Struktur Data & Algoritme
(*Data Structures & Algorithms*)

Red Black Tree

Suryana Setiawan setiawan@cs.ui.ac.id
Denny denny@cs.ui.ac.id
Fakultas Ilmu Komputer
Universitas Indonesia

Semester Ganjil - 2006/2007
Version 2.0 - Internal Use Only

Motivation

- History: invented by Rudolf Bayer (1972) who called them "symmetric binary B-trees", its modern name by Leo J. Guibas and Robert Sedgewick (1978).
- Red-black trees, along with AVL trees, offer the best possible worst-case guarantees for insertion time, deletion time, and search time -- O(log n).
- many data structures used in computational geometry can be based on red-black trees.
- in functional programming, used to construct associative arrays and sets which can retain previous versions after mutations.

Objectives

- Understand the definition, properties and operations of Red-Black Trees.

Outline

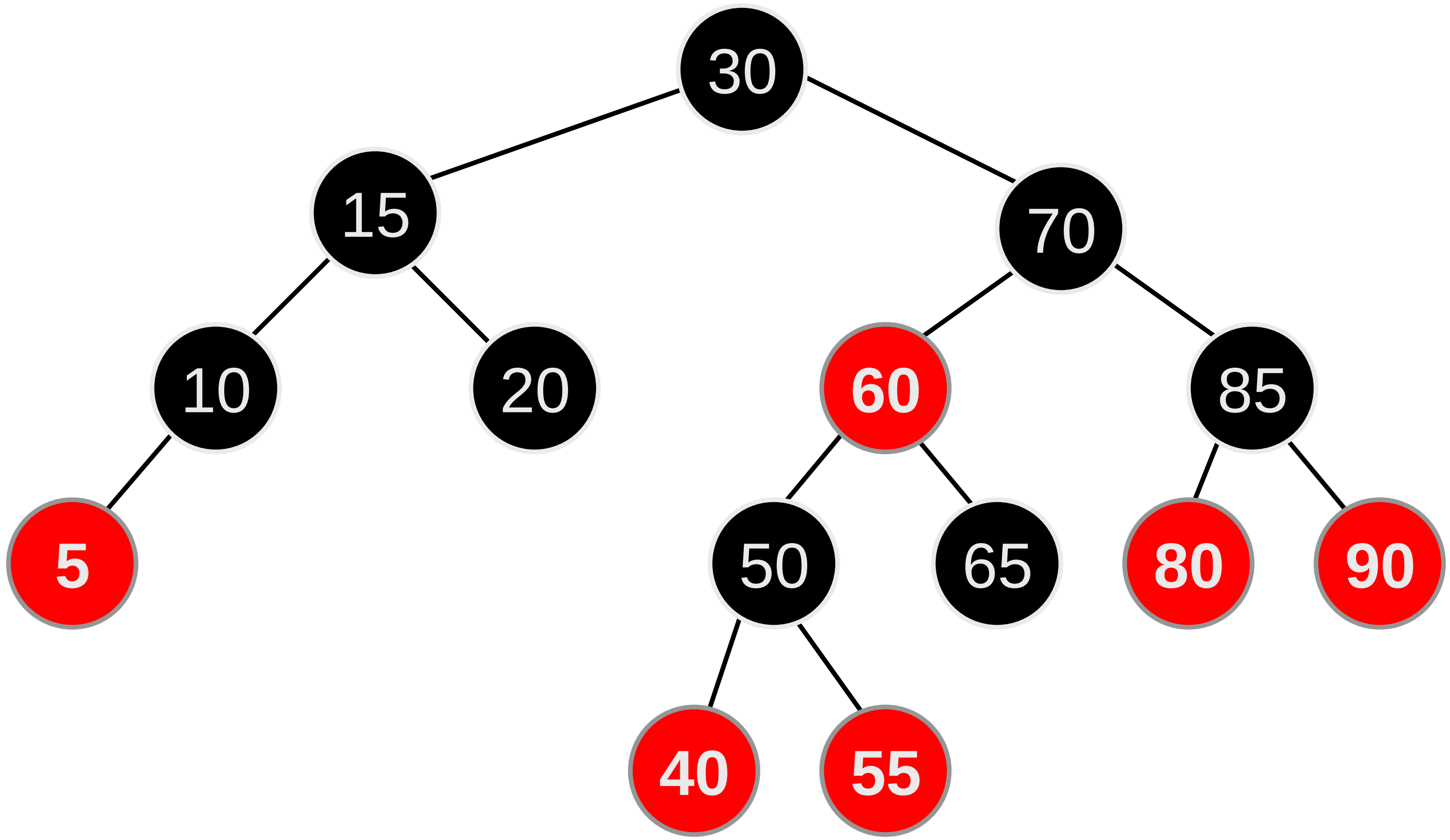
- **Red-Black Trees**
  - Definition
  - Operation

Red-Black Trees: Definition

- The red-black tree is a **binary search tree** whose nodes colored either **black** or **red**, under properties**:**
  - Property#1. Every node is colored either **red** or **black**
  - Property#2. The root is **black**
  - Property#3. If a node is **red**, its children must be **black**
  - Property#4. Every path from a node to a null reference must contain the same number of **black** nodes

Red-Black Trees

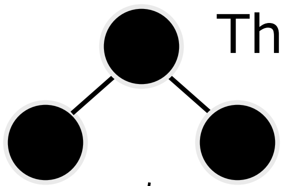
- Example: (insertion sequence: 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55)



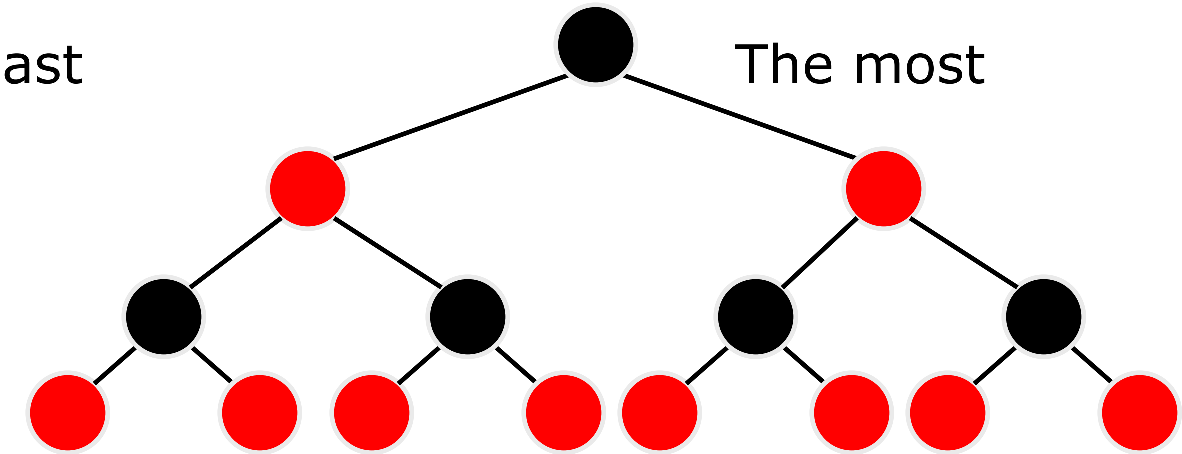
Implications

- Consecutive **red** nodes are disallowed (Pr#3)
- Every **red** node must have a **black** parent (Pr#3)
- The longest possible path from the root to a leaf is no more than twice as long as the shortest possible path. (Pr#3 & Pr#4)

The least



The most



$$2^B - 1 \le N \le 2^{2B} - 1$$

$$2^B \le N + 1 \le 2^{2B}$$

$$\log 2^B = B \qquad \log 2^{2B} = 2B$$

$$B \le \log(N+1) \qquad \log(N+1) \le 2B$$

$$\frac{1}{2}\log(N+1) \le B \le \log(N+1)$$

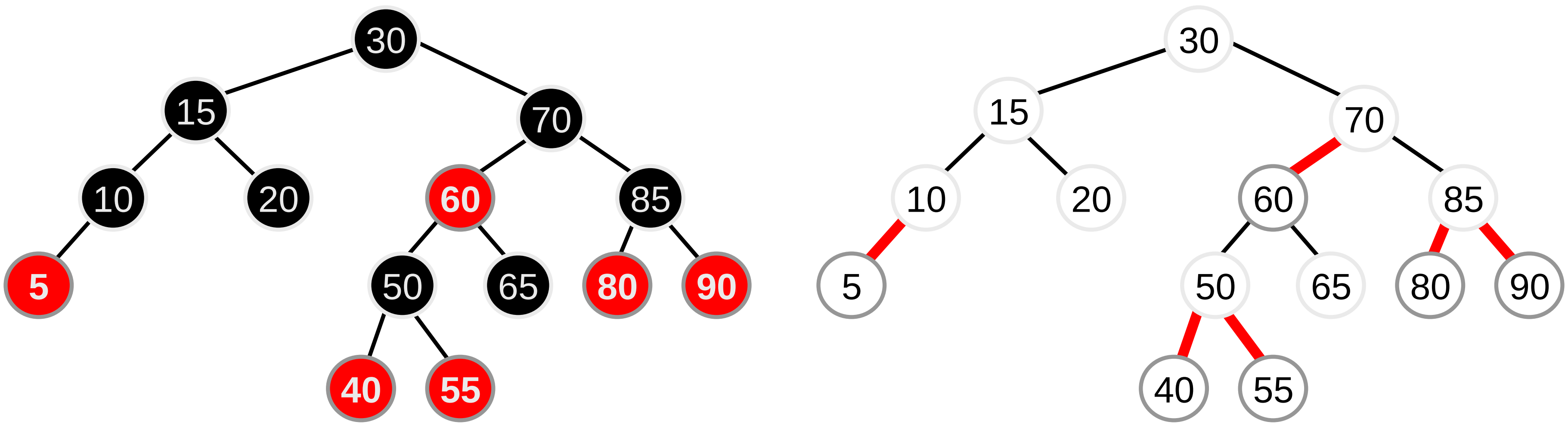
$$\log(N+1) \le H \le 2\log(N+1)$$

- B = total black nodes from root to leaf
- N = total all nodes
- H = height

All operation guaranteed logarithmic.

Variants of the description

- A red-black tree as a **binary search tree** whose <u>edges</u> (instead of nodes) are colored in red or black.
  - The color of a node in our terminology corresponds to the <u>color of the edge connecting the node to its parent</u>, except that the root node is always black.



Algorithm: Finding A node

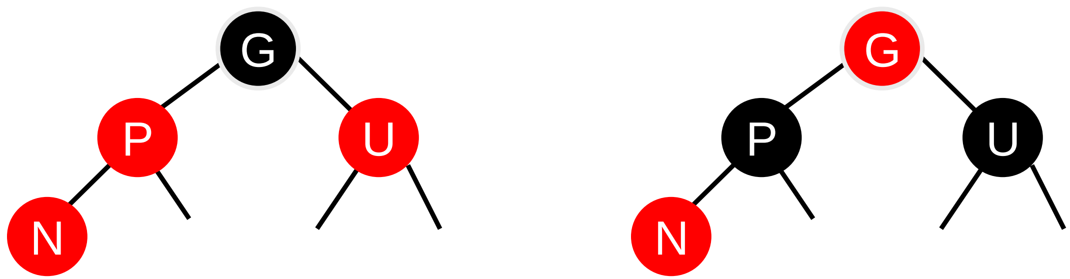
- As in the binary search node…

Algorithm: Inserting a Node (1)

- Inserted as in binary search tree as a new leaf node, but the node must be colored **red**
  - why?
    - new item is always inserted as leaf in the tree
    - if we color a new item black, then we will have a longer path of black nodes (violate property #4)
- Is the resulting tree still a red-black tree?
  - if the parent is **black**, *no problemo*
  - If it is the only node in the tree (i.e., it is a **root**), repaint it to be a **black** node.
  - if the parent is **red**
    (two consecutive red nodes → violate Pr #3),
    then should be restructured…..

Algorithm: Inserting a Node (2)

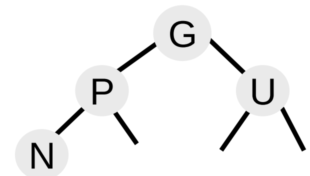
- Let, the new node is N, the Parent node is P, and the sibling of the parent node is U (from 'uncle'), and the parent of P is G (from 'grandparent'). If P and U are **red** and G is **black**, repaint P and U to be black, and G to be **red** (color flipping)



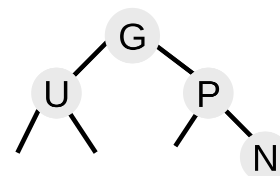
- G, then,
  - violate Pr#2 (when it is the root) → G is changed just to be black (as in the first page).
  - Violate Pr#3 (when G's parent is **red**) → perform the algo. with G as N.

Algorithm: Inserting a Node (3)

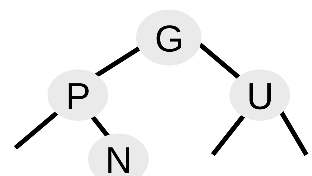
- <u>If P is **red** but U is **black** (or empty substree)</u>, then there are four subcases:
  - N is left child of P and U is right-sibling of P → SRR



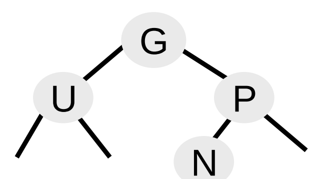
  - N is right child of P and U is left-sibling of P → SLR



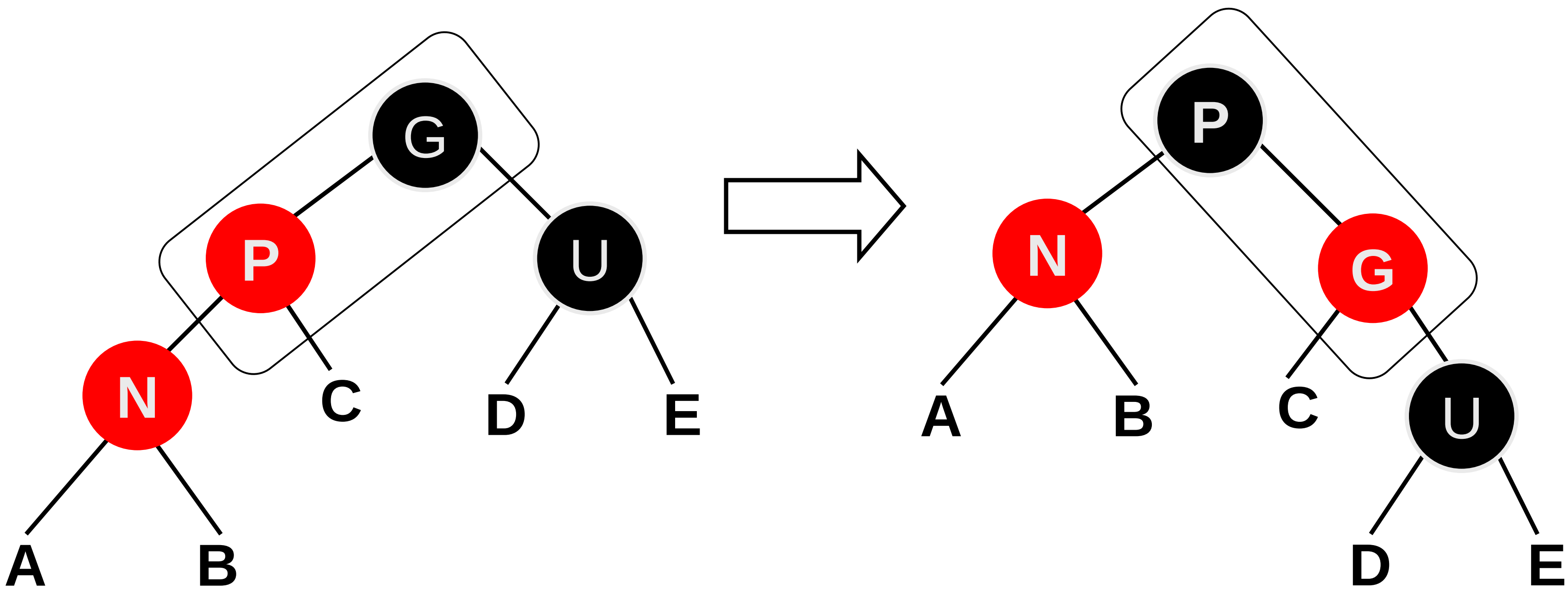
  - N is right child if P and U is right-sibling of P → DRR



  - N is left child if P and U is left-sibling of P → DLR



Single Rotation



N: new node

P: parent

U: uncle

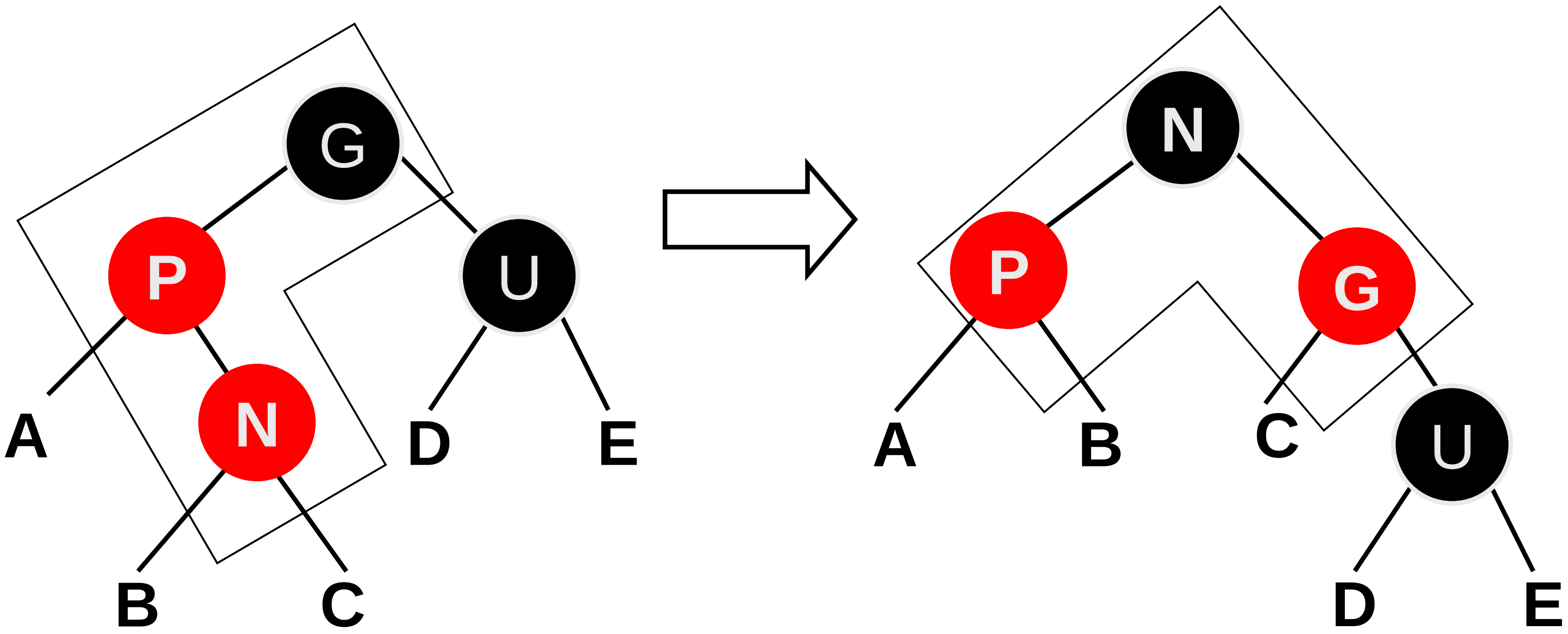
G: grandparent

Color Changes:

P: red → black

G: black → red

Double Rotation



N: new node

P: parent

U: uncle

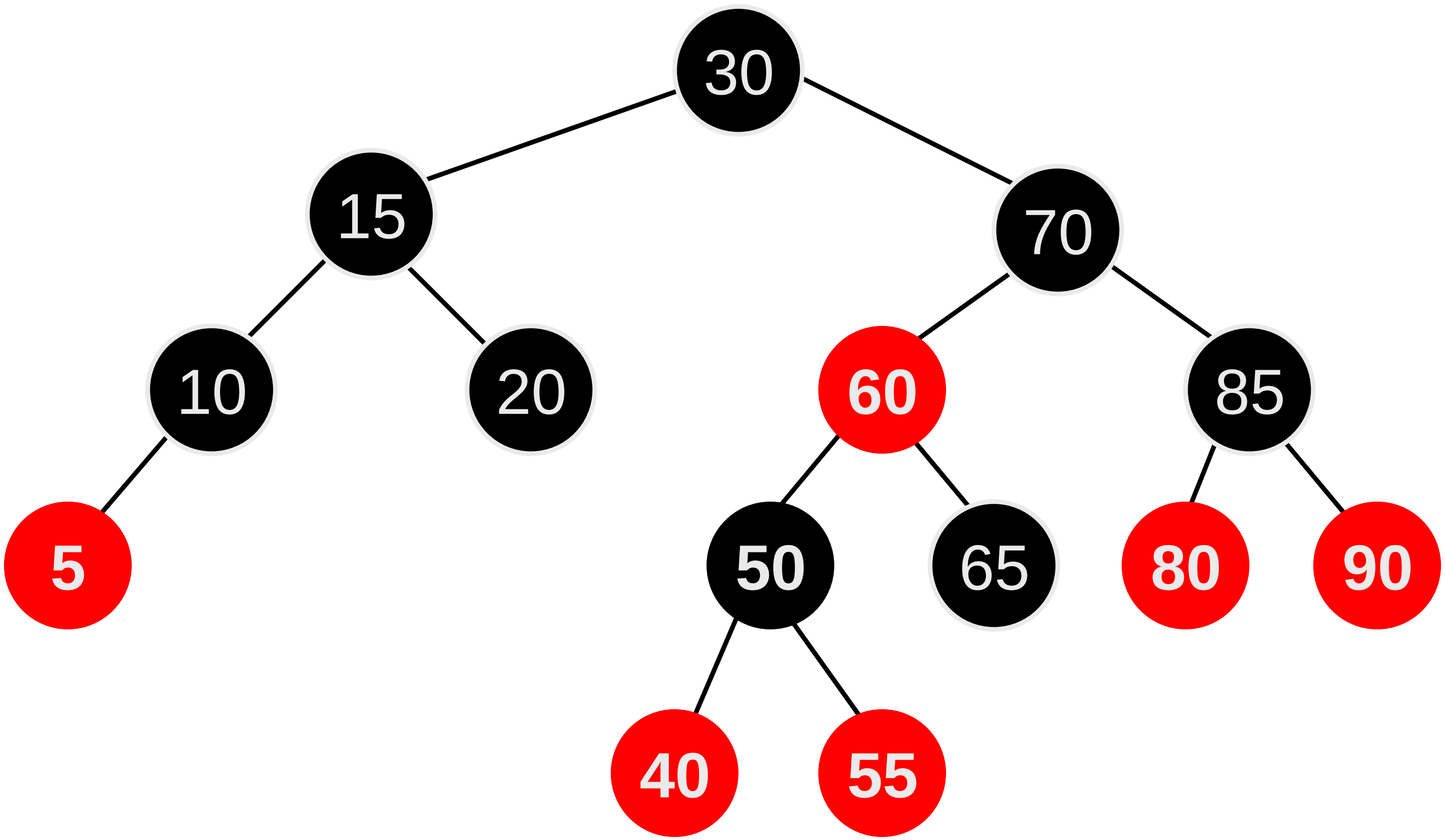
G: grandparent

Color Changes:

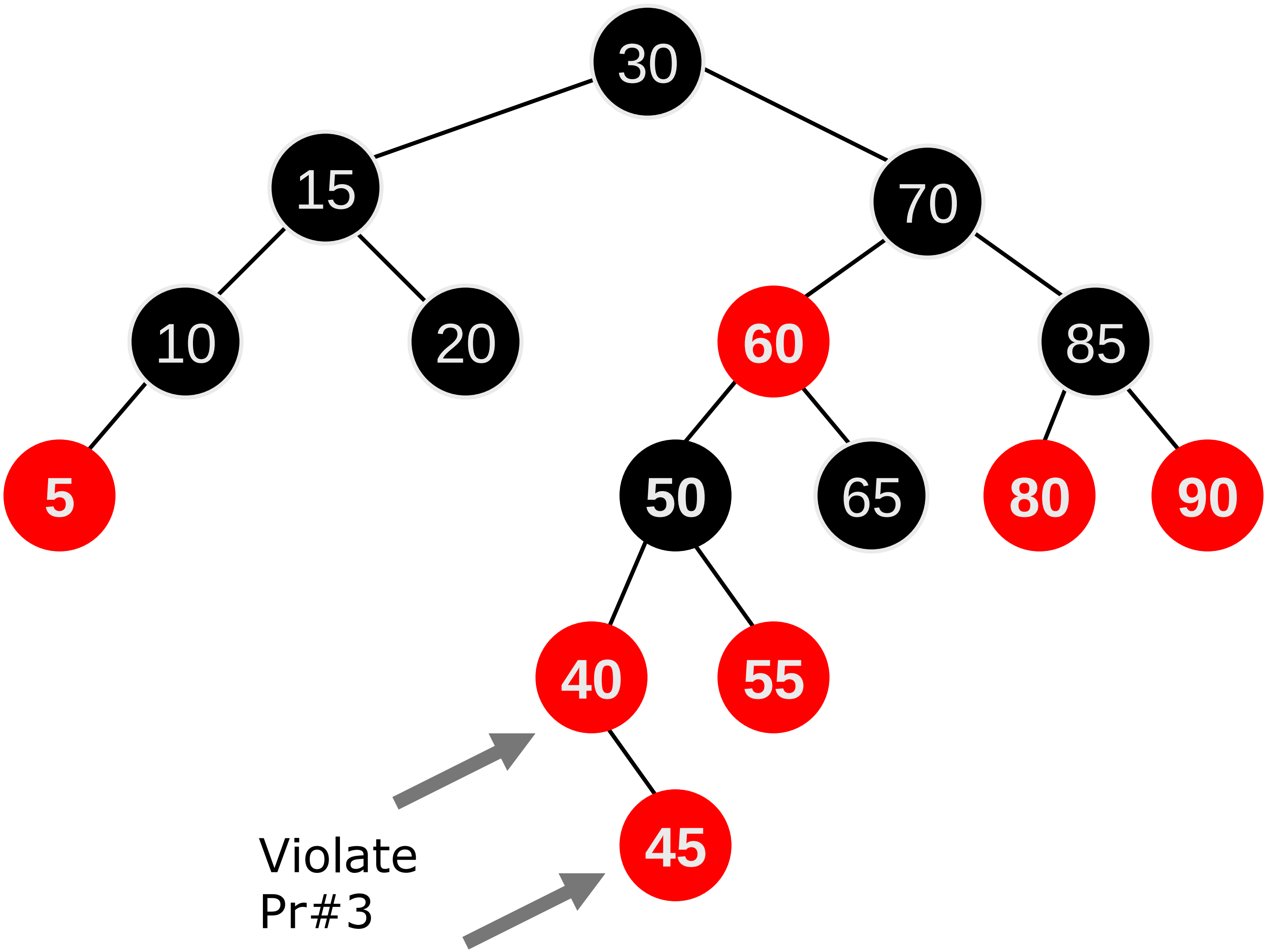
N: red → black

G: black → red

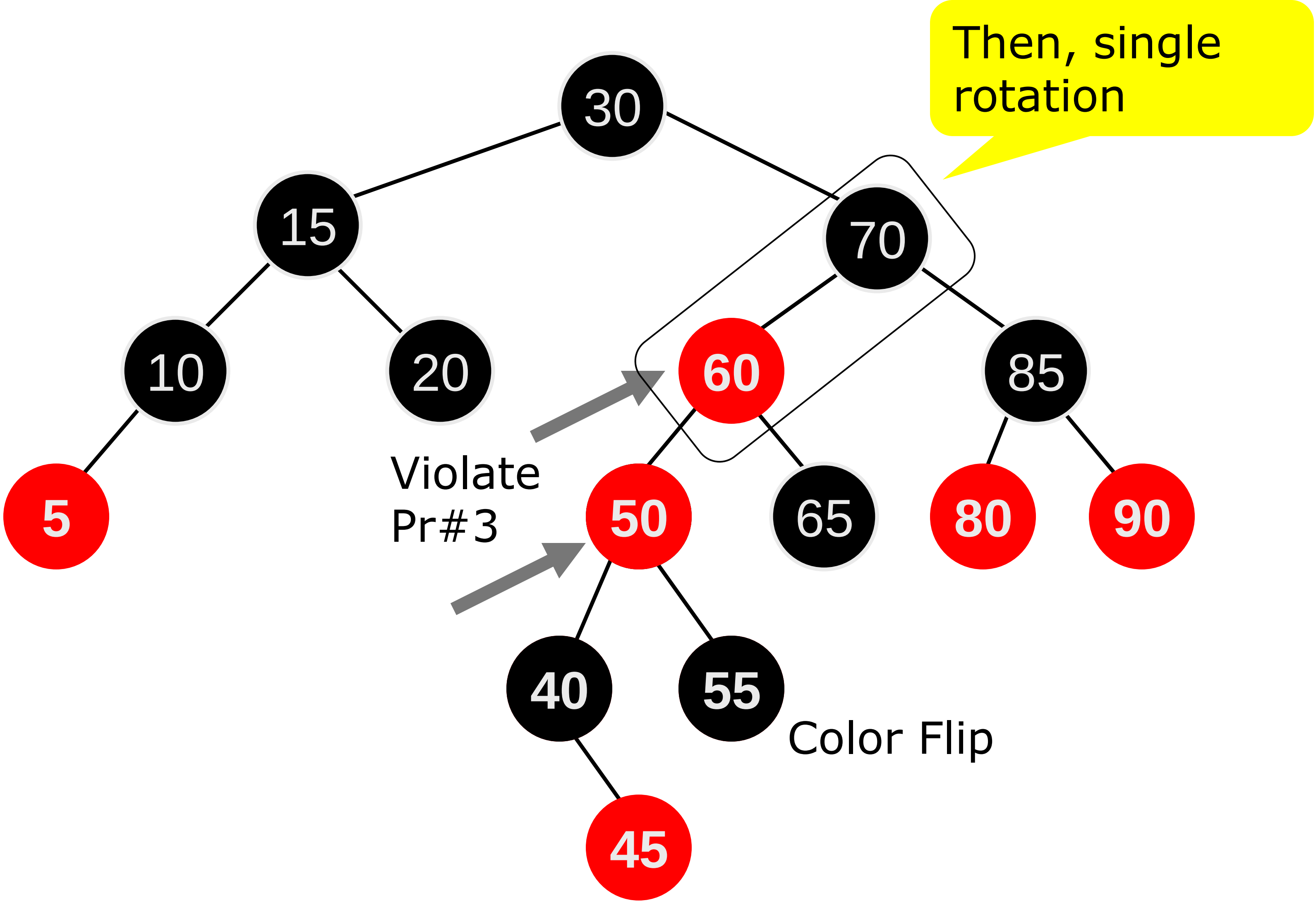
Insert 45 (original)



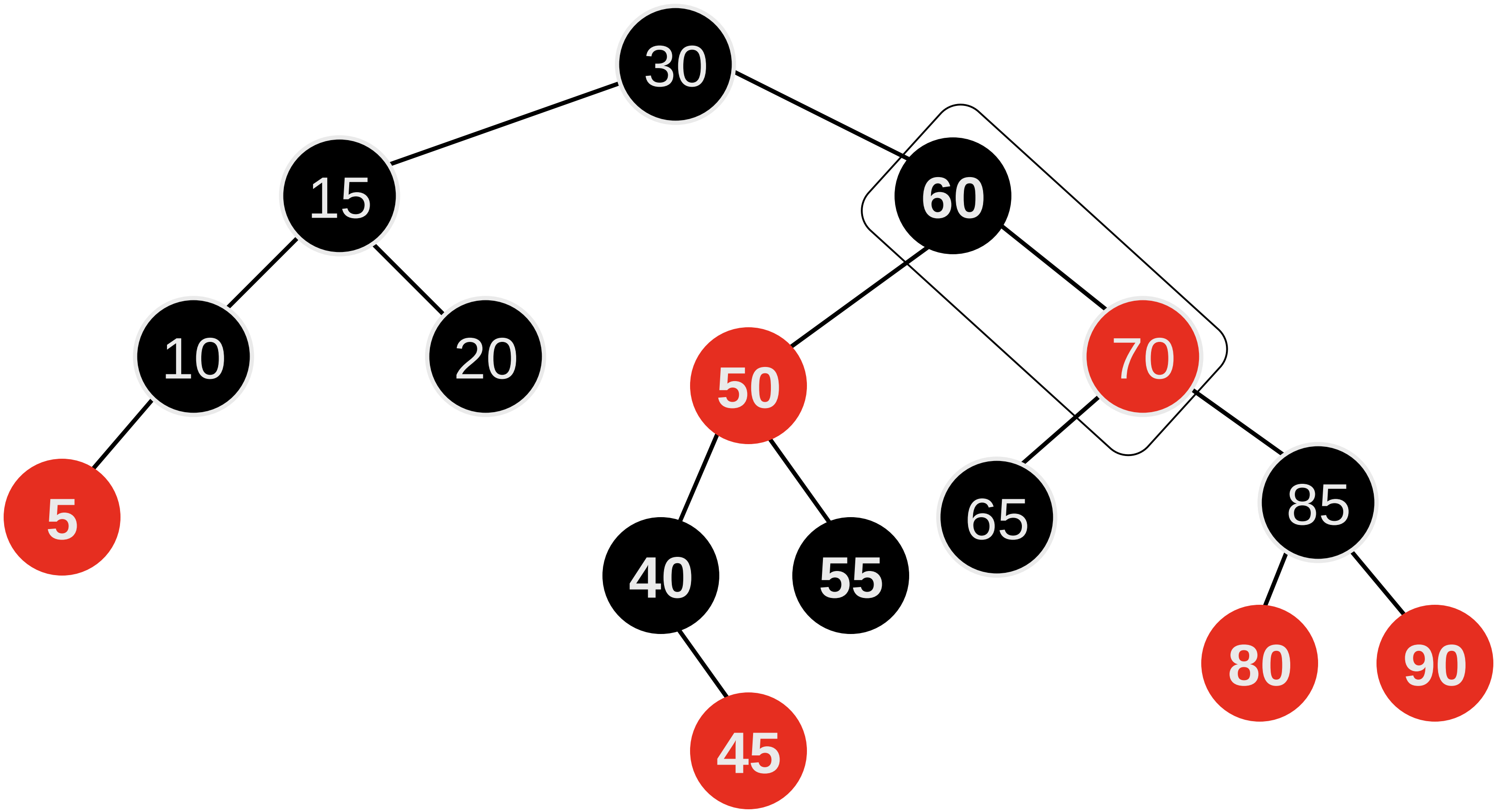
Insert 45



Insert 45 (color flip)



Insert 45 (single rotation)



Top-down algorithm

- Color-flipping described previously performed buttom-up wise (from the insertion position up to above).
- Weiss'book recommends to do **color-flipping top-down wise**:
  - On the way going down from the root **during in finding the insertion position**, if the current node has two red children,
    color flip them (the current and the children).
  - The flipping will be possibly followed by other flipping as well as the rotation
  - When the insertion position reached and the new red node inserted, the next possibly operation is just the rotation

Algorithm: deletion

- Firstly, the deletion following **the binary seach tree's deletion algorithms**
  - Every deletion causes in deleting **a leaf node** or deleting **an internal node with one child**
- Deleting **red node → no problemo**
- Deleting **black node** whose a **red child →** will also be OK by switching its color with its child's color
- The problem, when both the **deleted node** and **its child are black**

Summary

- **Red-Black trees use color as balancing information instead of height in AVL trees.**
- **An insertion may cause a local perturbation (two consecutive red nodes)**
- **The pertubation is either**
  - resolved locally (rotations), or
  - propagated to a higher level in the tree by recoloring (color flip)
- **O(1) for a rotation or color flip**
- **At most one restructuring per insertion.**
- **O(log n) color flips**
- **Total time: O(log n)**