



Introduction to

# *Algorithm Design and Analysis*

[8] *log n* search



*Yu Huang*

<http://cs.nju.edu.cn/yuhuang>  
Institute of Computer Software  
Nanjing University



# In the last class...

- **Selection – warm up**
  - Max and min
  - Second largest
- **Selection – rank  $k$  (median)**
  - Expected linear time
  - Worst-case linear time
- **Adversary argument**
  - Lower bound



# The Searching Problem

- **Searching vs. Selection**
  - *Search* for “Alice” or “Bob”
    - The key itself matters
  - *Select* the “rank 2” student
    - The partial order relation matters
- **Expected cost for searching**
  - Brute force case:  $O(n)$
  - Ideal case:  $O(1)$
  - Can we achieve  $O(\log n)$ ?



# The Searching Problem

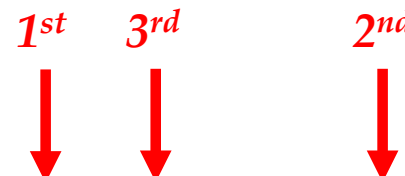
- Essential of *searching*
  - How to *organize the data* to enable efficient search
  - *logn* search
    - Each search cuts off half of the search space
    - How to organize the data to enable *logn* search
- *logn* search techniques
  - Warmup
    - Binary search over *sorted* sequences
  - *Balanced* Binary Search Tree (BST)
    - Red-black tree



# Binary Search by Example

- **Binary search for “24”**
  - Divide the search space
  - Cut off half the space after each search

The sequence is  
already sorted



3	5	6	20	21	24	30	40	45	50
---	---	---	----	----	----	----	----	----	----

Pseudo code in  
p.129 [Baase01]

# Binary Search Generalized

- **Peak-number**
  - Uni-modal array
- **Least number not in the array**
  - Sorted array of natural numbers
- **$A[i]=i$** 
  - Sorted array of integers



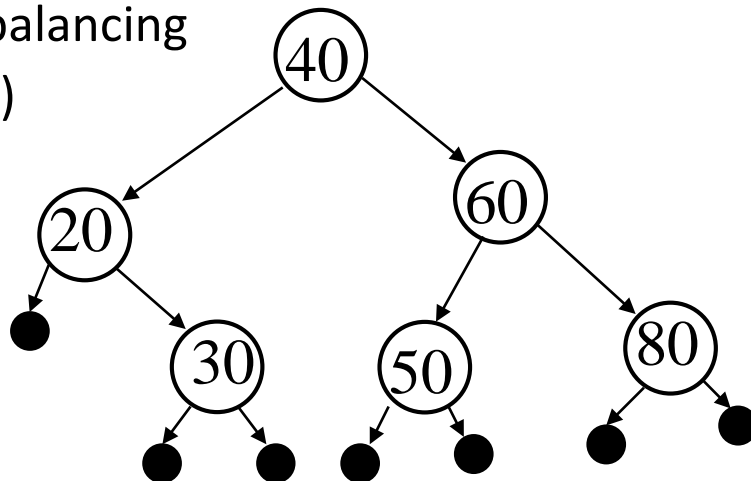
# Balanced Binary Search Tree

- **Binary search tree (BST)**
  - Definitions and basic operations
- **Definition of Red-Black Tree (RBT)**
  - Black height
- **RBT operations**
  - Insertion into a red-black tree
  - Deletion from a red-black tree

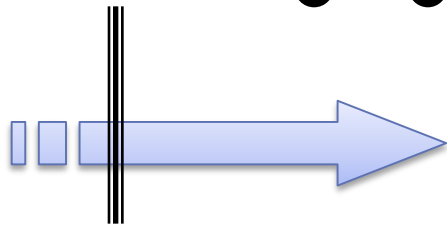
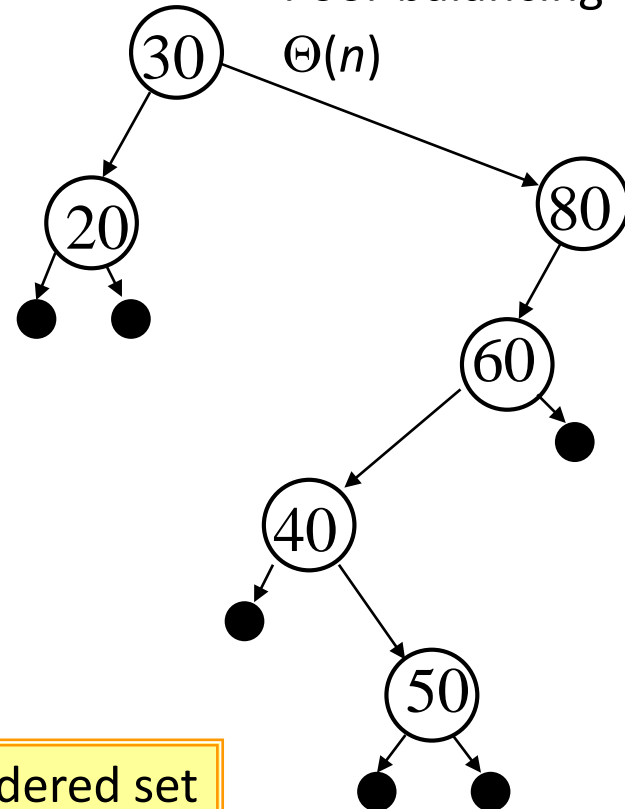


# Binary Search Tree Revisited

Good balancing  
 $\Theta(\log n)$



Poor balancing  
 $\Theta(n)$



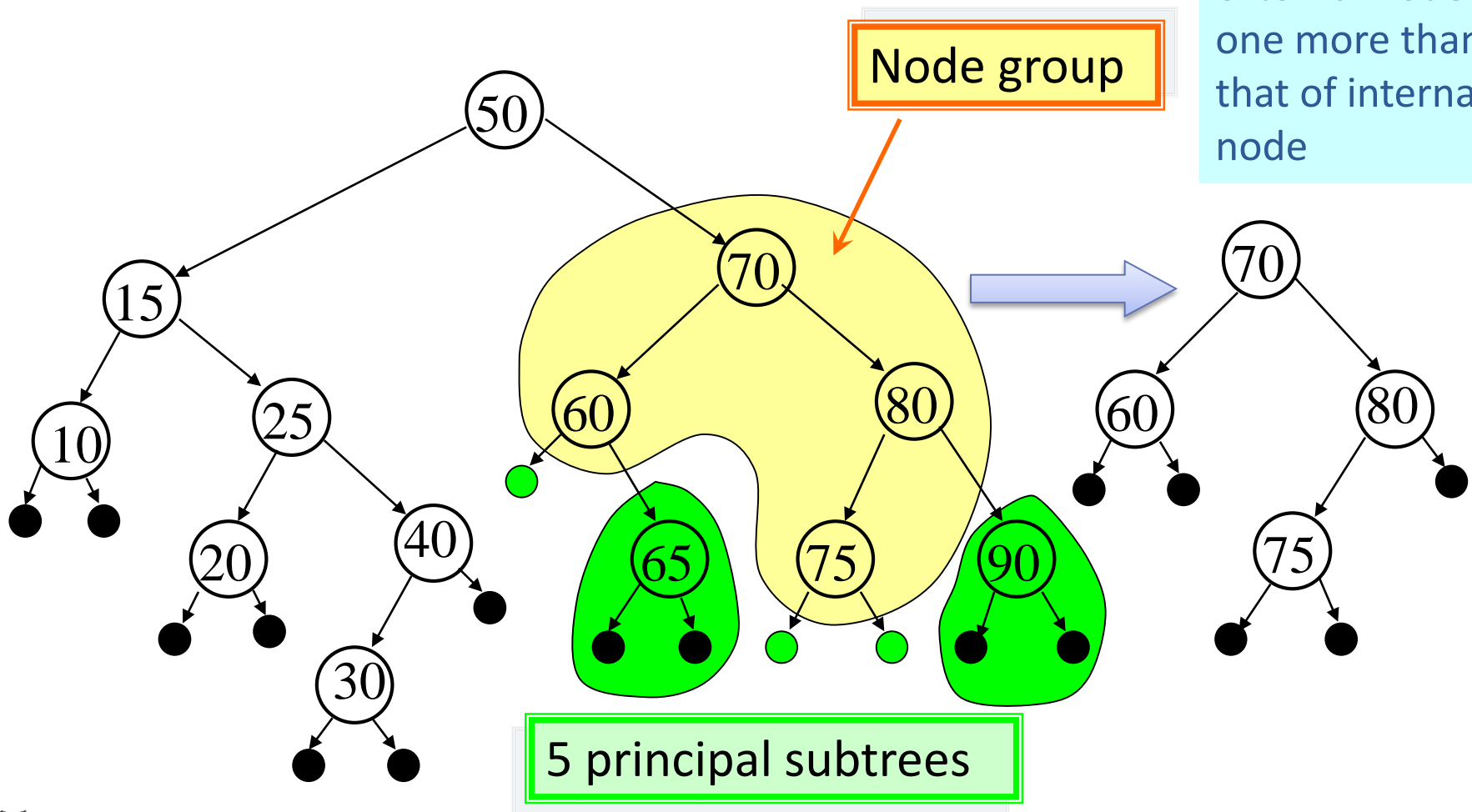
*In a properly drawn tree, pushing forward to get the ordered list.*

- Each node has a key, belonging to a linear ordered set
- An inorder traversal produces a sorted list of the keys

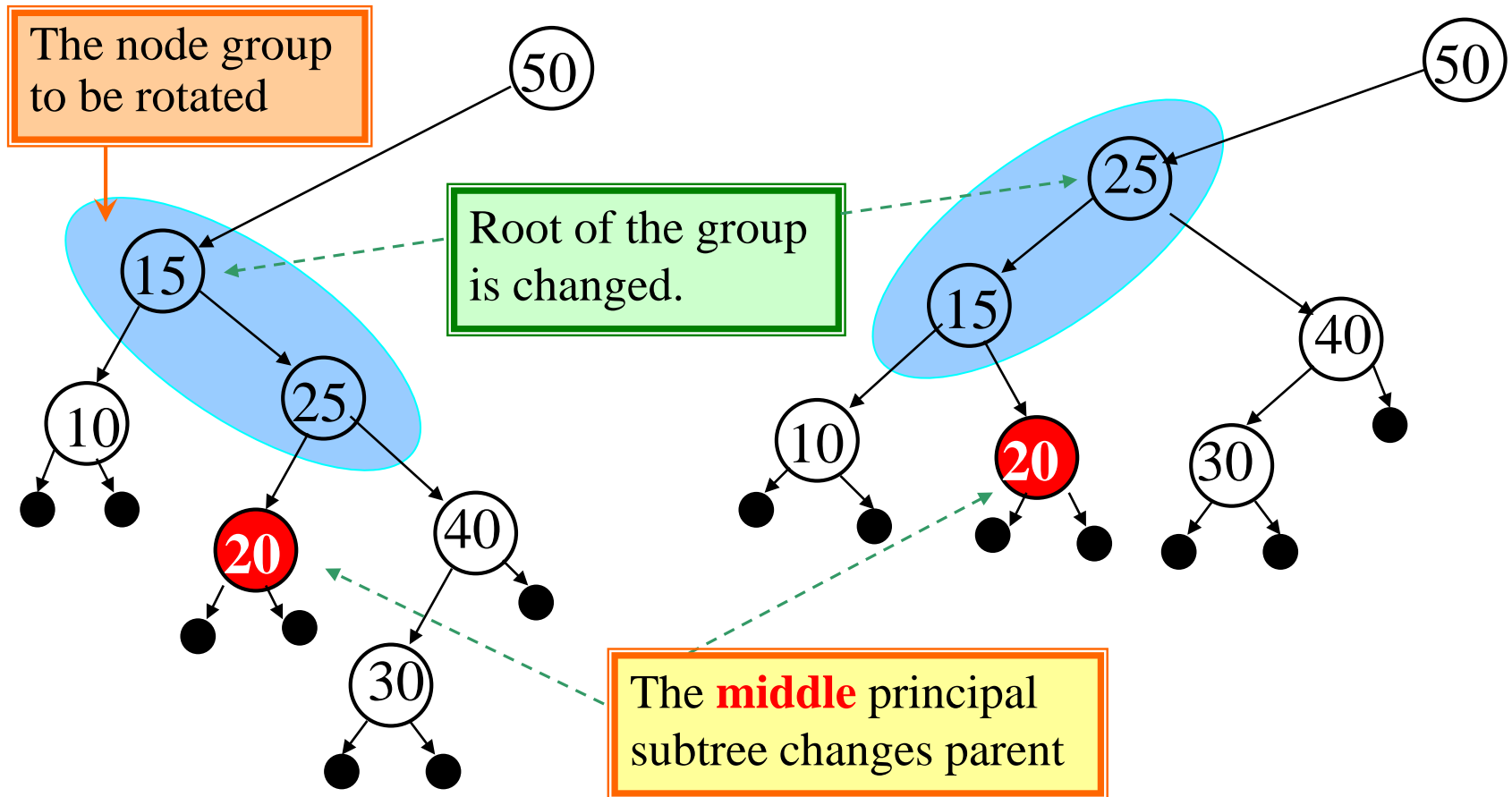


# Node Group

As in 2-tree, the number of external node is one more than that of internal node




# Balancing by Rotation



# Red-Black Tree: Definition

- If  $T$  is a **binary search tree** in which each node has a color, red or black, and all external nodes are black, then  $T$  is a **red-black tree** if and only if:
  - [*Color constraint*] No red node has a red child
  - [*Black height constraint*] The **black length** of all external paths from a given node  $u$  is the same (the black height of  $u$ )
  - The root is black.
- ***Almost*-red-black tree (ARB tree)**
  - Root is red, satisfying the other constraints.



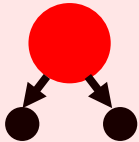
Balancing is under control

# $RB_i$ and $ARB_i$

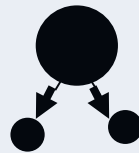
$RB_0$



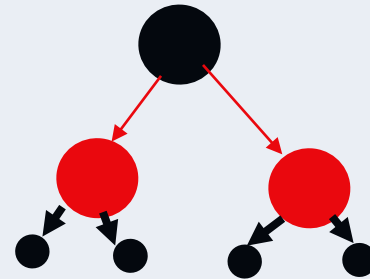
$ARB_1$



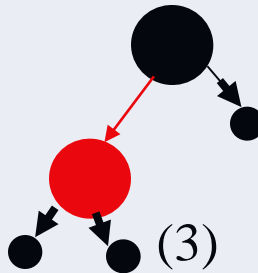
(1)



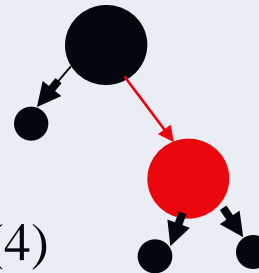
(2)



(3)

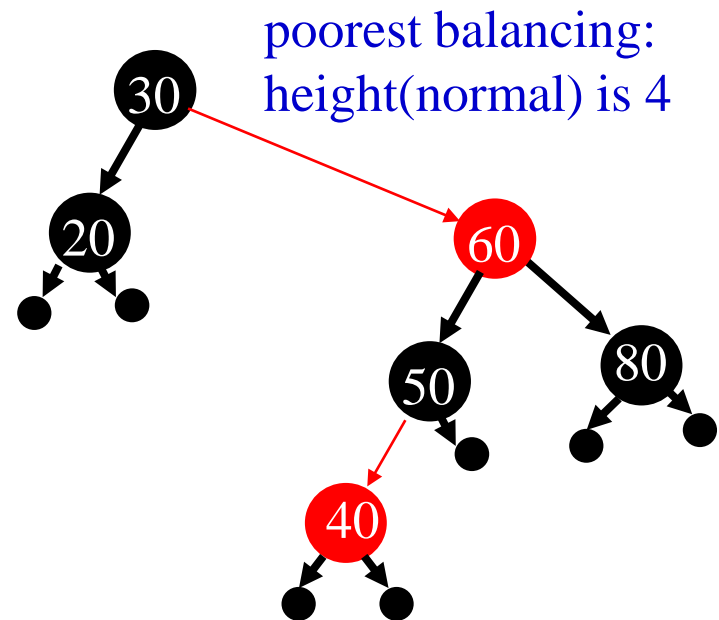
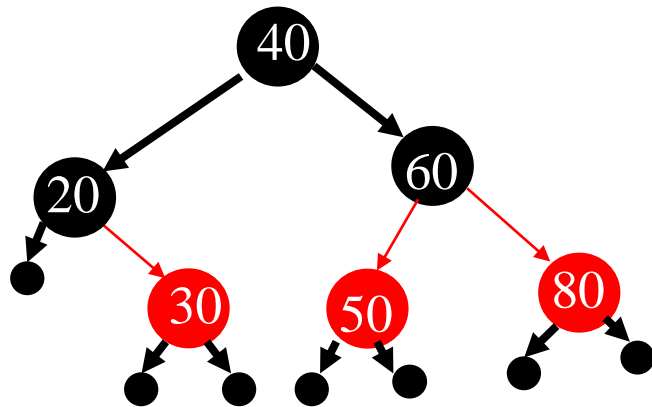


(4)

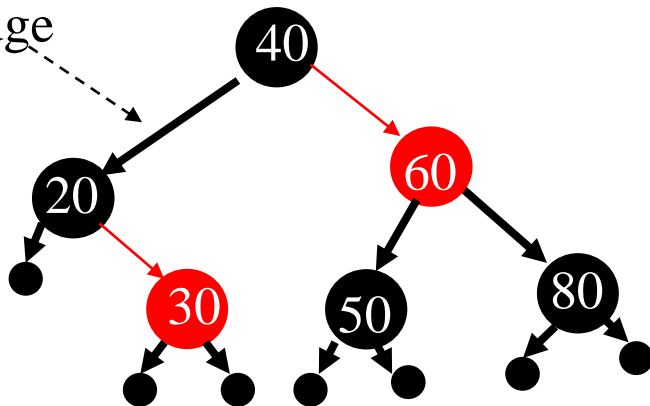


$RB_1$

# Red-Black Tree with 6 Nodes



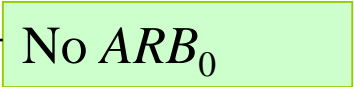
Black edge



# Recursive Definition of RBT

(A red-black tree of black height  $h$  is denoted as  $RB_h$ )

- **Definition:**

- An external node is an  $RB_0$  tree, and the node is black.
- A binary tree is an  $ARB_h$  ( $h \geq 1$ ) tree if:  No  $ARB_0$ 
  - Its root is red, and
  - Its left and right subtrees are each an  $RB_{h-1}$  tree.
- A binary tree is an  $RB_h$  ( $h \geq 1$ ) tree if:
  - Its root is black, and
  - Its left and right subtrees are each either an  $RB_{h-1}$  tree or an  $ARB_h$  tree.

# Well-defined Black Height

- That “the **black height** of any  $RB_h$  tree or  $ARB_h$  tree is well defined” means *the black length of all external paths from the root is the same.*
- Proof: induction on  $h$
- Base case:  $h=0$ , that is  $RB_0$  (there is no  $ARB_0$ )
- In  $ARB_{h+1}$ , its two subtrees are both  $RB_h$ . Since the root is red, the black length of all external paths from the root is  $h$ , that’s the same as its two subtrees.
- In  $RB_{h+1}$ :
  - Case 1: two subtrees are  $RB_h$ ’s
  - Case 2: two subtrees are  $ARB_{h+1}$ ’s
  - Case 3: one subtree is an  $RB_h$  (black height= $h$ ), and the another is an  $ARB_{h+1}$  (black height= $h+1$ )



# Properties of Red-Black Tree

- The **black height** of any  $RB_h$  tree or  $ARB_h$  tree is well-defined and is  $h$ .
- Let  $T$  be an  $RB_h$  tree, then:
  - $T$  has at least  $2^h - 1$  internal black nodes.
  - $T$  has at most  $4^h - 1$  internal nodes.
  - The depth of any black node is at most twice its black depth.
- Let  $A$  be an  $ARB_h$  tree, then:
  - $A$  has at least  $2^{h-2}$  internal black nodes.
  - $A$  has at most  $(4^h)/2 - 1$  internal nodes.
  - The depth of any black node is at most twice its black depth.

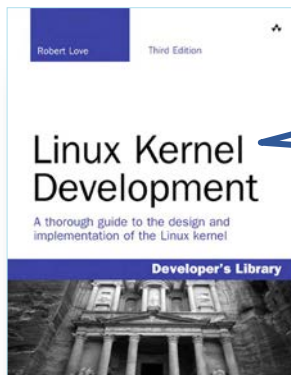




# Bound on Depth of Node in RBTre

- Let  $T$  be a red-black tree with  $n$  internal nodes. Then no node has black depth greater than  $\log(n+1)$ , which means that the height of  $T$  in the usual sense is at most  $2\log(n+1)$ .
  - Proof:
    - Let  $h$  be the black height of  $T$ . The number of internal nodes,  $n$ , is at least the number of internal black nodes, which is at least  $2^h - 1$ , so  $h \leq \log(n+1)$ . The node with greatest depth is some external node. All external nodes are with black depth  $h$ . So, the depth is at most  $2h$ .

# RBT in Practice



CFS uses a **red-black tree** to manage the list of runnable processes and efficiently find the process with the smallest `vruntime`. A red-black tree, called an *rbtree* in Linux, is a type of *self-balancing binary search tree*. We discuss self-balancing binary search trees in general and red-black trees in particular in Chapter 6. For now, if you are unfamiliar, you need to know only that red-black trees are a data structure that store *nodes* of arbitrary data, identified by a specific *key*, and that they enable efficient search for a given *key*. (Specifically, obtaining a node identified by a given key is logarithmic in time as a function of total nodes in the tree.)

```
C++ Containers library std::set

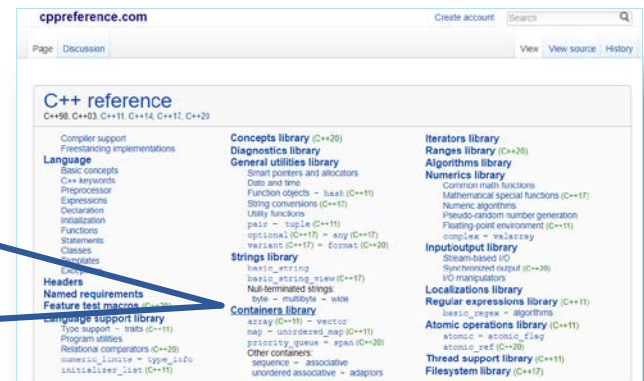
std::set

Defined in header <set>

template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class set;

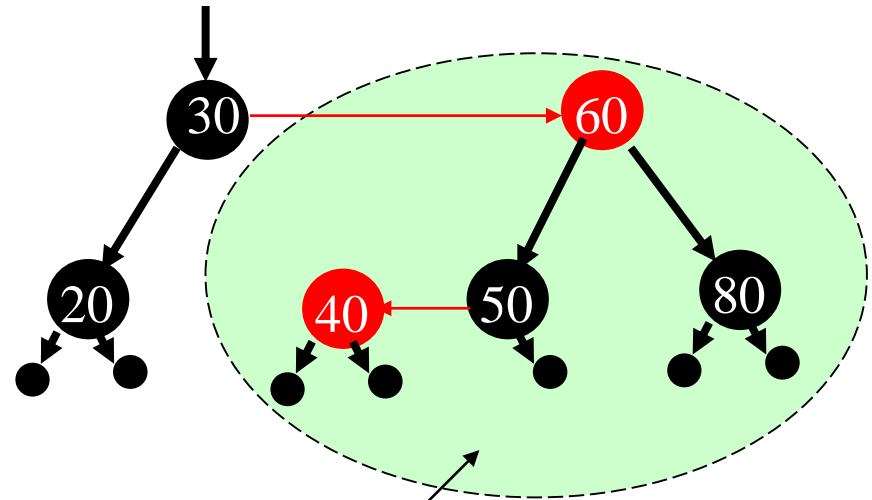
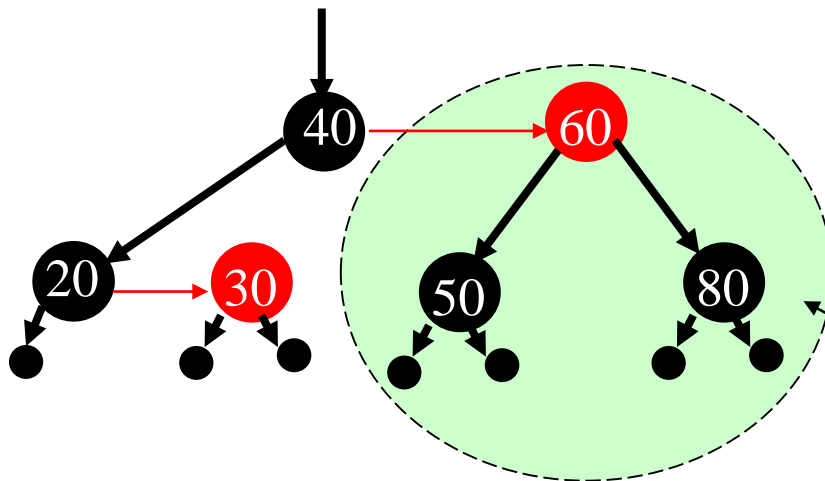
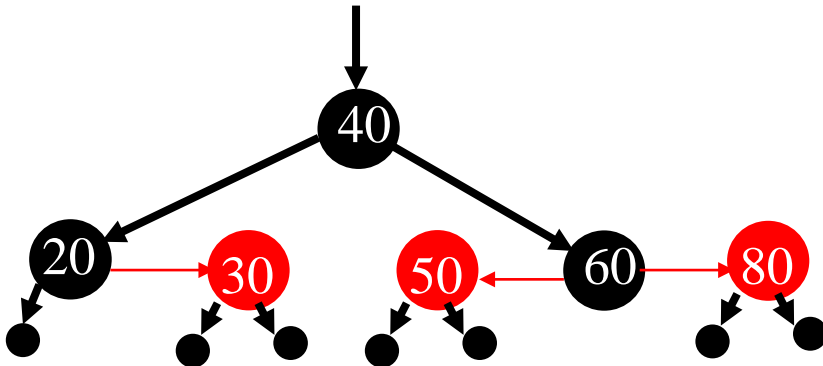
namespace pmr {
    template <class Key, class Compare = std::less<Key>>
        using set = std::set<Key, Compare, std::pmr::polymorphic_allocator<Key>>;
}

std::set is an associative container that contains a sorted set of unique objects of type Key. Sorting is done using the key comparison function Compare. Search, removal, and insertion operations have logarithmic complexity. Sets are usually implemented as red-black trees.
```



# Black-depth Convention

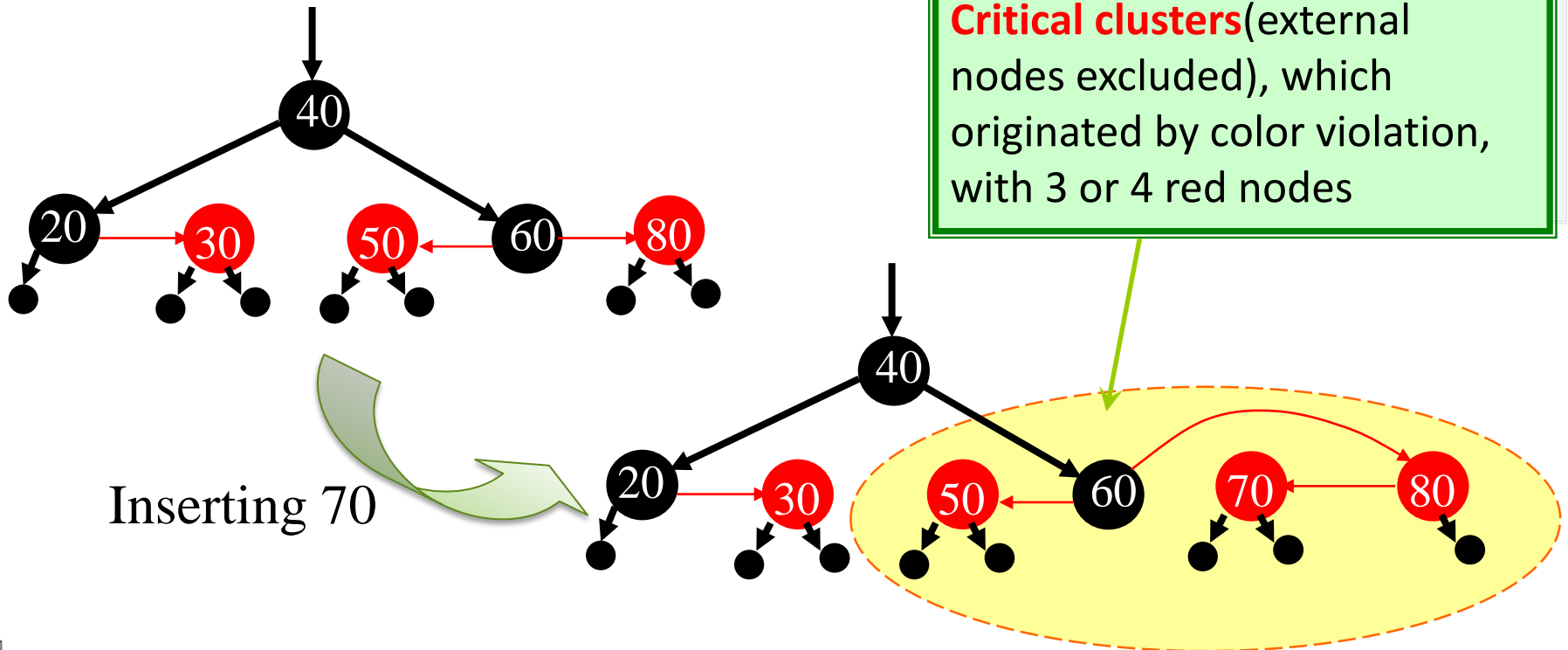
All with the same  
largest black depth: 2



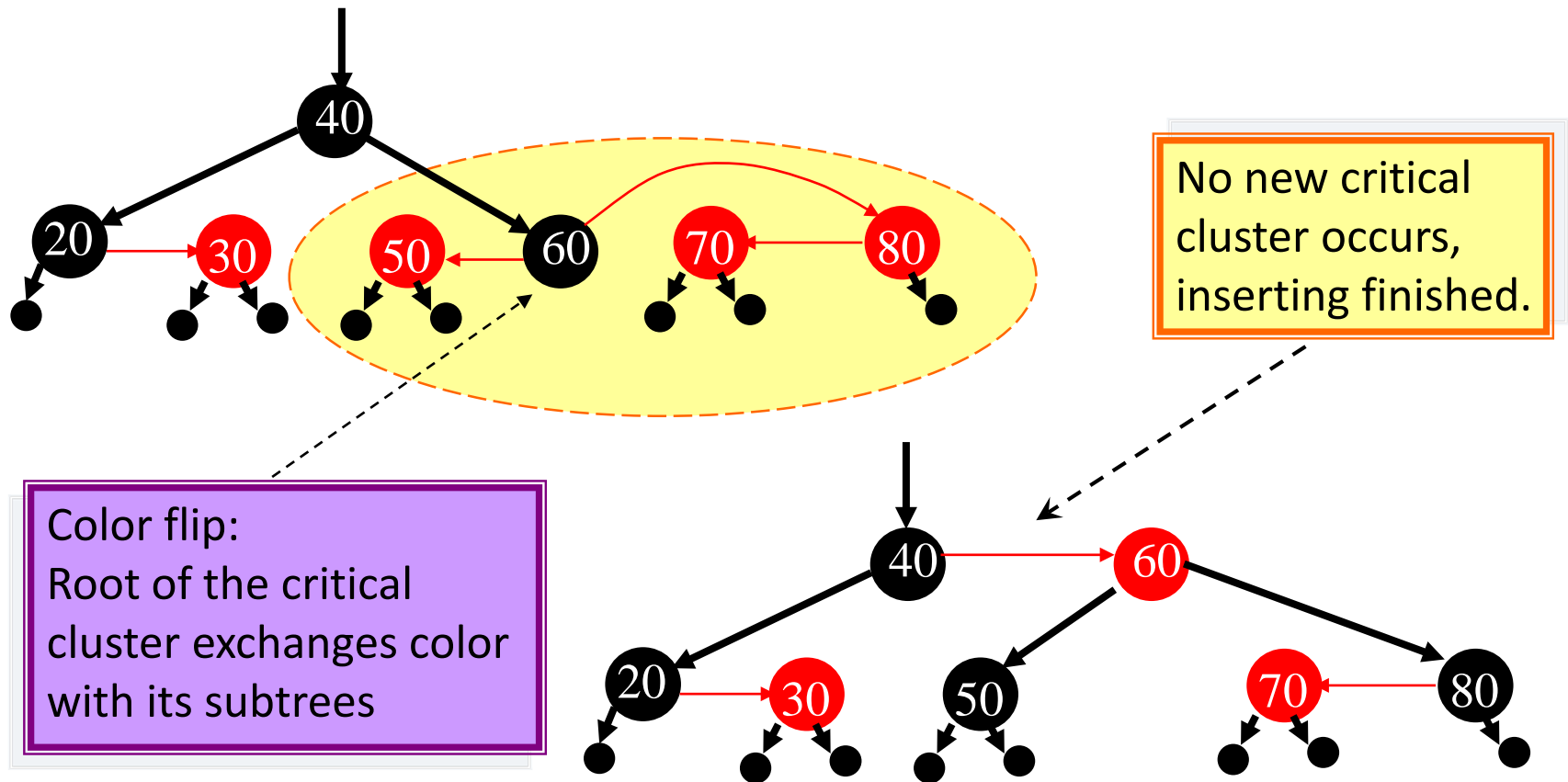
ARB Trees

# Influences of Insertion to an RBT

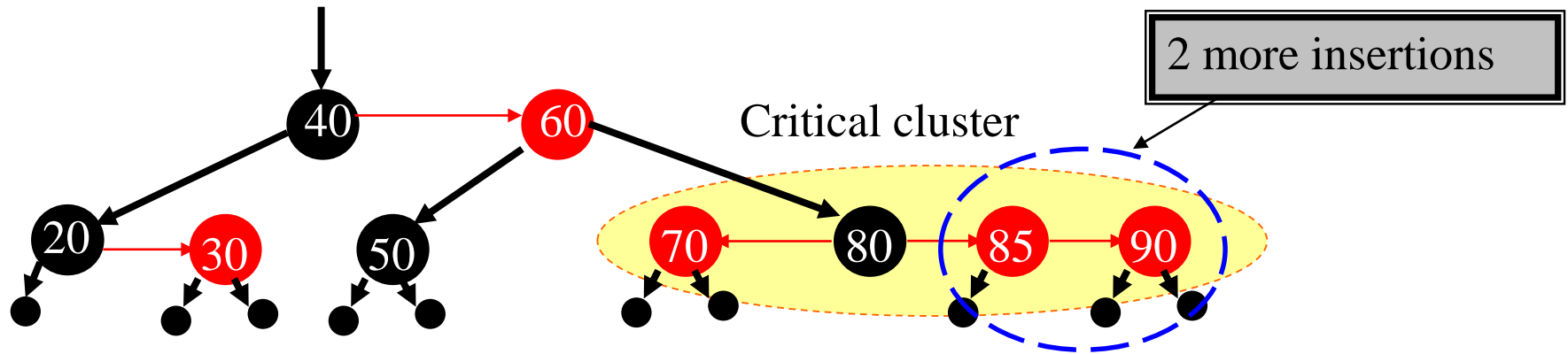
- Black height constraint:
  - No violation *if* inserting a red node.
- Color constraint:



# Repairing 4-node Critical Cluster

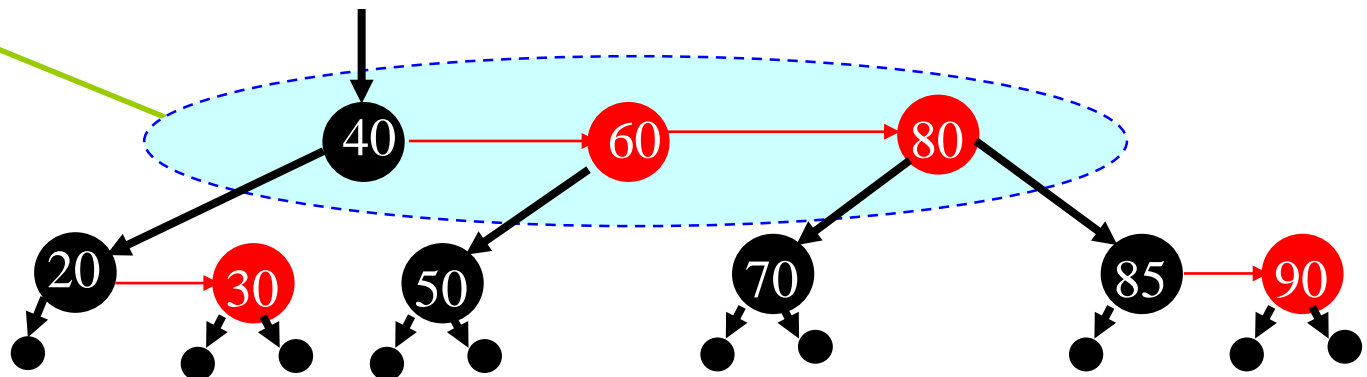


# Repairing 4-node Critical Cluster

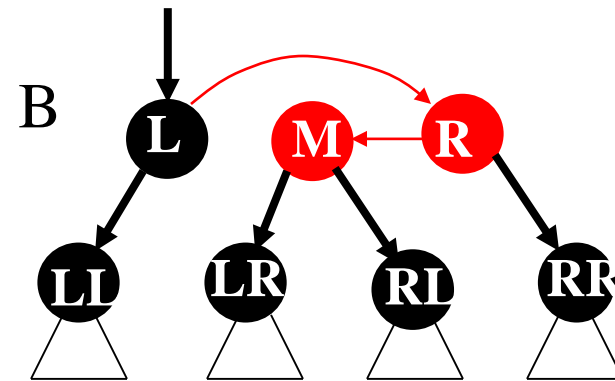
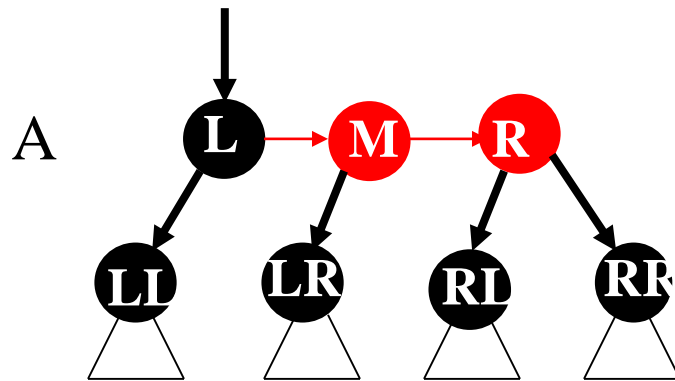


New critical cluster with 3 nodes.

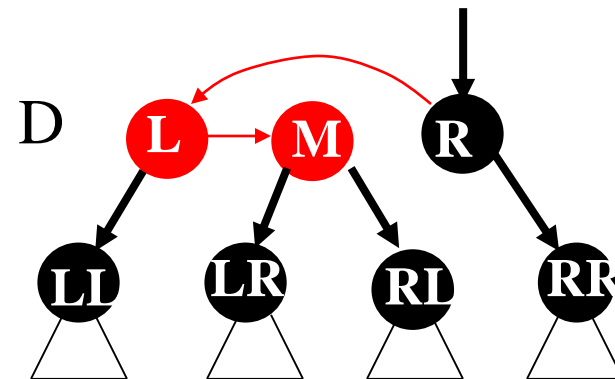
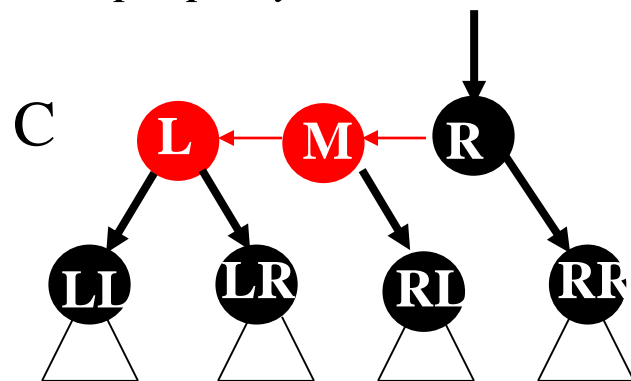
Color flip doesn't work, **Why?**



# Patterns of 3-node Critical Cluster

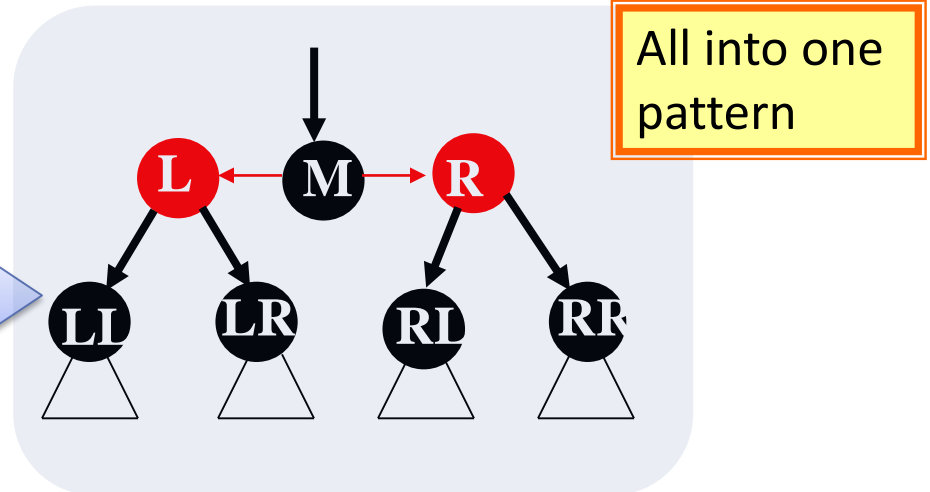


*Shown as properly drawn*

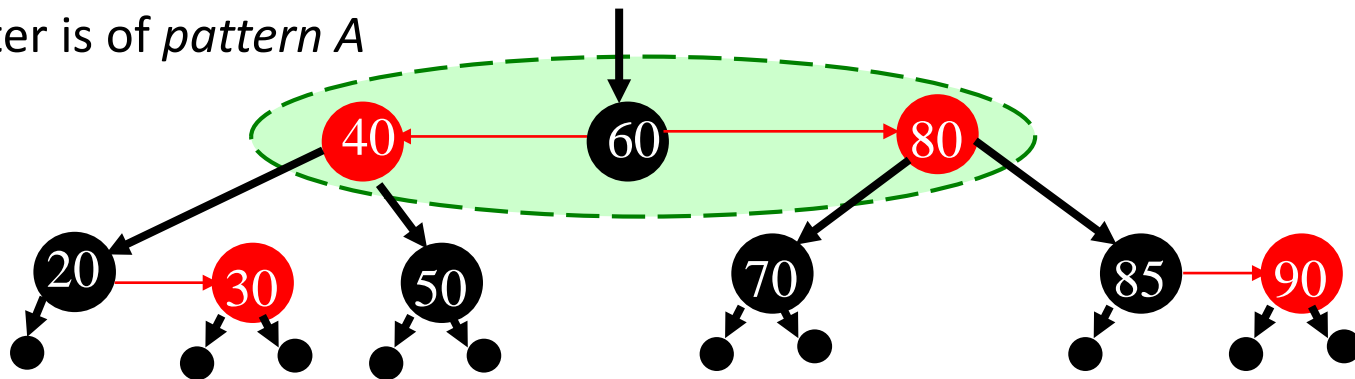


# Repairing 3-Node Critical Cluster

Root of the critical cluster is changed to **M**, and the parentship is adjusted accordingly



The incurred critical cluster is of *pattern A*





# Implementing Insertion: Class

```
class RBtree
```

```
    Element root;  
    RBtree leftSubtree;  
    RBtree rightSubtree;  
    int color; /* red, black */
```

```
    static class InsReturn
```

```
        public RBtree newTree;  
        public int status /* ok, rbr, brb, rrb, brr */
```

*Color pattern*



# Implementing Insertion: Procedure

```
RBtree rbtInsert (RBtree oldRBtree, Element newNode)
```

```
  InsReturn ans, ansLeft, ansRight; (LDA: Node)
```

```
  If (ans.newTr
```

```
    ans.newTr
```

```
  return ans.ne
```

*the wrapper*

```
  InsReturn rbtIns(RBtree oldRBtree, Element newNode)
```

```
    InsReturn ans, ansLeft, ansRight;
```

```
    if (oldRBtree = nil) then <Inserting simply>;
```

```
    else
```

```
      if (newNode.key < oldRBtree.root.key)
```

```
        ansLeft = rbtIns (oldRBtree.leftSubtree, newNode);
```

```
        ans = repairLeft(oldRBtree, ansLeft);
```

```
      else
```

```
        ansRight = rbtIns(oldRBtree.rightSubtree, newNode);
```

```
        ans = repairRight(oldRBtree, ansRight);
```

```
  return ans
```

*the recursive function*



# Correctness of Insertion

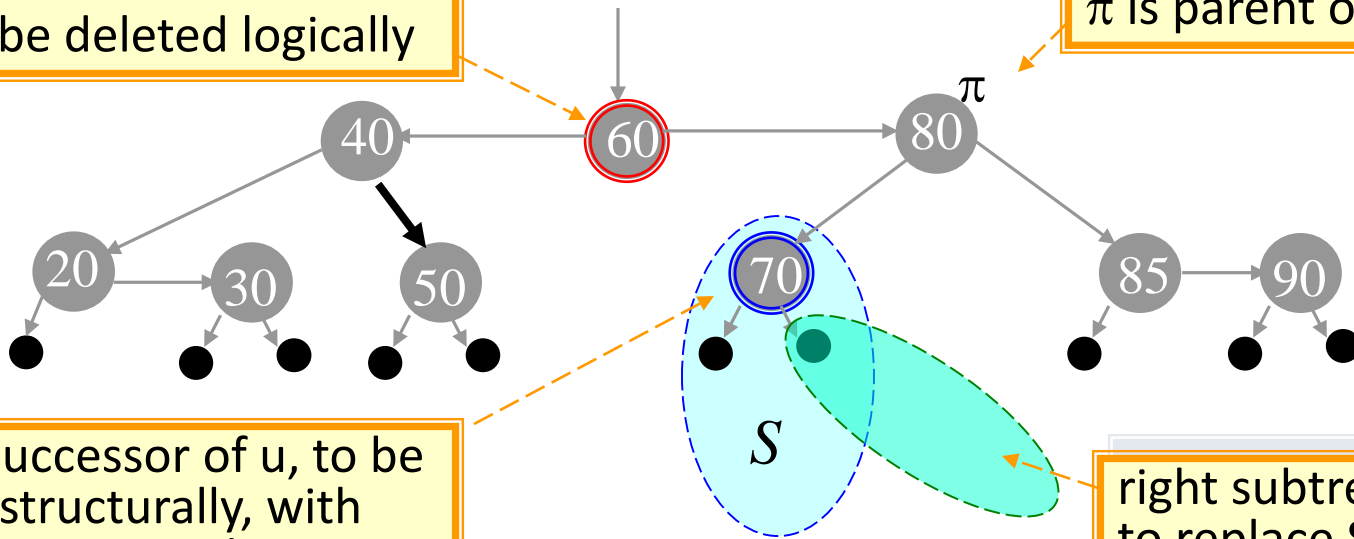
- If the parameter `oldRBtree` of `rbtIns` is an  $RB_h$  tree or an  $ARB_{h+1}$  tree (which is true for the recursive calls on `rbtIns`), then the `newTree` and `status` fields returned are one of the following combinations:
  - `Status=ok`, and `newTree` is an  $RB_h$  or an  $ARB_{h+1}$  tree,
  - `Status=rbr`, and `newTree` is an  $RB_h$ ,
  - `Status=brb`, and `newTree` is an  $ARB_{h+1}$  tree,
  - `Status=rrb`, and `newTree.color=red`, `newTree.leftSubtree` is an  $ARB_{h+1}$  tree and `newTree.rightSubtree` is an  $RB_h$  tree,
  - `Status=brr`, and `newTree.color=red`, `newTree.rightSubtree` is an  $ARB_{h+1}$  tree and `newTree.leftSubtree` is an  $RB_h$  tree
- For those cases with red root, the color will be changed to black, with other constraints satisfied by repairing subroutines.



# Deletion: Logical and Structural

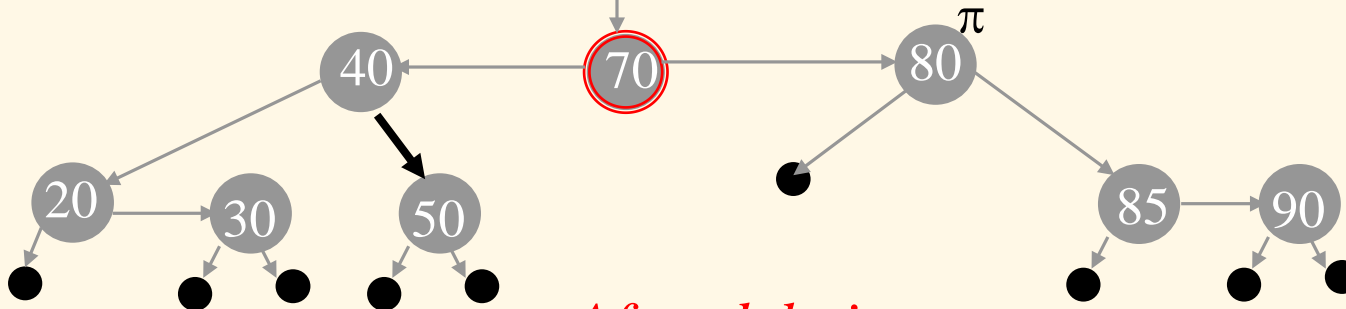
$u$ : to be deleted logically

$\pi$  is parent of  $\sigma$



$\sigma$ : tree successor of  $u$ , to be deleted structurally, with information moved into  $u$

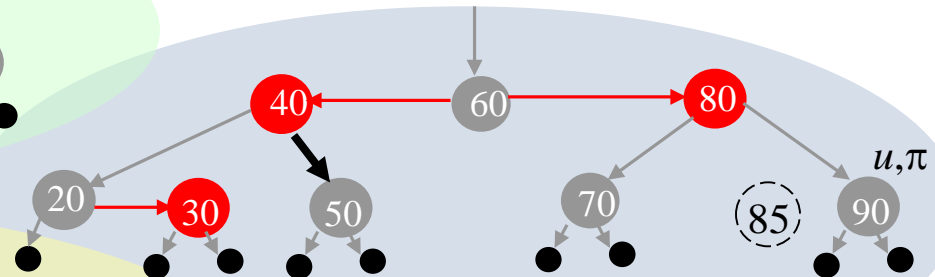
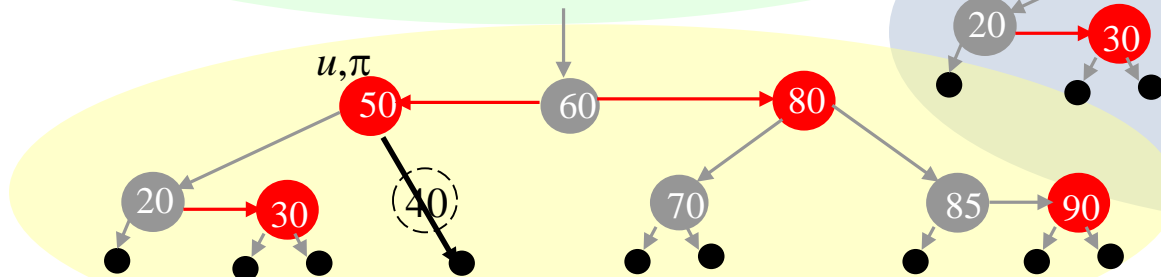
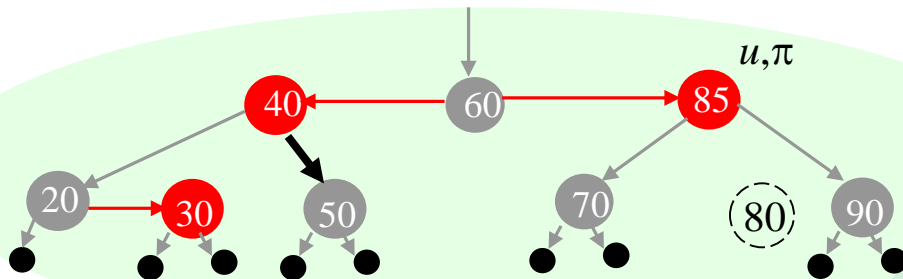
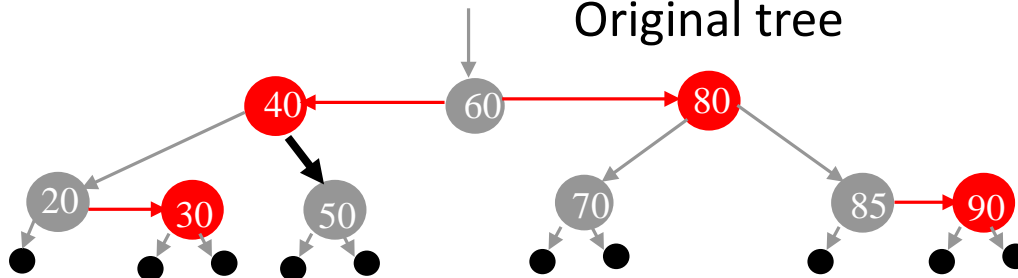
right subtree of  $S$ , to replace  $S$



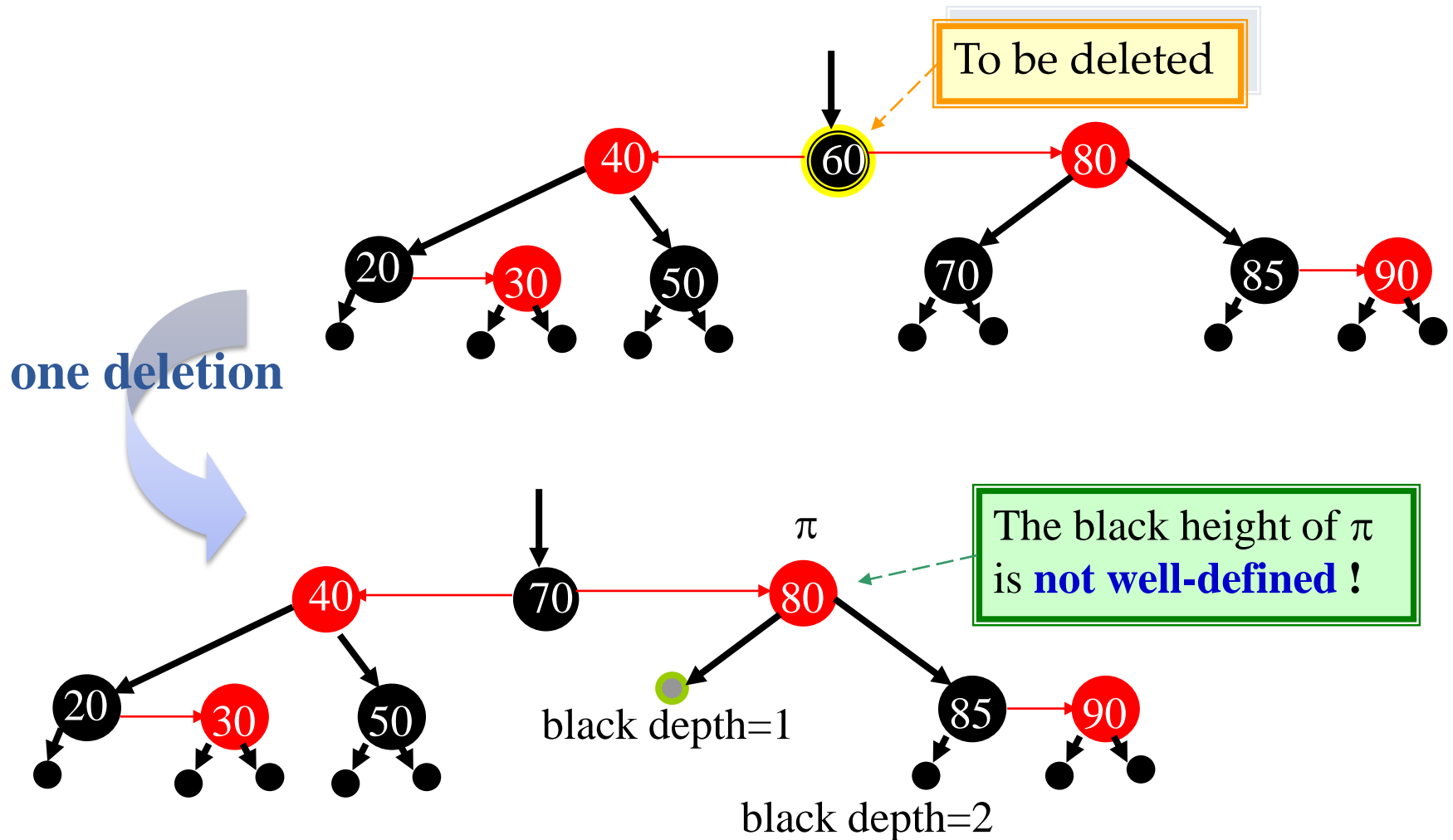
*After deletion*

# Deletion from RBT - Examples

Original tree



# Deletion in RBT

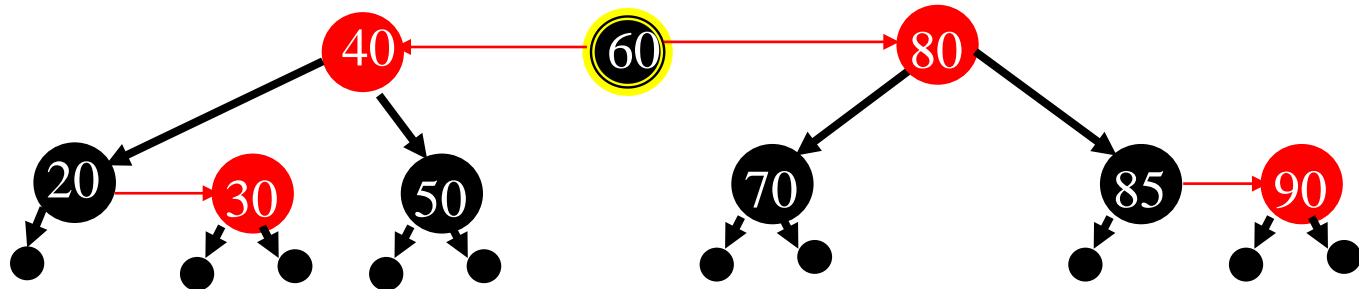


# Procedure of Red-Black Deletion

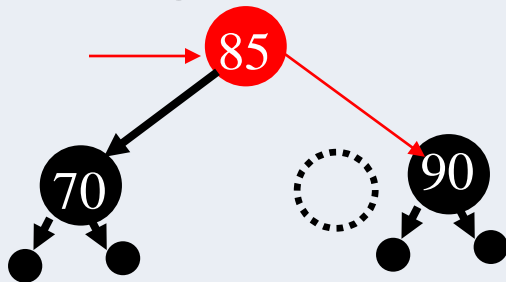
1. Do a standard BST search to locate the node to be logically deleted, call it  $u$
2. If the right child of  $u$  is an external node, identify  $u$  as the node to be structurally deleted.
3. If the right child of  $u$  is an internal node, find the tree successor of  $u$ , call it  $\sigma$ , copy the key and information from  $\sigma$  to  $u$ . (color of  $u$  not changed) Identify  $\sigma$  as the node to be deleted structurally.
4. Carry out the structural deletion and repair any imbalance of black height.



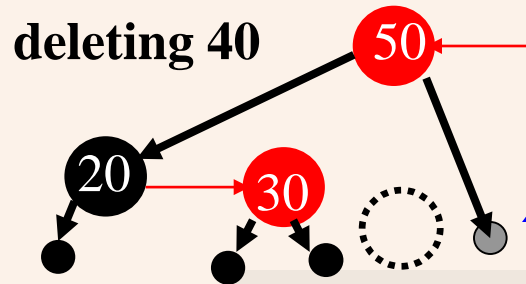
# Imbalance of Black Height



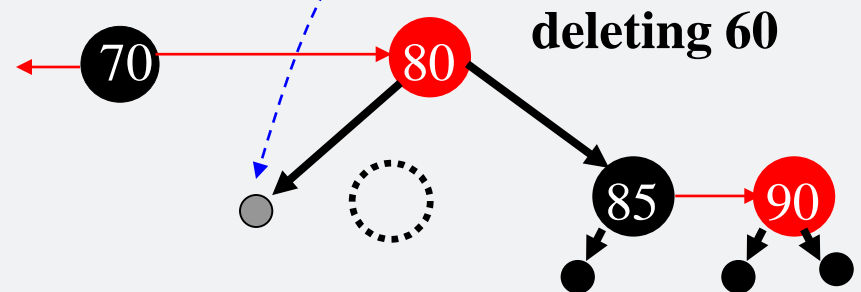
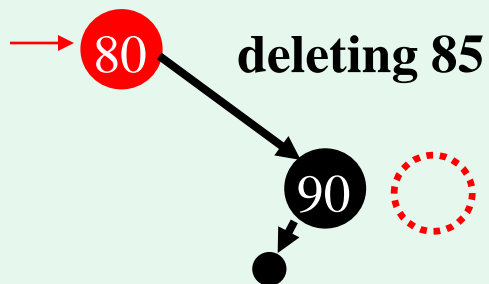
deleting 80



deleting 40



Black height has to be restored



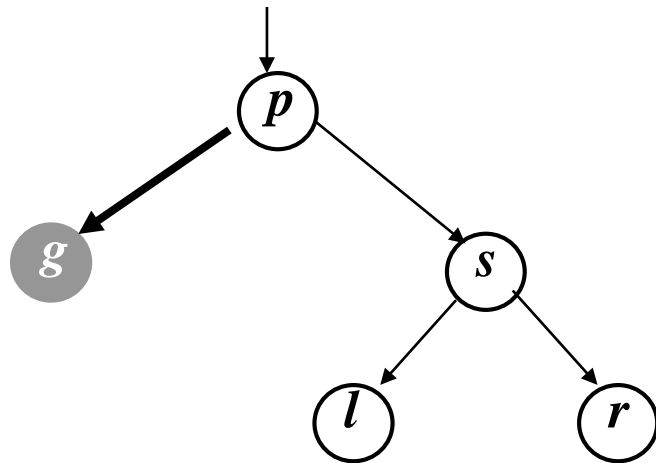


# Analysis of Black Imbalance

- **The imbalance occurs when:**
  - A black node is deleted structurally, and
  - Its right subtree is black (external)
- **The result is:**
  - An  $RB_{h-1}$  occupies the position of an  $RB_h$  as required by its parent, coloring it as a “gray” node.
- **Solution:**
  - Find a red node and turn it black as locally as possible.
  - The gray color might propagate up the tree.

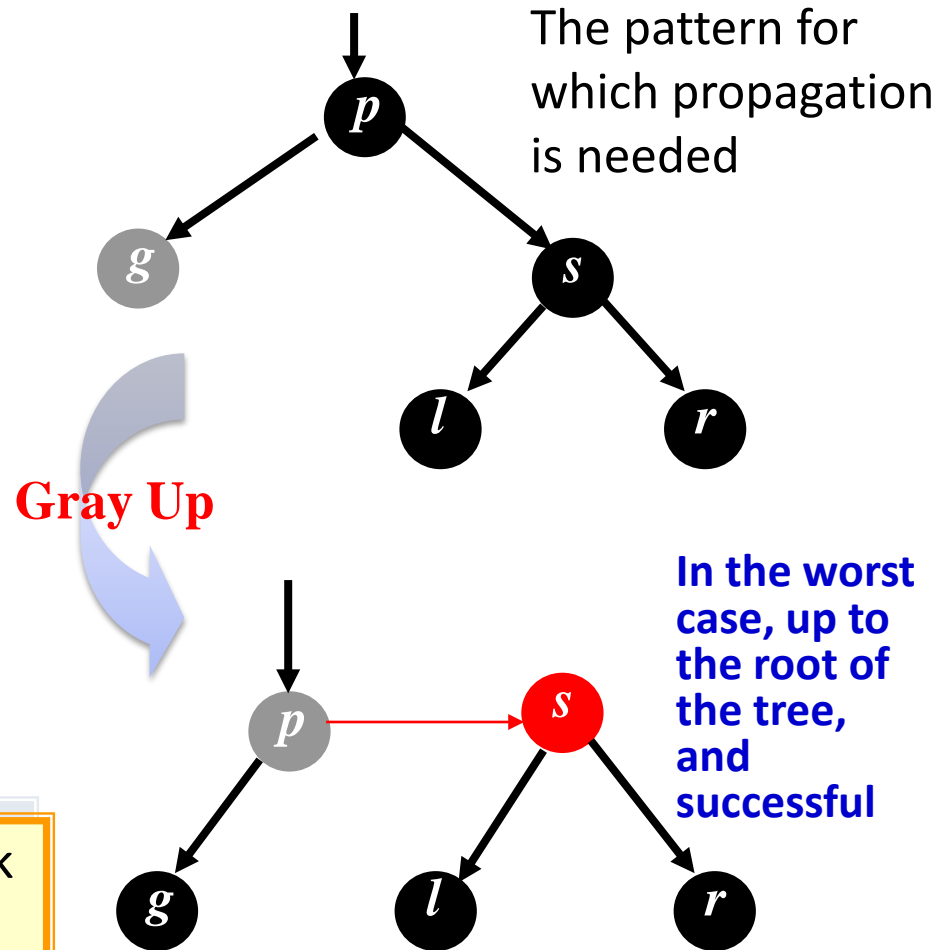


# Propagation of Gray Node

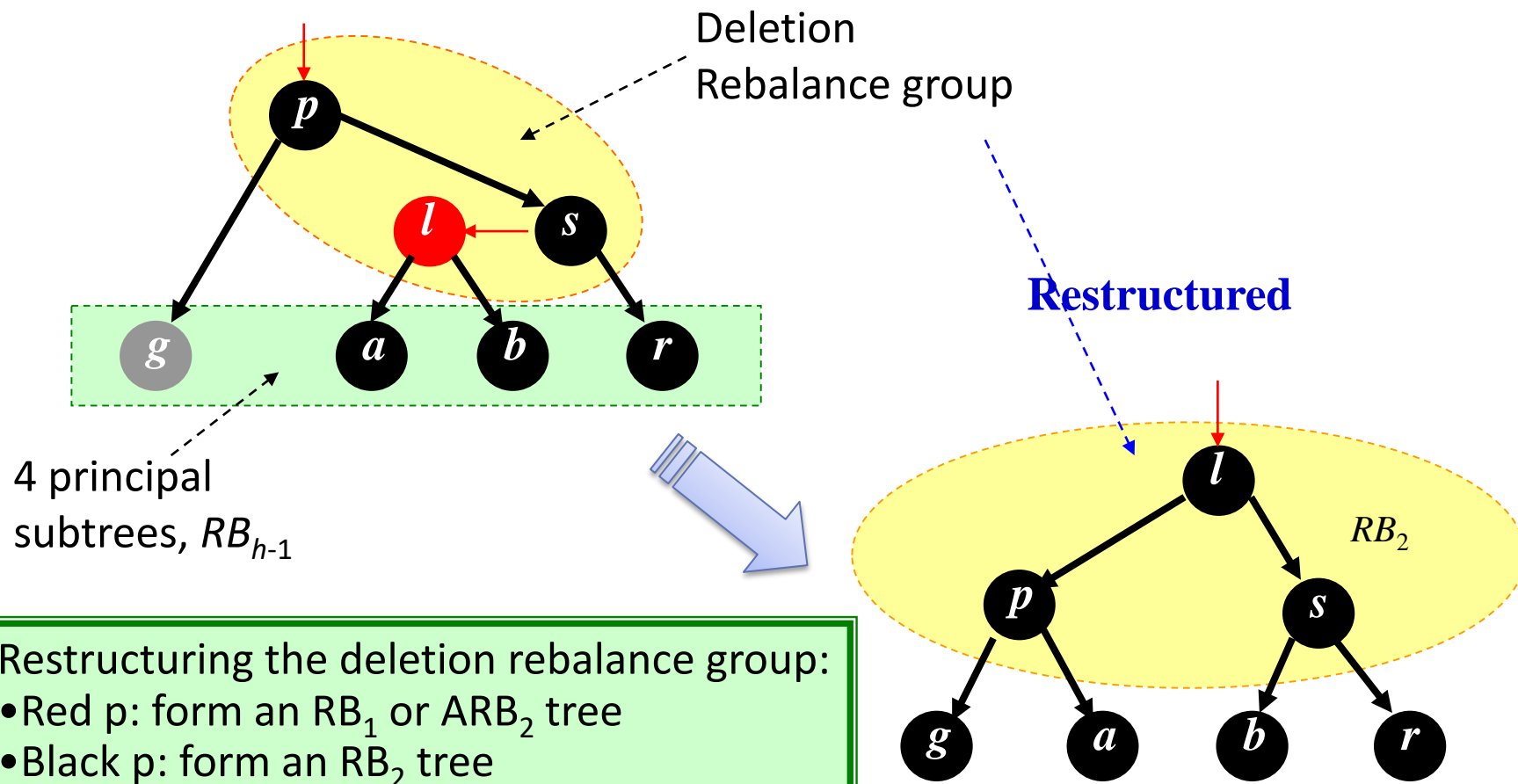


Map of the vicinity of  $g$ , the gray node

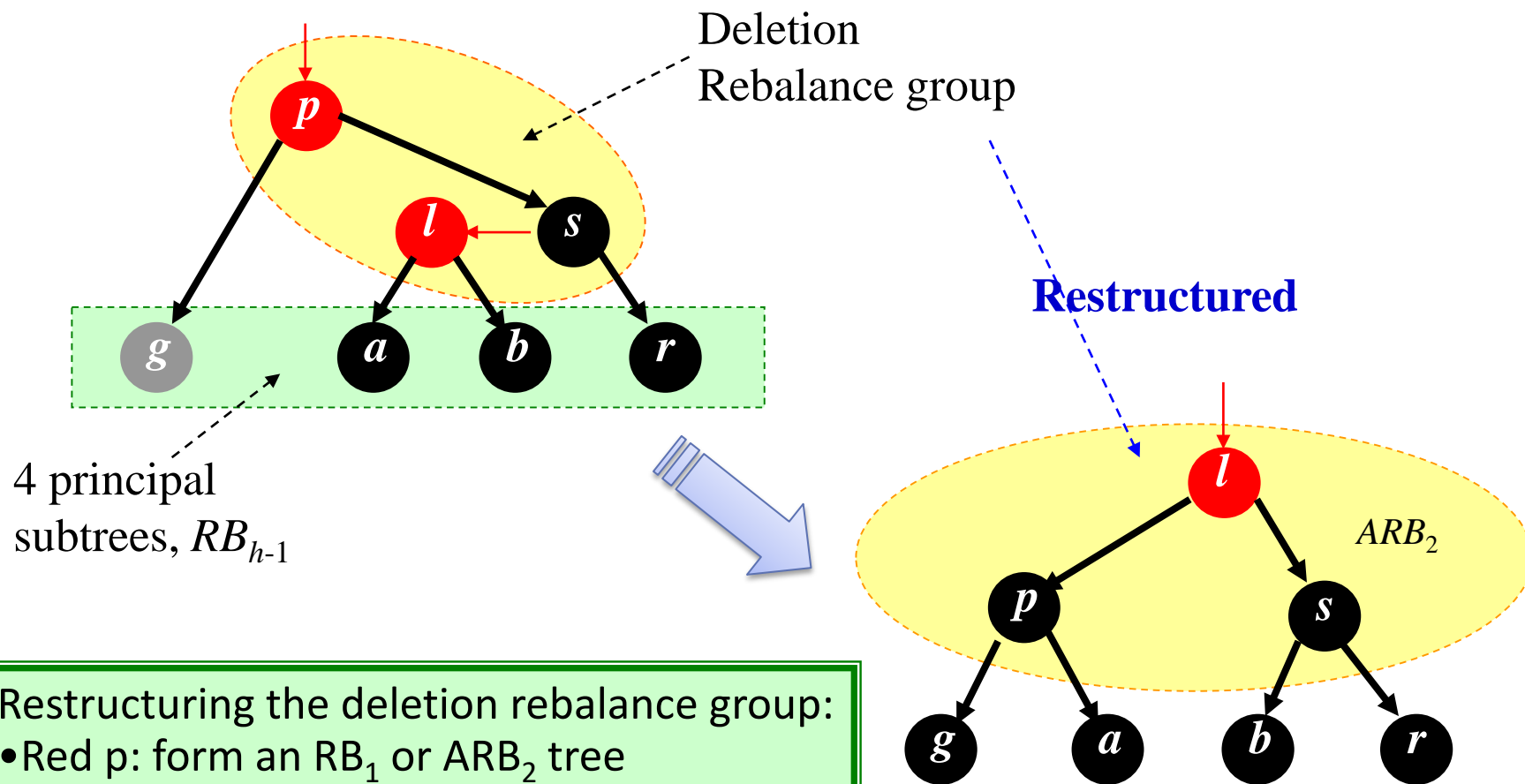
$g$ -subtree gets well-defined black height, but that is less than that required by its parent



# Repairing without Propagation



# Repairing without Propagation



Restructuring the deletion rebalance group:

- Red p: form an  $RB_1$  or  $ARB_2$  tree
- Black p: form an  $RB_2$  tree

# Complexity of Operations on RBT

- **With reasonable implementation**
  - A new node can be inserted correctly in a red-black tree with  $n$  nodes in  $\Theta(\log n)$  time in the worst case.
  - Repairs for deletion do  $O(1)$  structural changes, but may do  $O(\log n)$  color changes.

*Thank you!*

*Q & A*

*Yu Huang*

<http://cs.nju.edu.cn/yuhuang>

