CS61B Lecture #29

rch structures (*DS(IJ)*, Chapter 9

om Numbers (*DS(IJ)*, Chapter 11)

Balanced Search: The Problem

rch trees important?

/deletion fast (on every operation, unlike hash table,
to expand from time to time).

ange queries, sorting (unlike hash tables)

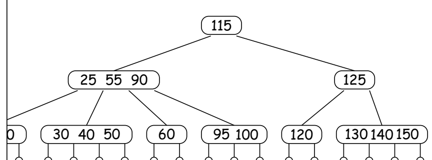
performance from binary search tree requires remaining
led $\approx$ by some some constant > 1 at each node.

ds, that tree be "bushy"

es (most inner nodes with one child) perform like linked

t heights of any two subtrees of a node always differ
han constant factor K.

mple of Direct Approach: B-Trees



tree is an M-ary search tree, $M > 2$.

h-tree property:

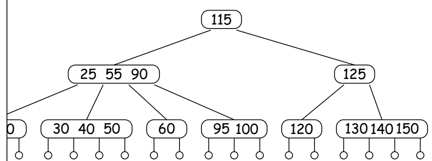
sorted in each node.

subtrees to left of a key, K, are $< K$, and all to right

bottom of tree are all empty (don't really exist) and
rom root.

simple generalization of binary search.

mple of Direct Approach: B-Trees



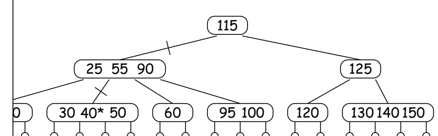
grows/shrinks only at root, then two sides always have

xcept root, has from $\lceil M/2 \rceil$ to M children, and one key
ch two children.

m 2 to M children (in non-empty tree).

dd just above bottom; split overfull nodes as needed,
ey up to parent.

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



s show path when finding 40.

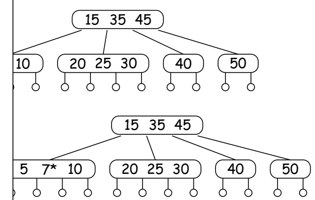
er side of each child pointer in path bracket 40.

as at least 2 children, and all leaves (little circles) are
m, so height must be $O(\lg N)$.

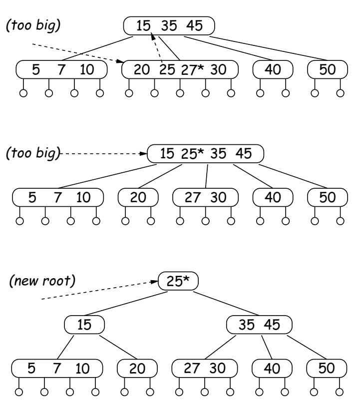
B-tree, order typically much bigger

le to size of disk sector, page, or other convenient unit

nserting in B-tree (Simple Case)

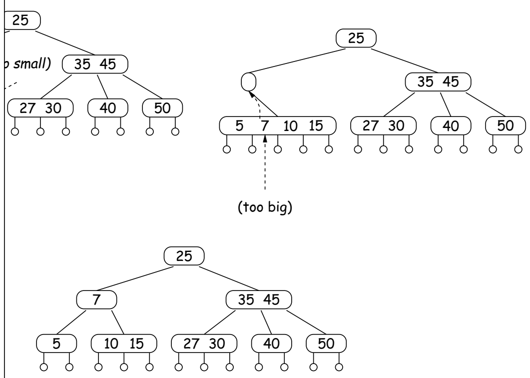


Inserting in B-Tree (Splitting)



Deleting Keys from B-tree

rom last tree.



Red-Black Trees

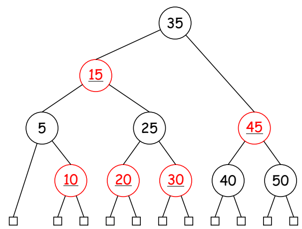
ee is a binary search tree with additional constraints w unbalanced it can be.

ing is always $O(\lg N)$.

a's `TreeSet` and `TreeMap` types.

are inserted or deleted, tree is *rotated* and *recolored* restore balance.

Red-Black Tree Constraints



(conceptually) colored red or black.

.

ode contains no data (as for B-trees) and is black.

as same number of black ancestors.

al node has two children.

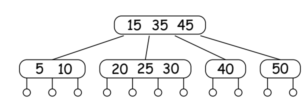
de has two black children.

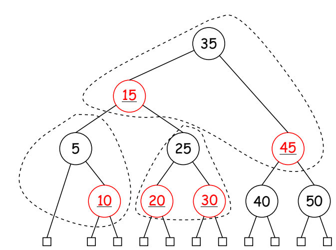
, 5, and 6 guarantee $O(\lg N)$ searches.

ed-Black Trees and (2,4) Trees

ack tree corresponds to a (2,4) tree, and the operations spond to those on the other.

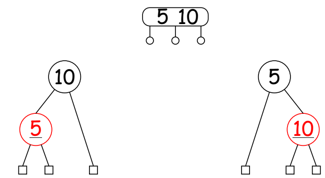
f (2,4) tree corresponds to a cluster of 1–3 red-black h the top node is black and any others are red.





onstraints: Left-Leaning Red-Black Trees

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



iderably simplify insertion and deletion in a red-black ys choosing the option on the left.

onstraint, there is a one-to-one relationship between nd red-black trees.

trees are called *left-leaning red-black trees.*

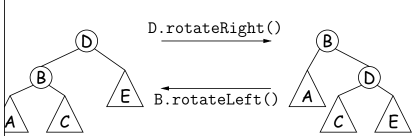
r simplification, let's restrict ourselves to red-black orrespond to (2,3) trees (whose nodes have no more en), so that no red-black node has two red children.

ed-Black Insertion and Rotations

ttom just as for binary tree (color red except when tree
y).

(and recolor) to restore red-black property, and thus

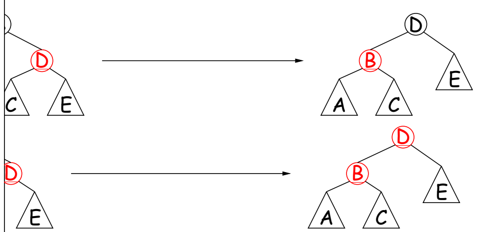
rees *preserves* binary tree property, but changes bal-



Rotations and Recolorings

oses, we'll augment the general rotation algorithms with
ing.

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

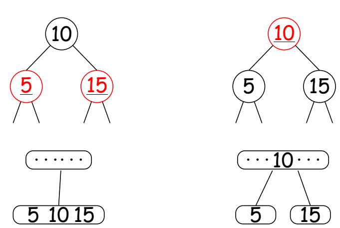


hese changes the number of black nodes along any path
root and the leaves.

Splitting by Recoloring

ms will temporarily create nodes with too many children,
t them up.

oloring allows us to split nodes. We'll call it colorFlip:



joins the parent node, splitting the original.

The Algorithm (Sedgewick)

binary-tree type RBTree: basically ordinary BST nodes

the same as for ordinary BSTs, but we add some fixups
he red-black properties.

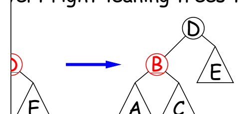
```
rt(RBTree tree, KeyType key) {
e == null)
urn new RBTree(key, null, null, RED);
 = key.compareTo(tree.label());
 (cmp < 0) tree.setLeft(insert(tree.left(), key));
           tree.setRight(insert(tree.right(), key));

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


Fixing Up the Tree

back up the BST, we restore the left-leaning red-black
nd limit ourselves to red-black trees that correspond
s by applying the following (in order) to each node:

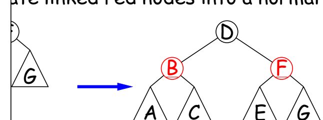
ert right-leaning trees to left-leaning:




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

node B will be red, so that both B and D end up red. This
.

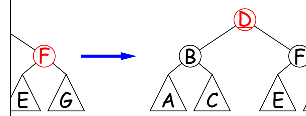
ate linked red nodes into a normal 4-node (temporarily).




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


Fixing Up the Tree (II)

ak up 4-nodes into 3-nodes or 2-nodes.

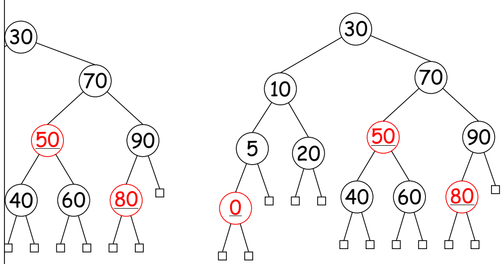



```
if (tree.left().isRed() &&
    tree.right().isRed())
       colorFlip(tree);
```

a result of other fixups, or of insertion into the empty
t may end up red, so color the root black after the rest
nd fixups are finished. (Not part of the fixup function;
the end).

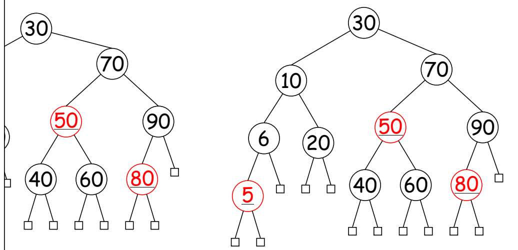
of Left-Leaning 2-3 Red-Black Insertion

initial tree on left. No fixups needed.



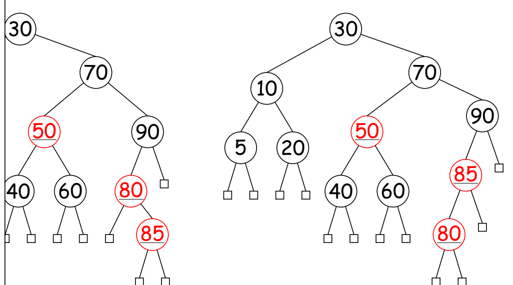
Insertion Example (II)

), let's insert 6, leading to the tree on the left. This is
, so apply Fixup 1:



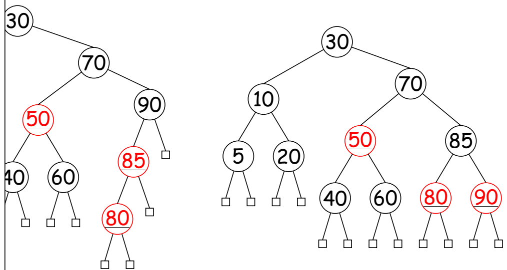
Insertion Example (III)

r inserting 85. We need fixup 1 first.



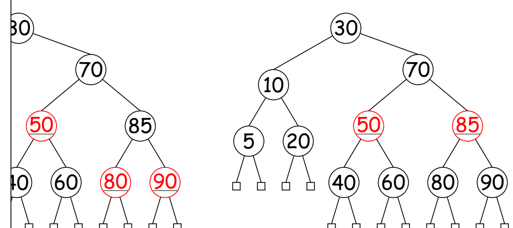
Insertion Example (IIIa)

xup 2.



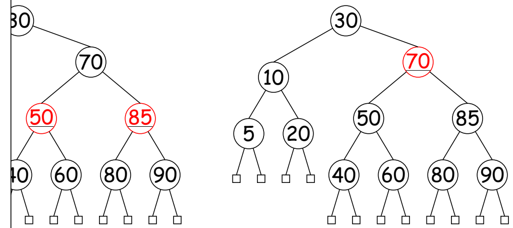
Insertion Example (IIIb)

a 4-node, so apply fixup 3.



Insertion Example (IIIc)

another 4-node, so apply fixup 3 again.



Insertion Example (IIId)

: a right-leaning tree, so apply fixup 1.