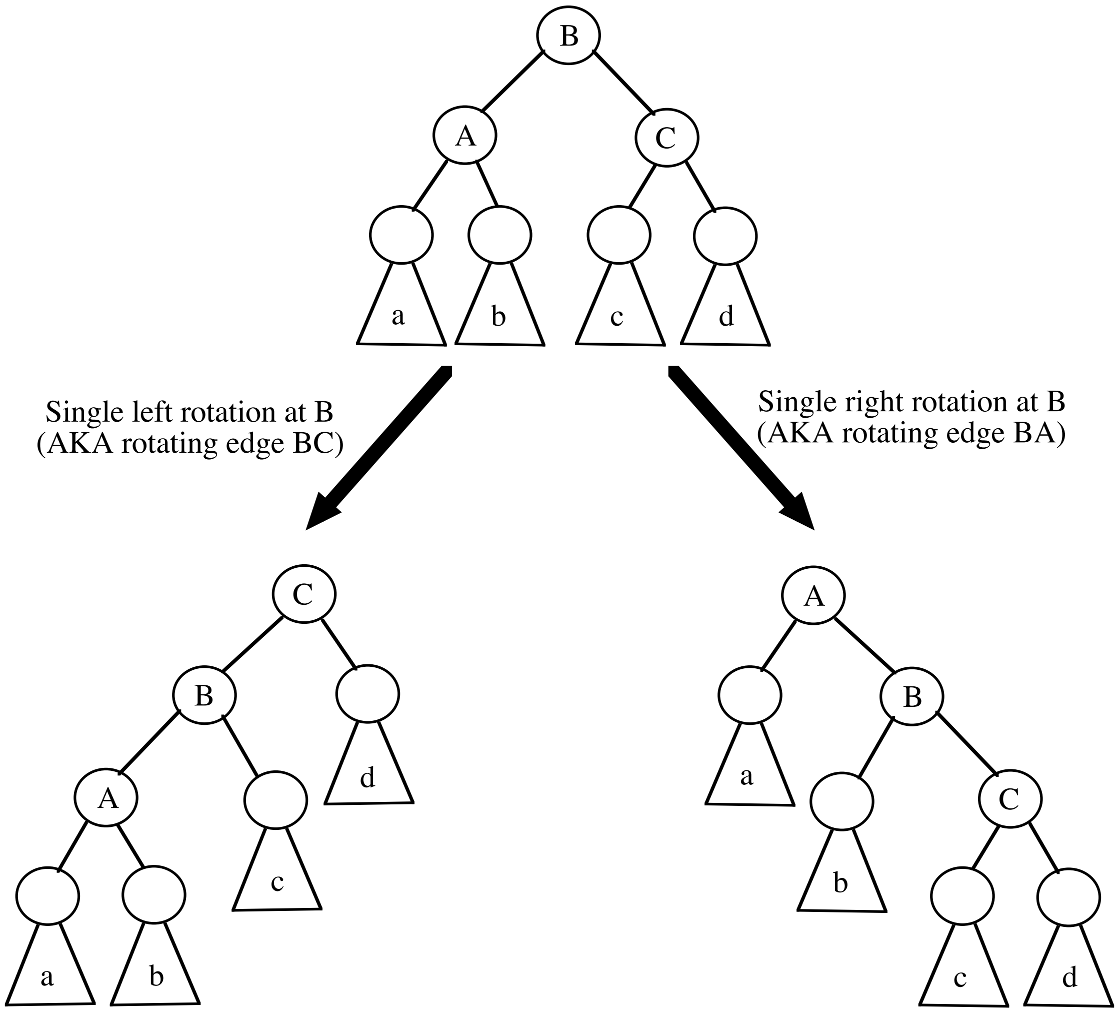
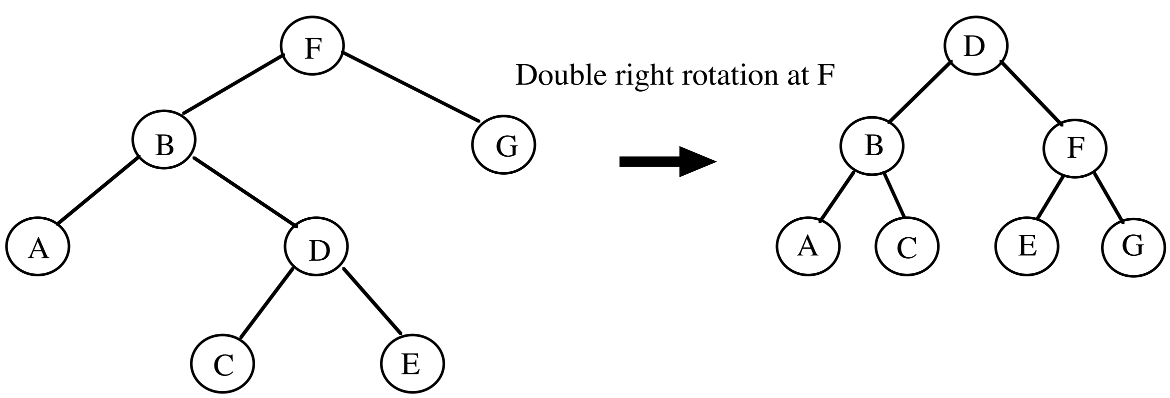
CSE 5311 Notes 2: Binary Search Trees

(Last updated 5/27/13 8:37 AM)

ROTATIONS





What two single rotations are equivalent?

(BOTTOM-UP) RED-BLACK TREES

A *red-black tree* is a binary search tree whose height is logarithmic in the number of keys stored.

1. Every node is colored red or black. (Colors are only examined during insertion and deletion)

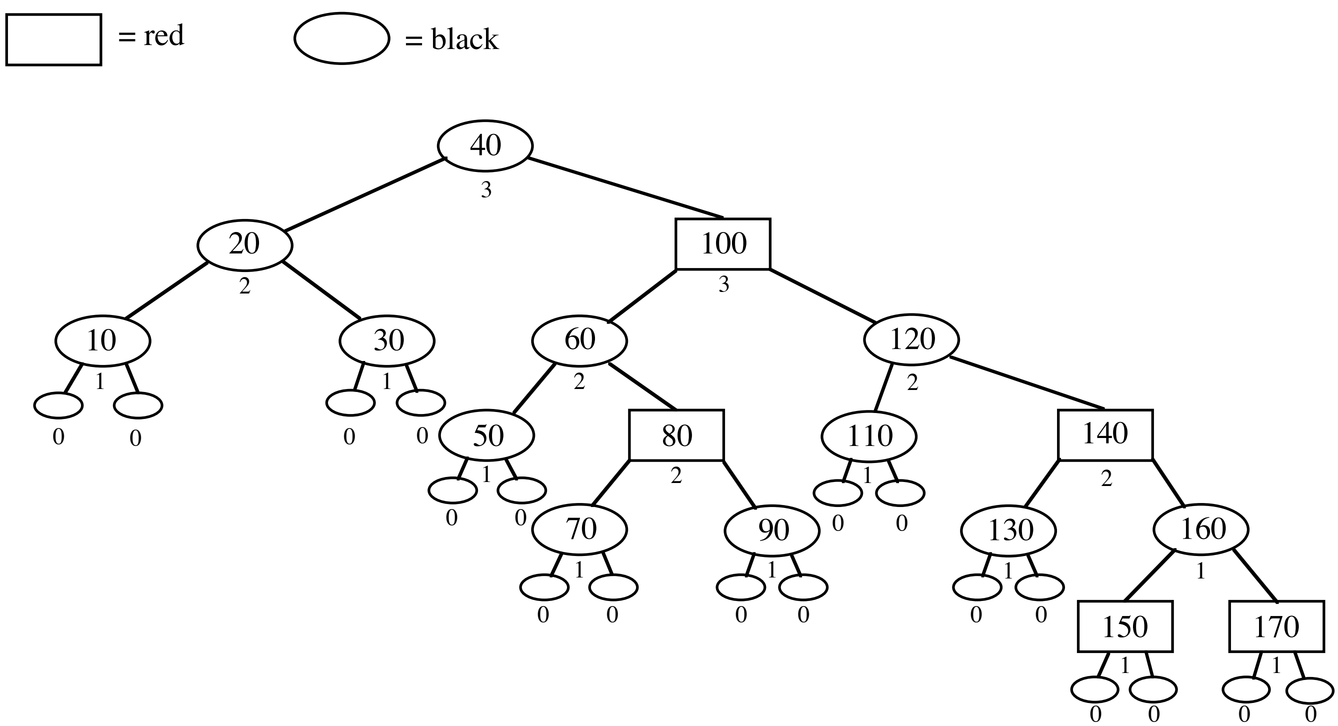
2. Every "leaf" (the sentinel) is colored black.

3. Both children of a red node are black.

4. Every simple path from a child of node X to a leaf has the same number of black nodes.

   This number is known as the *black-height* of X (bh(X)).

Example:



Observations:

1. A red-black tree with n internal nodes ("keys") has height at most 2 lg(n+1).

2. If a node X is not a leaf and its sibling is a leaf, then X must be red.

3. There may be many ways to color a binary search tree to make it a red-black tree.

4. If the root is colored red, then it may be switched to black without violating structural properties.

Insertion

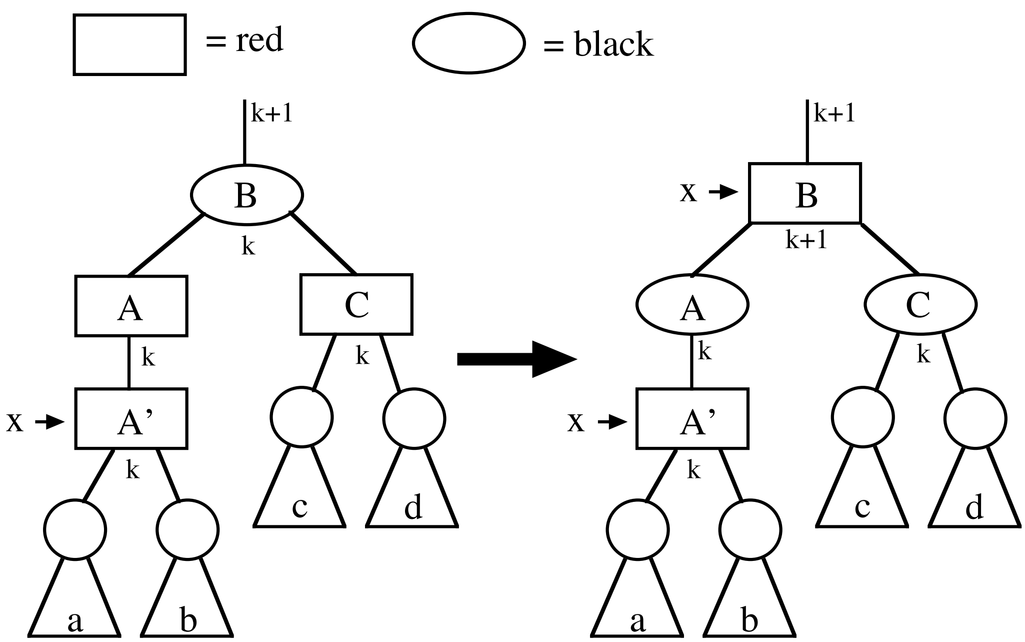
1. Start with unbalanced insert of a "data leaf" (both children are the sentinel).

2. Color of new node is _________.

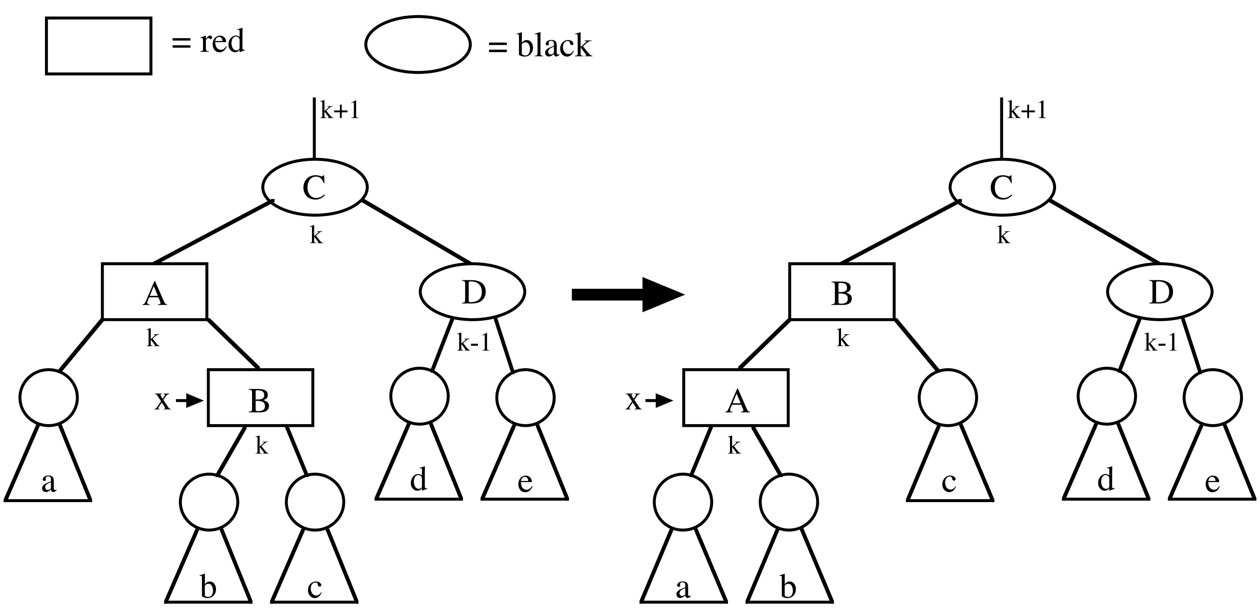
3. May violate structural property 3. Leads to three cases, along with symmetric versions.

   The x pointer points at a red node whose parent might also be red.

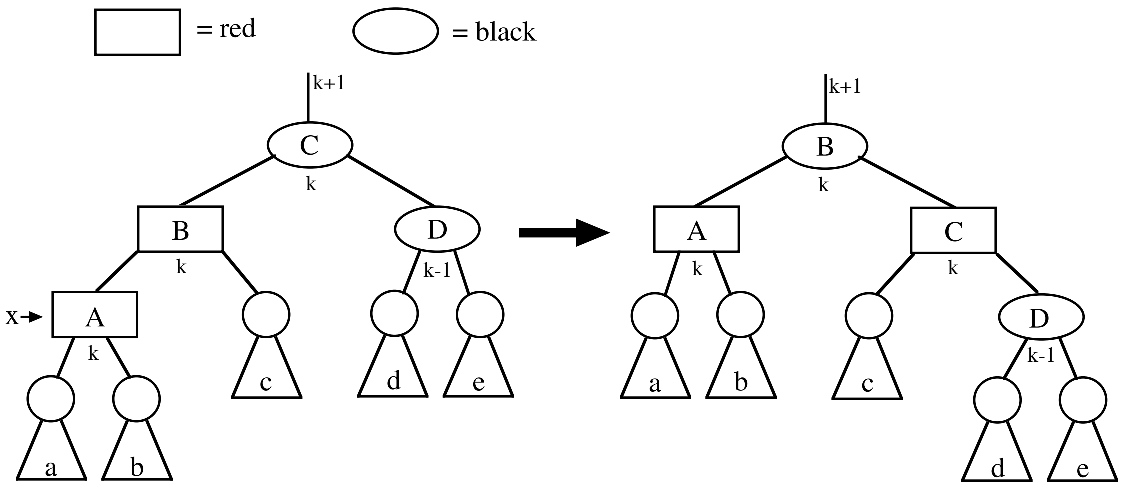
Case 1: (in 2320 Notes 12, using Sedgewick's book, this is done *top-down* before attaching new leaf)



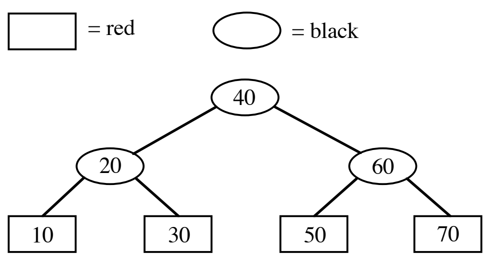
Case 2:



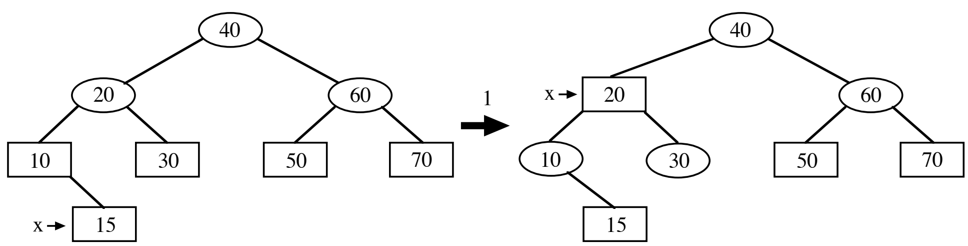
Case 3:



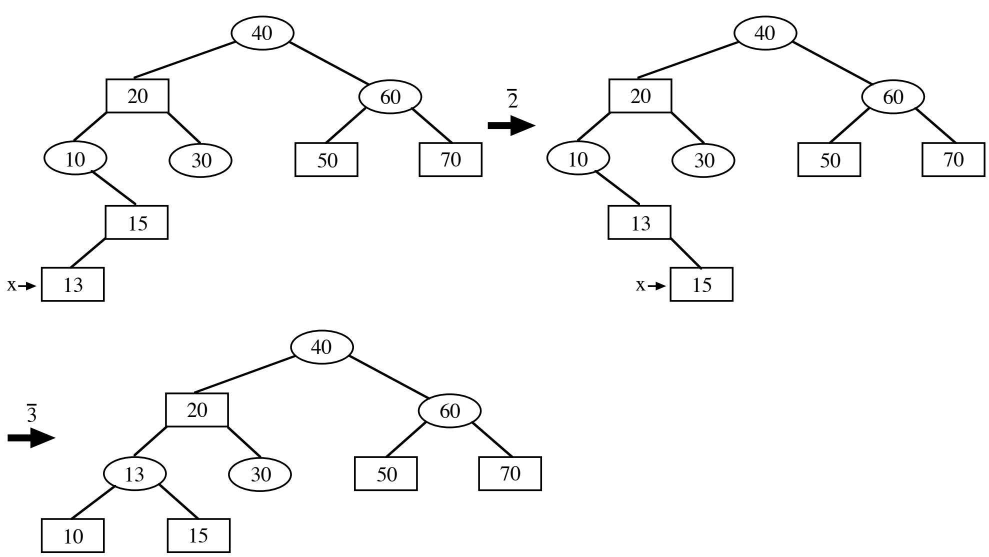
Example:



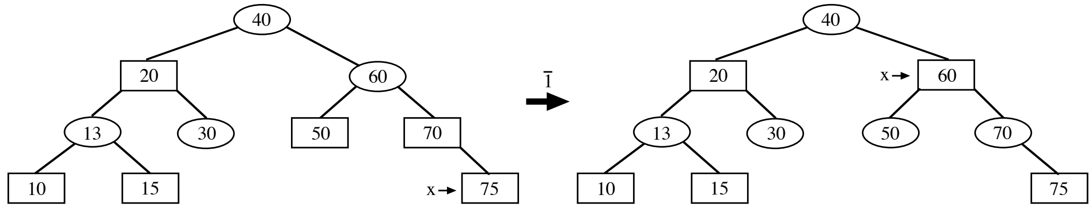
Insert 15



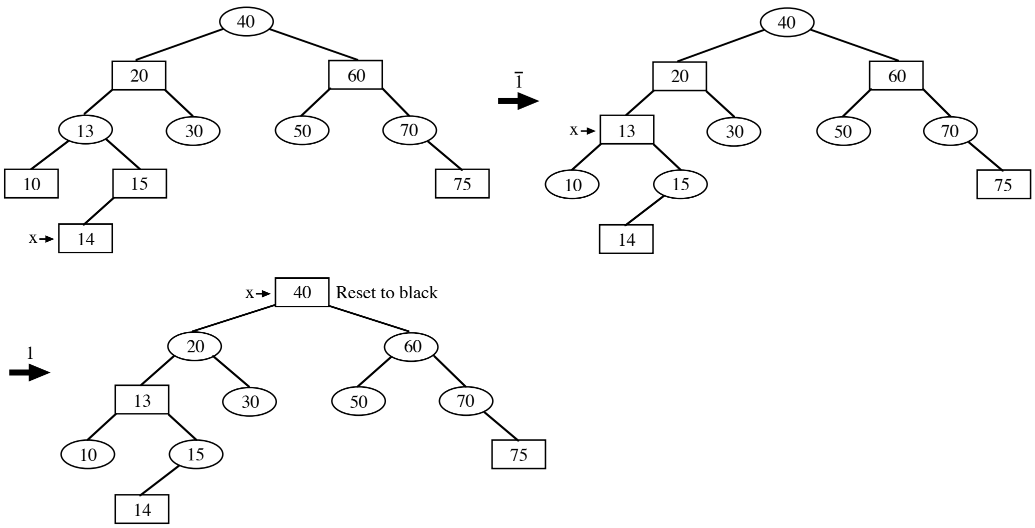
Insert 13



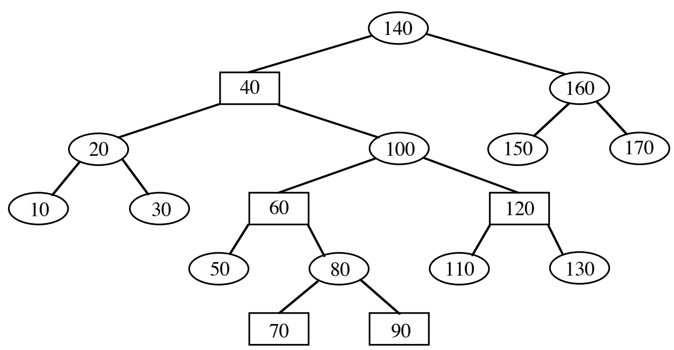
Insert 75



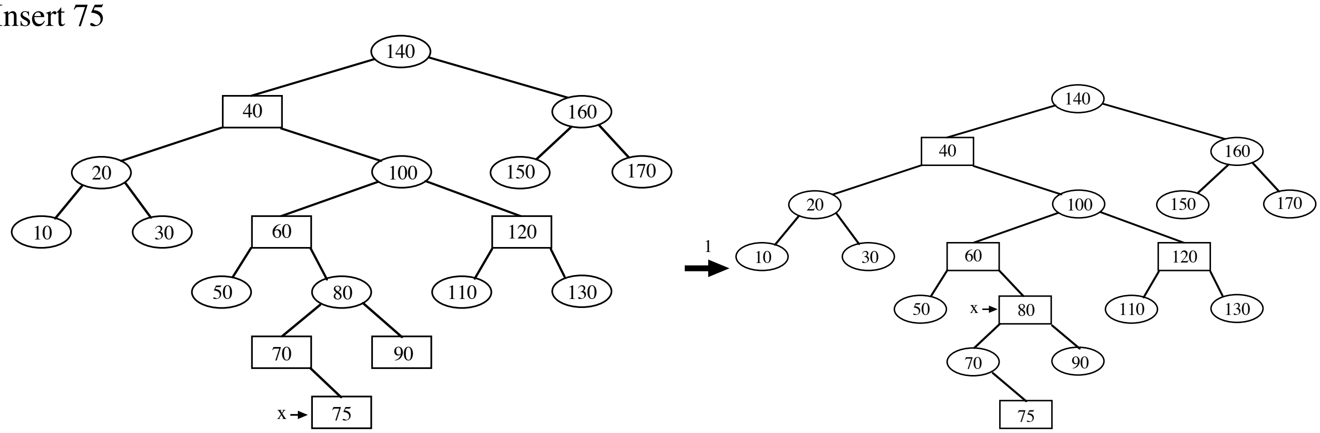
Insert 14

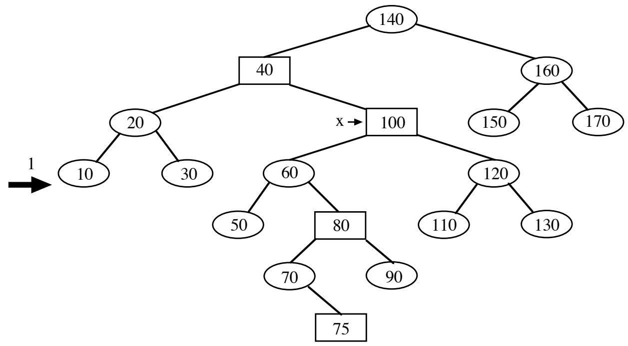


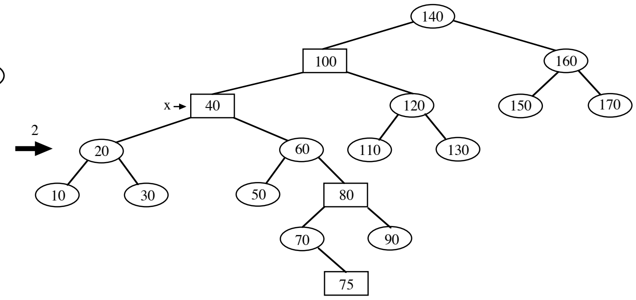
Example:

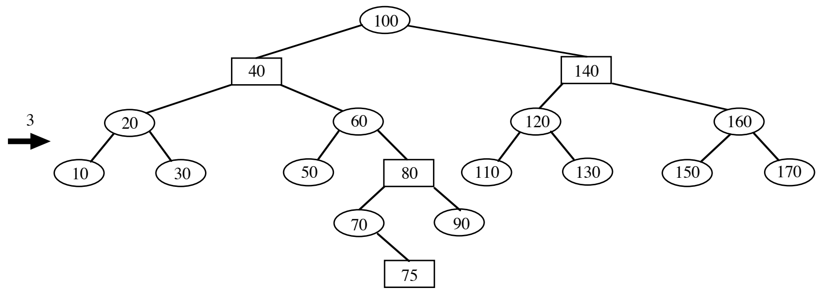


Insert 75









Deletion

Start with one of the unbalanced deletion cases:

1. Deleted node is a "data leaf".

   a. Splice around to sentinel.

   b. Color of deleted node?

      Red ⇒ Done

      Black ⇒ Set "double black" pointer at sentinel.
      Determine which of four rebalancing cases applies.

2. Deleted node is parent of one "data leaf".

   a. Splice around to "data leaf"

   b. Color of deleted node?

      Red ⇒ Not possible

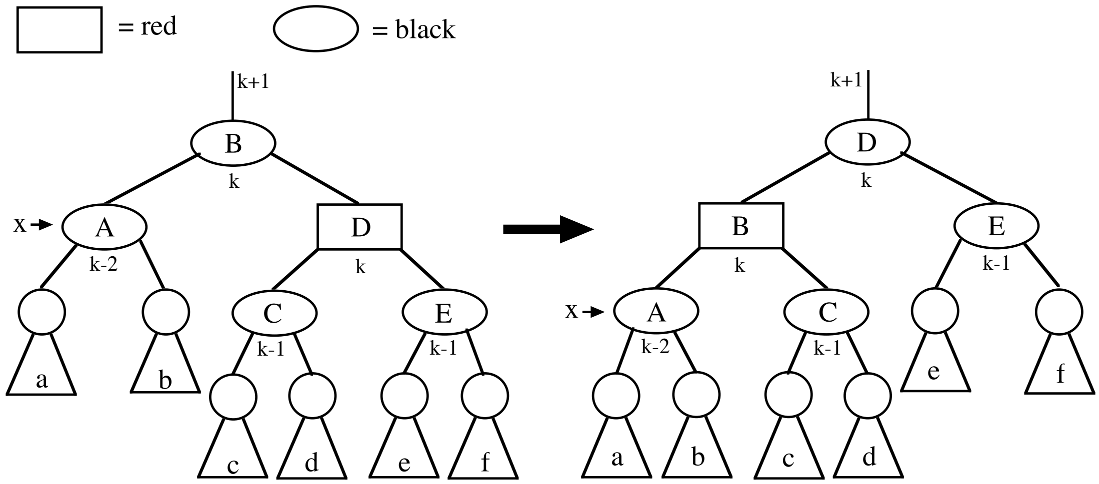
      Black ⇒ "data leaf" must be red. Change its color to black.

3. Node with key-to-delete is parent of two "data nodes".

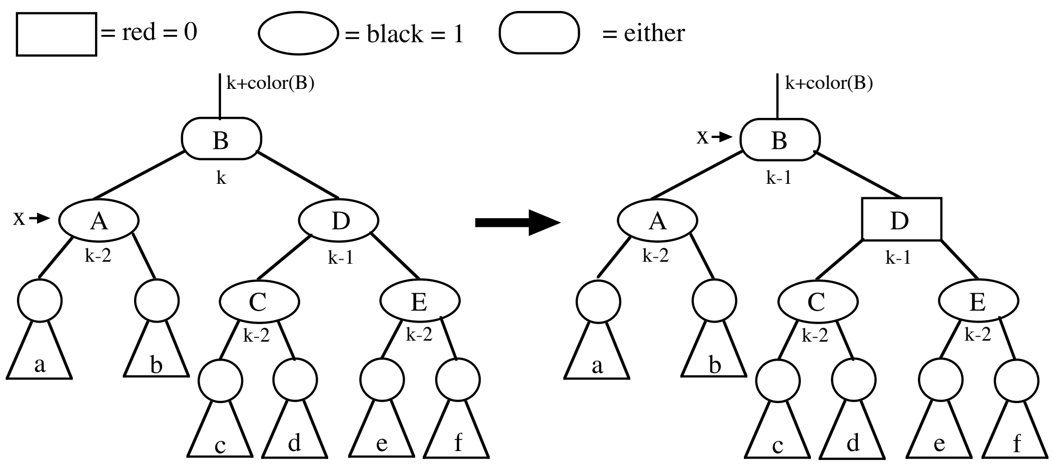
   a. "Steal" key and data from successor (but not the color).

   b. "Delete" successor using the appropriate one of the previous two cases.

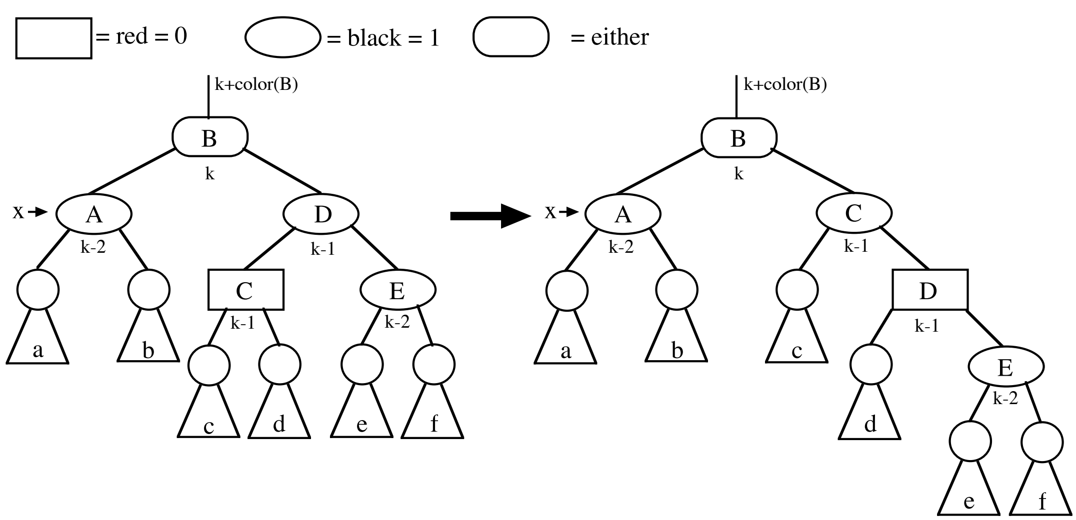
Case 1:



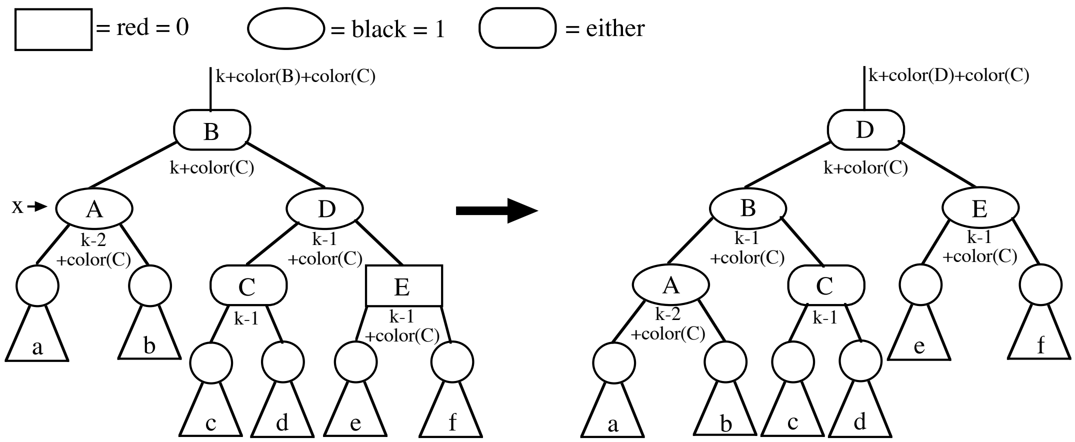
Case 2:



Case 3:

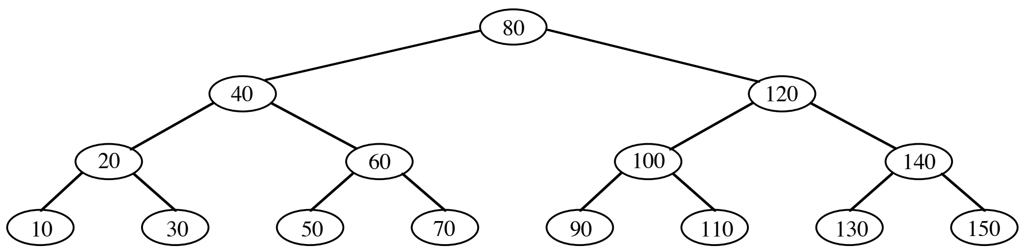


Case 4:

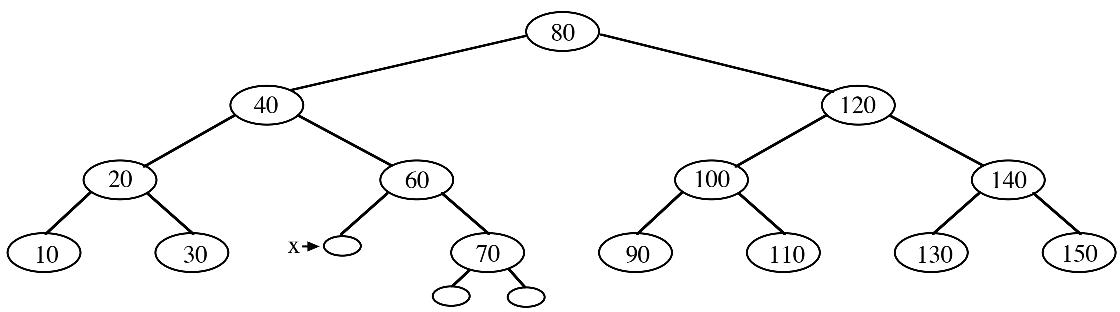


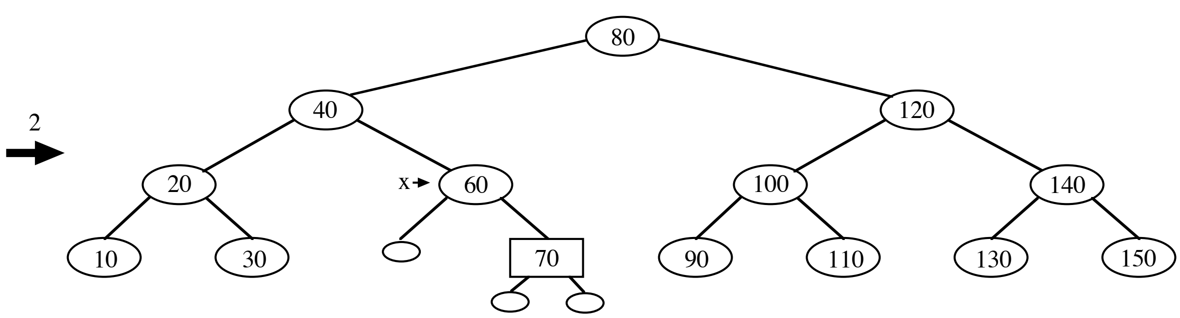
(At most three rotations occur while processing the deletion of one key)

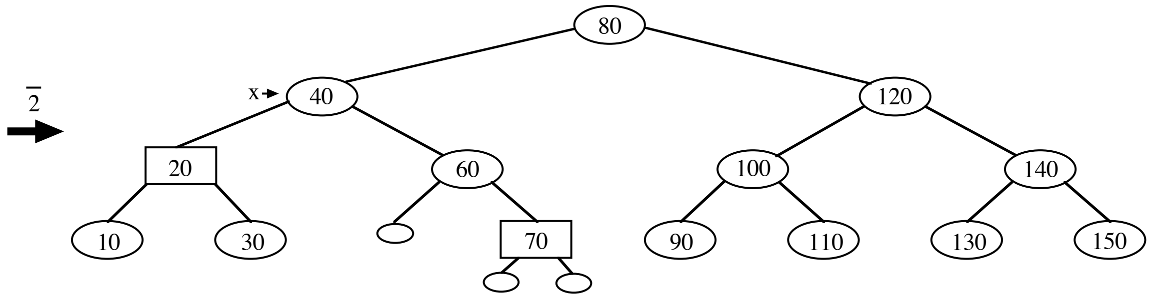
Example:

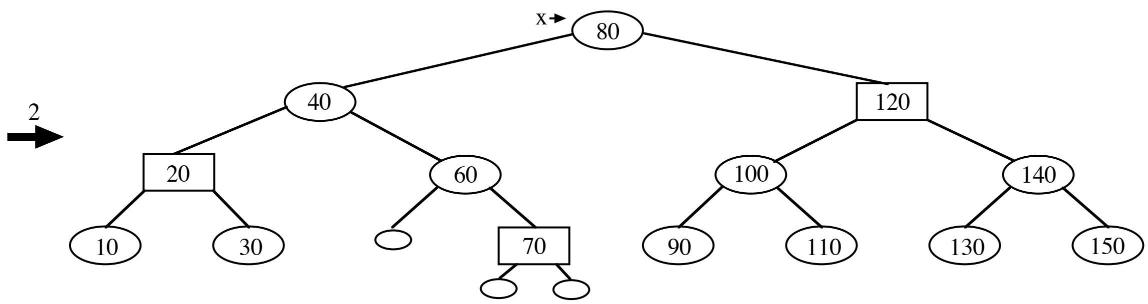


Delete 50



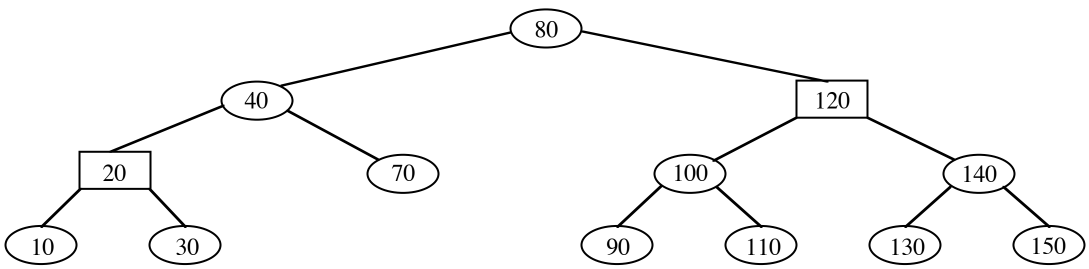




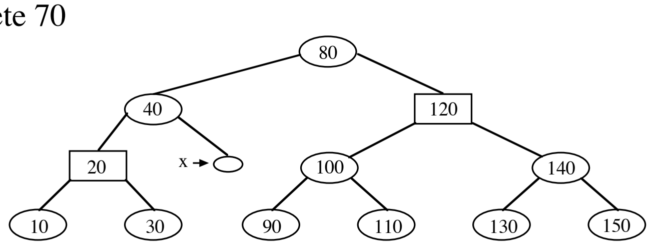


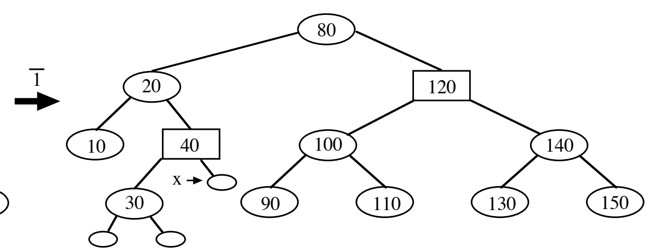
If x reaches the root, then done. Only place in tree where this happens.

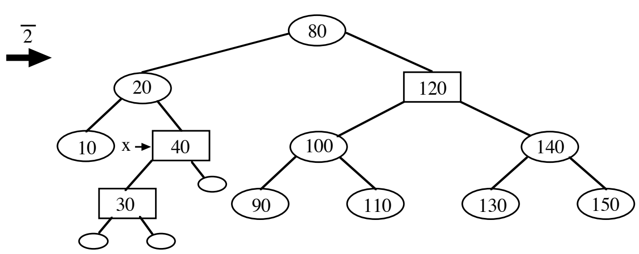
Delete 60



Delete 70

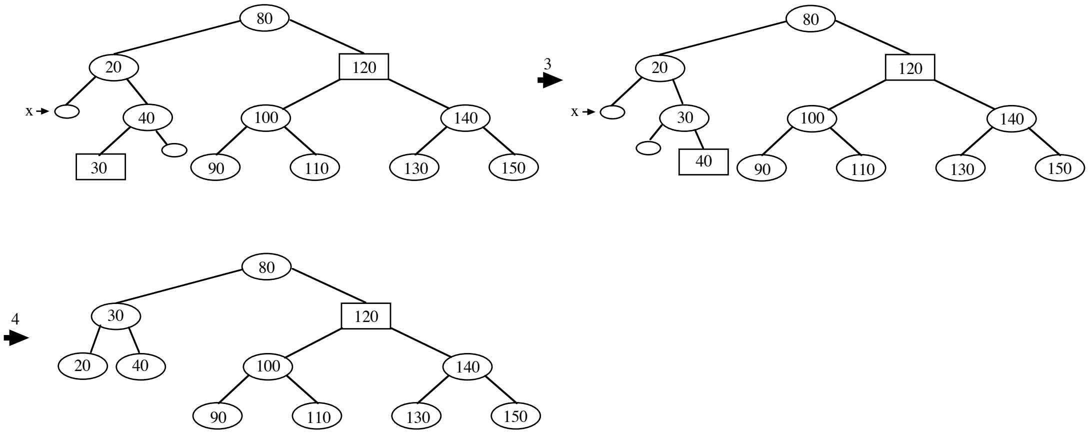




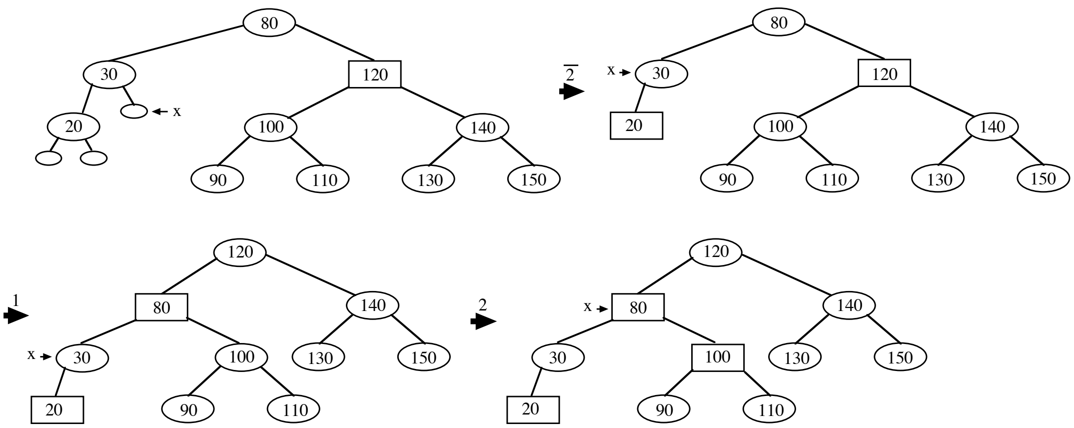


If x reaches a red node, then change color to black and done.

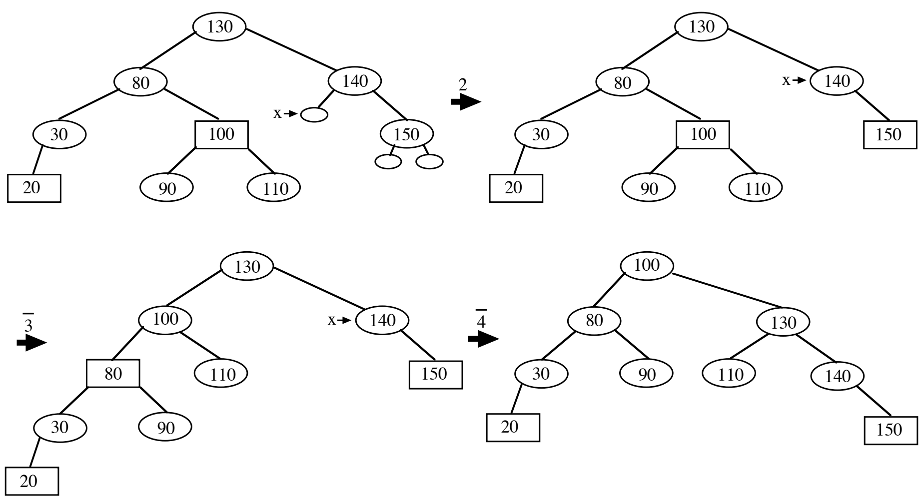
Delete 10



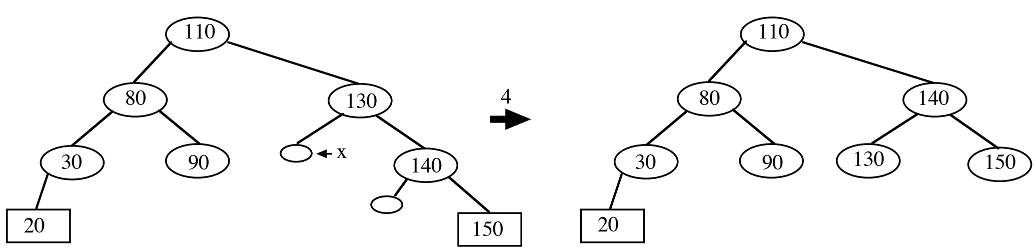
Delete 40



Delete 120



Delete 100

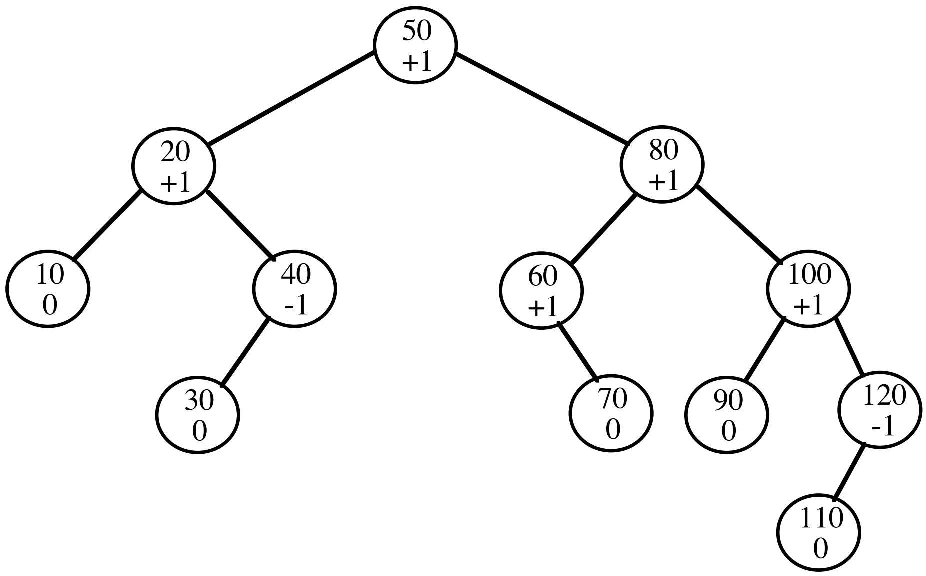


AVL TREES

An *AVL tree* is a binary search tree whose height is logarithmic in the number of keys stored.

1. Each node stores the difference of the heights (known as the *balance factor*) of the right and left subtrees rooted by the children:

$$\text{height}_{\text{right}} - \text{height}_{\text{left}}$$



2. A balance factor must be +1, 0, -1 (leans right, "balanced", leans left).

3. An insertion is implemented by:

    a. Attaching a leaf

    b. Rippling changes to balance factor:

        1. Right child ripple

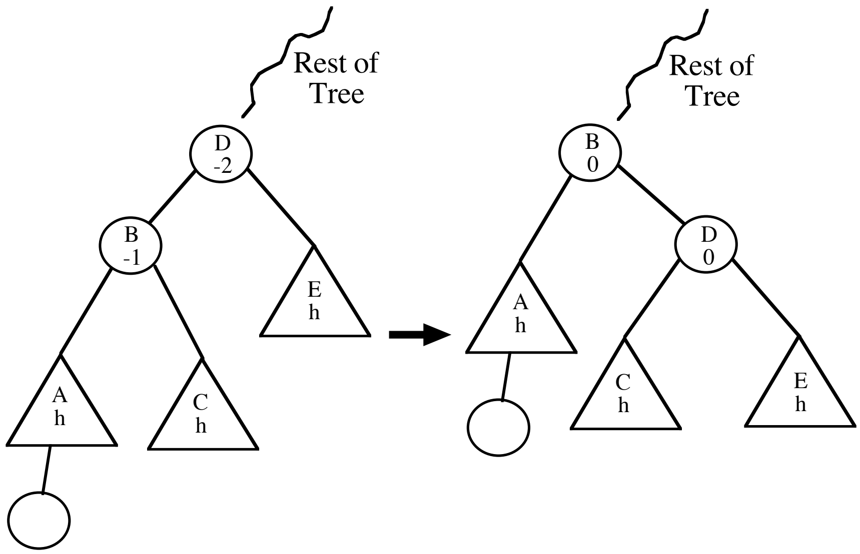
           Parent.Bal = 0 ⇒ +1 and ripple to parent
           Parent.Bal = -1 ⇒ 0 to complete insertion
           Parent.Bal = +1 ⇒ +2 and ROTATION to complete insertion

        2. Left child ripple

           Parent.Bal = 0 ⇒ -1 and ripple to parent
           Parent.Bal = +1 ⇒ 0 to complete insertion
           Parent.Bal = -1 ⇒ -2 and ROTATION to complete insertion

4. Rotations

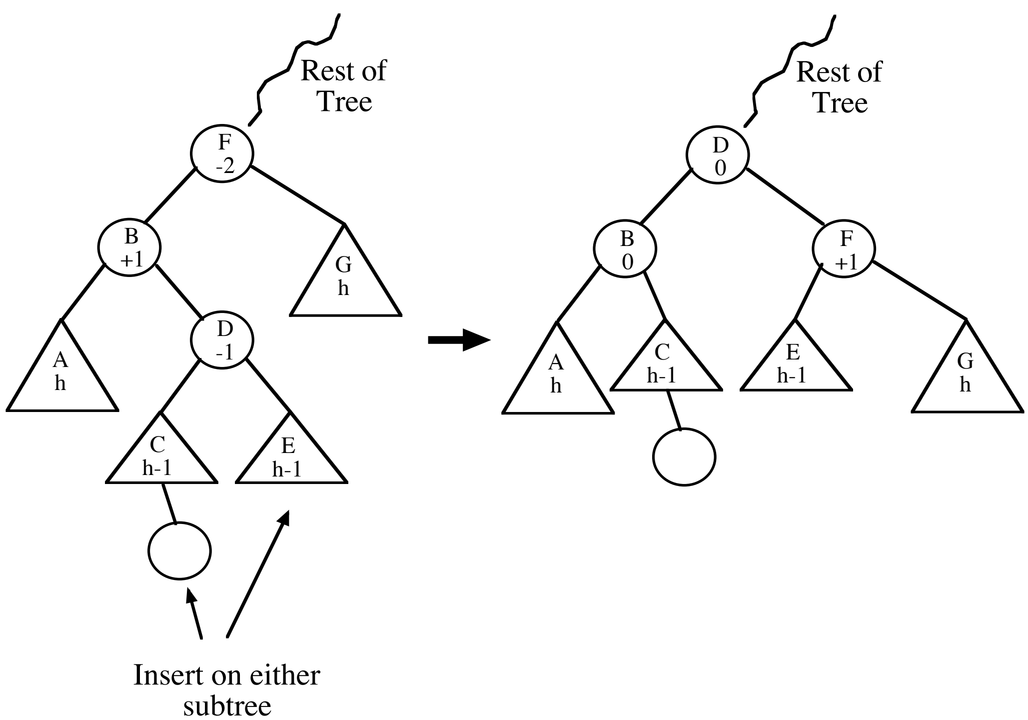
a. Single (LL) - right rotation at D



Restores height of subtree to pre-insertion number of levels

RR case is symmetric

b. Double (LR)



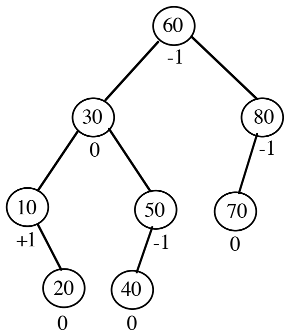
Restores height of subtree to pre-insertion number of levels

RL case is symmetric

Deletion -

Still have RR, RL, LL, and LR, but two addditional (symmetric) cases arise.

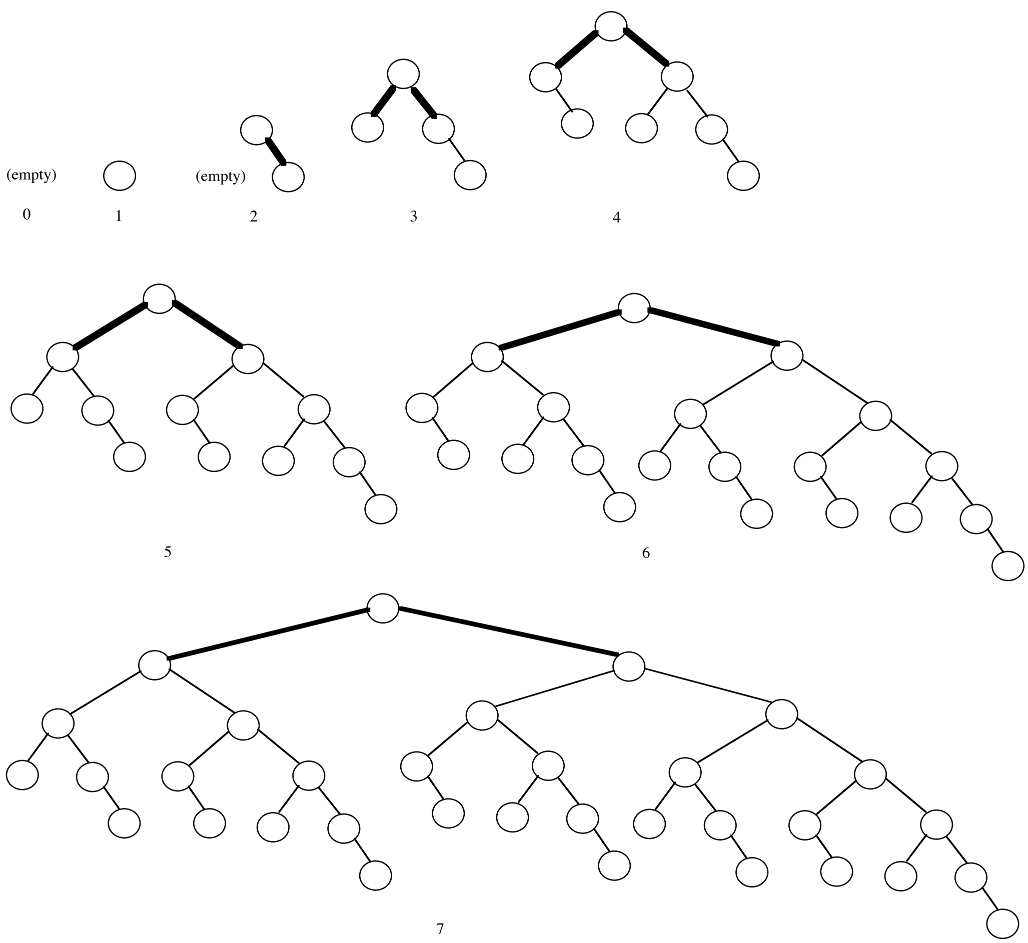
Suppose 70 is deleted from this tree. Either LL or LR may be applied.



Fibonacci Trees - special case of AVL trees exhibiting two worst-case behaviors -

1. Maximally skewed. (max height is roughly $\log_{1.618} n = 1.44 \lg n$, expected height is $\lg n + .25$)

2. $\theta(\log n)$ rotations for a single deletion.



TREAPS (CLRS, p. 333)

Hybrid of BST and min-heap ideas

Gives code that is clearer than RB or AVL (but comparable to skip lists)

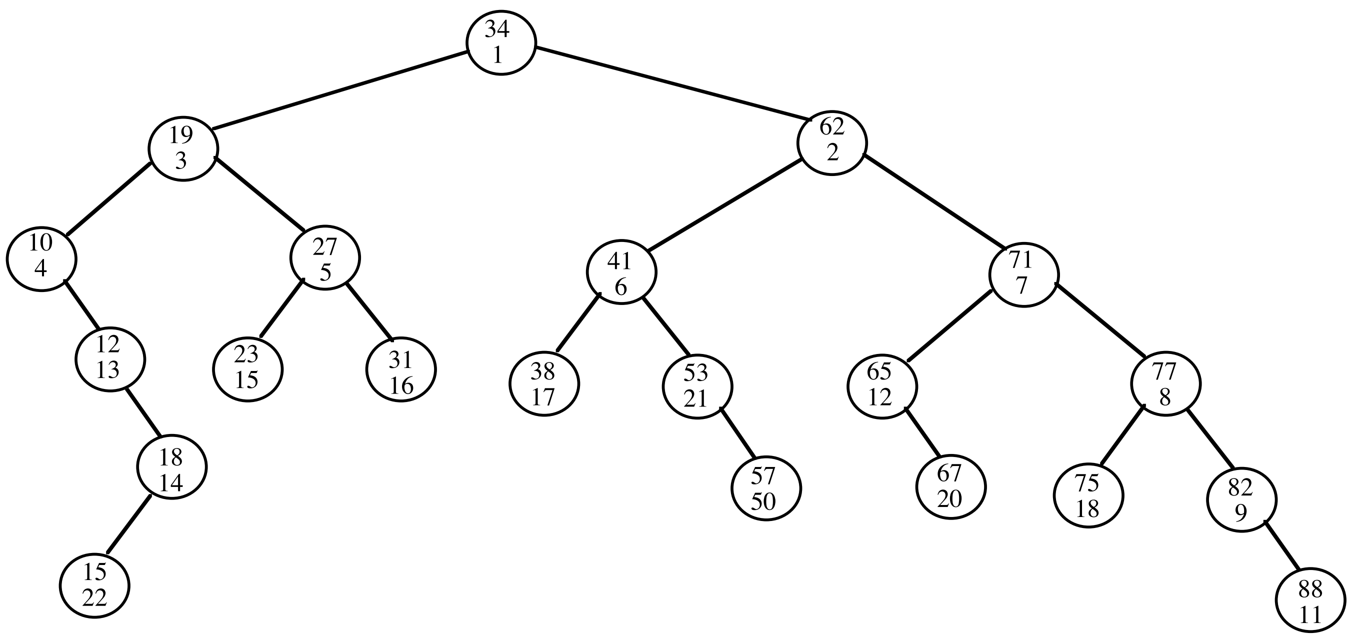
Expected height of tree is logarithmic (2.5 lg n)

Keys are used as in BST

Tree also has min-heap property based on each node having a priority:

Randomized priority - generated when a new key is inserted

Virtual priority - computed (when needed) using a function similar to a hash function



Asides: the first published such hybrid were the *cartesian trees* of J. Vuillemin, "A Unifying Look at Data Structures", *C. ACM 23 (4)*, April 1980, 229-239. A more complete explanation appears in E.M. McCreight, "Priority Search Trees", *SIAM J. Computing 14 (2)*, May 1985, 257-276 and chapter 10 of M. de Berg et.al. These are also used in the elegant implementation in M.A. Babenko and T.A. Starikovskaya, "Computing Longest Common Substrings" in E.A. Hirsch, *Computer Science - Theory and Applications*, LNCS 5010, 2008, 64-75.

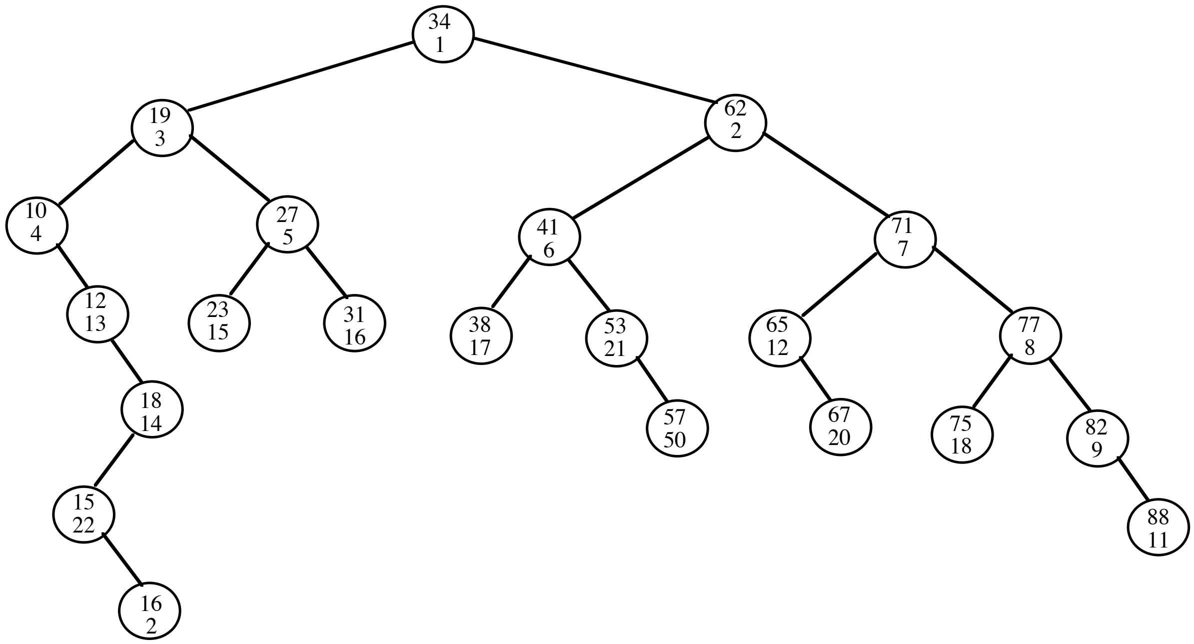
Insertion

Insert as leaf

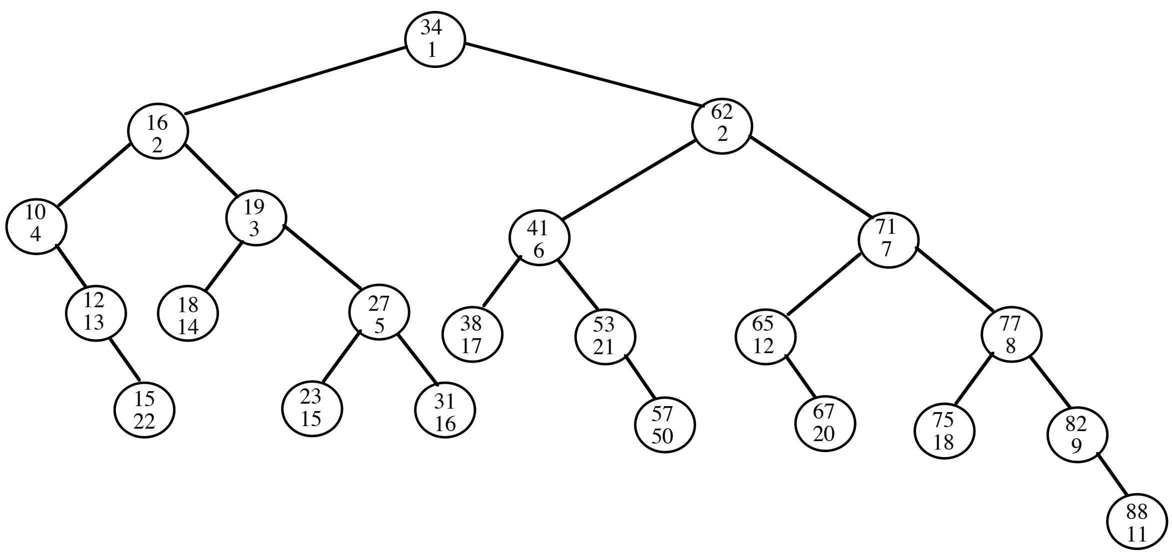
Generate random priority (large range to minimize duplicates)

Single rotations to fix min-heap property

Example: Insert 16 with a priority of 2



After rotations:



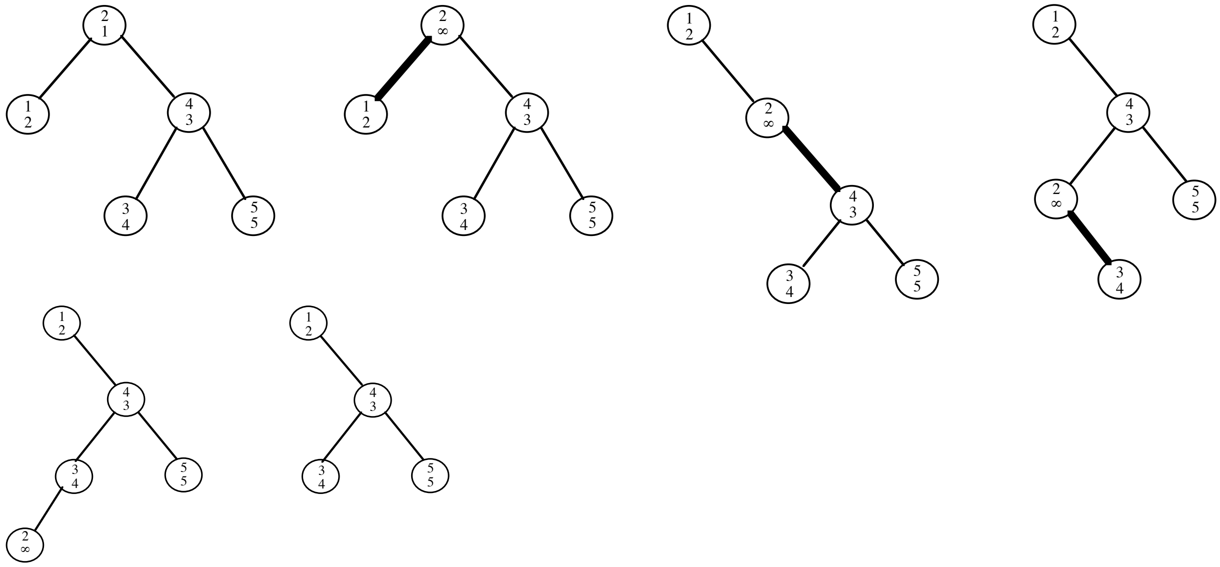
Deletion

Find node and change priority to ∞

Rotate to bring up child with lower priority. Continue until min-heap property holds.

Remove leaf.

Delete key 2:



AUGMENTING DATA STRUCTURES

Read CLRS, section 14.1 on using RB tree with *ranking* information for order statistics.

Retrieving an element with a given rank

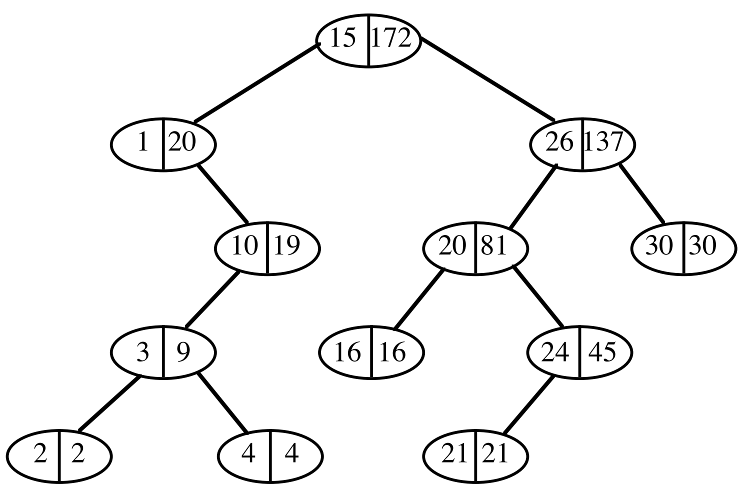
Determine the rank of an element

Problem: Maintain summary information to support an aggregate operation on the k smallest (or largest) keys in $O(\log n)$ time.

Example: Prefix Sum

Given a key, determine the sum of all keys ≤ given key (prefix sum).

Solution: Store sum of all keys in a subtree at the root of the subtree.



Key	Prefix Sum
1	1
2	3
3	6
4	10
10	20
15	35
16	51
20	71
21	92
24	116
26	142
30	172

To compute prefix sum for a key:

Initialize sum to 0

Search for key, modifying total as search progresses:

Search goes left - leave total alone

Search goes right or key has been found - add present node's key and left child's sum to total

Key is 24: (15 + 20) + (20 + 16) + (24 + 21) = 116

Key is 10: (1 + 0) + (10 + 9) = 20

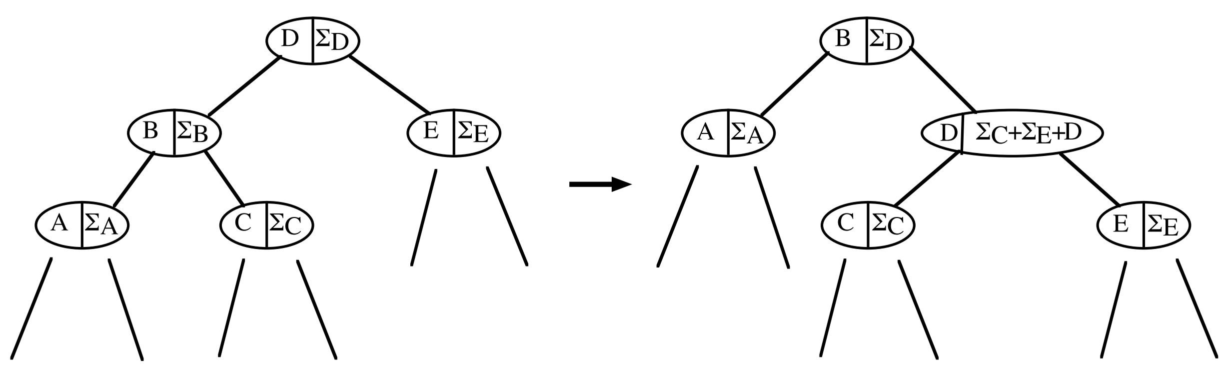
Key is 16: (15 + 20) + (16 + 0) = 51

Variation: Determine the smallest key that has a prefix sum ≥ a specified value.

Updates to tree:

Non-structural (attach/remove node) - modify node and every ancestor

Single rotation (for prefix sum)



(Similar for double rotation)

General case - see CLRS 14.2, especially "Theorem" 14.1

Interval trees (CLRS 14.3) - a more significant application

Set of (closed) intervals [*low*, *high*] - *low* is the key, but duplicates are allowed

Each subtree root contains the *max* value appearing in any interval in that subtree

Aggregate operation to support - find any interval that overlaps a given interval [*low'*, *high'*]

Modify BST search . . .

if *ptr* == *nil*
    no interval in tree overlaps [*low'*, *high'*]

if *high'* ≥ *ptr->low* and *ptr->high* ≥ *low'*
    return *ptr* as an answer

if *ptr->left* != *nil* and *ptr->left->max* ≥ *low'*
    ptr := *ptr->left*
else
    ptr := *ptr->right*

Updates to tree - similar to prefix sum, but replace additions with maximums