ter 9

**Coming Up:**

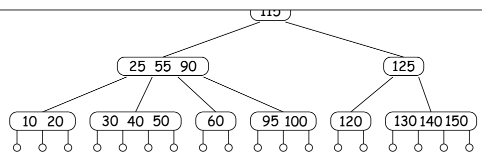
- Pseudo-random Numbers (*DS(IJ)*, Chapter 11)



- Insertion/deletion fast (on every operation, unlike hash table, which has to expand from time to time).
- Support range queries, sorting (unlike hash tables)

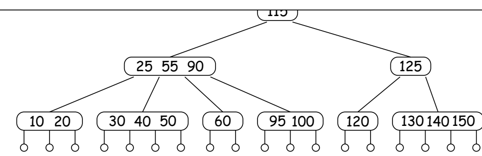
- But $O(\lg N)$ performance from binary search tree requires remaining keys be divided $\approx$ by some some constant $> 1$ at each node.
- In other words, that tree be "bushy"
- "Stringy" trees (most inner nodes with one child) perform like linked lists.
- Suffices that heights of any two subtrees of a node always differ by no more than constant factor $K$.






- *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 to right are $> K$.
- Children at bottom of tree are all empty (don't really exist) and equidistant from root.
- Searching is simple generalization of binary search.

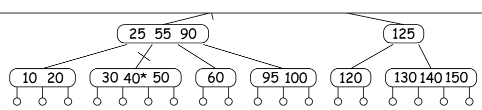





**Idea:** If tree grows/shrinks only at root, then two sides always have same height.

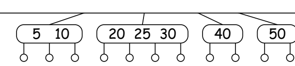
- Each node, except root, has from $\lceil M/2 \rceil$ to $M$ children, and one key "between" each two children.
- Root has from 2 to $M$ children (in non-empty tree).
- Insertion: add just above bottom; split overfull nodes as needed, moving one key up to parent.

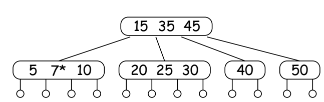


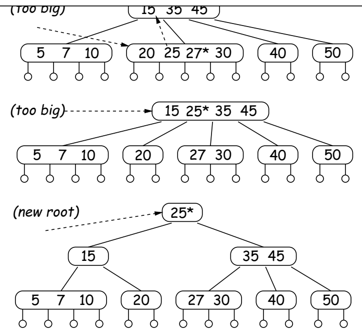



- Crossed lines show path when finding 40.
- Keys on either side of each child pointer in path bracket 40.
- Each node has at least 2 children, and all leaves (little circles) are at the bottom, so height must be $O(\lg N)$.
- In real-life B-tree, order typically much bigger
  - comparable to size of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

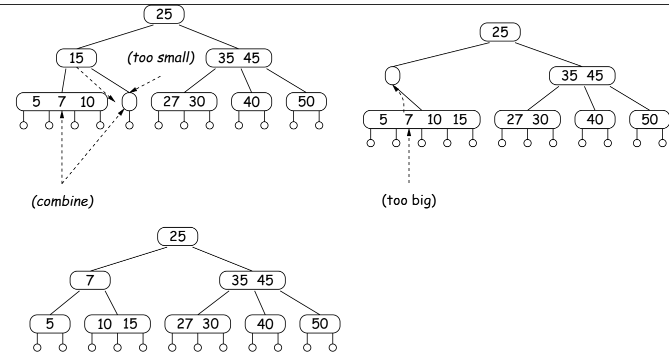


- Start: 

- Insert 7: 





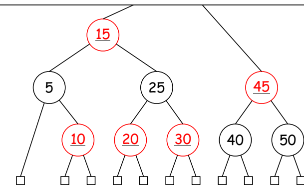






additional constraints that limit how unbal-
anced it can be.

- 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 *recolored* as needed to restore balance.





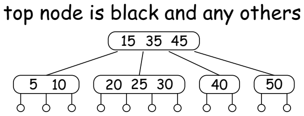

1. Each node is (conceptually) colored red or black.
2. Root is black.
3. Every leaf node contains no data (as for B-trees) and is black.
4. Every leaf has same number of black ancestors.
5. Every internal node has two children.
6. Every red node has two black children.

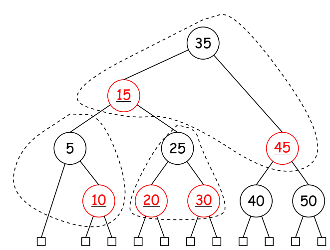




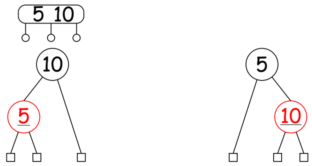
tree, and the operations on one correspond to those on the other.

- 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.



- 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*.

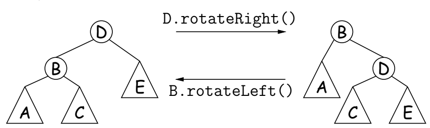


than 3 children), so that no red-black node
has two red children.



(color red except when tree initially empty).

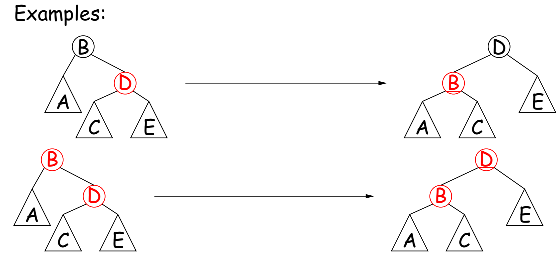
- Then rotate (and recolor) to restore red-black property, and thus balance.
- Rotation of trees *preserves* binary tree property, but changes balance.






rotation algorithms with some recoloring.

- Transfer the color from the original root to the new root, and color the original root red. Examples:

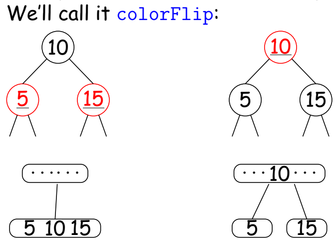



- Neither of these changes the number of black nodes along any path between the root and the leaves.



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.



cally ordinary BST nodes plus color.

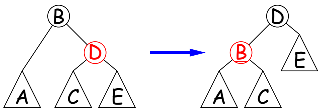
- Insertion is the same as for ordinary BSTs, but we add some fixups to restore the red-black properties.

```
  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!
  }
```

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...

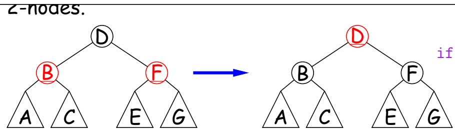
- Fixup 2: Rotate linked red nodes into a normal 4-node (temporarily).




```
tree.rotateRight();
```



2-nodes.

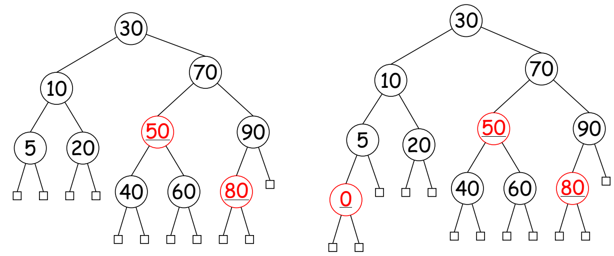



```
if (tree.left().isRed() &&
    tree.right().isRed())
       colorFlip(tree);
```

- 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).

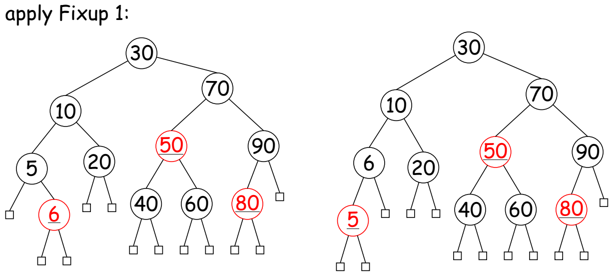


- Insert 0 into initial tree on left. No fixups needed.



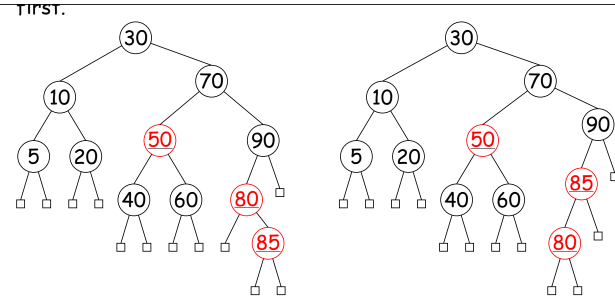


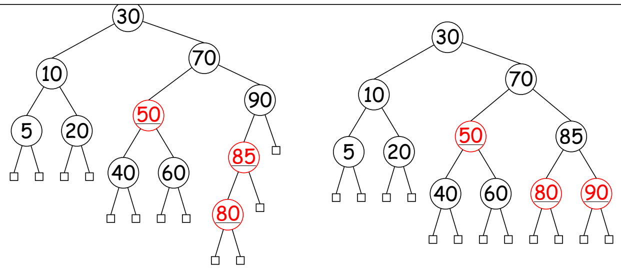

tree on the left. This is right-leaning, so apply Fixup 1:



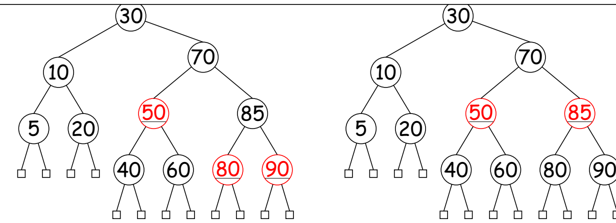



first.



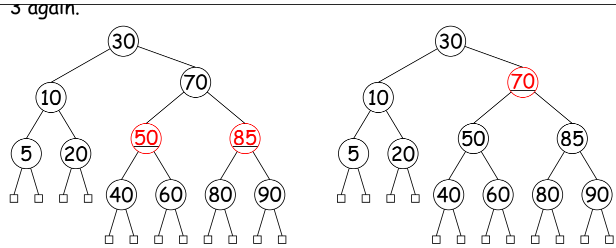







3 again.






fixup 1.