

# Analysis of Algorithms 1 (Fall 2011) Istanbul Technical University Computer Eng. Dept.



## Chapter 13 Red-Black Trees

Last updated: December 03, 2009

# Purpose

Review Binary Search Trees

Introduce Red-Black Trees

Review 2-3 and 2-3-4 trees

Rotations and other operations in RB  
Trees

# Outline

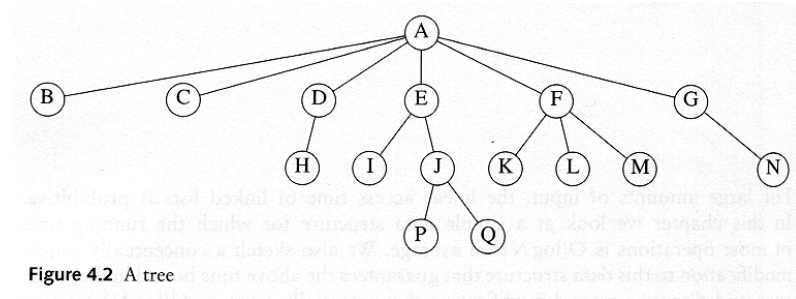
Binary Search Tree (BST) review

Red and Black Trees

2-3 and 2-3-4 trees

Operations on Red and Black Trees

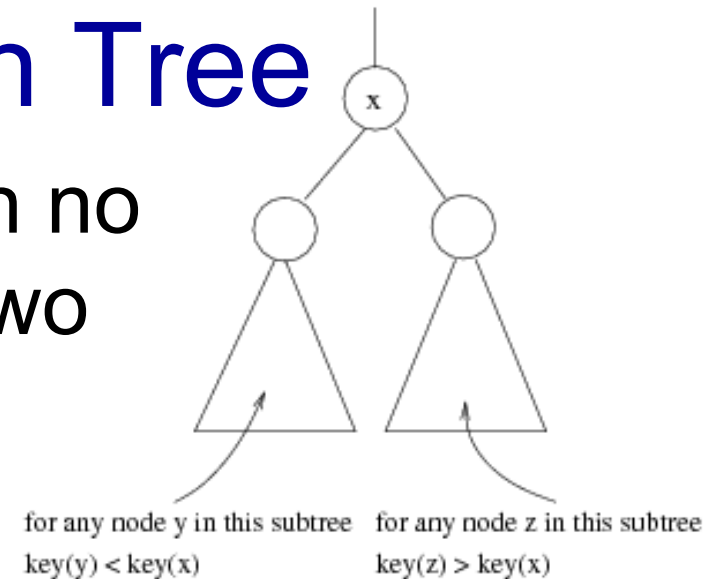
# Tree



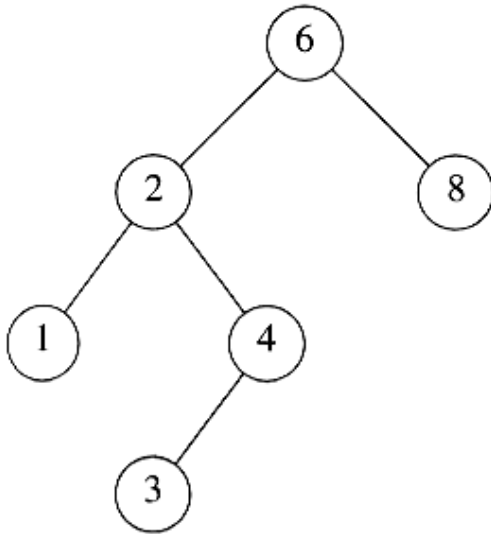
- **Child and parent**
  - Every node except the root has one parent
  - A node can have an arbitrary number of children
- **Leaves:** Nodes with no children
- **Sibling:** nodes with same parent
- **Path Length:** number of edges on the path
- **Depth of a node:** length of the unique path from the root to that node. The depth of a tree is equal to the depth of the deepest leaf
- **Height of a node:** length of the longest path from that node to a leaf, all leaves are at height 0, The height of a tree is equal to the height of the root
- **Ancestor and descendant**

# Binary Search Tree

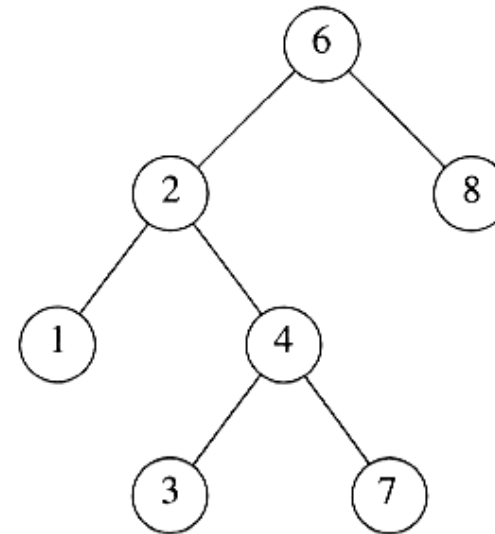
- **Binary tree:** A tree in which no node can have more than two children
- **Binary search tree:**
  - Stores keys in the nodes in a way so that searching, insertion and deletion can be done efficiently.
  - For every node  $X$ , all the keys in its left subtree are smaller than the key value in  $X$ , and all the keys in its right subtree are larger than the key value in  $X$



# Binary Search Tree

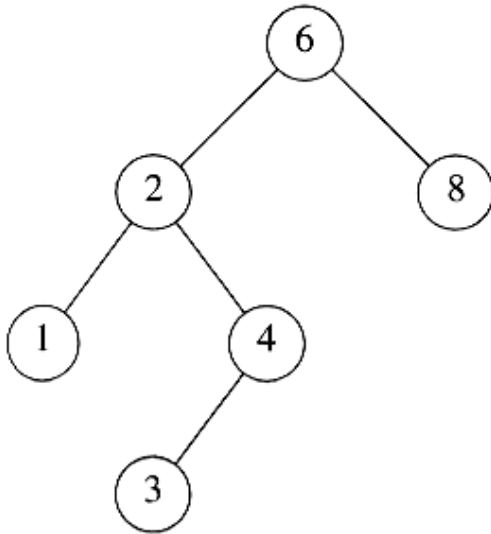


A binary search tree

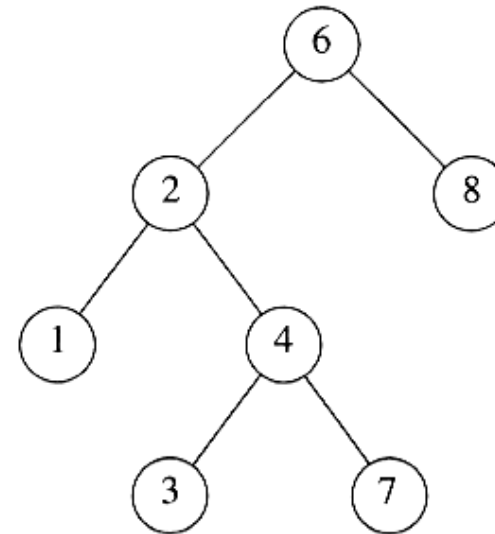


Not a binary search tree  
WHY?

# Binary Search Tree



A binary search tree



Not a binary search tree  
Because  $7 > 6$

# Binary Search Tree Operations

- **Tree Traversal:** Used to print out the data in a tree in a certain order
- **Pre-order traversal (node-left-right)**
  - Print the data at the root
  - Recursively print out all data in the left subtree
  - Recursively print out all data in the right subtree
- See also, **post-order: left-right-node** and **in-order:left-node-right** traversal.



# Binary-search-tree sort

$T \leftarrow \emptyset$  ▷ Create an empty BST

**for**  $i = 1$  to  $n$

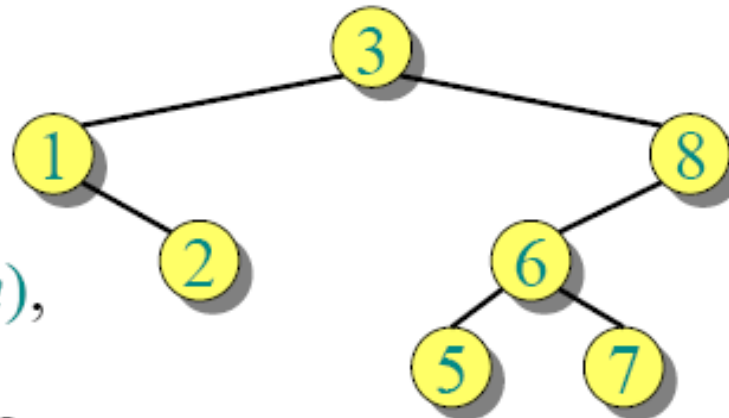
**do** TREE-INSERT ( $T, A[i]$ )

Perform an inorder tree walk (traversal) of  $T$ .

**Example:**

$A = [3 \ 1 \ 8 \ 2 \ 6 \ 7 \ 5]$

Tree-walk time =  $O(n)$ ,  
but how long does it  
take to build the BST?



# Node depth

The depth of a node = the number of comparisons made during TREE-INSERT. Assuming all input permutations are equally likely, we have

Average node depth

$$\begin{aligned} &= \frac{1}{n} E \left[ \sum_{i=1}^n (\# \text{ comparisons to insert node } i) \right] \\ &= \frac{1}{n} O(n \lg n) \quad (\text{quicksort analysis}) \\ &= O(\lg n) . \end{aligned}$$

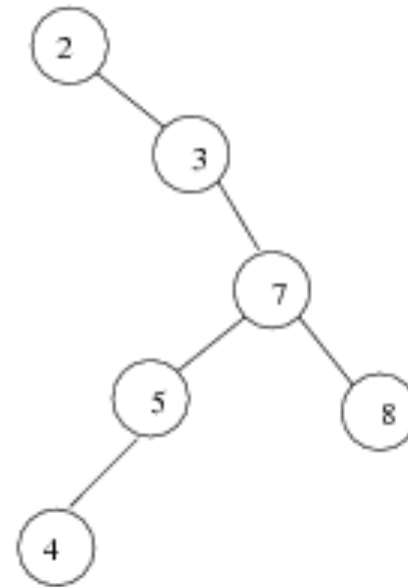
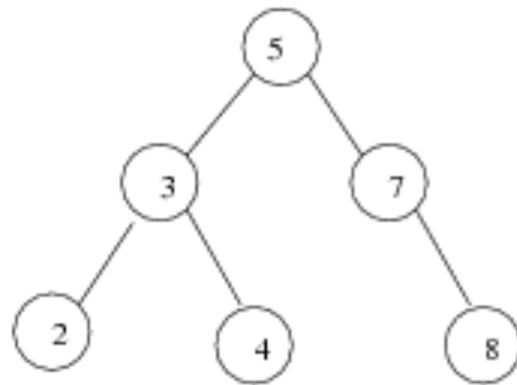
# Height of a randomly built binary search tree

## Outline of the analysis:

- Use *Jensen's inequality*, which says that  $f(E[X]) \leq E[f(X)]$  for any convex function  $f$  and random variable  $X$ .
- Analyze the *exponential height* of a randomly built BST on  $n$  nodes, which is the random variable  $Y_n = 2^{X_n}$ , where  $X_n$  is the random variable denoting the height of the BST.
- Prove that  $2^{E[X_n]} \leq E[2^{X_n}] = E[Y_n] = O(n^3)$ , and hence that  $E[X_n] = O(\lg n)$ .

# Binary Search Tree Operations

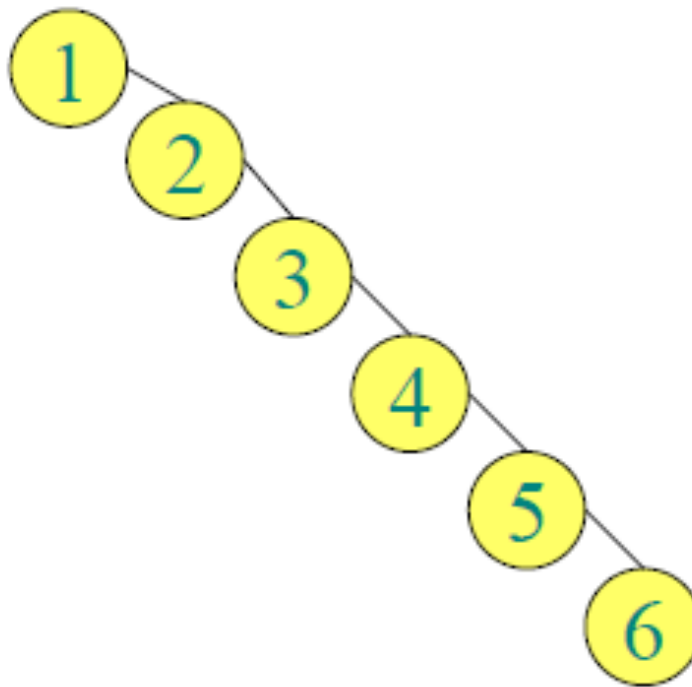
- Linear access time of linked lists is prohibitive
- Does there exist any simple data structure for which the running time of most operations (search, insert, delete) is  $O(\log N)$ ?



Two binary search trees representing the same set.

Average depth of a node is  $O(\log N)$ ;  
maximum depth of a node is  $O(N)$

**Balanced search trees**, or how to avoid **this** even in the worst case



# Balanced search tree

**Balanced search tree:** A search-tree data structure for which a height of  $O(\lg n)$  is guaranteed when implementing a dynamic set of  $n$  items.

## Examples:

- AVL trees
- 2-3 trees
- 2-3-4 trees
- B-trees
- Red-black trees

# Red-black trees

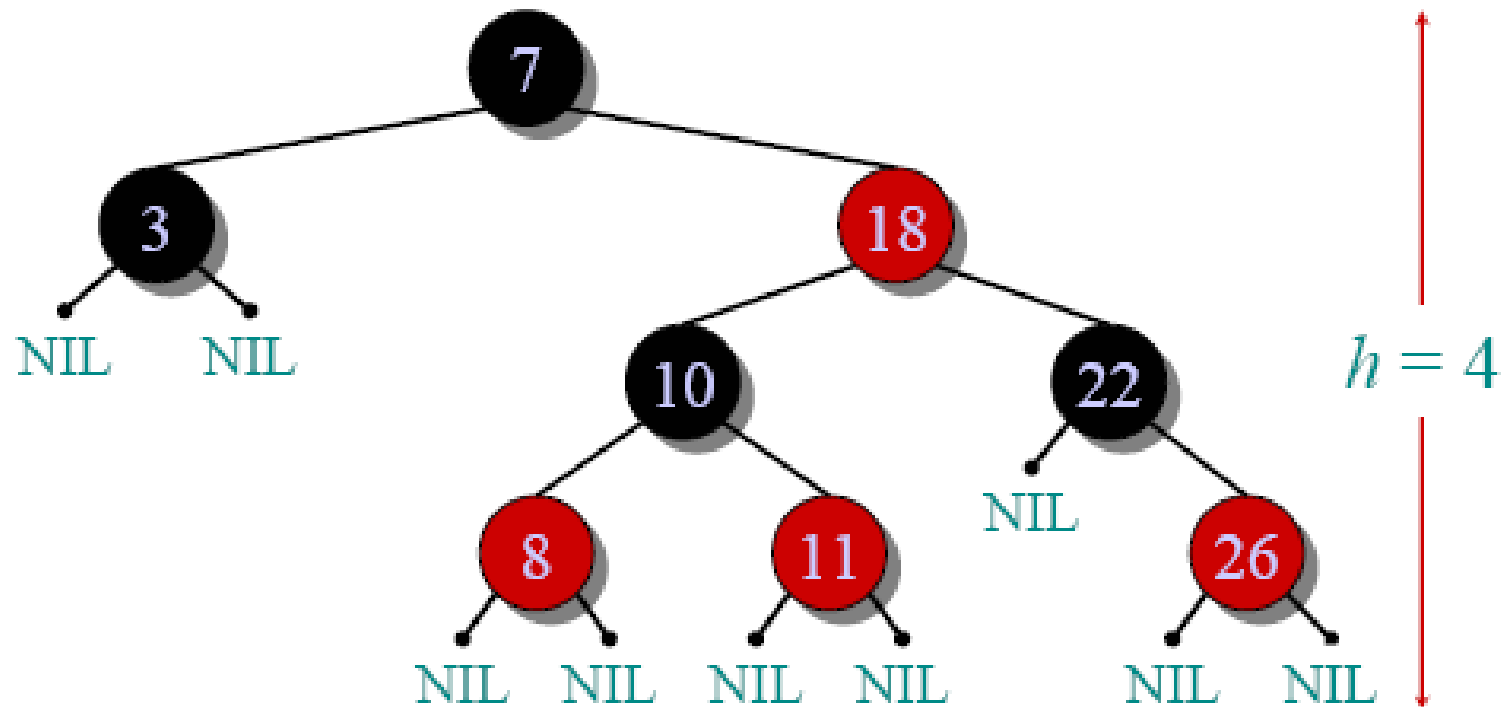
BSTs (Binary Search Tree) with an extra one-bit color field in each node.

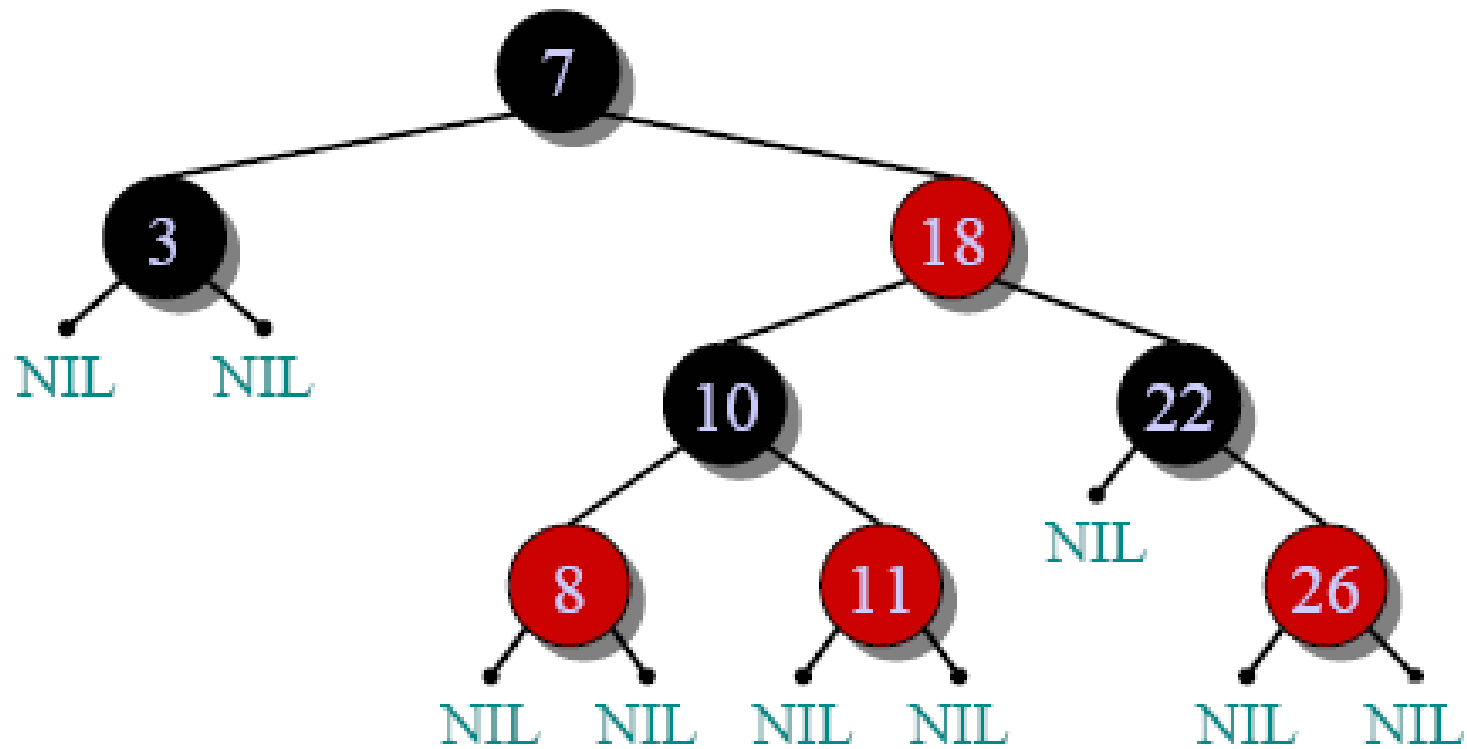
## *Red-black properties:*

1. Every node is either red or black.
2. The root and leaves (NIL's) are black.
3. If a node is red, then its parent is black.
4. All simple paths from any node  $x$  to a descendant leaf have the same number of black nodes =  $\text{black-height}(x)$ .

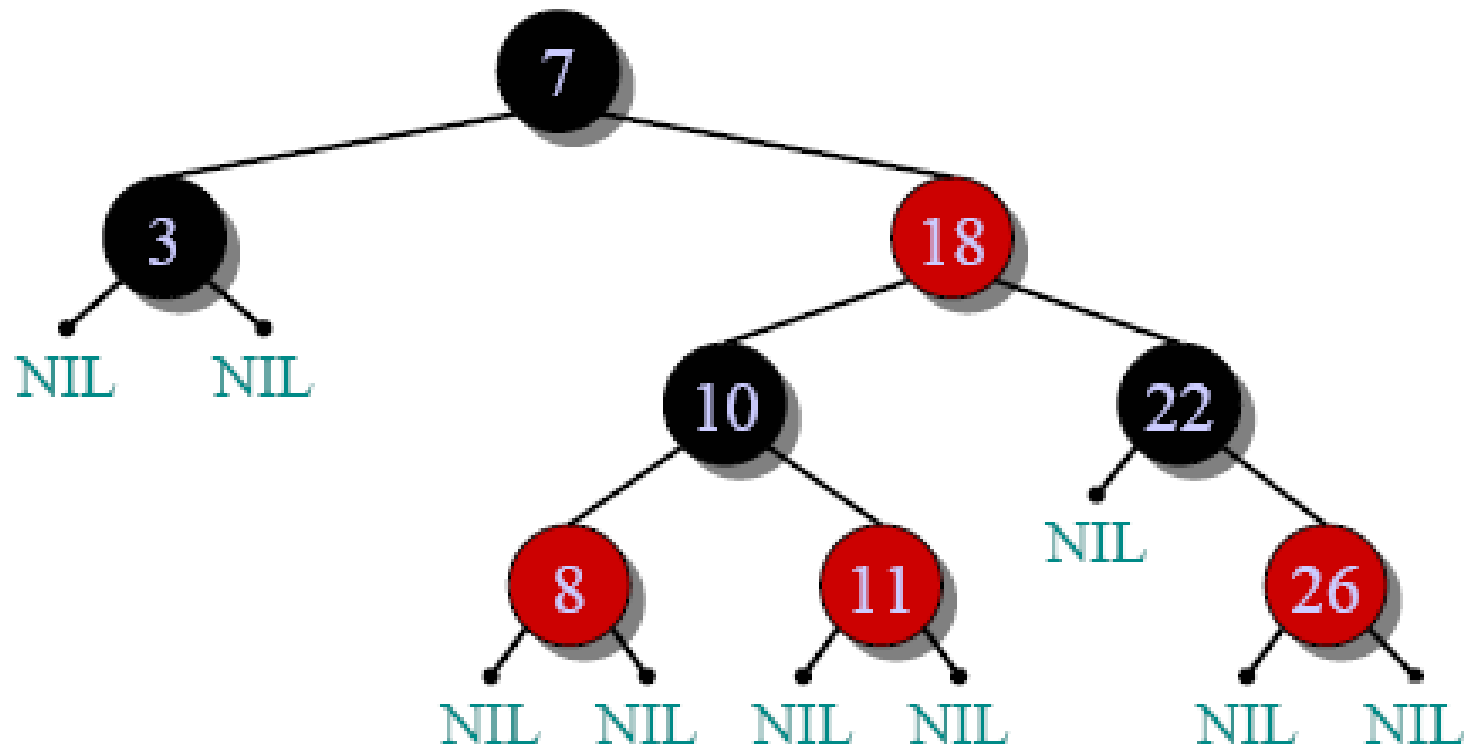


# Red-black tree example

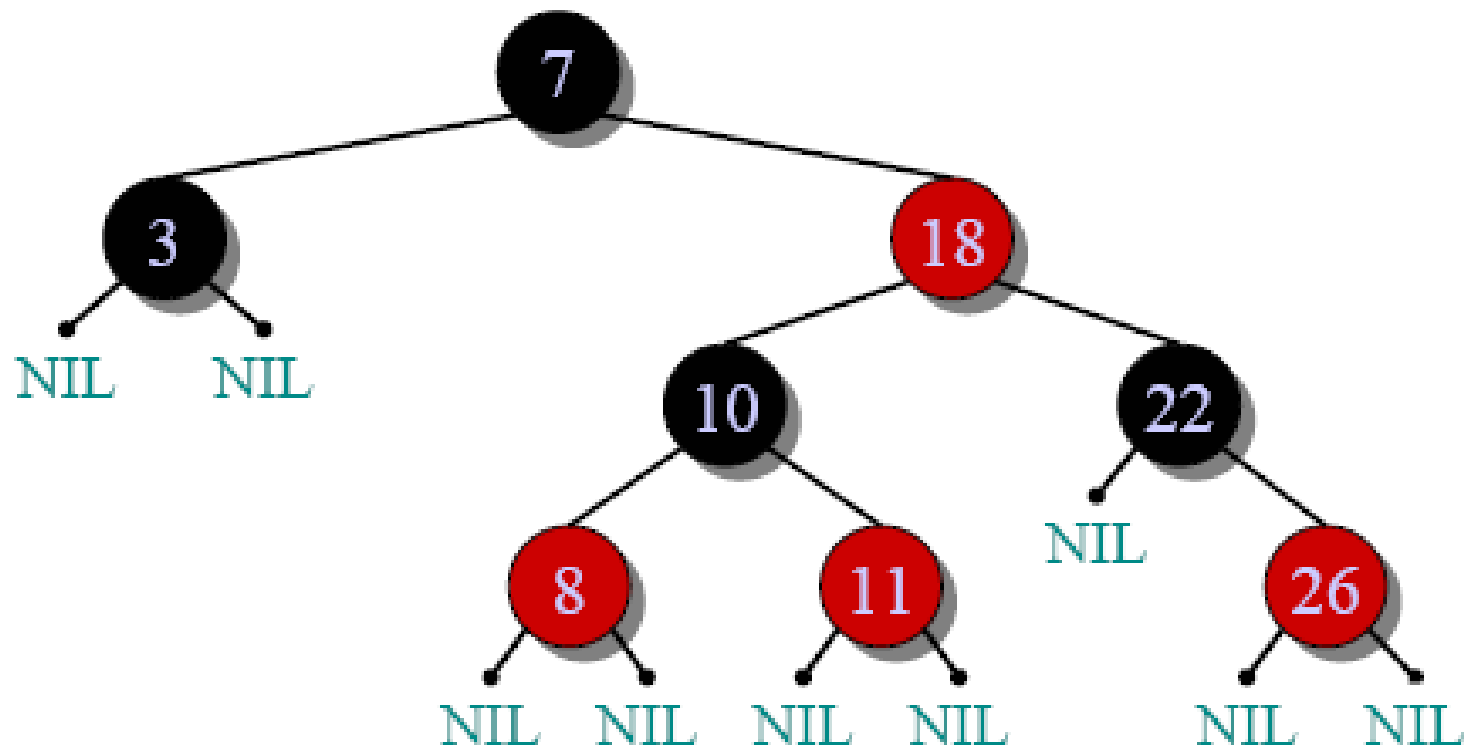




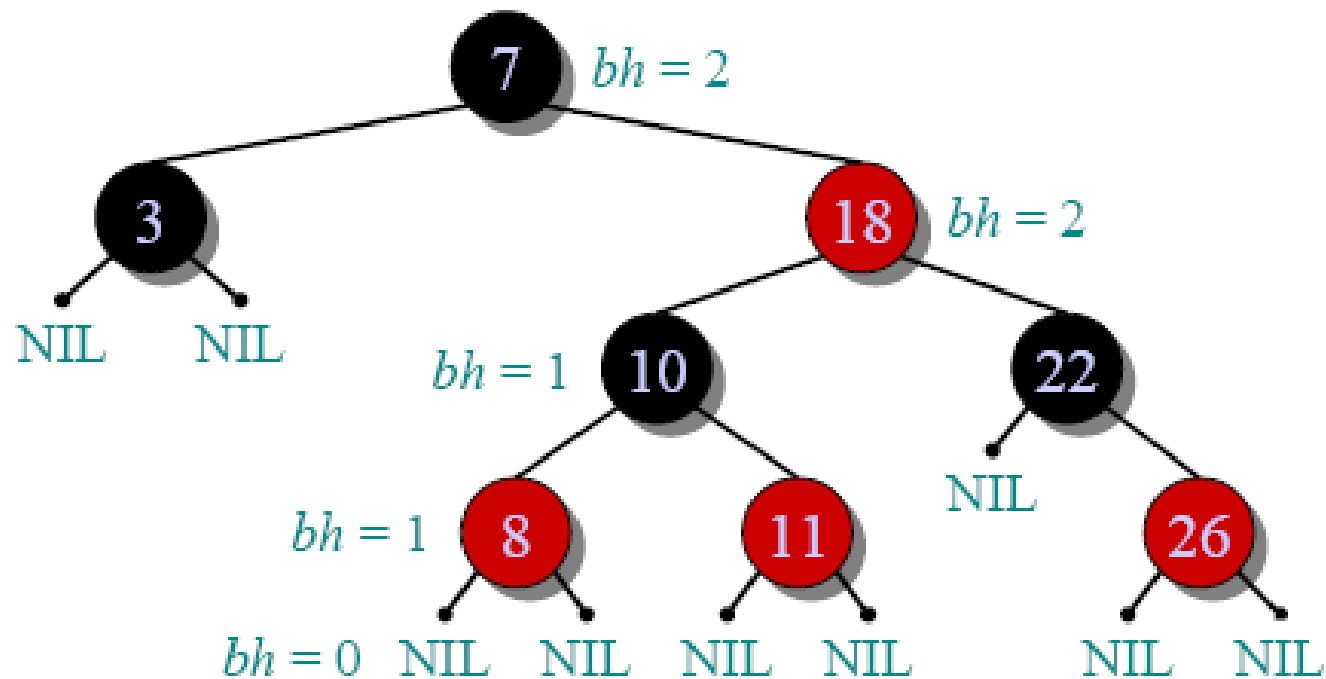
1. Every node is either red or black.



2. The root and leaves (NIL's) are black.



3. If a node is red, then its parent is black.



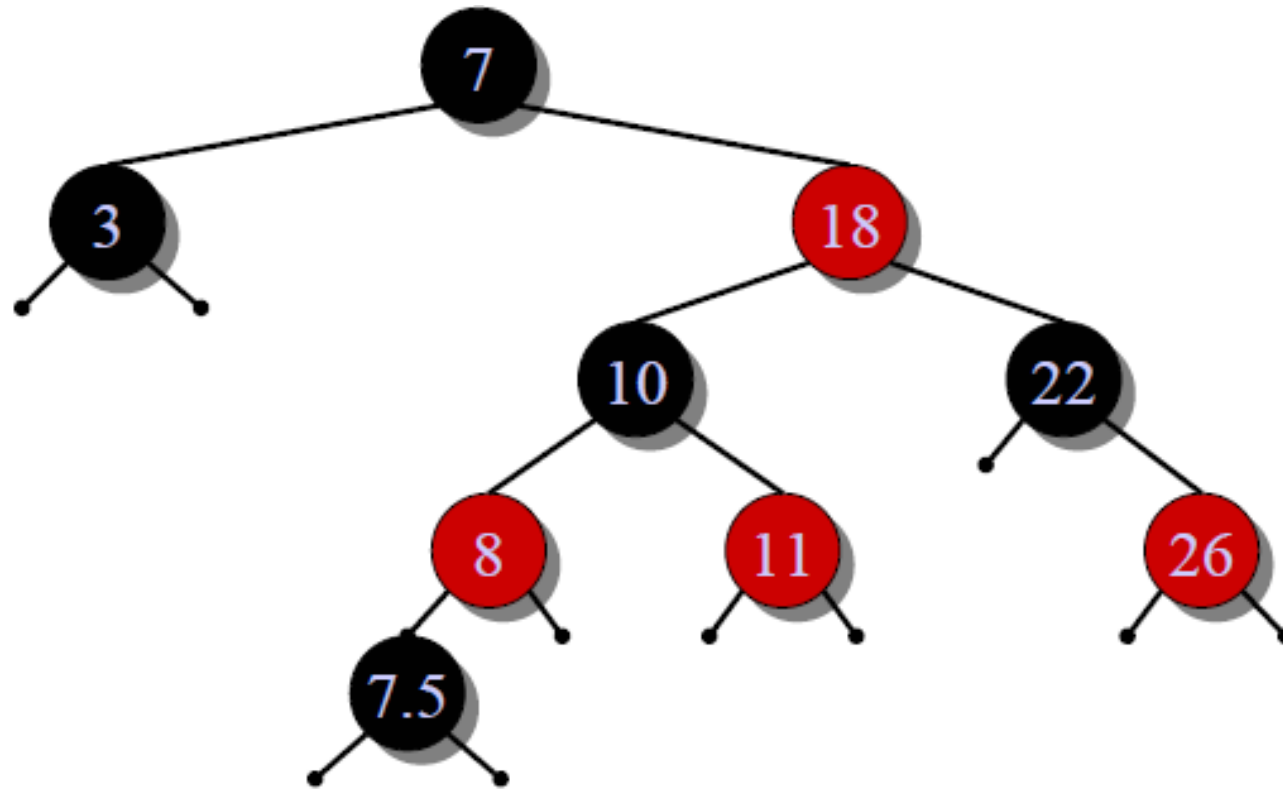
4. All simple paths from any node  $x$  to a descendant leaf have the same number of black nodes = *black-height*( $x$ ).

What properties would we like to prove about red-black trees ?

They always have  **$O(\log n)$  height**

There is an  **$O(\log n)$  time insertion** procedure which preserves the red-black properties

- Is it true that, after we add a new element to a tree, we can always recolor the tree to keep it red-black ?

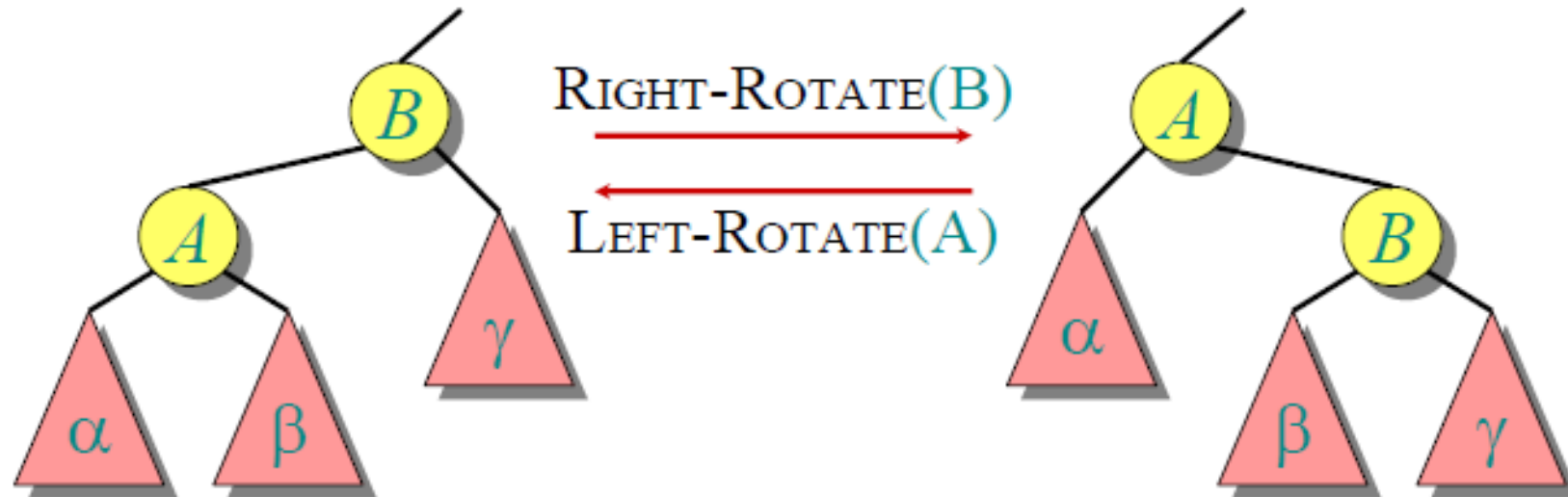


- Is it true that, after we add a new element to a tree, we can always recolor the tree to keep it red-black ?

NO. After insertions, sometimes we need to juggle nodes around



# Rotations



Rotations maintain the inorder ordering of keys:

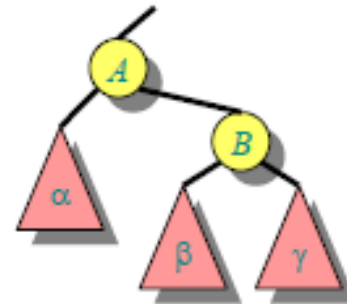
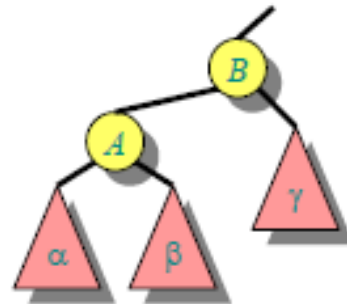
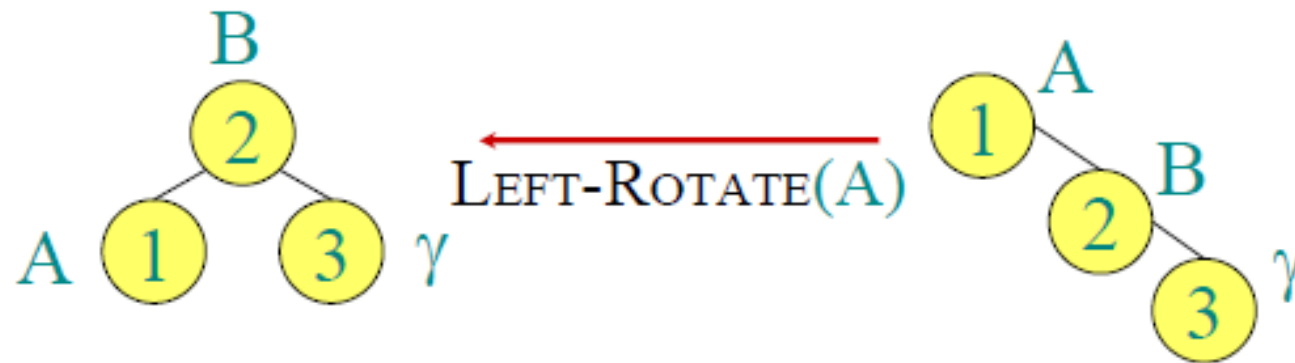
- $a \in \alpha, b \in \beta, c \in \gamma \Rightarrow a \leq A \leq b \leq B \leq c.$

A rotation can be performed in  $O(1)$  time.





# Rotations can reduce height





# Red-black tree wrap-up

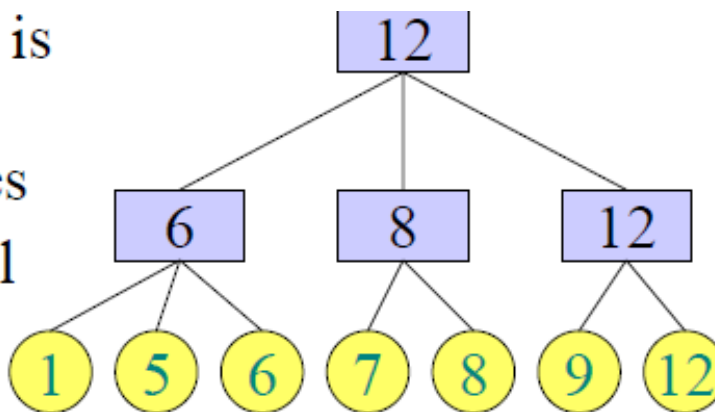
- Can show how
  - $O(\log n)$  re-colorings
  - 1 rotationcan restore red-black properties after an insertion
- Instead, we will see 2-3 trees (but will come back to red-black trees at the end)

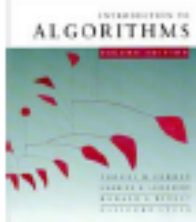
# 2-3 and 2-3-4 Trees



## 2-3 Trees

- The simplest balanced trees on the planet!
- Although a little bit more wasteful
- Degree of each node is either 2 or 3
- Keys are in the leaves
- All leaves have equal depth
- Leaves are sorted
- Each node  $x$  contains maximum key in the sub-tree, denoted  $x.max$





# Internal nodes

- Internal nodes:
  - Values:
    - `x.max`: maximum key in the sub-tree
  - Pointers:
    - `left[x]`
    - `mid[x]`
    - `right[x]` : can be null
    - `p[x]` : can be null for the root
    - ...
- Leaves:
  - `x.max` : the key



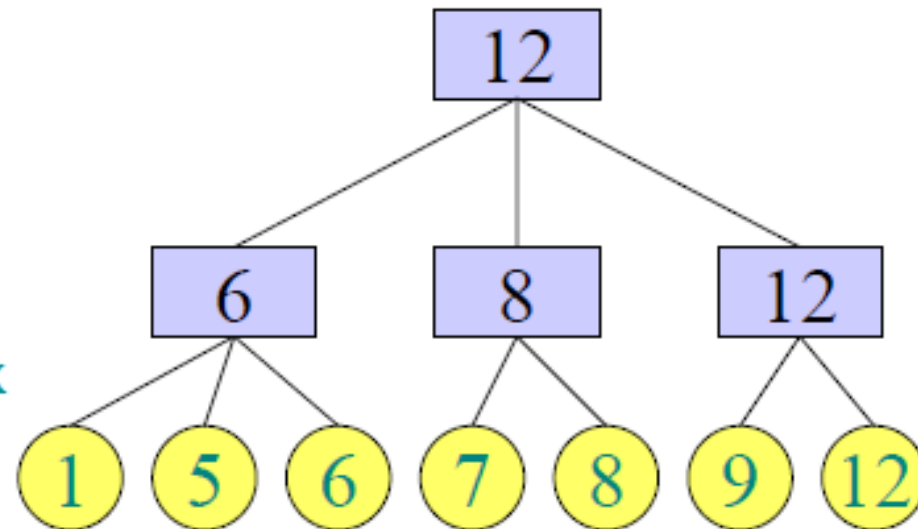
## Height of 2-3 tree

- What is the maximum height  $h$  of a 2-3 tree with  $n$  nodes ?
- Alternatively, what is the minimum number of nodes in a 2-3 tree of height  $h$  ?
- It is  $1+2+2^2+2^3+\dots+2^h = 2^{h+1}-1$
- $n \geq 2^{h+1}-1 \Rightarrow h = O(\log n)$
- Full binary tree is the worst-case example!



# Searching<sup>^</sup>

- How can we search for a key  $k$  ?
- Search( $x, k$ ):
- If  $x = \text{NIL}$  then return  $\text{NIL}$
  - Else if  $x$  is a leaf then
    - If  $x.\text{max} = k$  then return  $x$
    - Else return  $\text{NIL}$
  - Else
    - If  $k \leq \text{left}[x].\text{max}$  then Search( $\text{left}[x], k$ )
    - Else if  $k \leq \text{mid}[x].\text{max}$  then Search( $\text{mid}[x], k$ )
    - Else Search( $\text{right}[x], k$ )

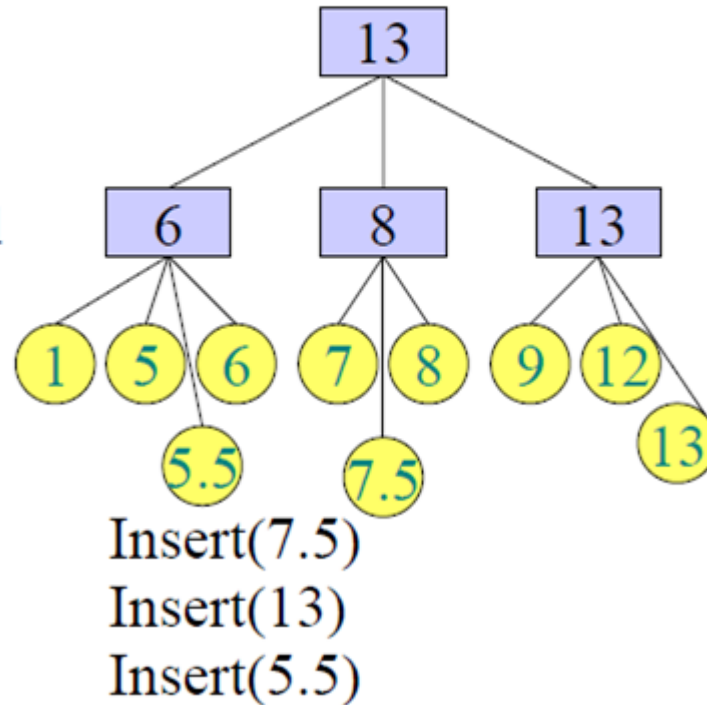


Search(8)  
Search(13)

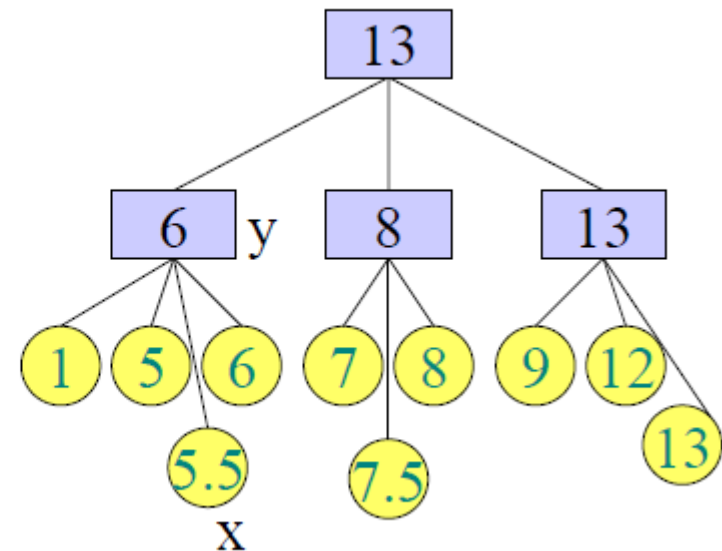


# Insertion

- How to insert  $x$  ?
- Perform Search for the key of  $x$
- Let  $y$  be the last internal node
- Insert  $x$  into  $y$  in a sorted order
- At the end, update the max values on the path to root



- If  $y$  has 4 children, then Split( $y$ )



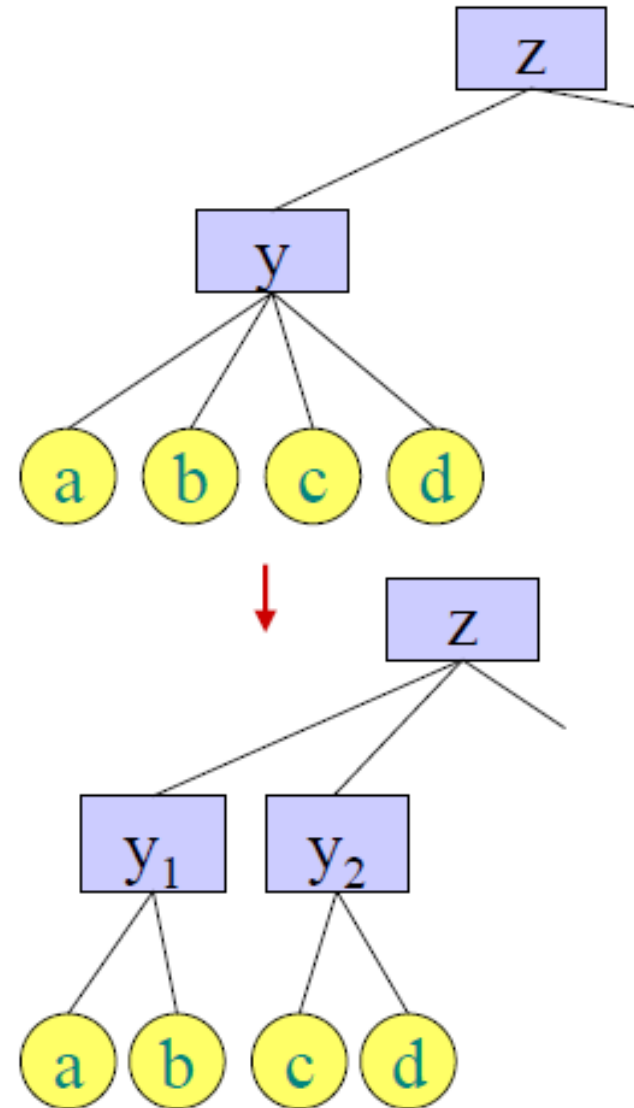


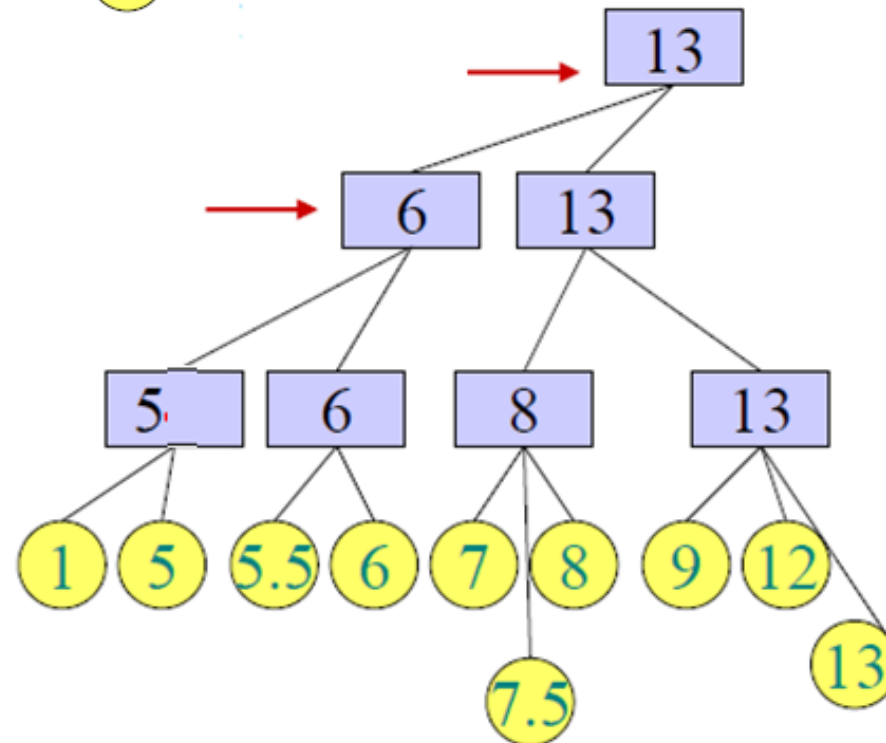
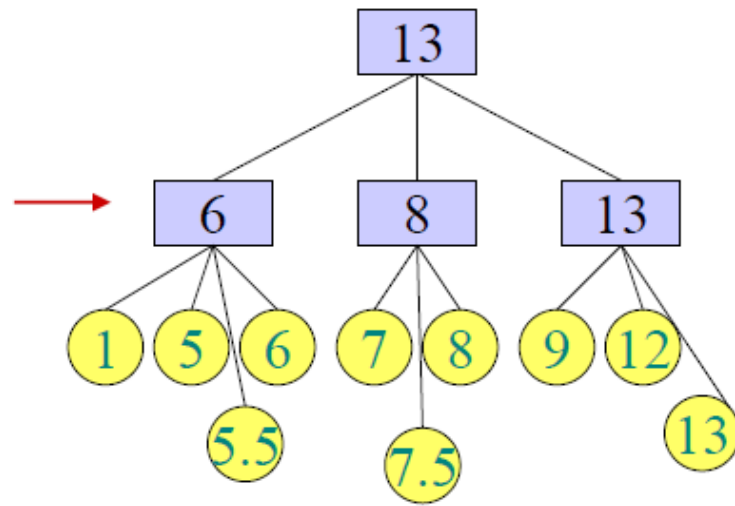


# Split

- Split  $y$  into two nodes  $y_1, y_2$
- Both are linked to  $z = \text{parent}(y)^*$
- If  $z$  has 4 children, split  $z$

\*If  $y$  is a root, then create new  $\text{parent}(y) = \text{new root}$



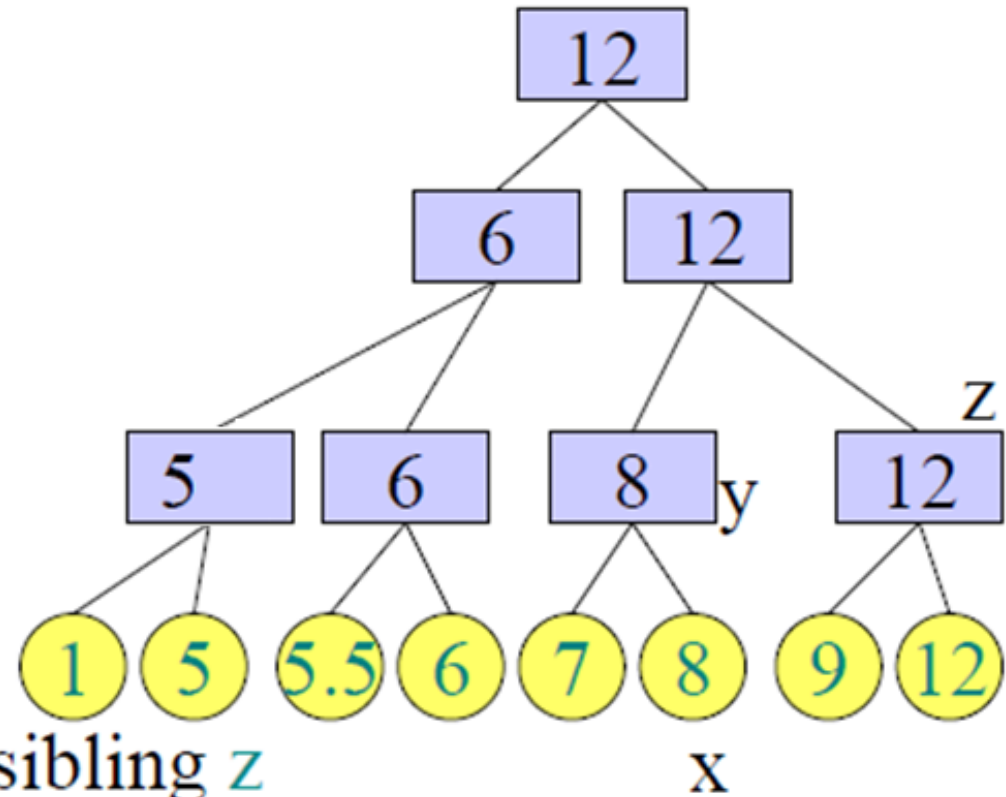


- Insert and Split preserve heights, unless new root is created, in which case all heights are increased by 1
- After Split, all nodes have 2 or 3 children
- Everything takes  $O(\log n)$  time



# Delete

- How to delete  $x$  ?
- Let  $y = p(x)$
- Remove  $x$  from  $y$
- If  $y$  has 1 child:
  - Remove  $y$
  - Attach to  $y$ 's sibling  $z$

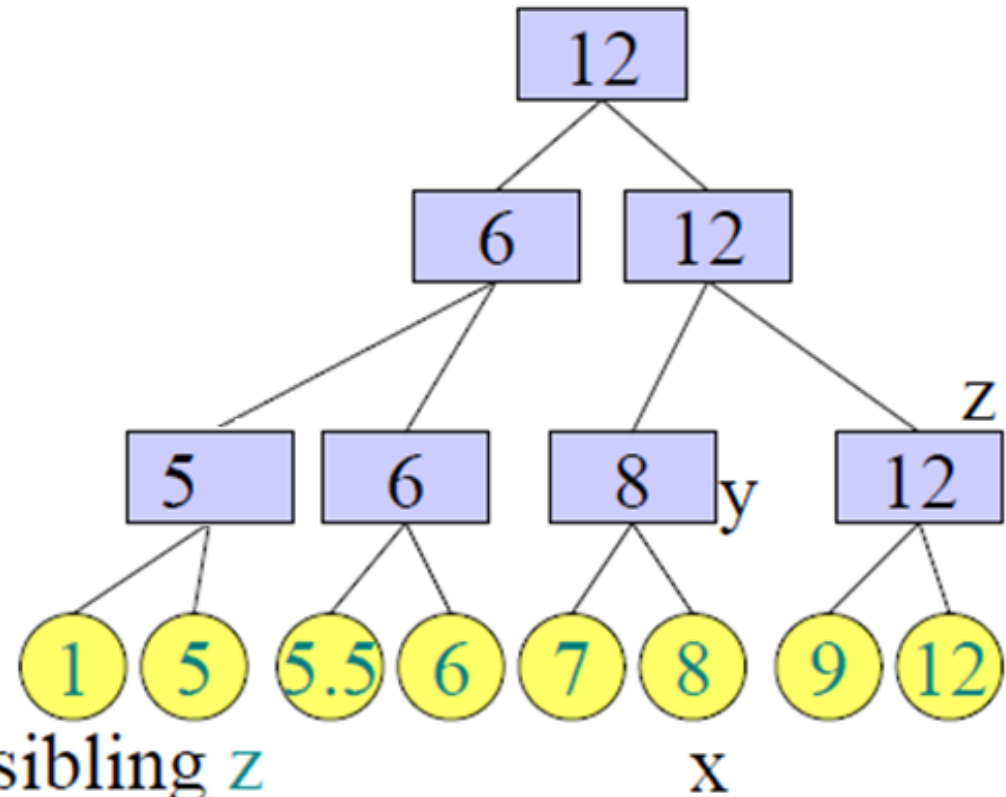


Delete(8)



# Delete

- How to delete  $x$  ?
- Let  $y = p(x)$
- Remove  $x$  from  $y$
- If  $y$  has 1 child:
  - Remove  $y$
  - Attach to  $y$ 's sibling  $z$



Delete(8)

If  $z$  has 4 children, then Split( $z$ )

INCOMPLETE – SEE THE END FOR FULL VERSION

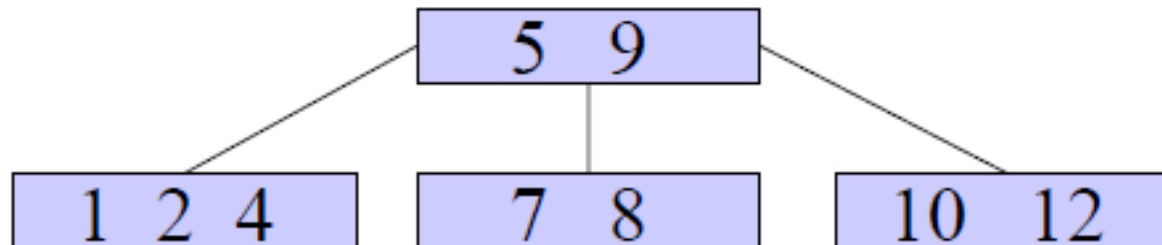


# Summing up

- 2-3 Trees:
  - $O(\log n)$  depth  $\Rightarrow$  Search in  $O(\log n)$  time
  - Insert, Delete (and Split) in  $O(\log n)$  time
- We will now see 2-3-4 trees
  - Same idea, but:
    - Each parent has 2,3 or 4 children
    - Keys in the inner nodes
    - More complicated procedures



# 2-3-4 Trees





# Back to Red and Black Trees

# Height of a red-black tree

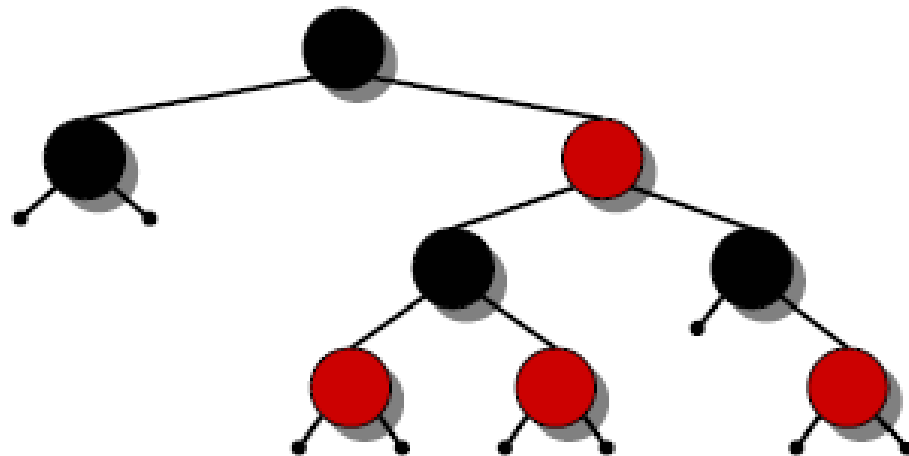
**Theorem.** A red-black tree with  $n$  keys has height

$$h \leq 2 \lg(n + 1).$$

**Proof.** (The book uses induction. Read carefully.)

**INTUITION:**

Merge red nodes into their black parents.





# Height of a red-black tree

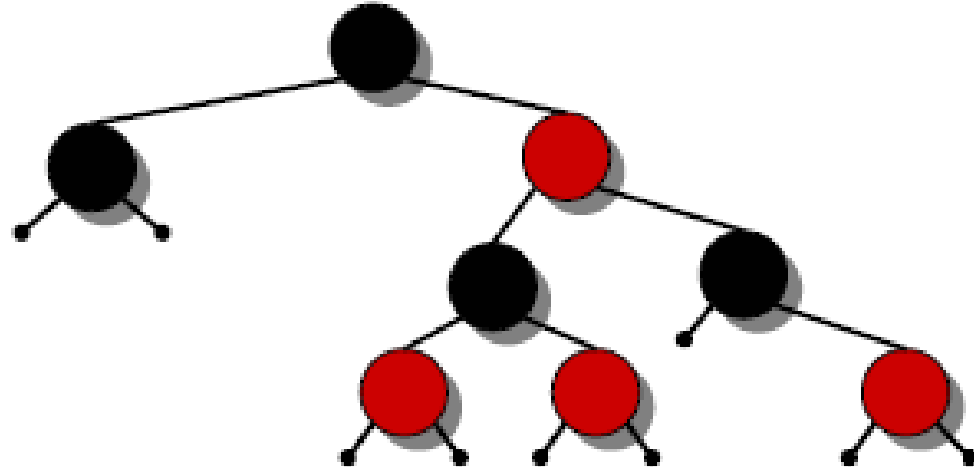
**Theorem.** A red-black tree with  $n$  keys has height

$$h \leq 2 \lg(n + 1).$$

**Proof.** (The book uses induction. Read carefully.)

**INTUITION:**

Merge red nodes into their black parents.



# Height of a red-black tree

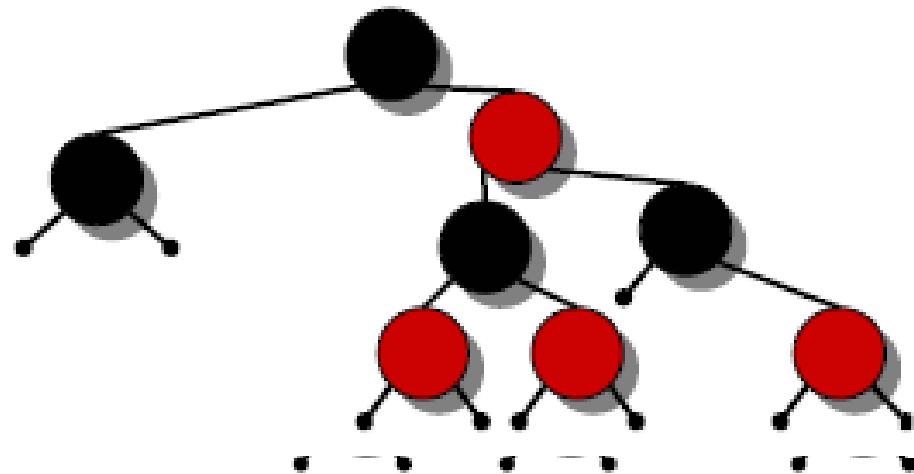
**Theorem.** A red-black tree with  $n$  keys has height

$$h \leq 2 \lg(n + 1).$$

**Proof.** (The book uses induction. Read carefully.)

**INTUITION:**

Merge red nodes into their black parents.



# Height of a red-black tree

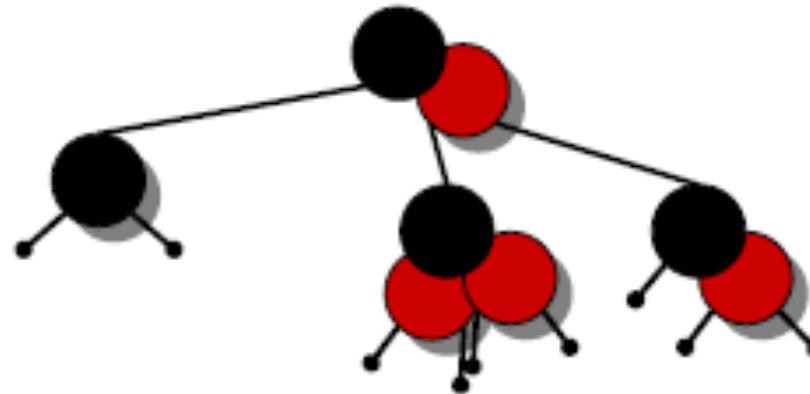
**Theorem.** A red-black tree with  $n$  keys has height

$$h \leq 2 \lg(n + 1).$$

**Proof.** (The book uses induction. Read carefully.)

## INTUITION:

Merge red nodes into their black parents.



# Height of a red-black tree

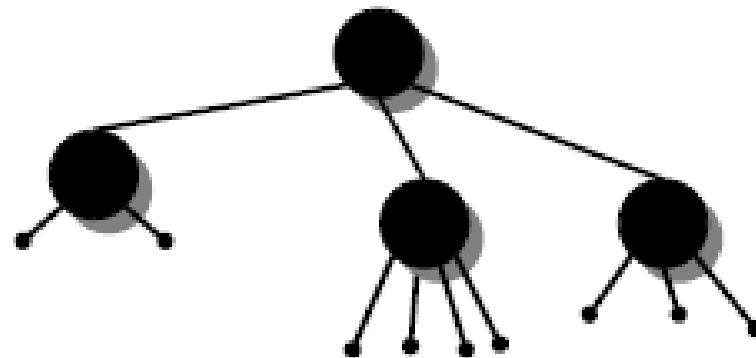
**Theorem.** A red-black tree with  $n$  keys has height

$$h \leq 2 \lg(n + 1).$$

**Proof.** (The book uses induction. Read carefully.)

## **INTUITION:**

Merge red nodes into their black parents.



# Height of a red-black tree

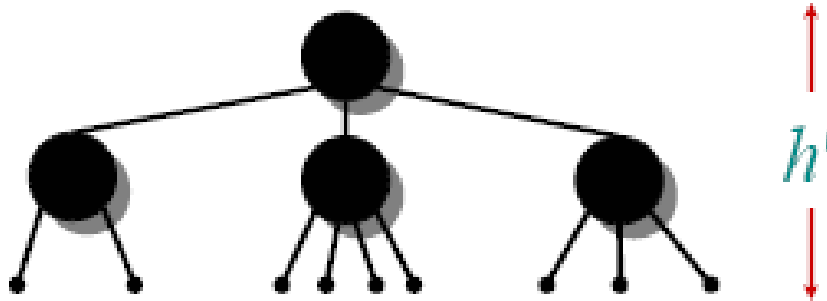
**Theorem.** A red-black tree with  $n$  keys has height

$$h \leq 2 \lg(n + 1).$$

*Proof.* (The book uses induction. Read carefully.)

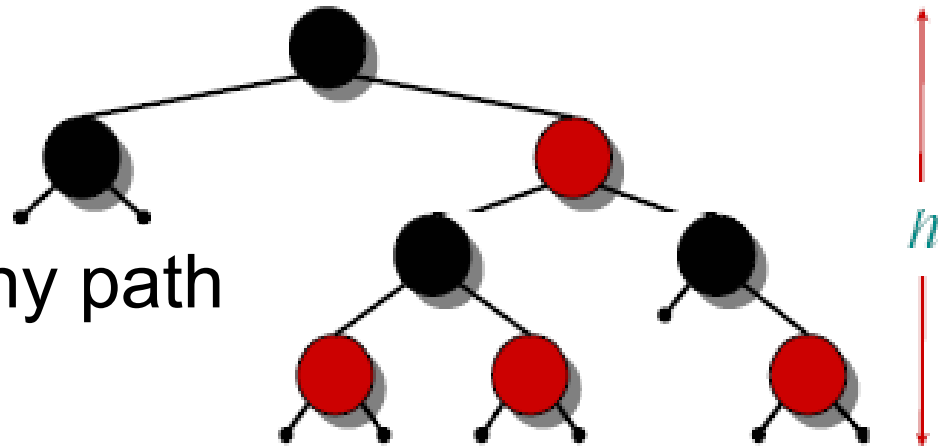
## INTUITION:

- Merge red nodes into their black parents.
- This process produces a tree in which each node has 2, 3, or 4 children.
- The 2-3-4 tree has uniform depth  $h'$  of leaves.



# Height of a red-black tree

- We have  $h' \geq h/2$ , since at most half the leaves on any path are red.

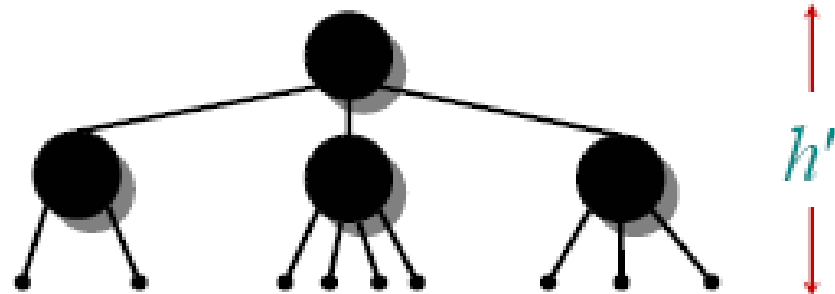


- The number of leaves in each tree is  $n + 1$

$$\Rightarrow n + 1 \geq 2^{h'}$$

$$\Rightarrow \lg(n + 1) \geq h' \geq h/2$$

$$\Rightarrow h \leq 2 \lg(n + 1). \quad \square$$



## Query operations

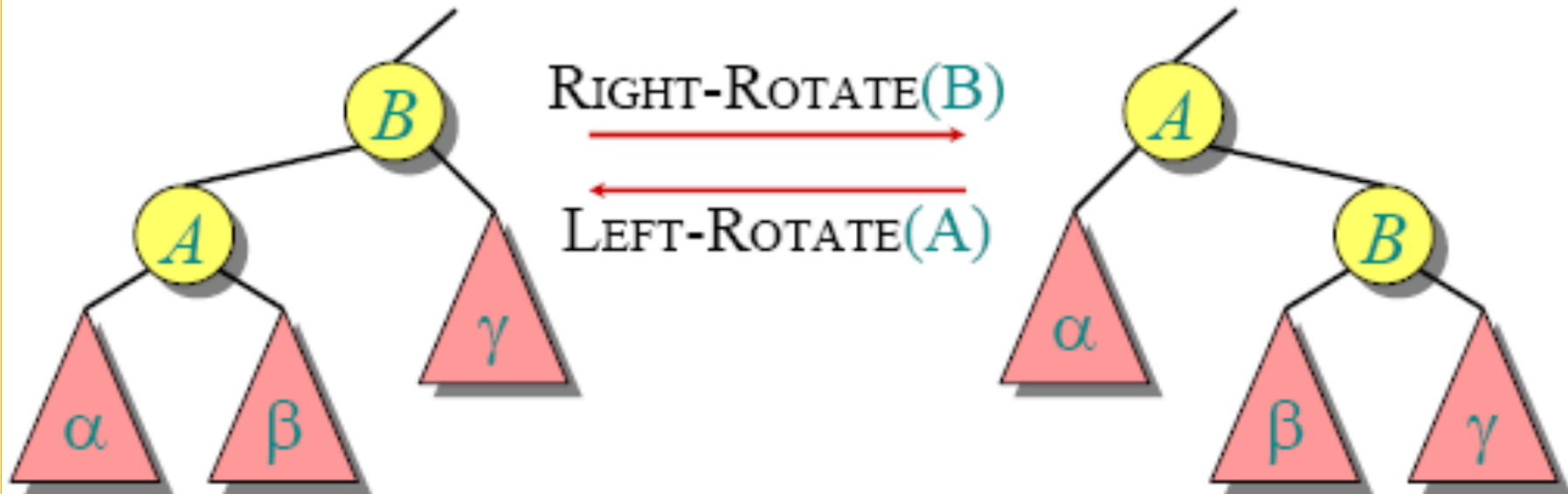
**Corollary.** The queries SEARCH, MIN, MAX, SUCCESSOR, and PREDECESSOR all run in  $O(\lg n)$  time on a red-black tree with  $n$  nodes.

## Modifying operations

The operations INSERT and DELETE cause modifications to the red-black tree:

- the operation itself,
- color changes,
- restructuring the links of the tree via ***“rotations”***.

# Rotations



Rotations maintain the inorder ordering of keys:

- $a \in \alpha, b \in \beta, c \in \gamma \Rightarrow a \leq A \leq b \leq B \leq c.$

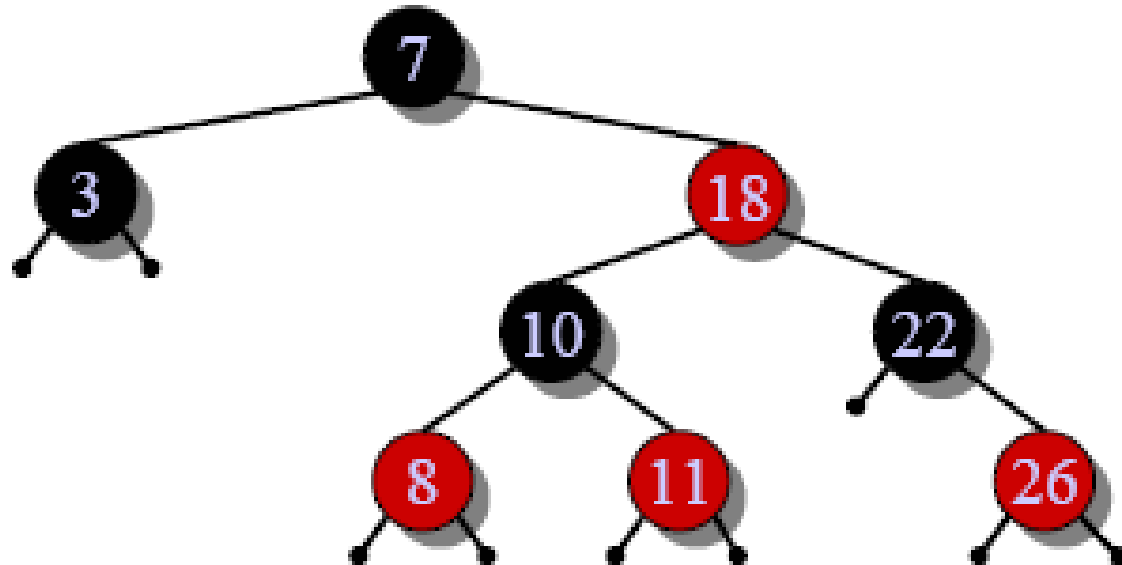
A rotation can be performed in  $O(1)$  time.



# Insertion into a red-black tree

**IDEA:** Insert  $x$  in tree. Color  $x$  red. Only redblack property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

**Example:**

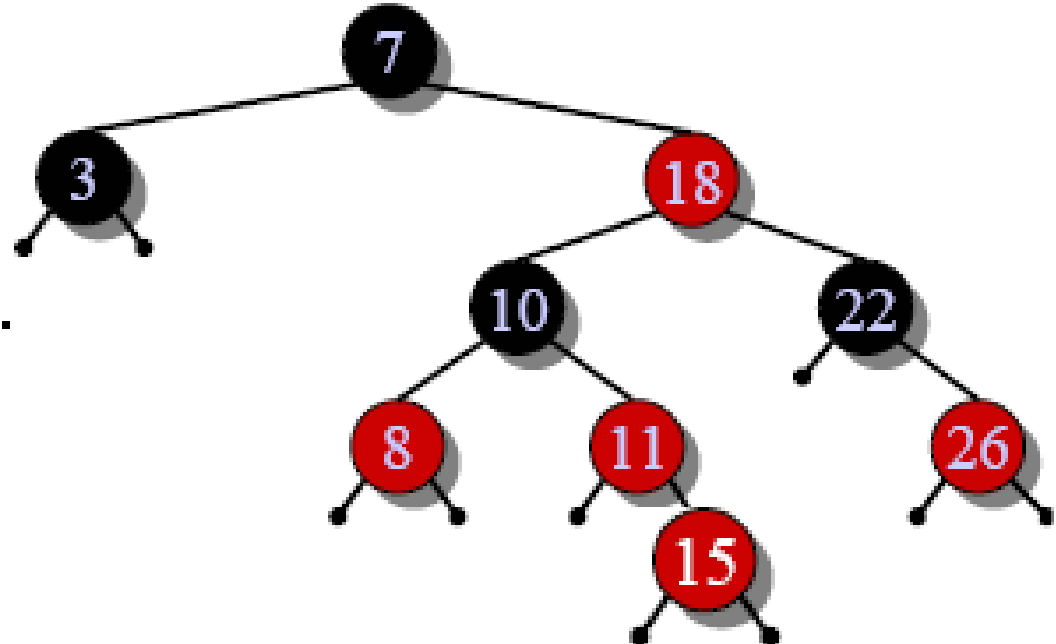


# Insertion into a red-black tree

**IDEA:** Insert  $x$  in tree. Color  $x$  red. Only redblack property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

## Example:

- Insert  $x = 15$ .
- Recolor, moving the violation up the tree.

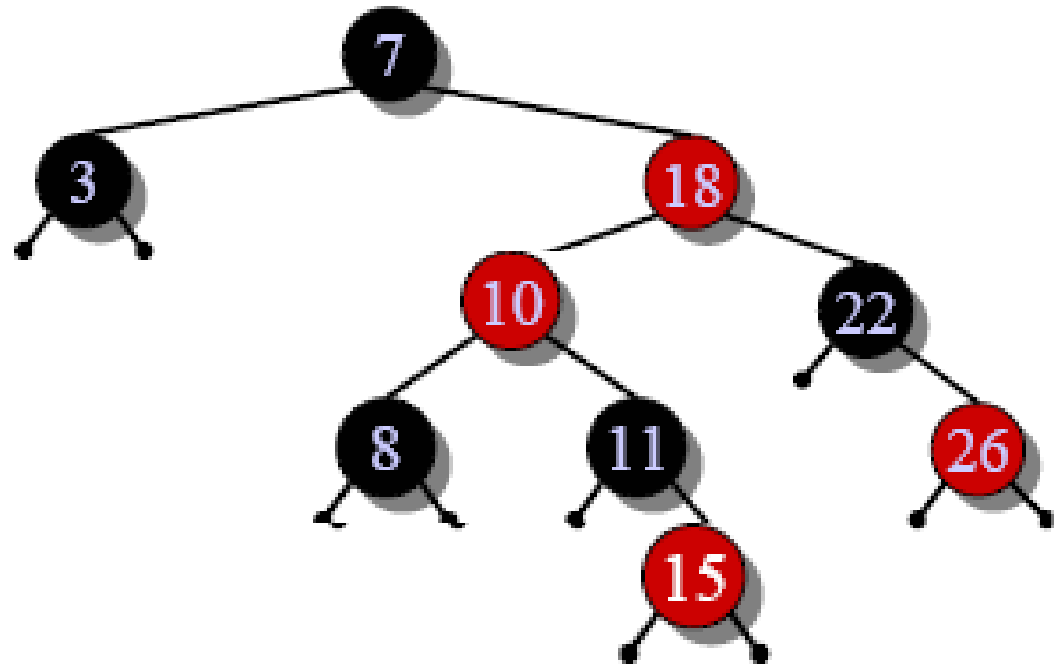


# Insertion into a red-black tree

**IDEA:** Insert  $x$  in tree. Color  $x$  red. Only redblack property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

## Example:

- Insert  $x = 15$ .
- Recolor, moving the violation up the tree
- RIGHT-ROTATE(18)

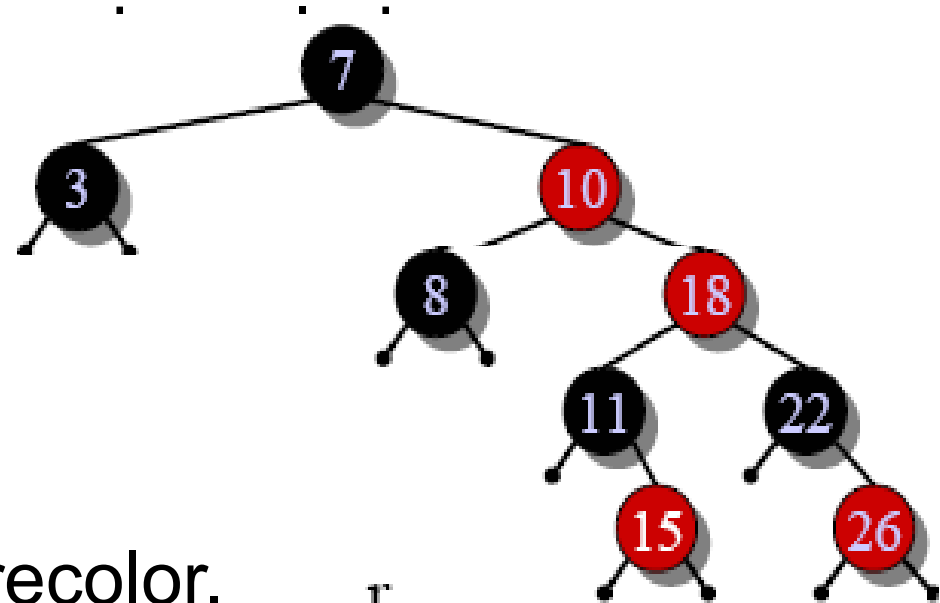


# Insertion into a red-black tree

**IDEA:** Insert  $x$  in tree. Color  $x$  red. Only redblack property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations

## Example:

- Insert  $x = 15$ .
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).
- LEFT-ROTATE(7) and recolor.

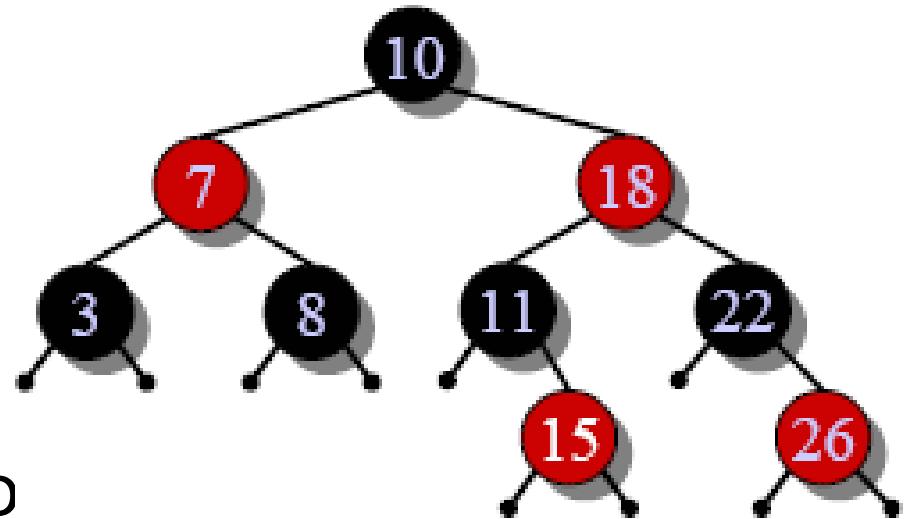


# Insertion into a red-black tree

**IDEA:** Insert  $x$  in tree. Color  $x$  red. Only redblack property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

**Example:**

- Insert  $x = 15$ .
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).
- LEFT-ROTATE(7) and recolor



# Pseudocode

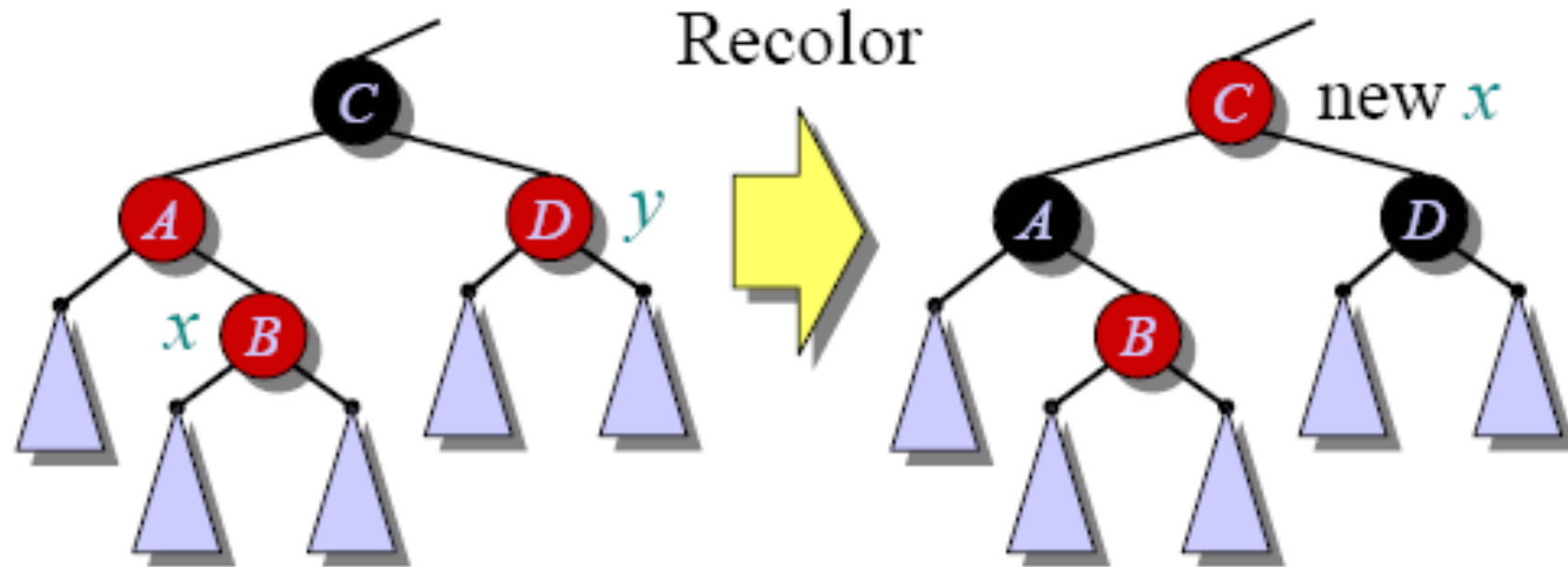
```
RB-INSERT( $T, x$ )
  TREE-INSERT( $T, x$ )
   $color[x] \leftarrow \text{RED}$   $\triangleright$  only RB property 3 can be violated
  while  $x \neq root[T]$  and  $color[p[x]] = \text{RED}$ 
    do if  $p[x] = left[p[p[x]]]$ 
      then  $y \leftarrow right[p[p[x]]]$   $\triangleright y = \text{aunt/uncle of } x$ 
        if  $color[y] = \text{RED}$ 
          then  $\langle \text{Case 1} \rangle$ 
          else if  $x = right[p[x]]$ 
            then  $\langle \text{Case 2} \rangle \triangleright \text{Case 2 falls into Case 3}$ 
             $\langle \text{Case 3} \rangle$ 
          else  $\langle \text{"then" clause with "left" and "right" swapped} \rangle$ 
         $color[root[T]] \leftarrow \text{BLACK}$ 
```

# Graphical notation

Let  denote a subtree with a black root.

All 's have the same black-height.

# Case 1

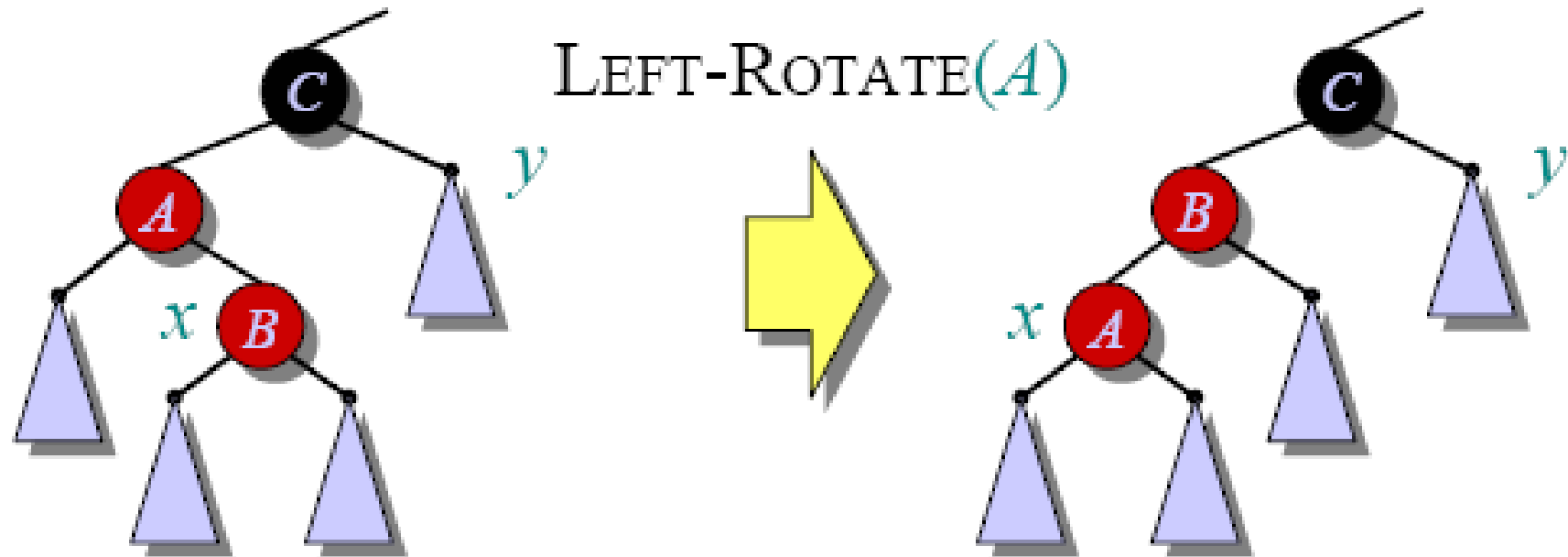


(Or, children of  $A$  are swapped.)

Push  $C$ 's black onto  $A$  and  $D$ , and recurse, since  $C$ 's parent may be red.

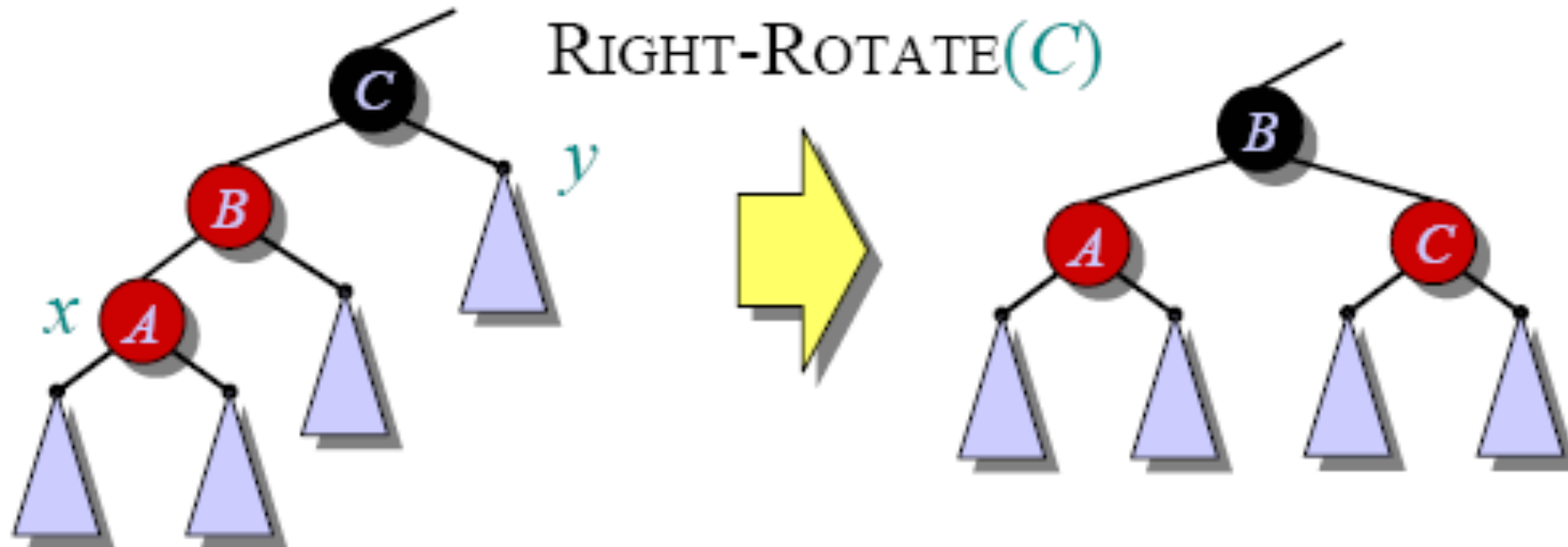


## Case 2



Transform to Case 3.

## Case 3



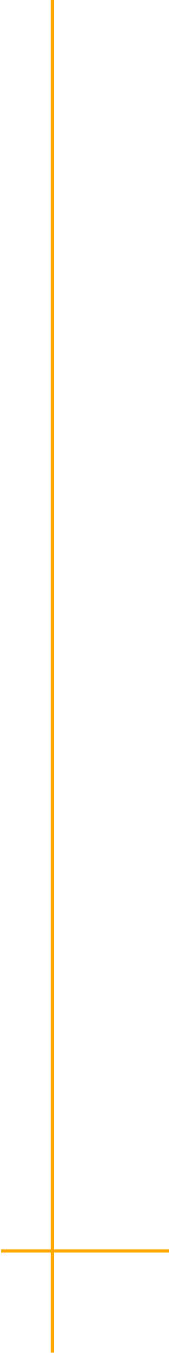
Done! No more violations of RB property 3 are possible.

# Analysis

- Go up the tree performing Case 1, which only recolors nodes.
- If Case 2 or Case 3 occurs, perform 1 or 2 rotations, and terminate.

**Running time:**  $O(\lg n)$  with  $O(1)$  rotations.

RB-DELETE— same asymptotic running time and number of rotations as RB-INSERT (see textbook).

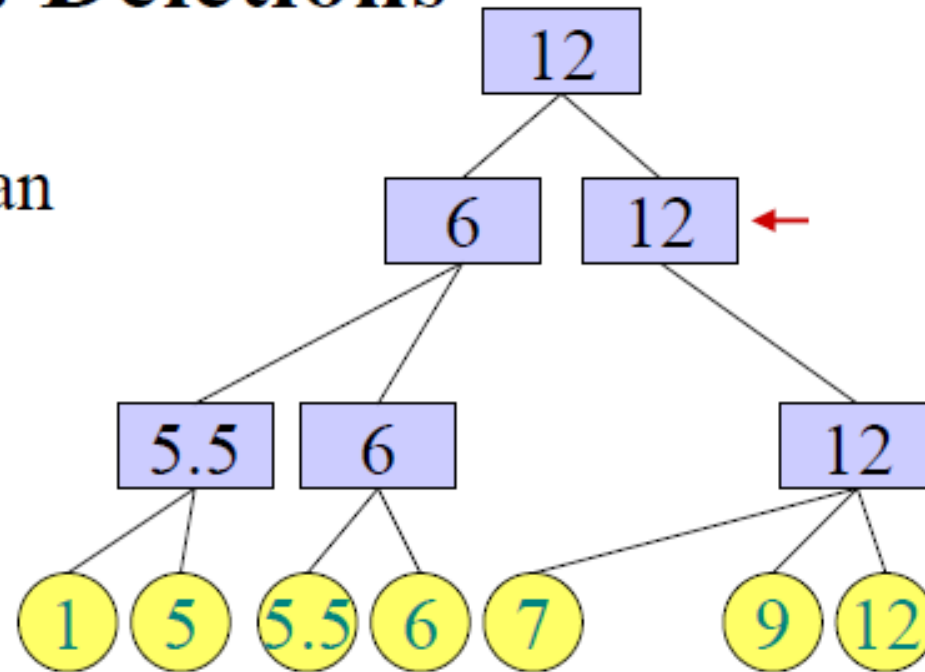


# Back to 2-3 Trees



## 2-3 Trees: Deletions

- Problem: there is an internal node that has only 1 child



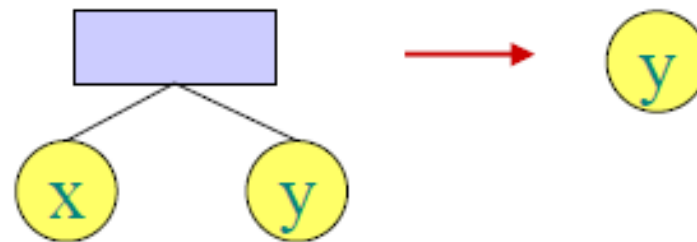


## Full procedure for Delete(x)

- Special case:  $x$  is the only element in the tree: delete everything



- Not-so-special case:  $x$  is one of two elements in the tree. In this case, the procedure on the next slide will delete  $x$

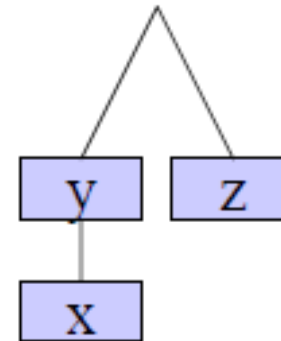


- Both  $\text{NIL}$  and  $y$  are special 2-3 trees

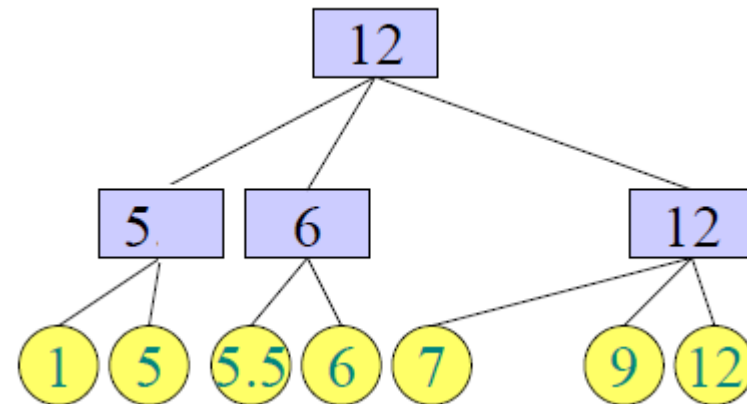
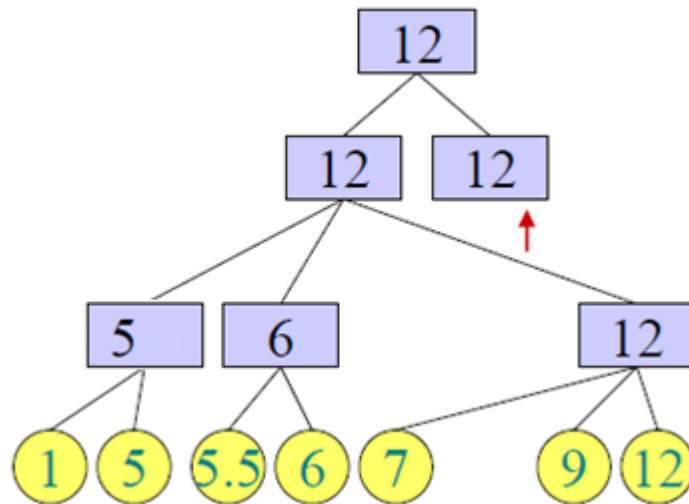
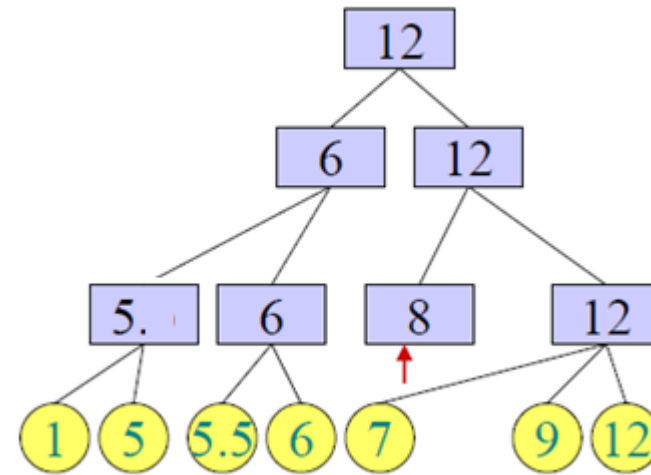
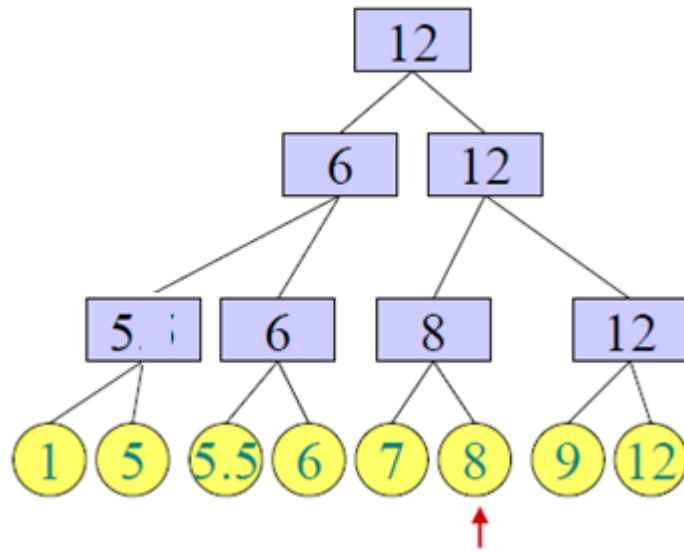


# Procedure for Delete(x)

- Let  $y = p(x)$
- Remove  $x$
- If  $y \neq \text{root}$  then
  - Let  $z$  be the sibling of  $y$ .
  - Assume  $z$  is the right sibling of  $y$ , otherwise the code is symmetric.
  - If  $y$  has only 1 child  $w$  left
    - Case 1:  $z$  has 3 children
      - Attach  $\text{left}[z]$  as the rightmost child of  $y$
      - Update  $y.\text{max}$  and  $z.\text{max}$
    - Case 2:  $z$  has 2 children:
      - Attach the child  $w$  of  $y$  as the leftmost child of  $z$
      - Update  $z.\text{max}$
      - Delete( $y$ ) (recursively\*)
  - Else
    - Update  $\text{max}$  of  $y$ ,  $p(y)$ ,  $p(p(y))$  and so on until root
- Else
  - If root has only one child  $u$ 
    - Remove root
    - Make  $u$  the new root



\*Note that the input of Delete does not have to be a leaf





# Summary

Binary Search Tree (BST) review

Red and Black Trees

2-3 and 2-3-4 trees

Operations on Red and Black Trees