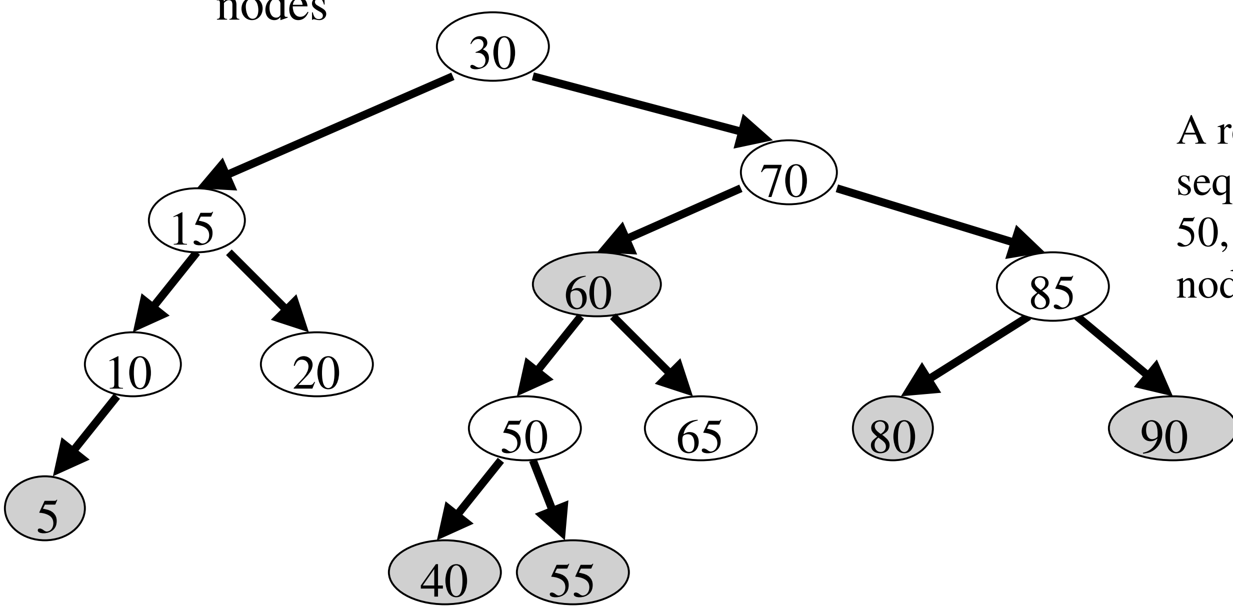
Red-Black Trees

- A good alternative to AVL tree
- A single top-down pass allows insertion and deletion
- A red-black tree is a binary tree such that

1. Every node is colored either red or black
2. The root is black
3. If a node is red, then its children are black (consecutive red nodes are disallowed)
4. Every path from a node to a NULL pointer must contain the same number of black nodes



A red-black tree: The insertion sequence is 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, and 55 (shaded nodes are red).

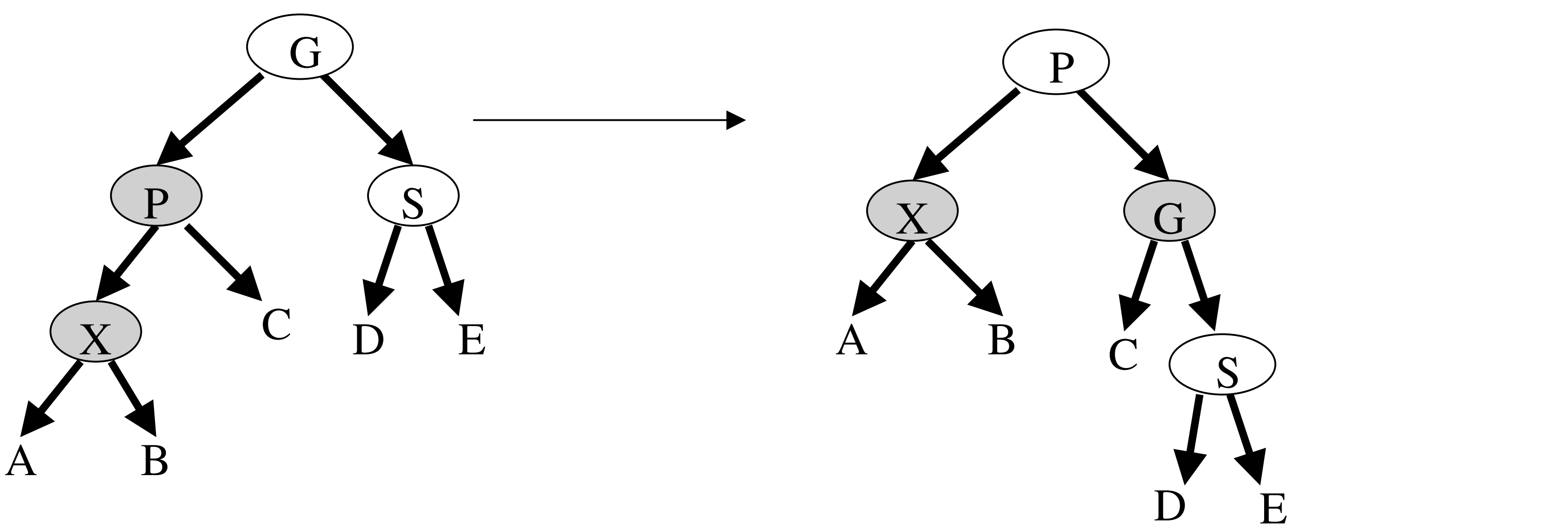
- The depth of a red-black tree is guaranteed to be logarithmic.
- The difficulty is that the insertion/deletion may alter the structure of red-black tree.

Bottom-Up Insertion

- A new insertion is always a leaf
- Coloring a new item black violates property 4 ➔new item is red
- If the parent is black, no problem
- If the parent is red, the insertion violates property 3
- This can be fixed by color changes and rotation
- Several cases:

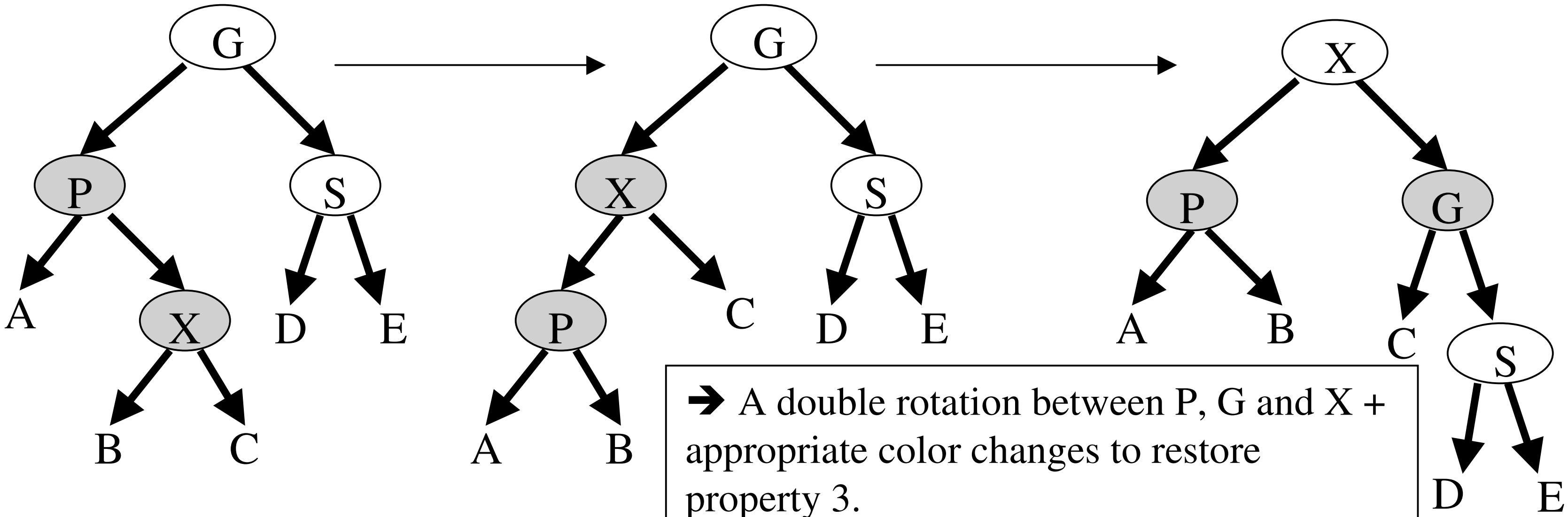
  1. The parent is red, the sibling of the parent is black, and the new node (X) is an outside grandchild (left-left, right-right cases)

     ➔ Single rotation between P and G + appropriate color change to restore property 3.



Bottom-Up Insertion (cont.)

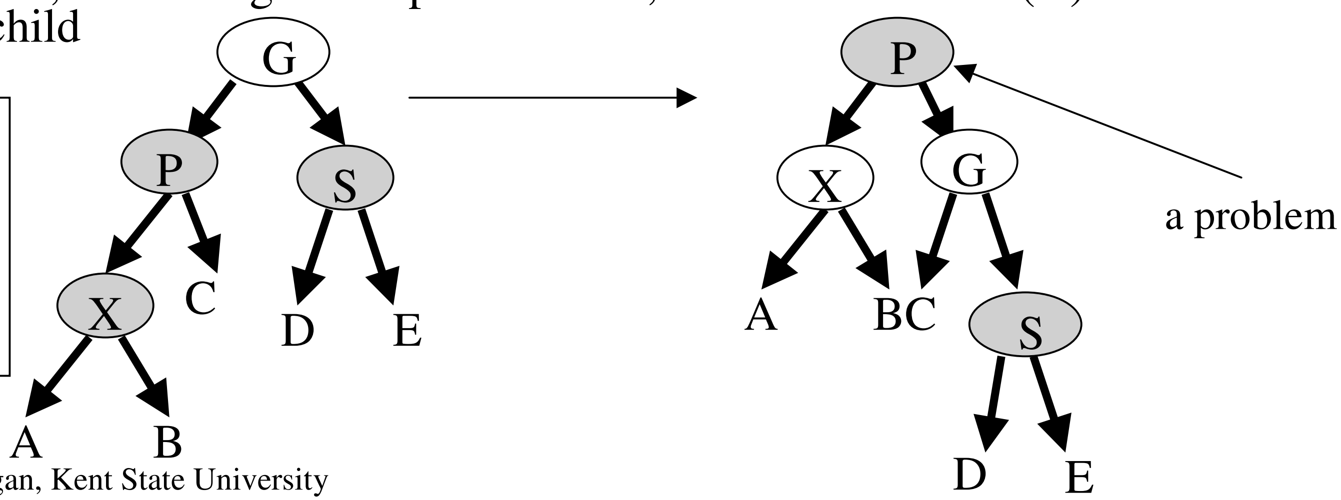
2. The parent is red, the sibling of the parent is black, and the new node (X) is an inside grandchild (left-right, right-left cases)



➔ A double rotation between P, G and X + appropriate color changes to restore property 3.

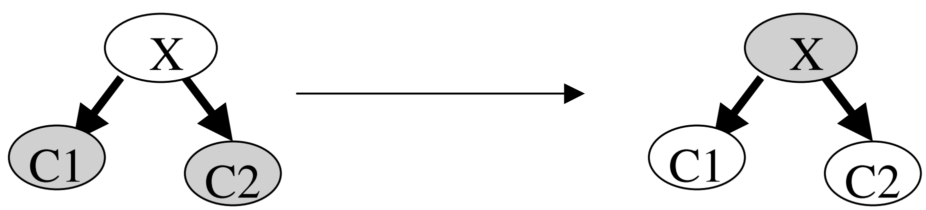
3. The parent is red, the sibling of the parent is red, and the new node (X) is an outside grandchild

➔ Single rotation between P and G + appropriate color change to restore property 3 (but…).



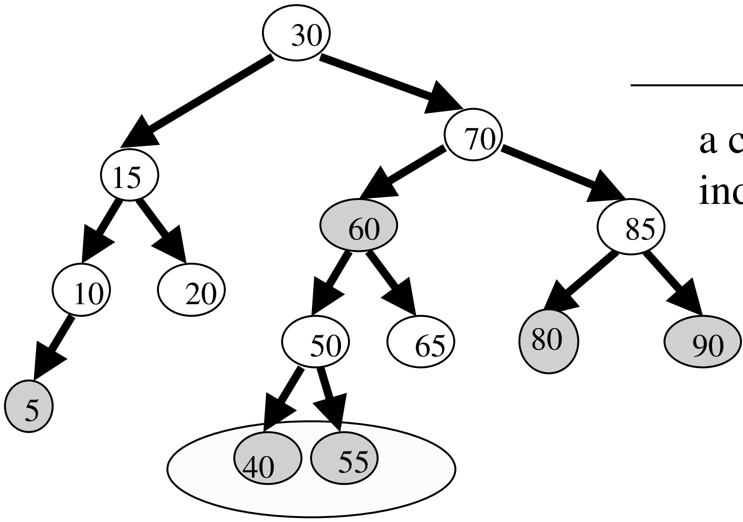
Top-Down Red-Black Trees

- To avoid the possibility to rotate up the tree
- When we arrive at a leaf and insert a node,
  - It is guaranteed that S is not red
  - add a red node
  - rotate single or double if necessary
- Procedure:
  - Color flip
    - On the way down if node X has two red children, flip colors
    - The number of black nodes on paths below X is unchanged

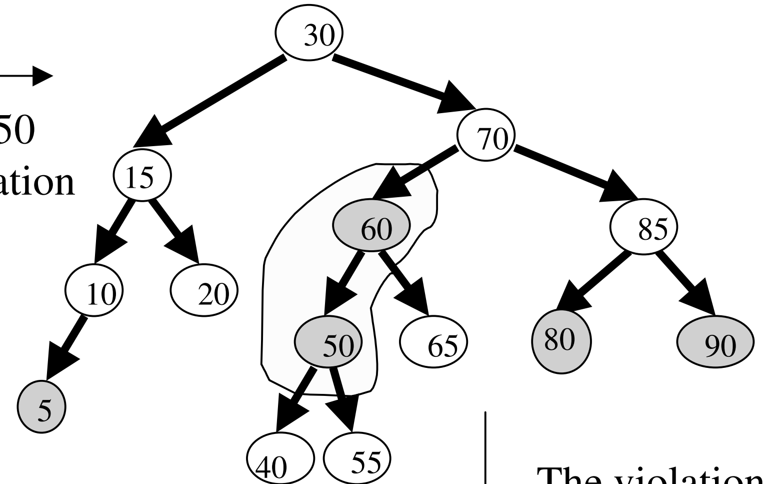


  - If X's parent is red ➔ we have two consecutive red nodes
    - A single/double rotation fixes that
  - Could X's parent's sibling be also red ? No, Why?
    - On the way down, when we saw a node with two red children we flipped the colors

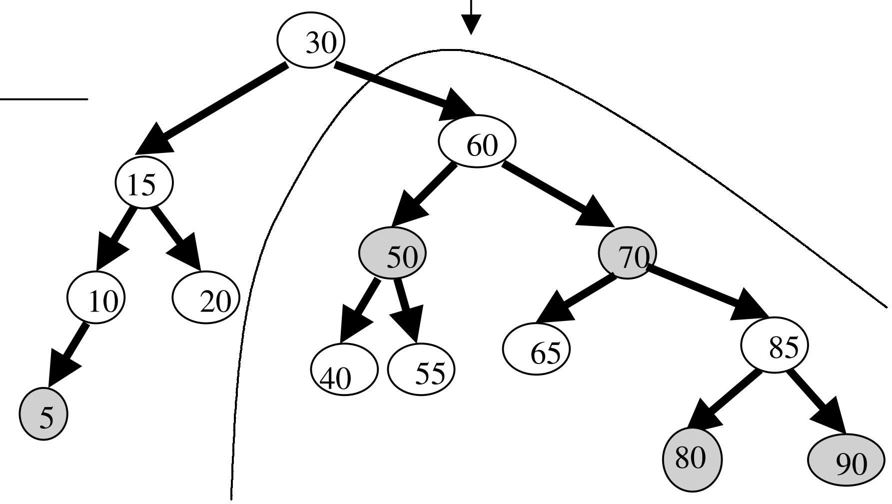
Example: Insert 45



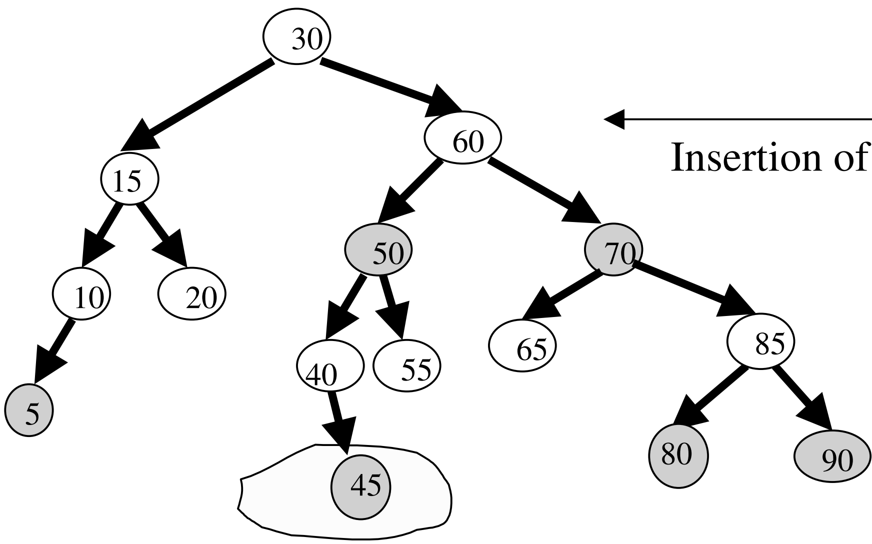
a color flip at 50
induces a violation



The violation is
outside. A single
rotation fixes it.

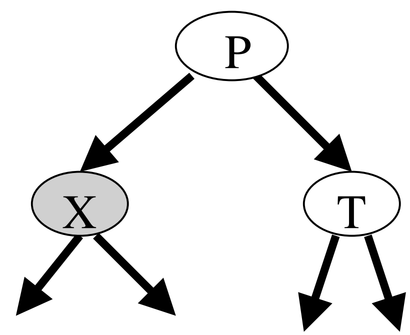


Insertion of 45



Top-Down Deletion

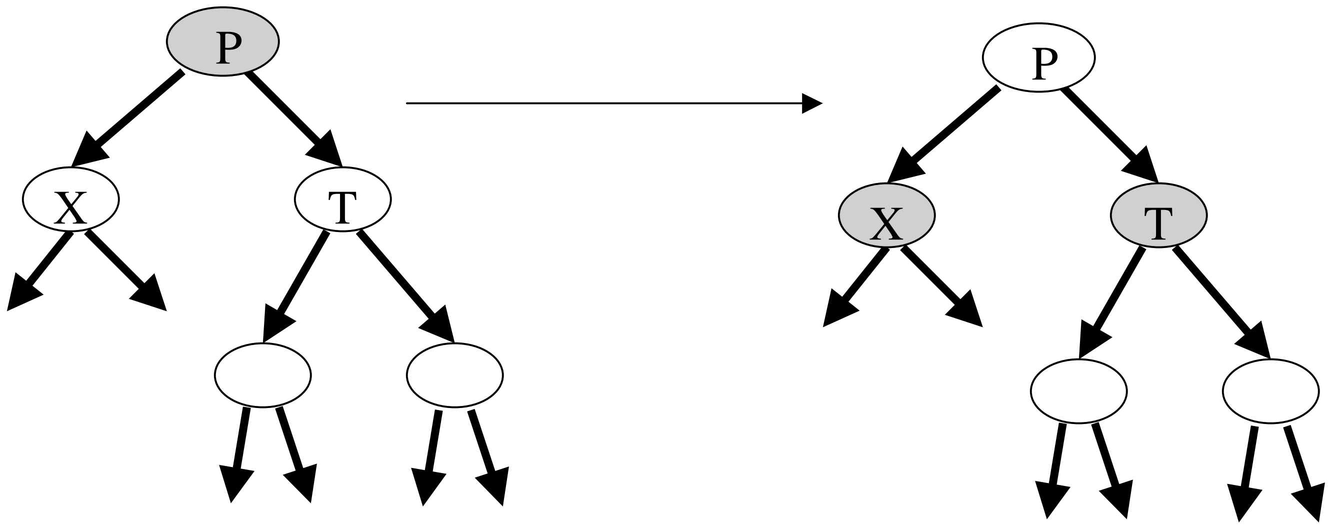
- The problem is not trivial
- In binary search tree, deletion removes nodes that are leaves or have one child
- Deleting a red node is no problem
- Deleting a black node violates property 4 ➔deletion must delete a red node
- Let
  - *X*: the current node
  - *T*: X's sibling
  - *P*: X's parent



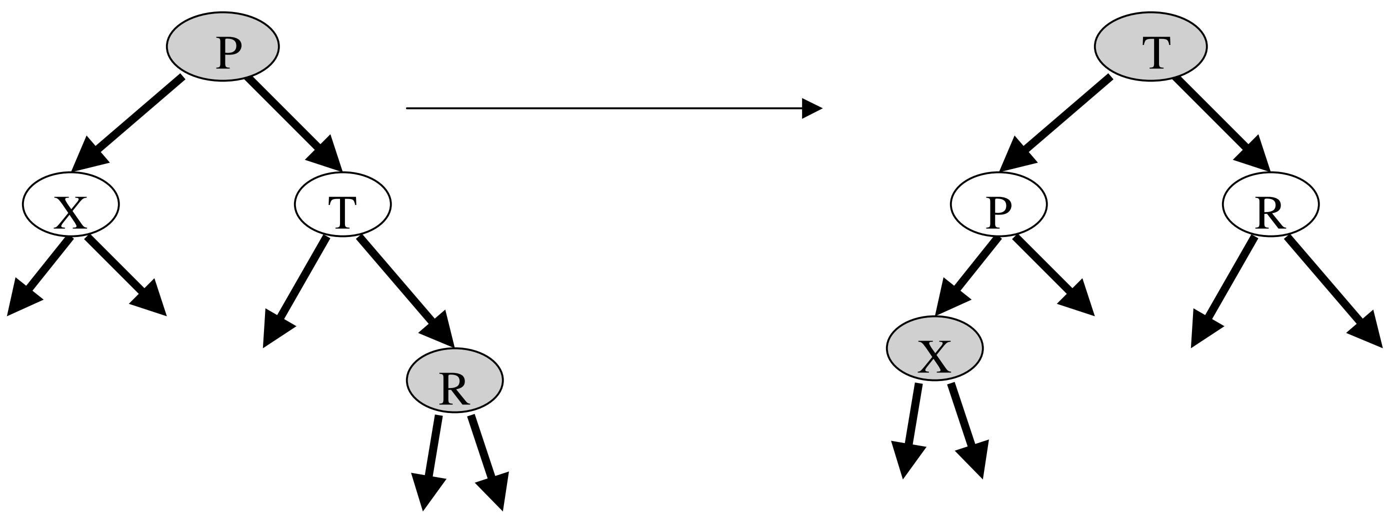
- Starting with the root, as we traverse the tree, we attempt to ensure that X is red (➔ two children are black)
- We consider 4 cases.

Top-Down Deletion (cont.)

1. X has two black children, and both of its sibling's children are black; do a color flip

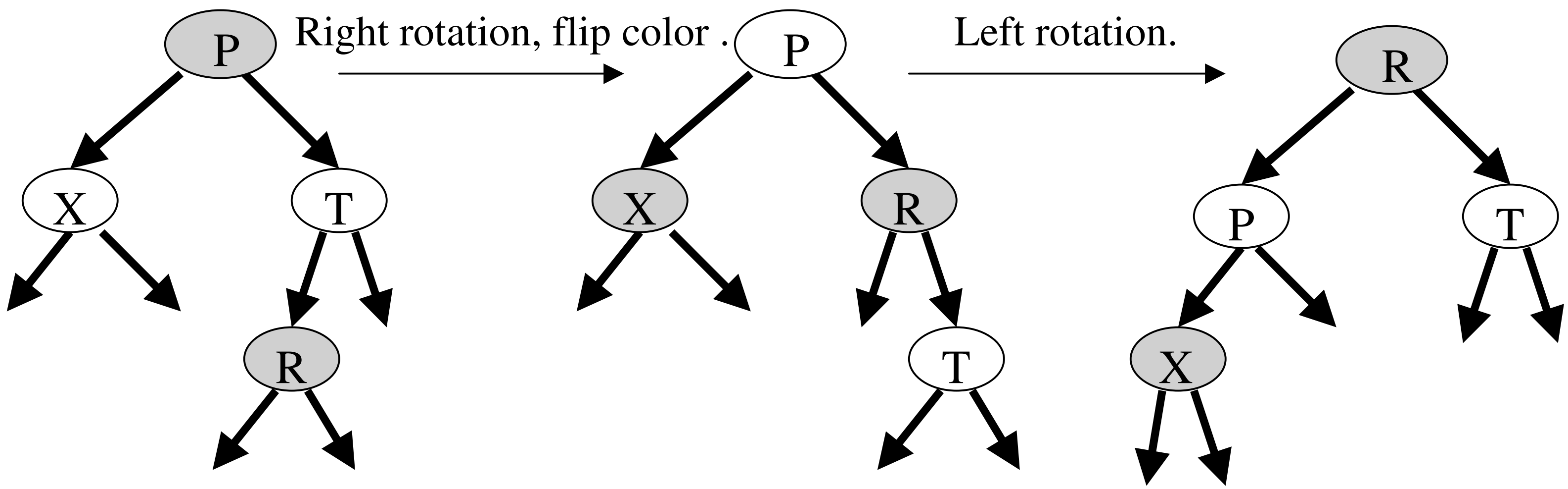


2. X has two black children, and the outer child of its sibling is red; do a single rotation



Top-Down Deletion (cont.)

3. X has two black children, and the inner child of its sibling is red; do a double rotation

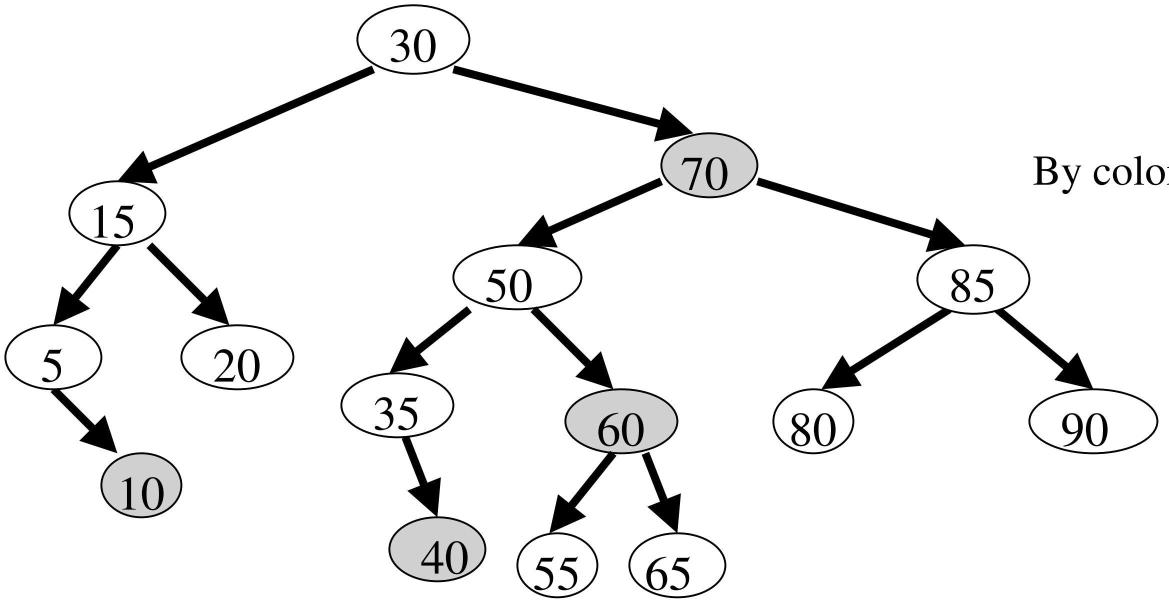


4. X is black and at least one child is red; if we fall through to next level and land on a red child, everything is good; if not, we rotate

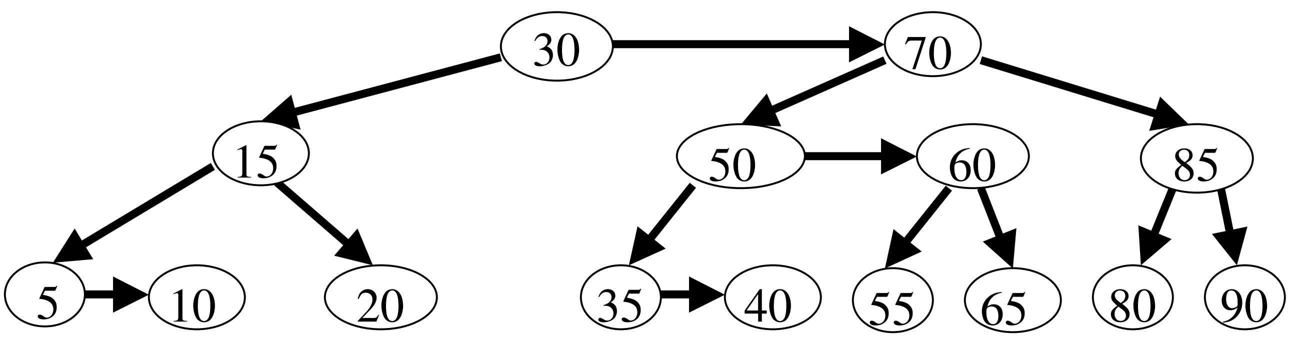
AA-Trees

- Because of many possible rotations in B/R trees and coding difficulty, an alternative to a balance search tree is an AA-tree
- An AA-tree is a R/B tree with a condition: **left children may not be red**
- This restriction greatly simplifies the R/B algorithms
    - Eliminates half of the restructuring cases
    - Simplifies the remove algorithm
- AA-trees are defined by **level of nodes** (not by their colors)
- The level of a node is
    1. 1 if the node is a leaf
    2. The level of its parent, if the node is red
    3. One less than the level of its parent, if the node is black
- Properties:
    - Horizontal links are right pointers (since only right children may be red)
    - There may not be two consecutive horizontal links (since there cannot be consecutive red nodes)
    - Nodes at level 2 or higher must have two children
    - If a node does not have a right horizontal link, then its two children are at the same level

Example



By colors (as a R/B tree)



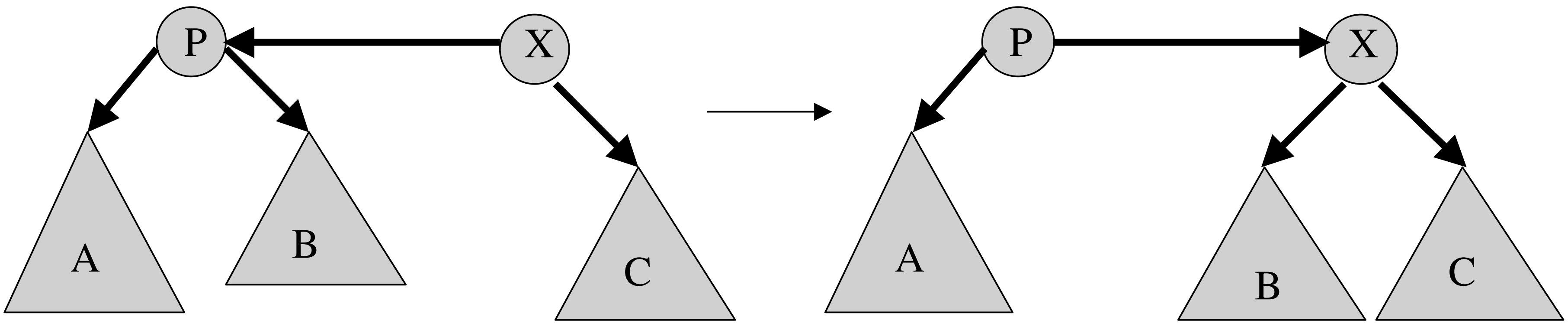
By levels

Insertion

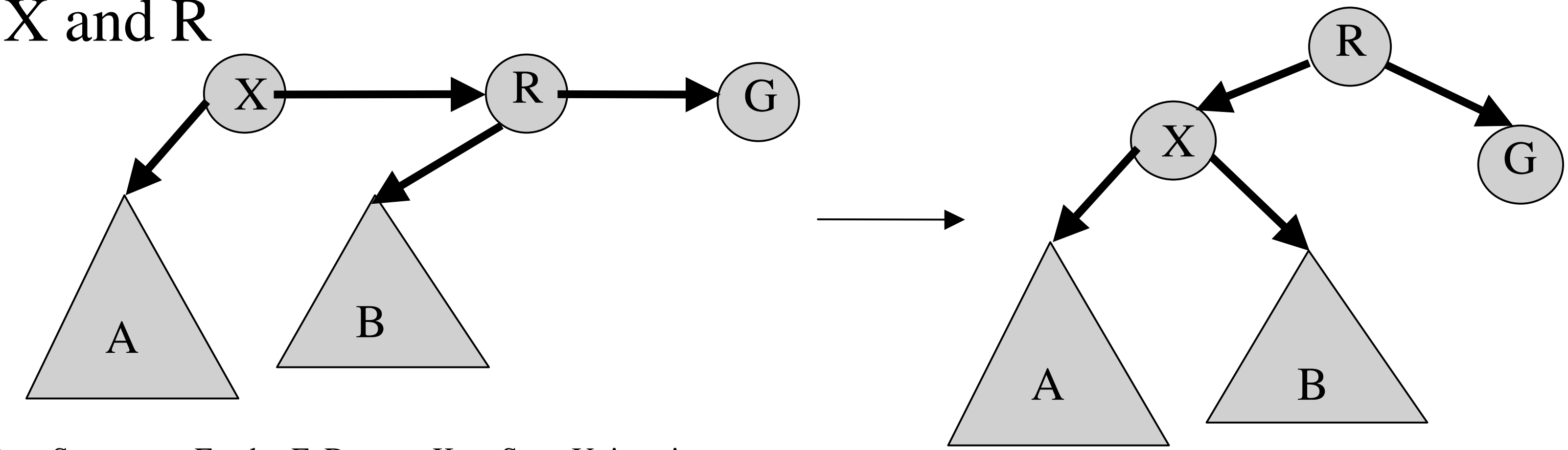
• Insertion of a new item is always done at the bottom level.

• It uses usual recursive algorithm and two method calls (skew, split)

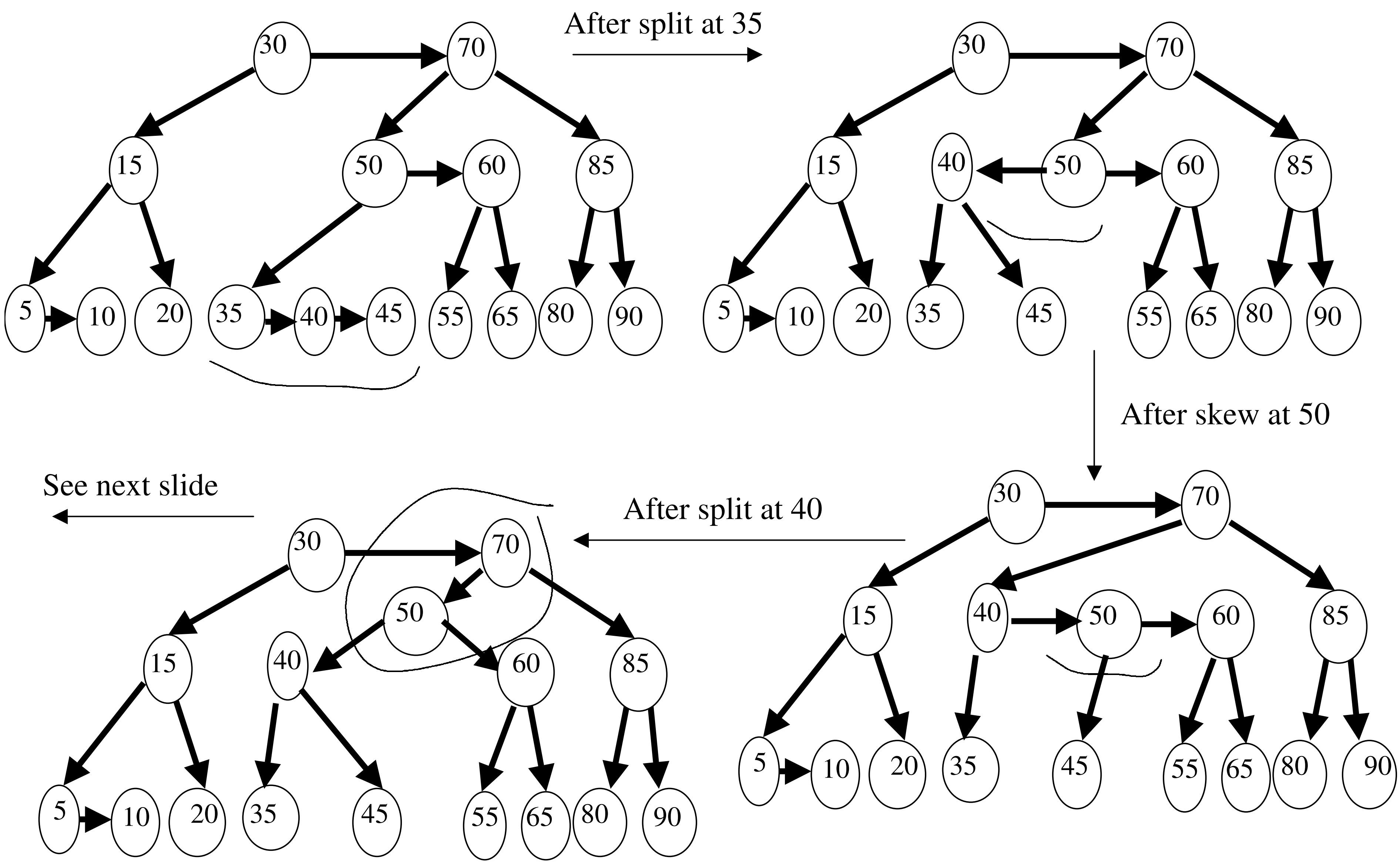
• **Skew** procedure (removes left horizontal links) ➔ rotation between X and P



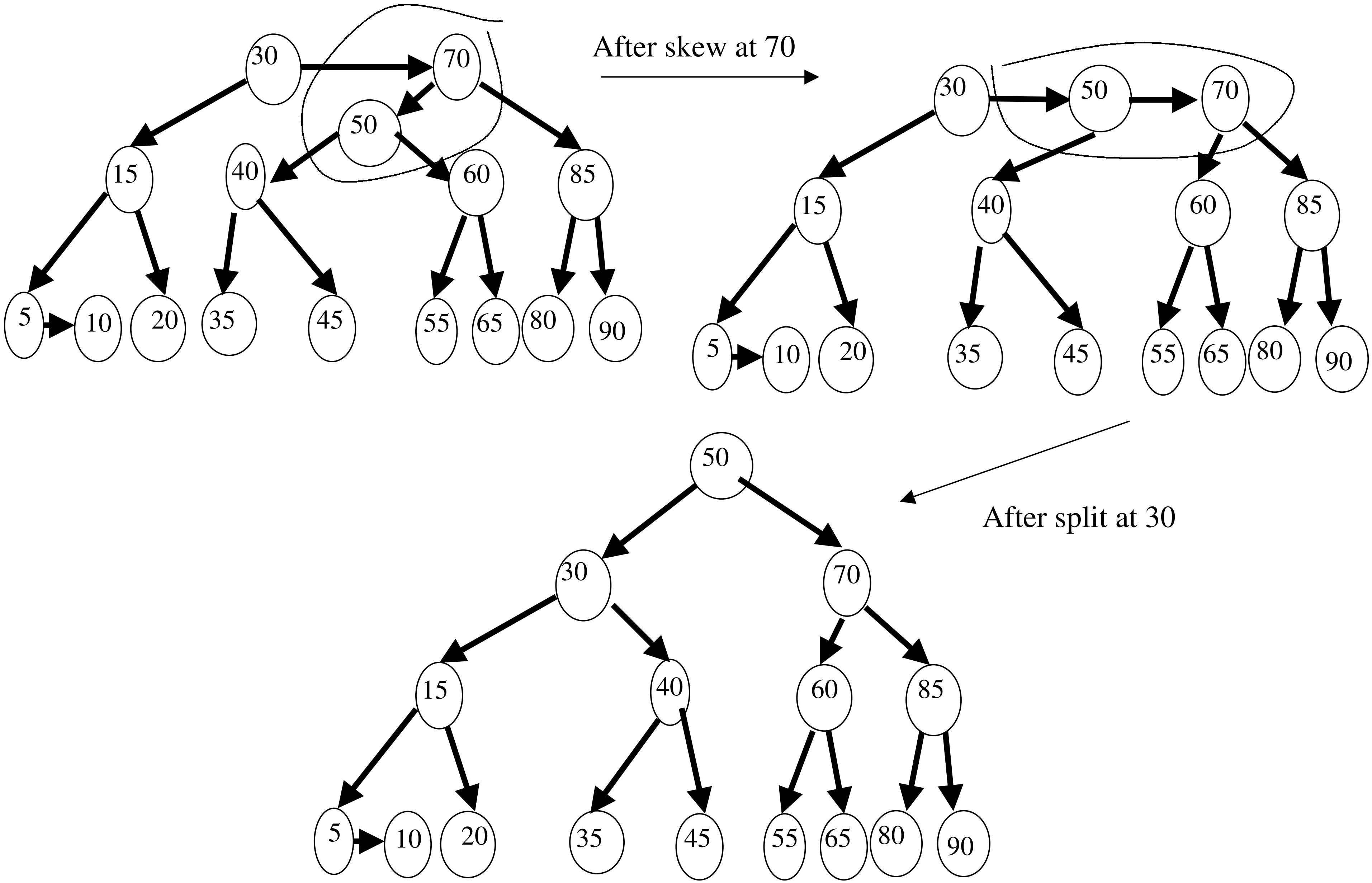
• **Split** (fixes consecutive right horizontal links) ➔ rotation between X and R



Example: Insert 45

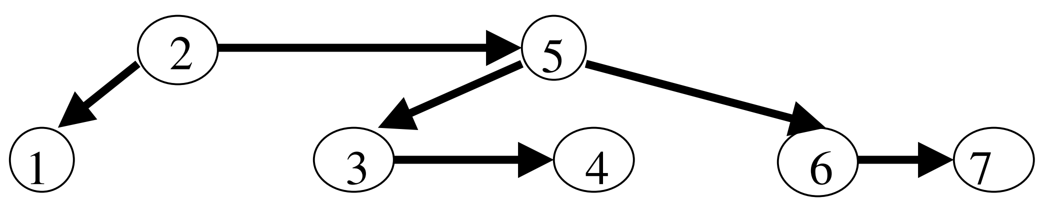


Example: Insert 45 (cont.)



Deletion

- Three cases could occur with deletion in general Binary Search Tree
  - the item is a leaf
  - the item has one child
  - the item has two children (➔ we have *removeMin* in subtree, i.e., again a node with at most one child)
- In AA-trees deletion is made easier because the one child case can occur only at level 1 and we are willing to use recursion.
- Every case boils down to remove a level 1 node
- After a recursive removal, three skews and two splits guarantee rebalancing



When 1 is deleted, all nodes become level1, thereby introducing horizontal left links.

Homework

- Read carefully sections 19.6, 19.7 and 19.8 of the textbook
- Problems:

  - Problem 18.1 page 637 of the text
  - Problem 18.2 pages 637-638 of the text
  - Problem 18.4 page 638 of the text
  - Problems 19.1 and 19.2 page 718 of the text
  - Problems 19.5 and 19.7 page 718 of the text