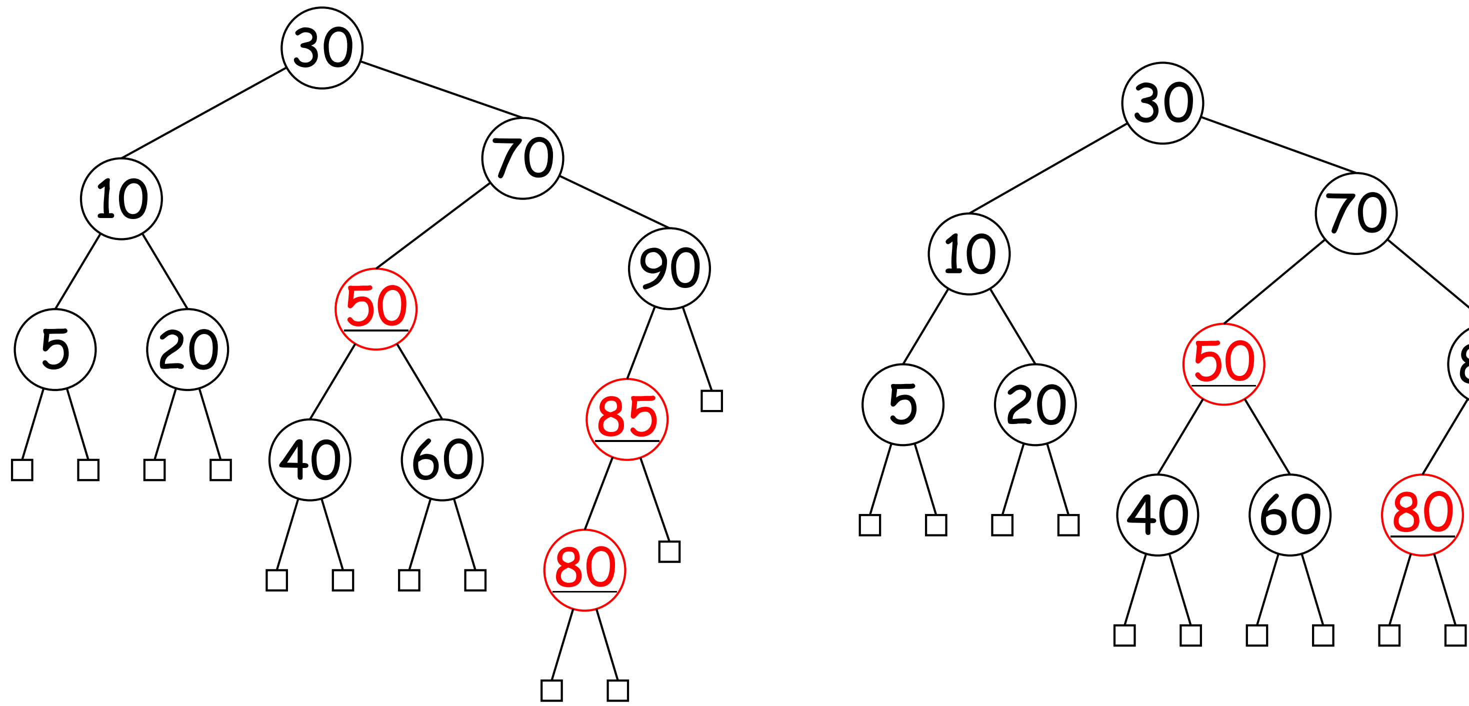
Insertion Example (III)

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



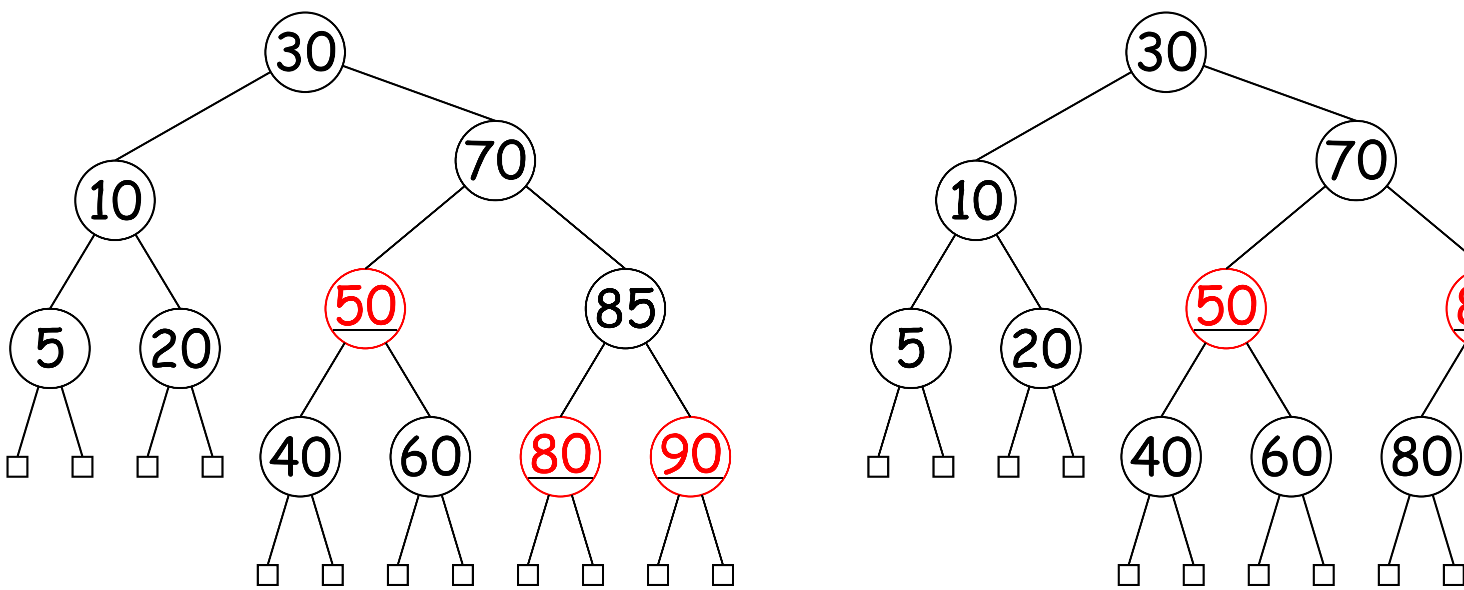
Insertion Example (IIIa)

- Now apply fixup 2.



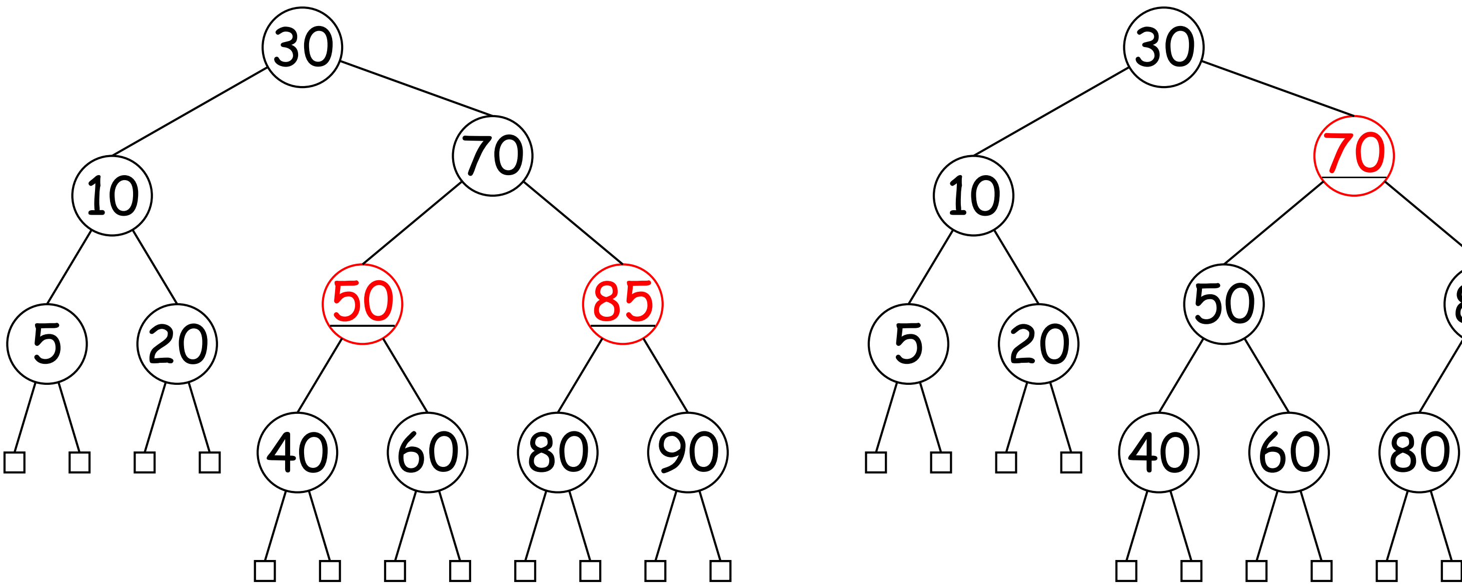
Insertion Example (IIb)

- 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.