CS 372: Red-Black Trees

Cormen et al - Chapter 13.1-13.4

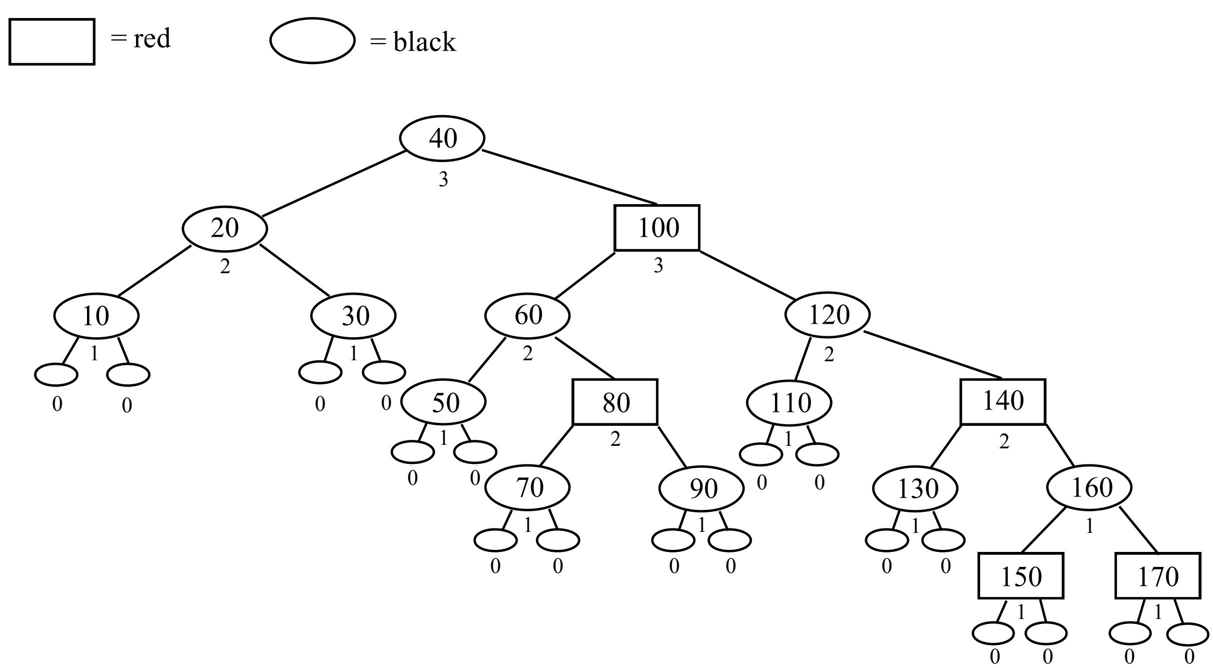
STRUCTURAL PROPERTIES

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. The root is black.
3. Every "leaf" (the sentinel) is colored black.
4. Both children of a red node are black.
5. 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)). These are not stored.

Example:

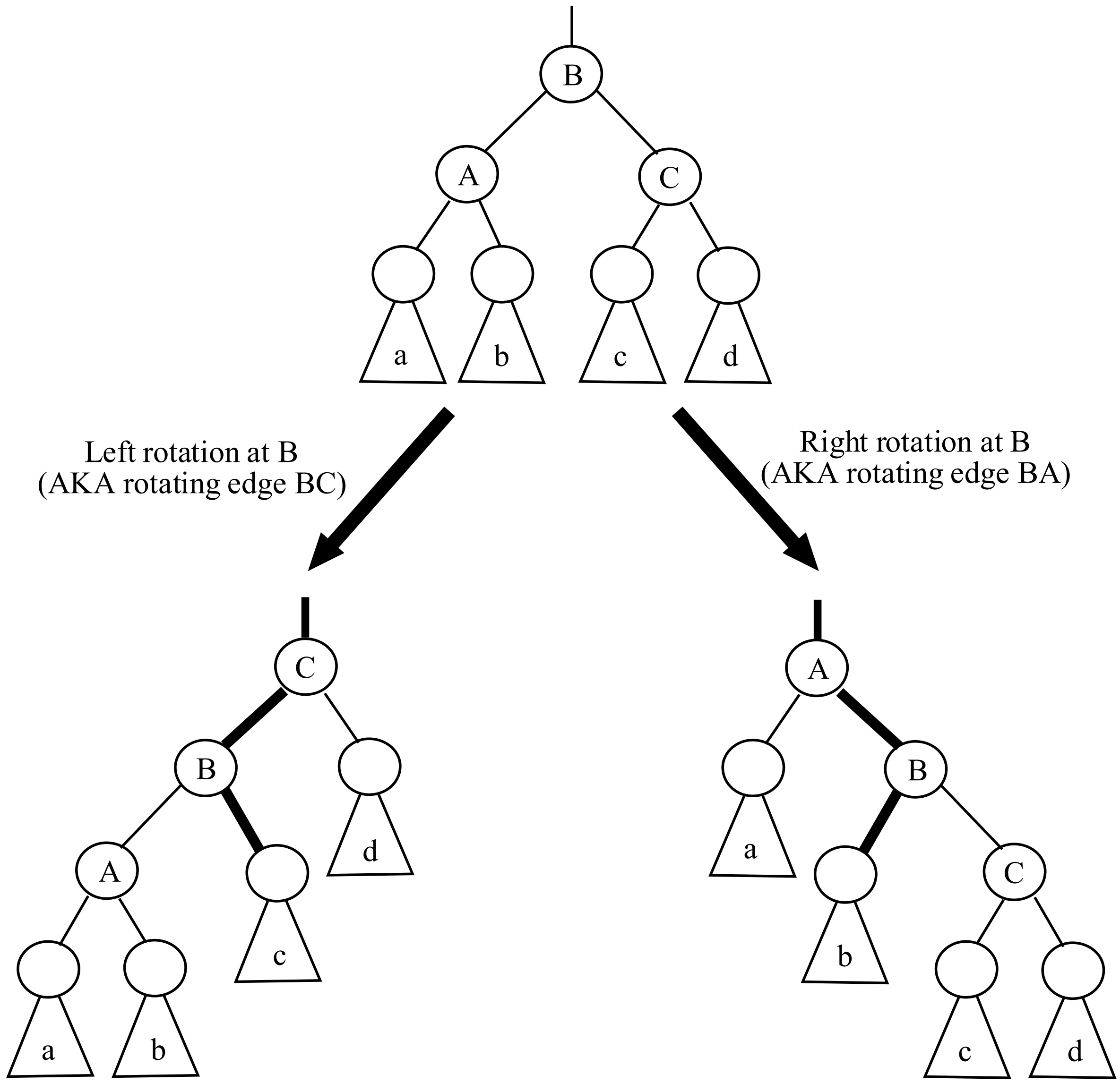


Observations:

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.

ROTATIONS

Technique for rebalancing in most balanced binary search tree schemes. Takes $\Theta(1)$ time.



INSERTION

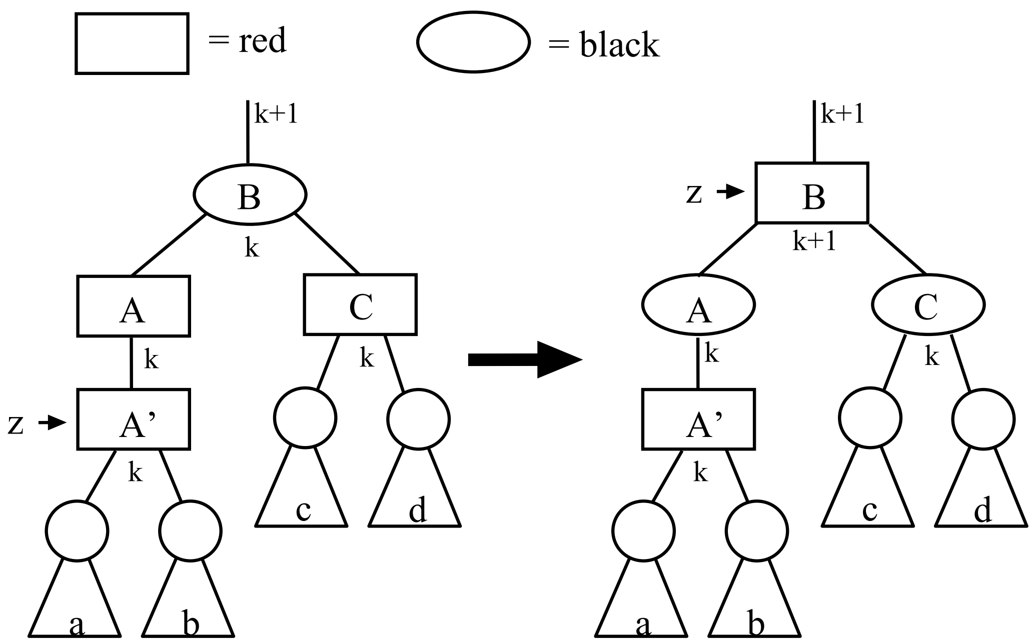
1. Start with unbalanced insert of a "data leaf" (both children are the sentinel).

2. Color of new node is red.

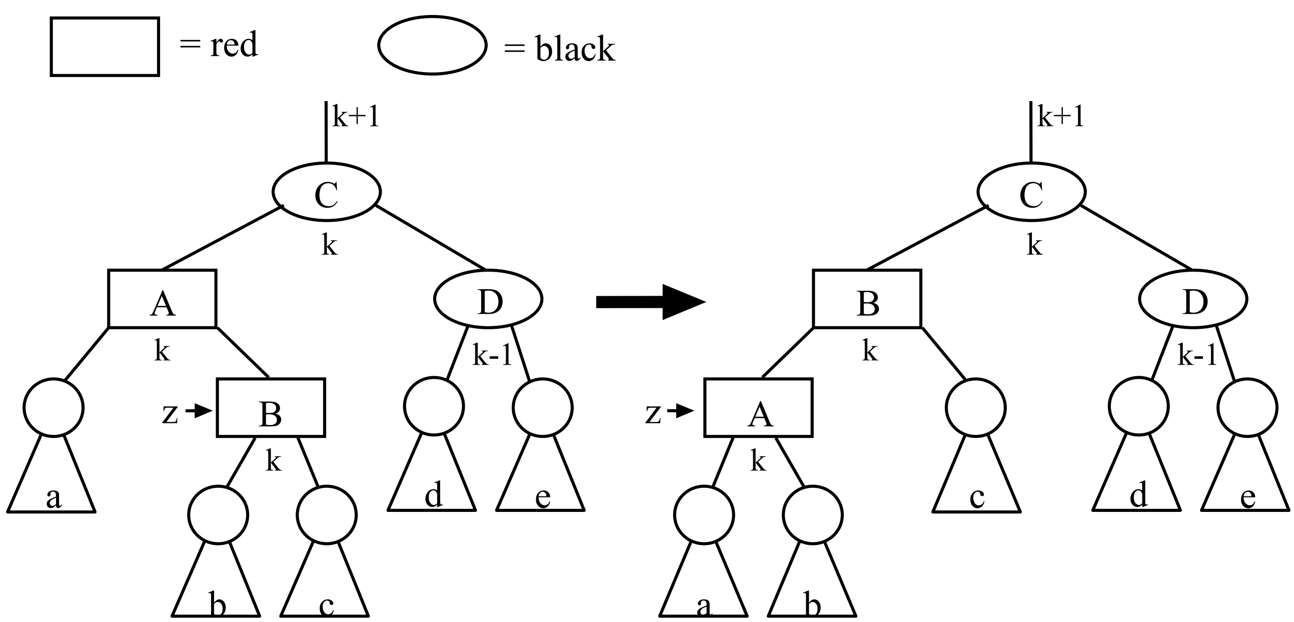
3. May violate structural properties 2 and 4. Leads to three cases, along with symmetric versions.

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

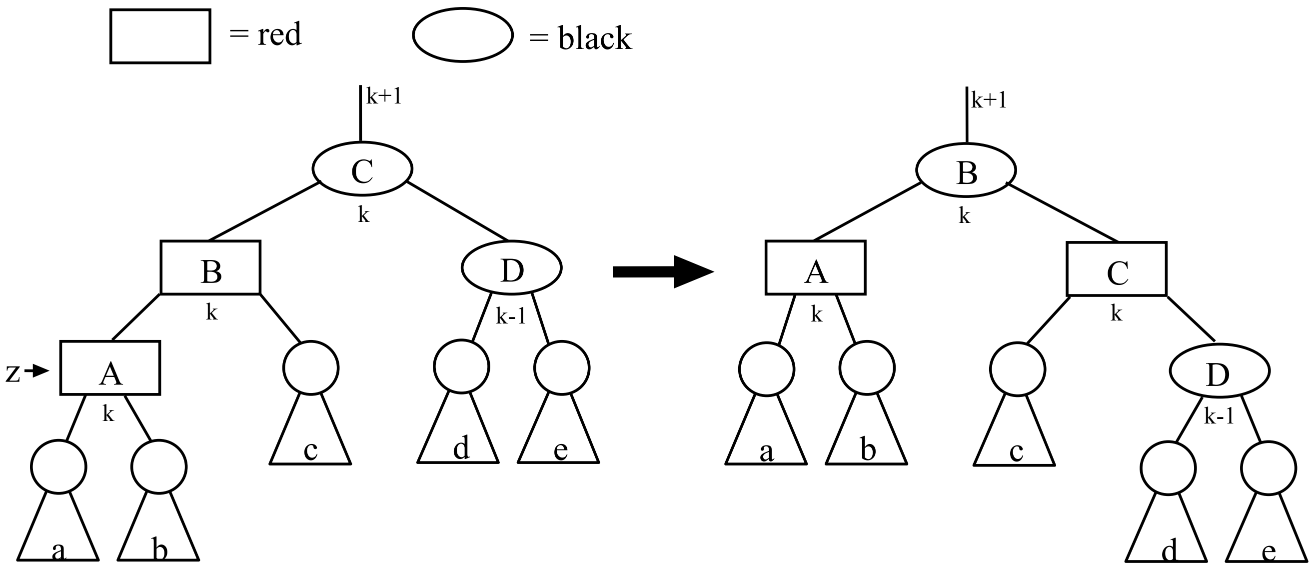
Case 1: z's uncle is red



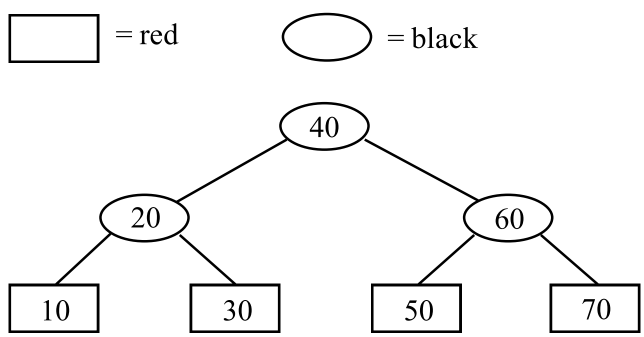
Case 2: (z'a uncle is black) and (z, z's parent, and z's grandparent do not line up straightly)



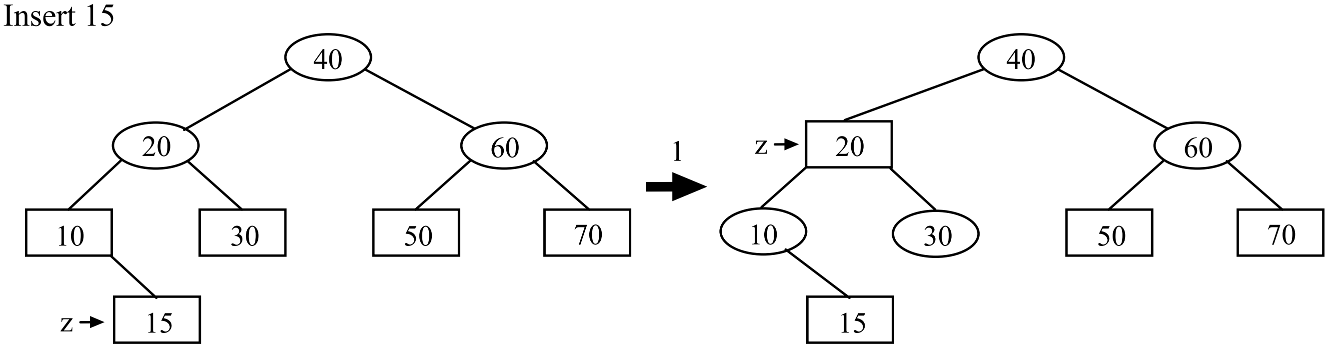
Case 3: (z'a uncle is black) and (z, z's parent, and z's grandparent line up straightly)



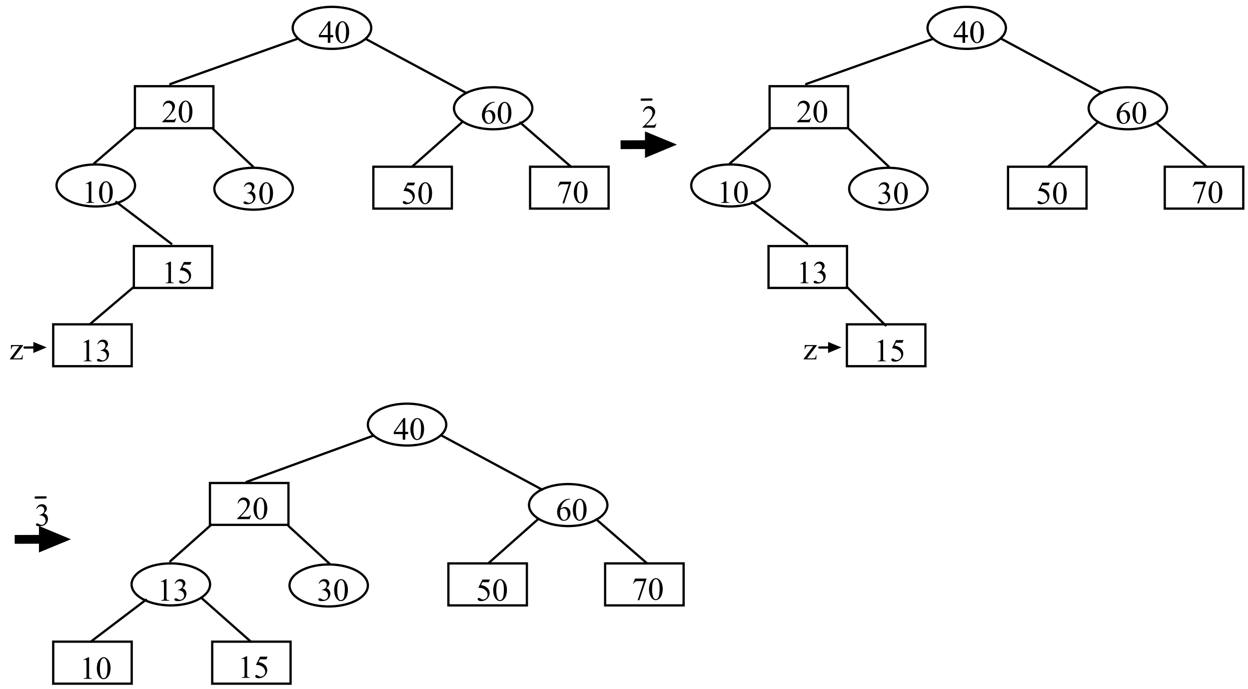
Example 1:



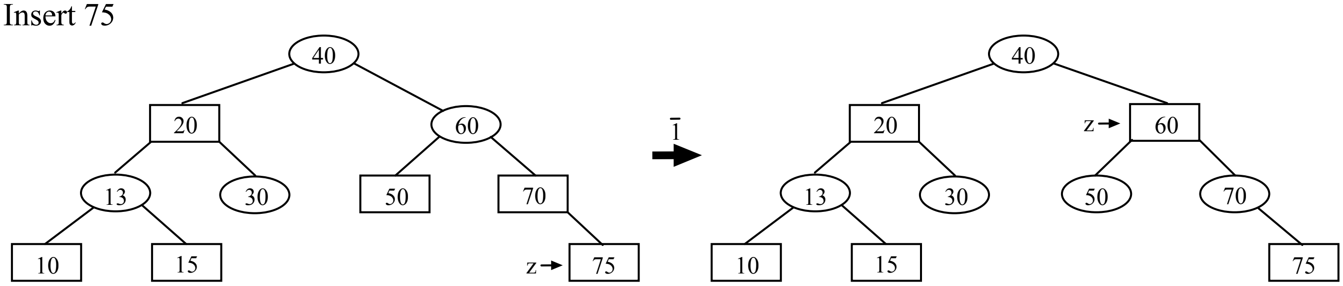
Insert 15



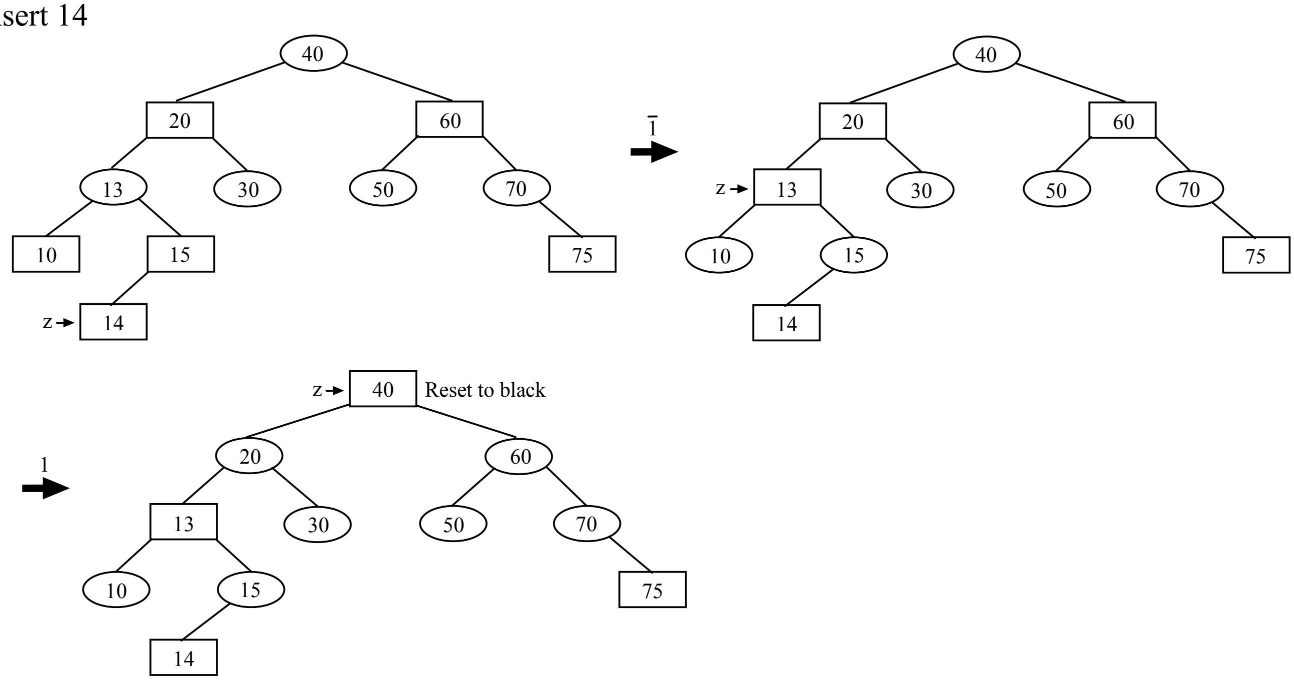
Insert 13



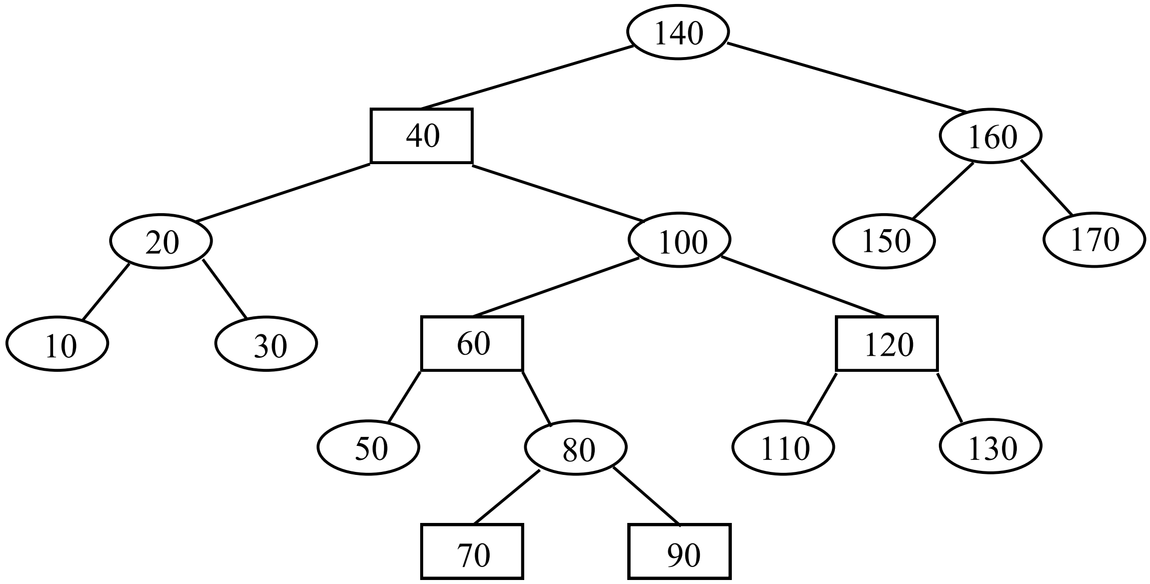
Insert 75



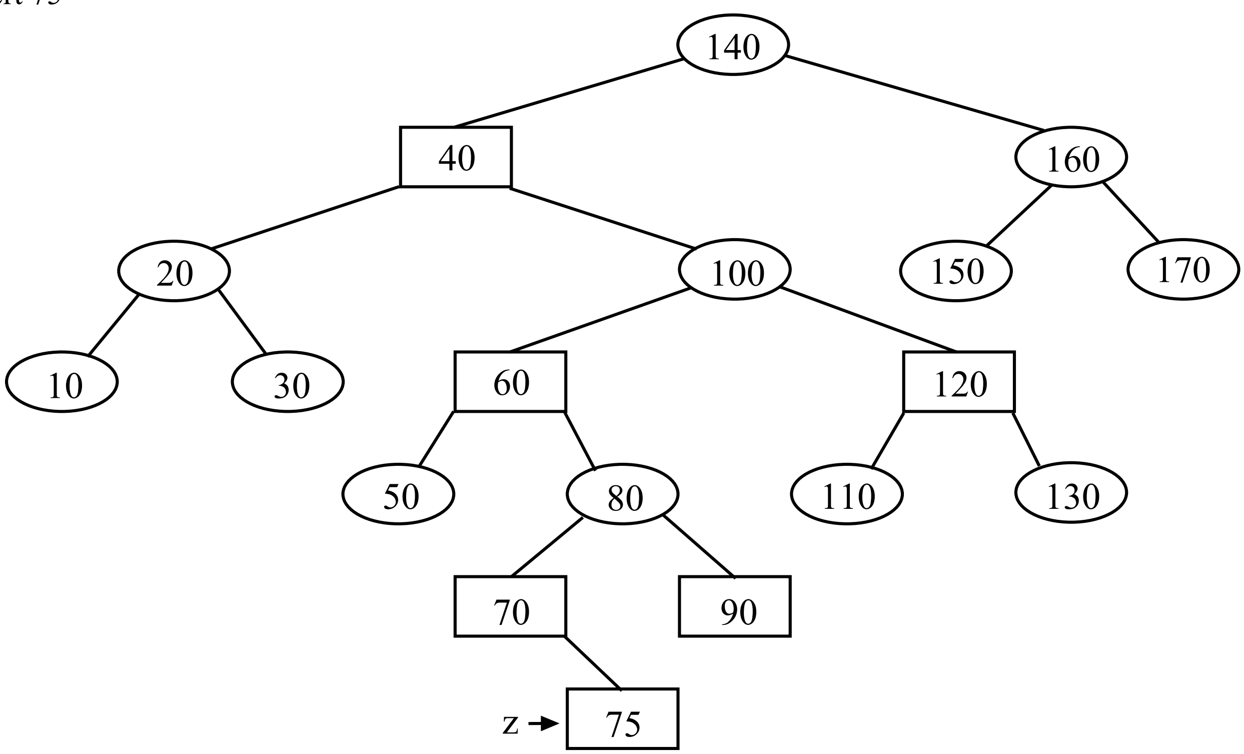
Insert 14

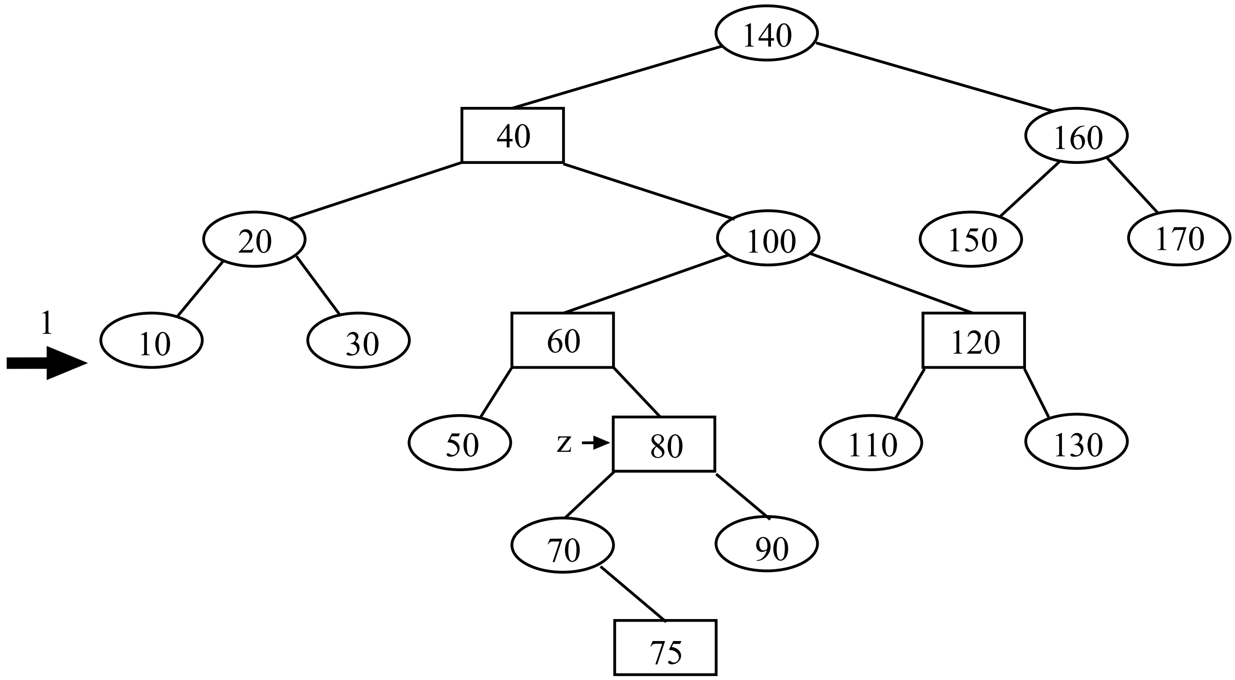


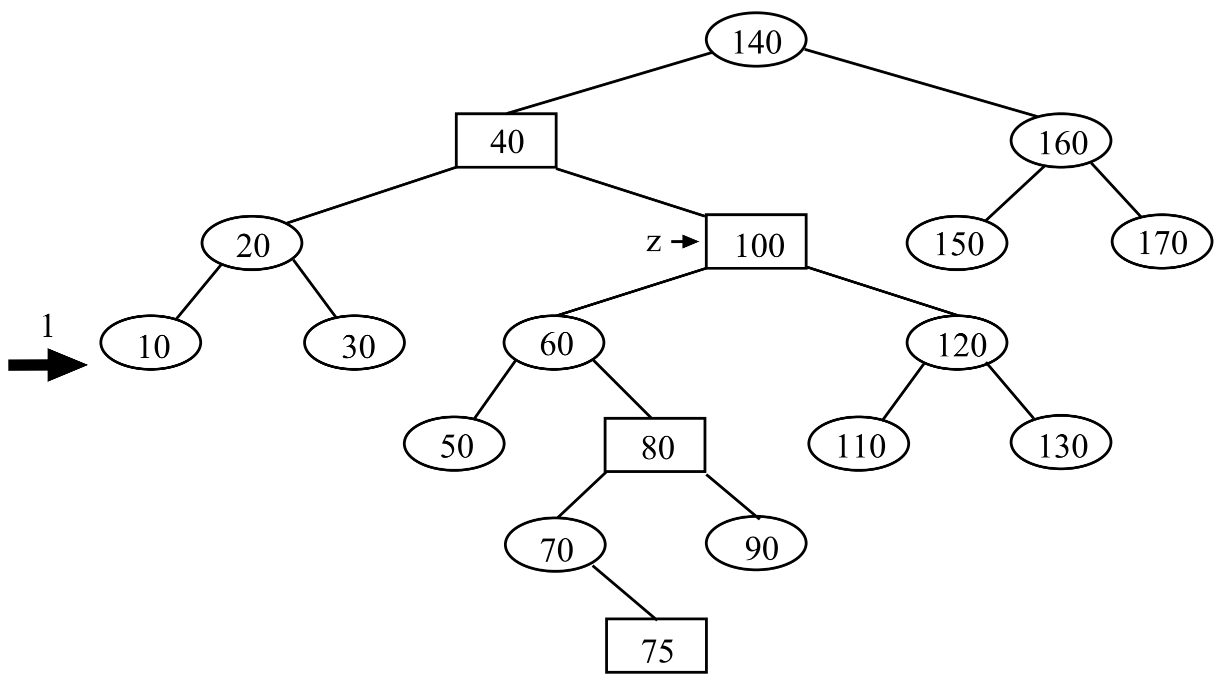
Example 2:

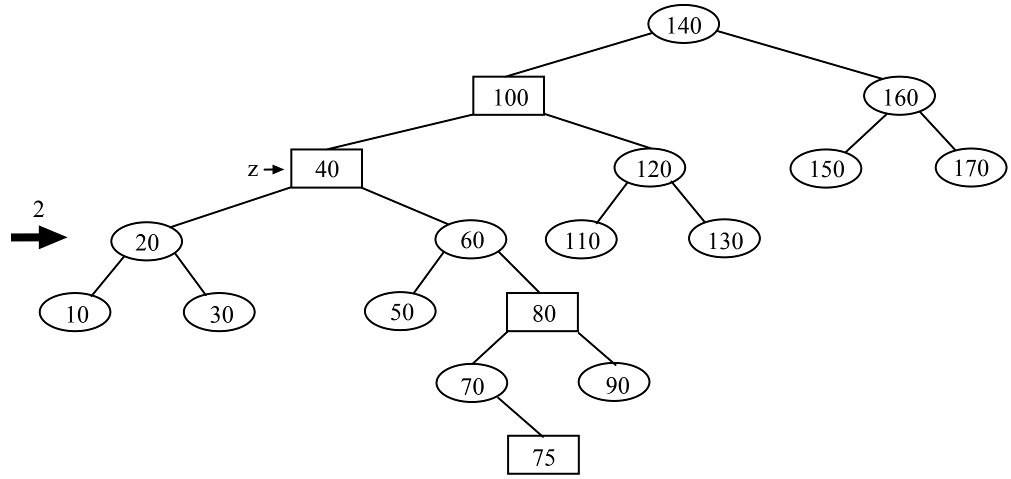


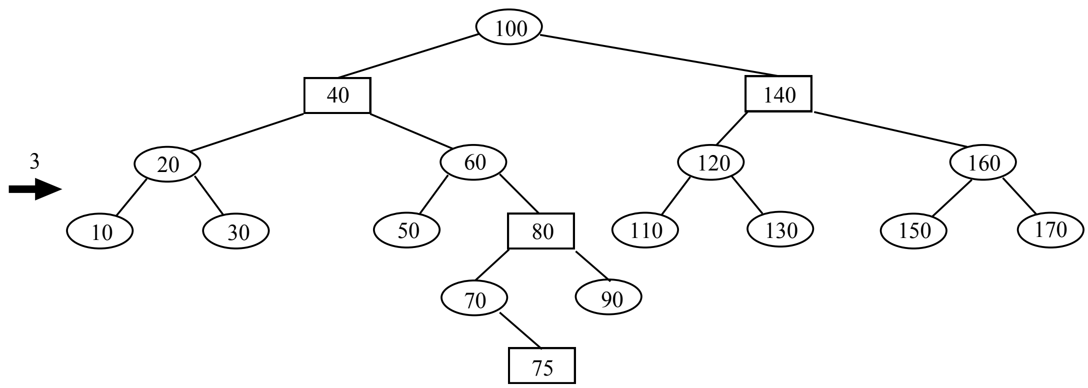
Insert 75




140
40
160
20
100
150
170
1
10
30
60
120
50
z
80
110
130
70
90
75


140
40
160
20
z
100
150
170
1
10
30
60
120
50
80
110
130
70
90
75





DELETION

First, delete a node as from a regular binary search tree. There are three cases for deletion:

1. Deleted node is a "data leaf".

   a. Splice around to sentinel.

   b. Color of deleted node?

      Red ⇒ Done

      Black ⇒ Set temporary "double black" pointer (x) at sentinel.
      Determine which of four rebalancing cases below 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.

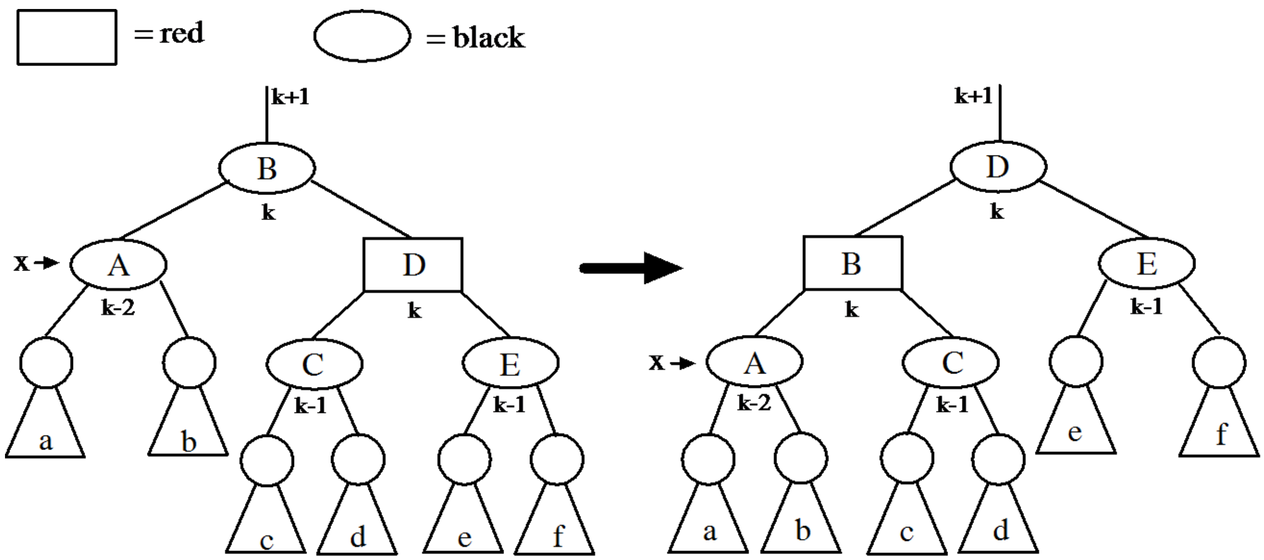
3. Node with key-to-delete is parent of two “data nodes”.

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

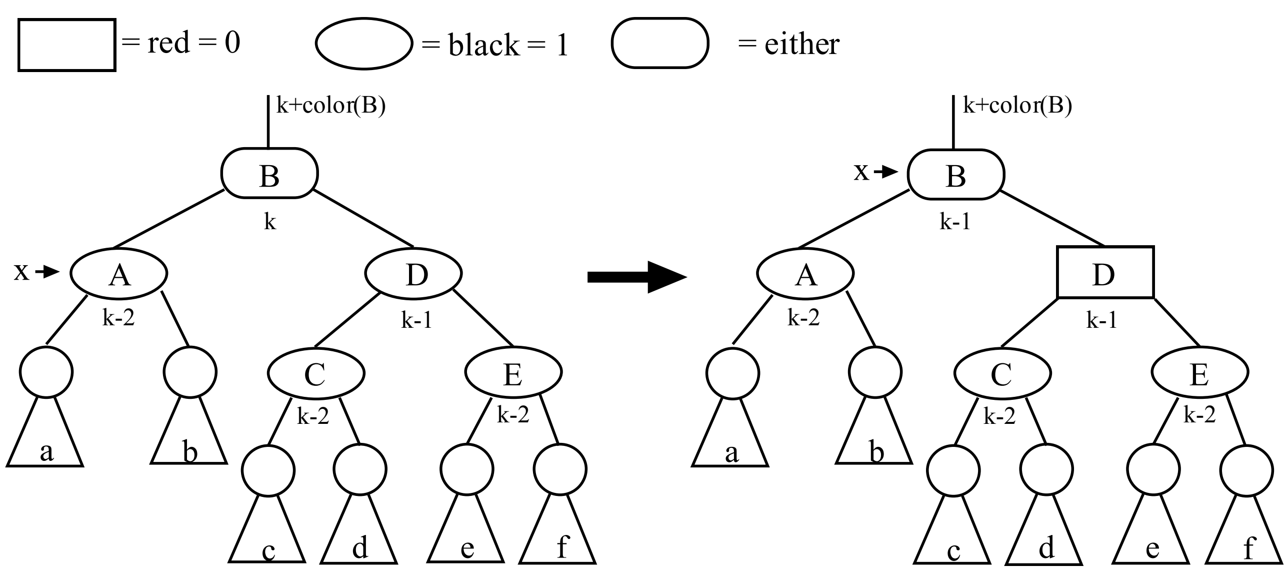
    b. Delete successor using the appropriate one of the previous two cases.

The following are four rebalancing cases which are used to eliminate “double black” pointer. Note that a node is already deleted (using the 3 cases mentioned above), so none of the nodes are deleted in the 4 cases below.
In all the cases below node x has an “extra” black.

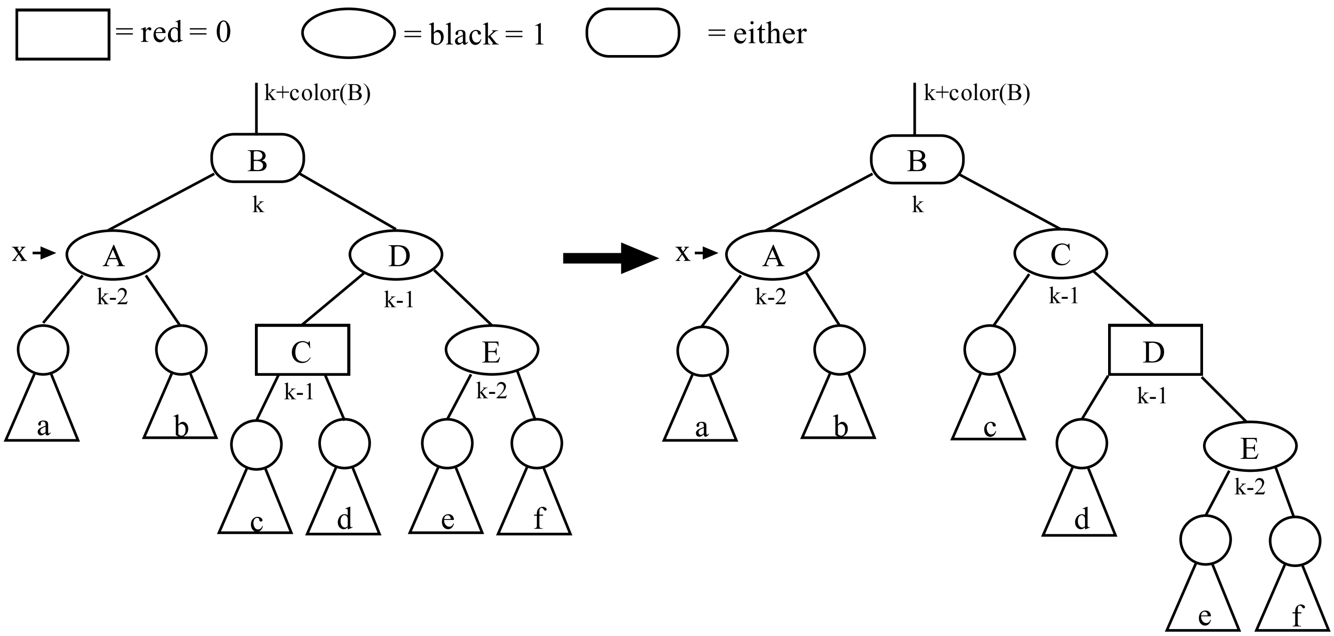
Case 1: x’s sibling is red



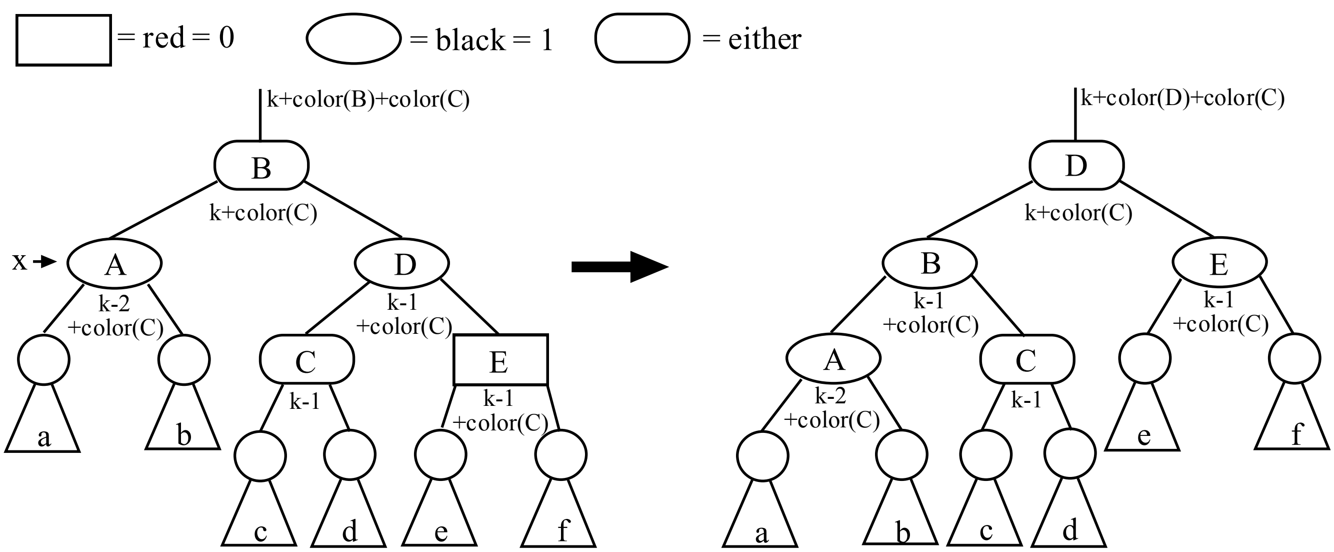
Case 2: x’s sibling w is black and both of w’s children are black



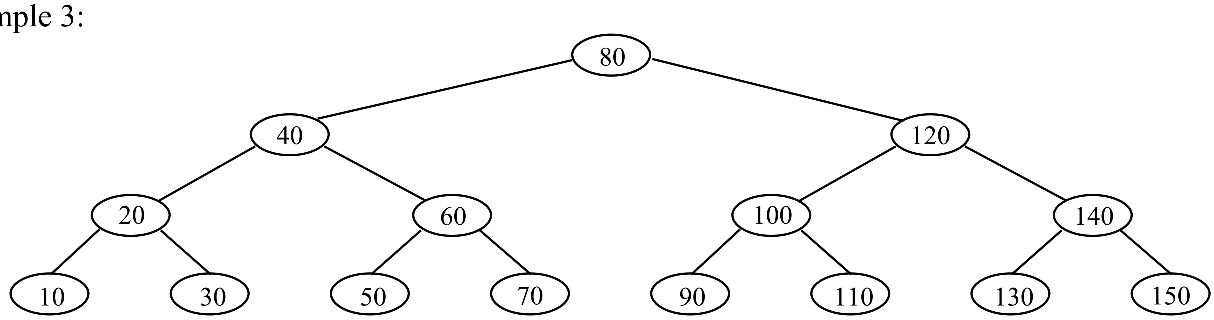
Case 3: x’s sibling w is black. w has one red child. w’s only red child, w, and w’s parent do not line up straightly.



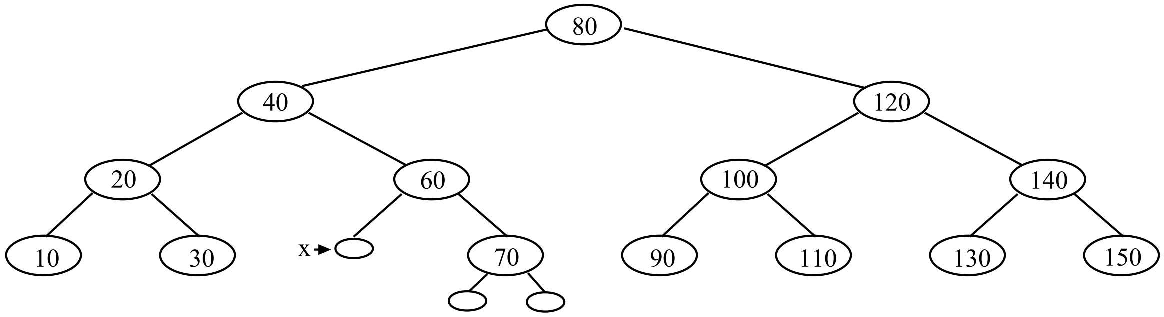
Case 4: x’s sibling w is black. w has a red child which lines up straightly with w and w’s parent. (The other child of w may be either red or black).

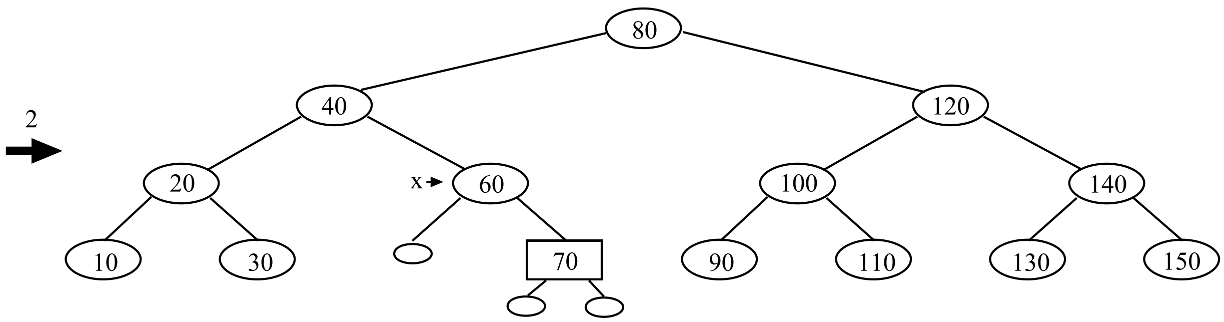


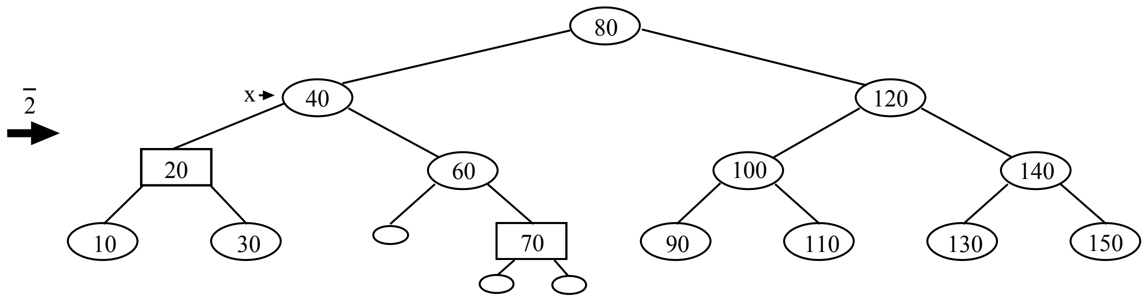
Example 3:

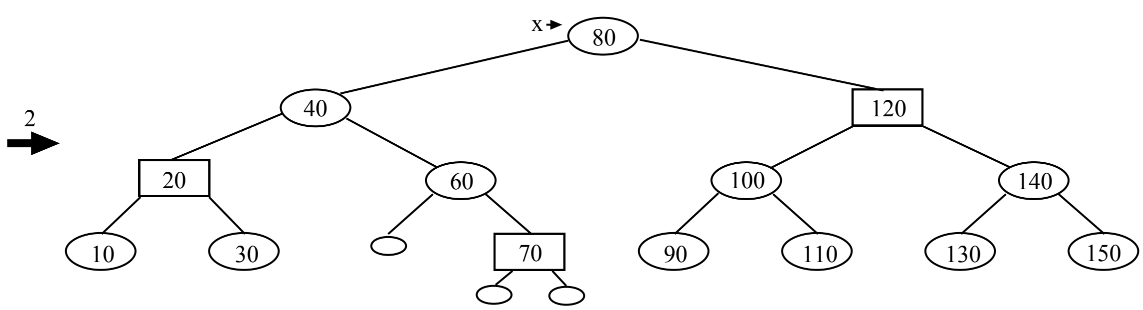


Delete 50



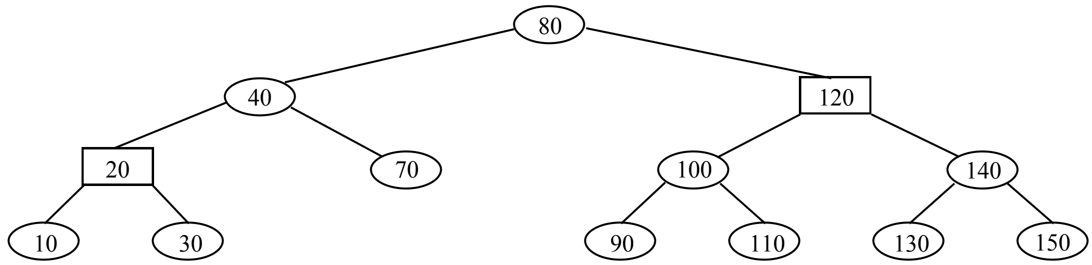




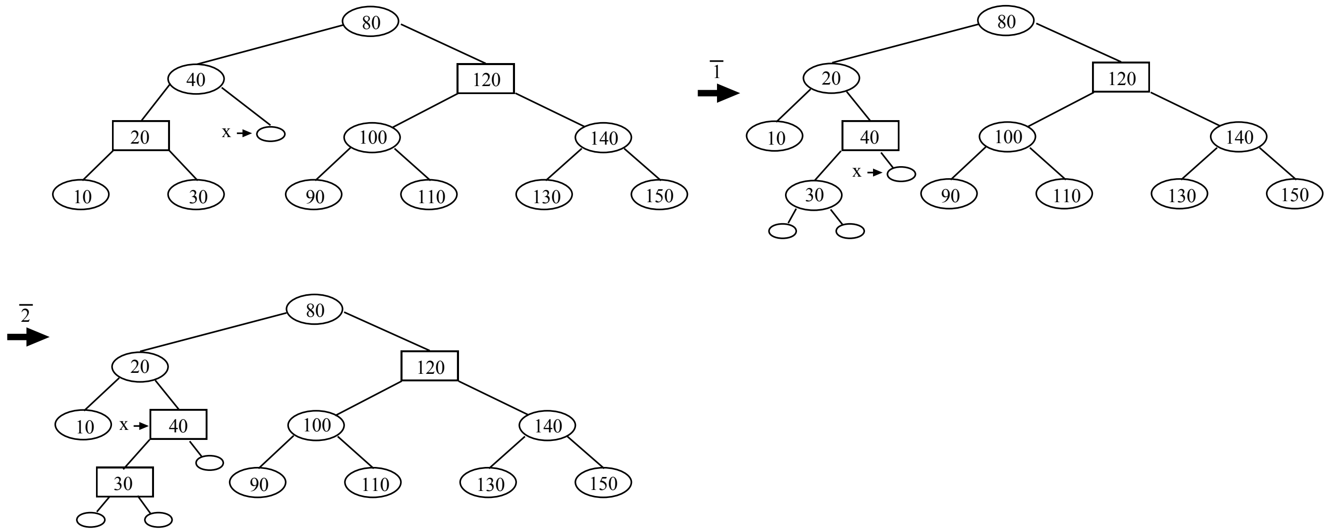


If x reaches the root, then done. Only place in tree where this happens.

Delete 60

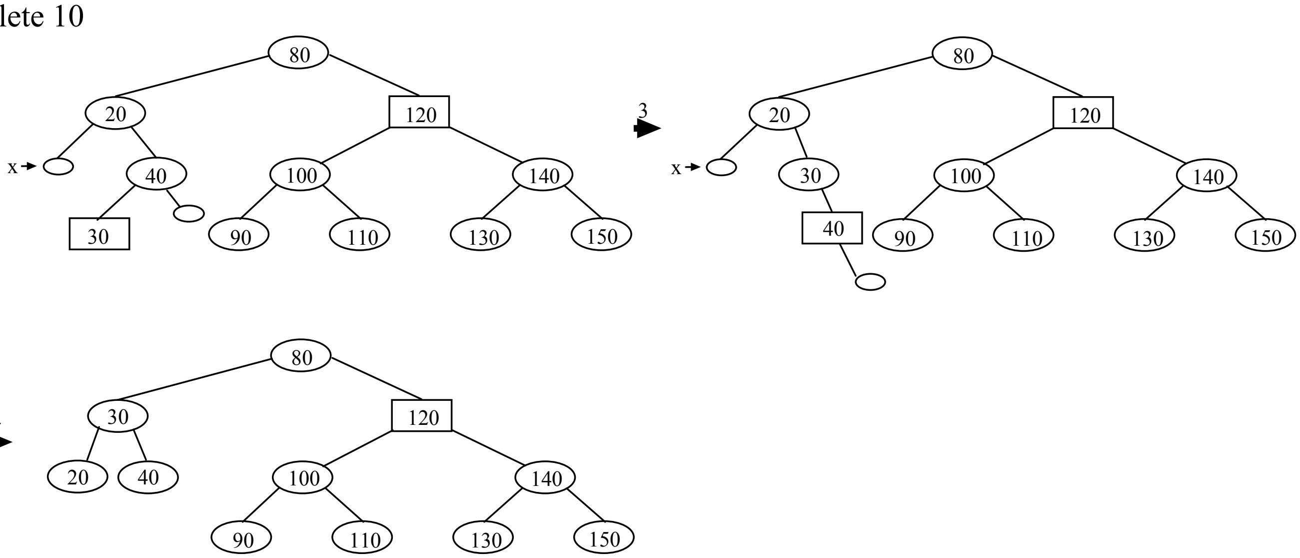


Delete 70

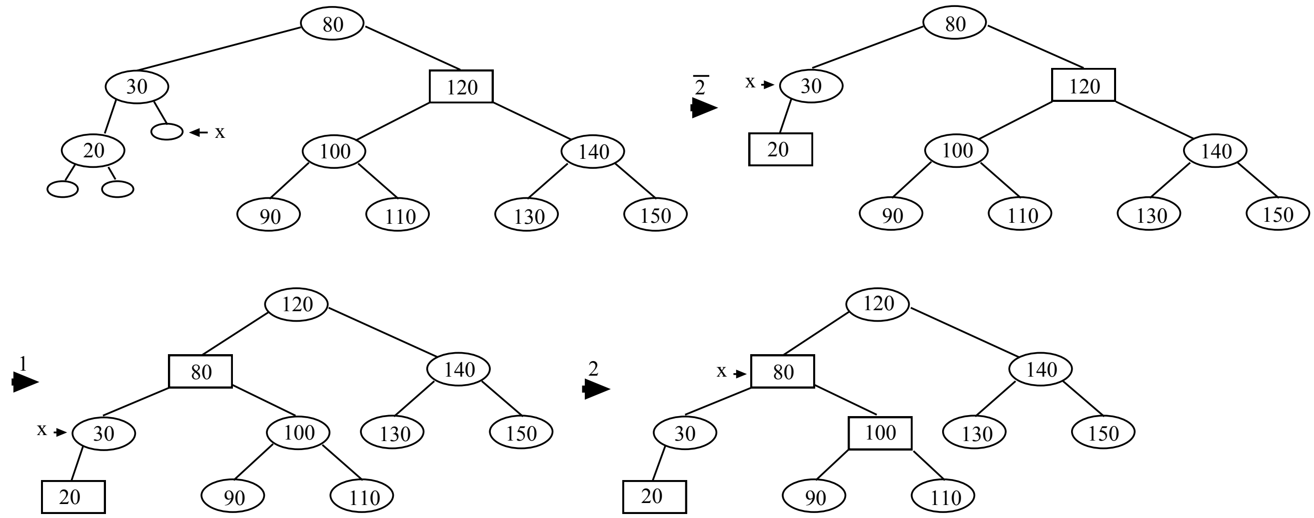


If x reaches a red node, then change color to black and done.

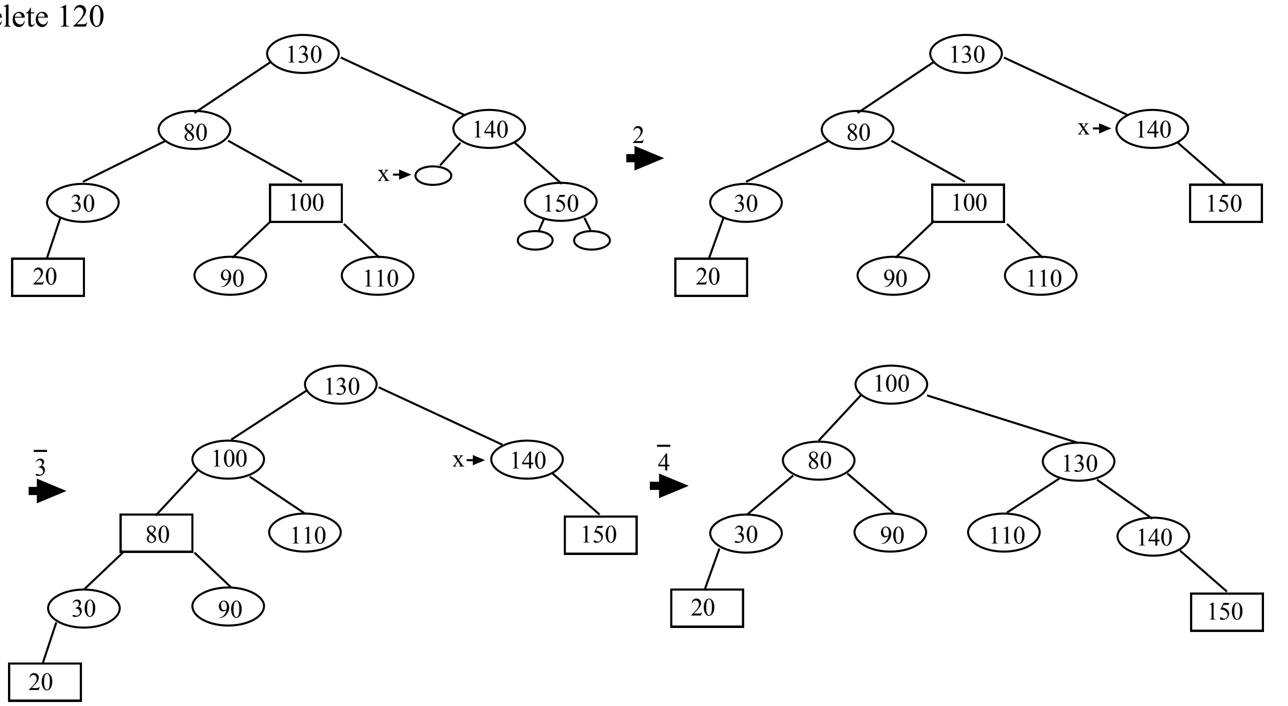
Delete 10



Delete 40



Delete 120



Delete 100