CS60020: Foundations of Algorithm Design and Machine Learning

Sourangshu Bhattacharya

Balanced search trees

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

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Red-black trees

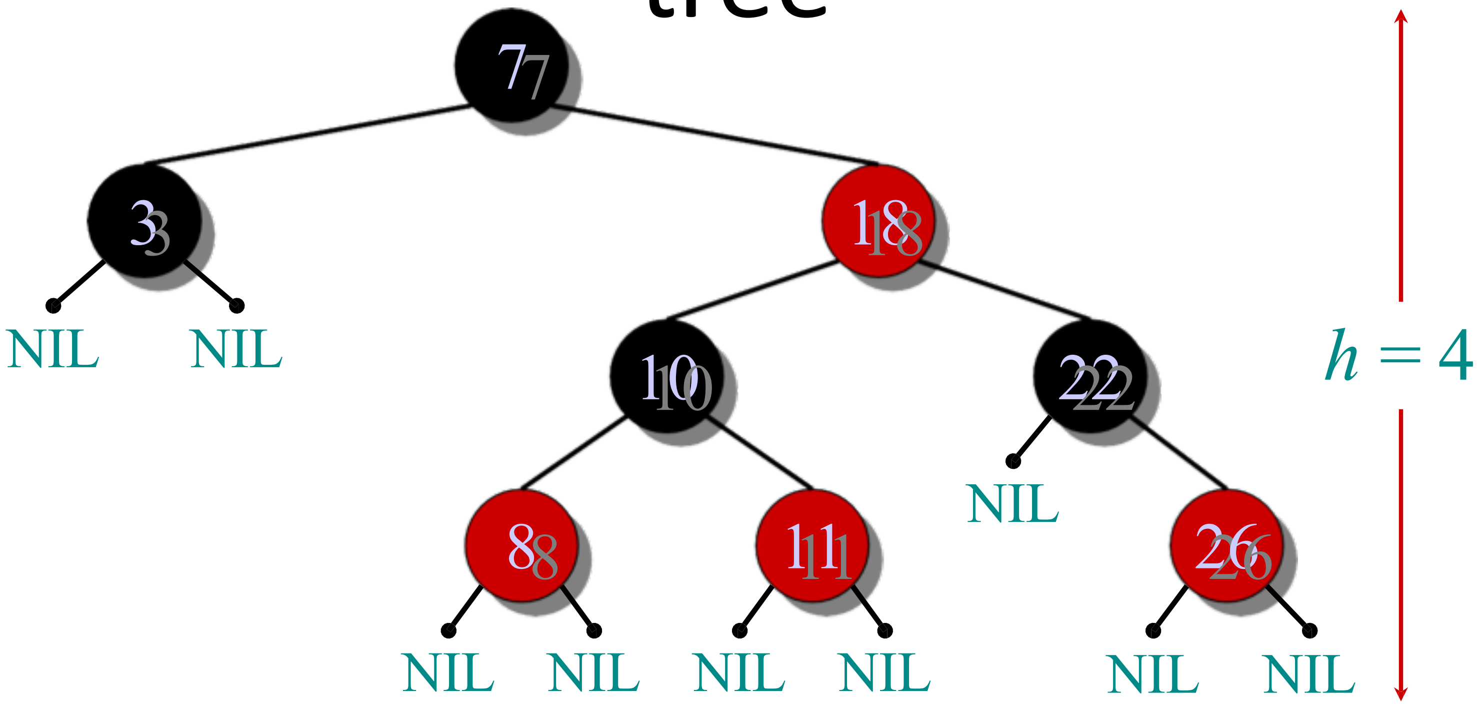
This data structure requires an extra one-bit color field in each node.

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 = black-height(x).

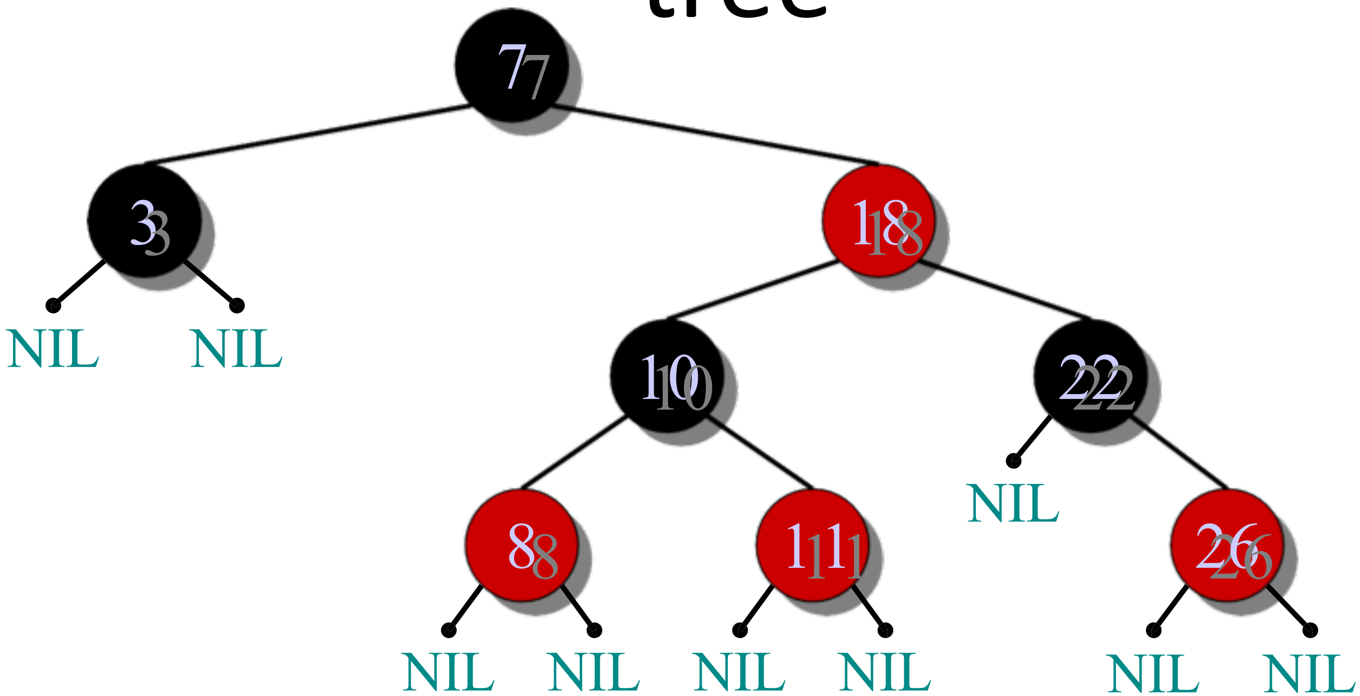
Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Example of a red-black tree



 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

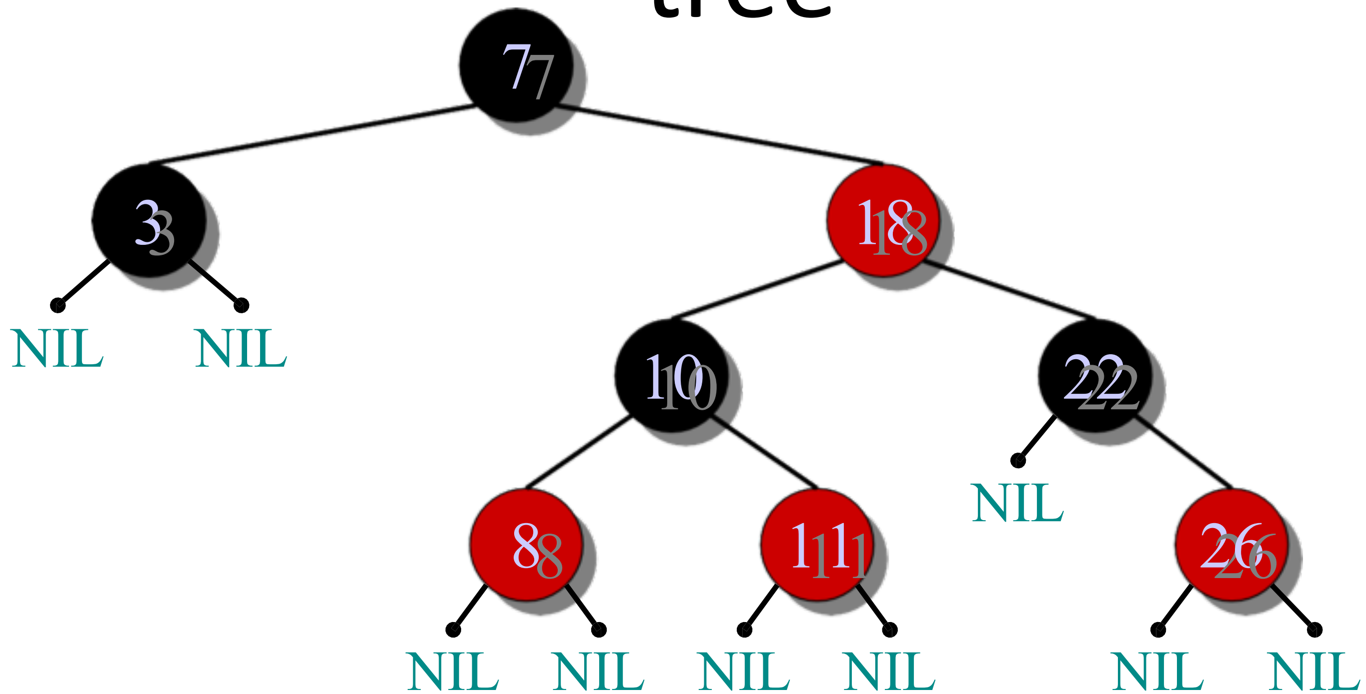
Example of a red-black tree



1. Every node is either red or black.

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

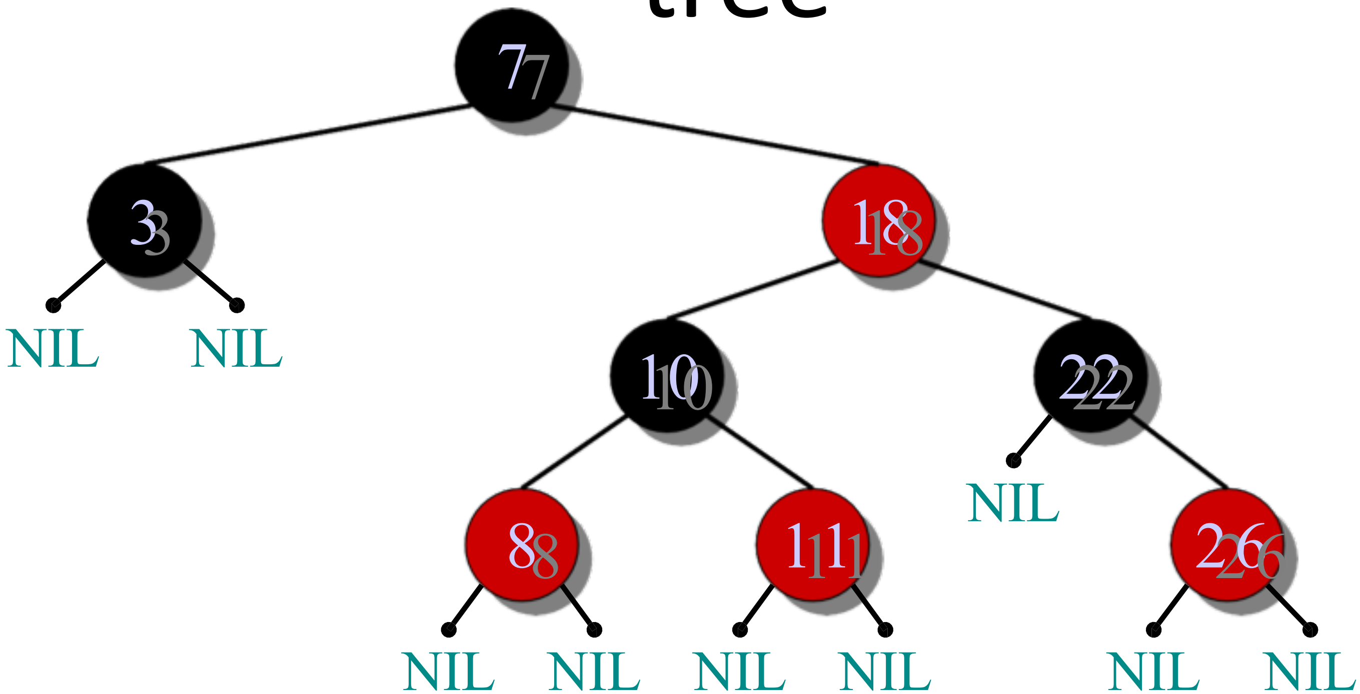
Example of a red-black tree



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

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

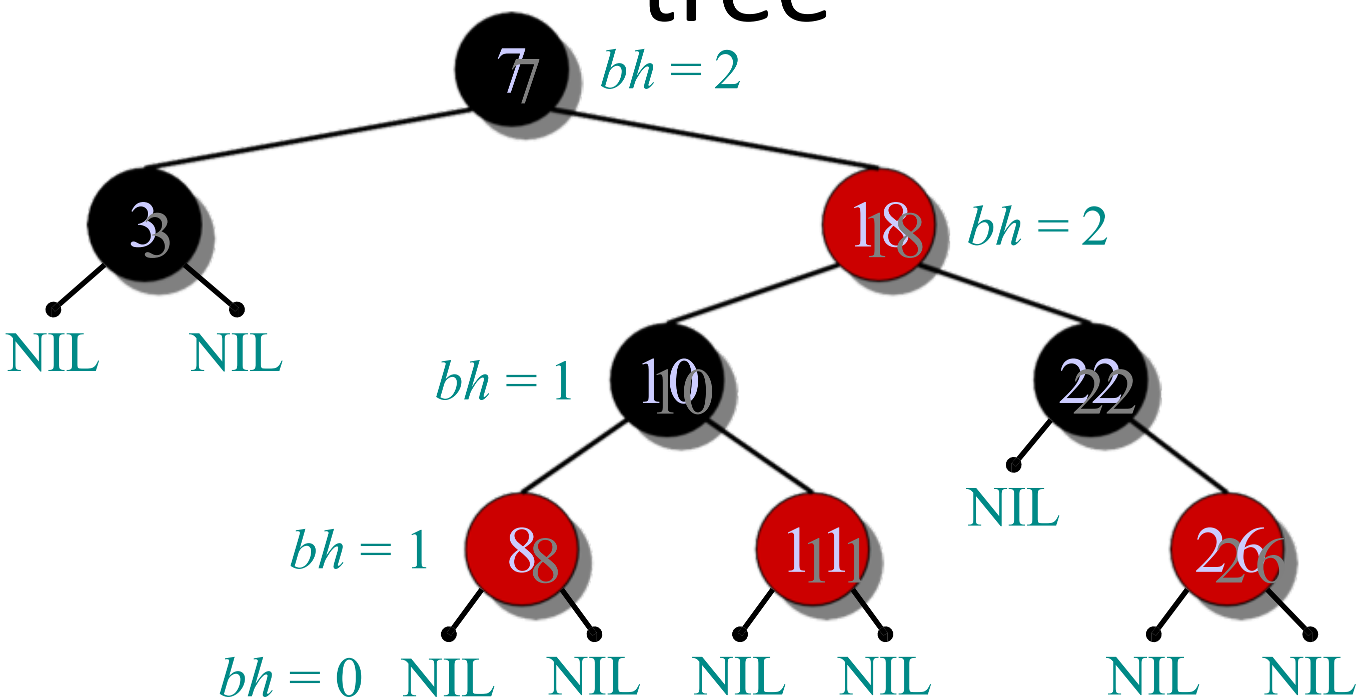
Example of a red-black tree



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

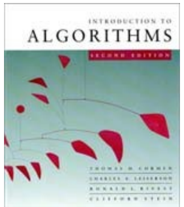
Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Example of a red-black tree



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

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Height of a red-black tree

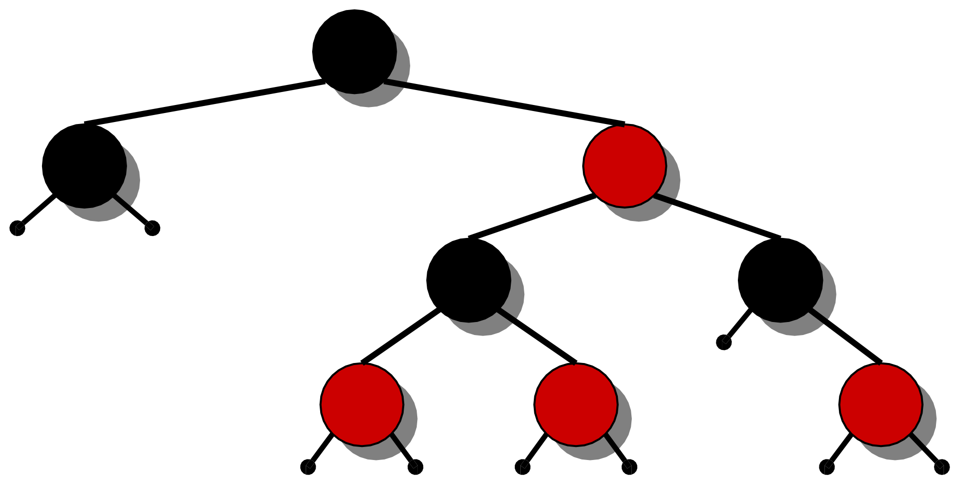
Theorem. A red-black tree with n keys has height

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

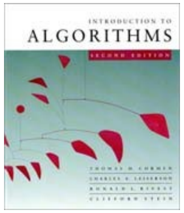
Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.



 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Height of a red-black tree

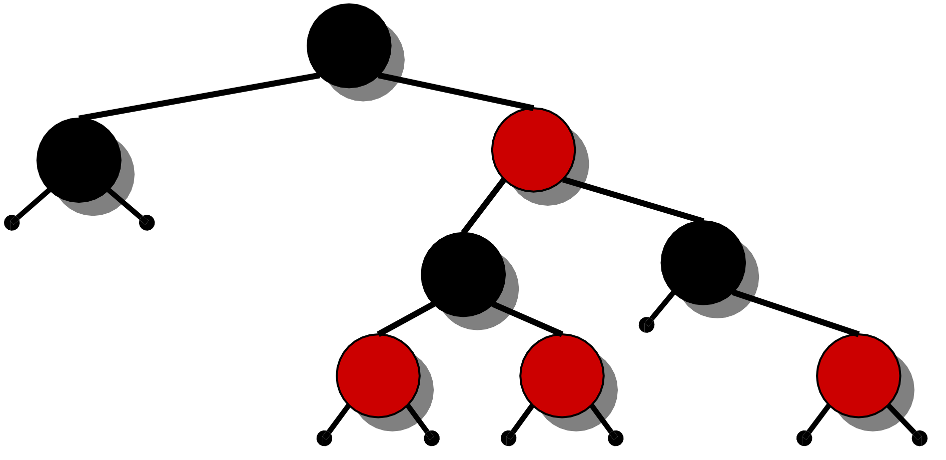
Theorem. A red-black tree with n keys has height

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

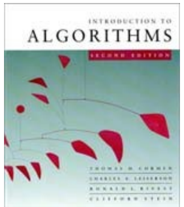
Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.



 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Height of a red-black tree

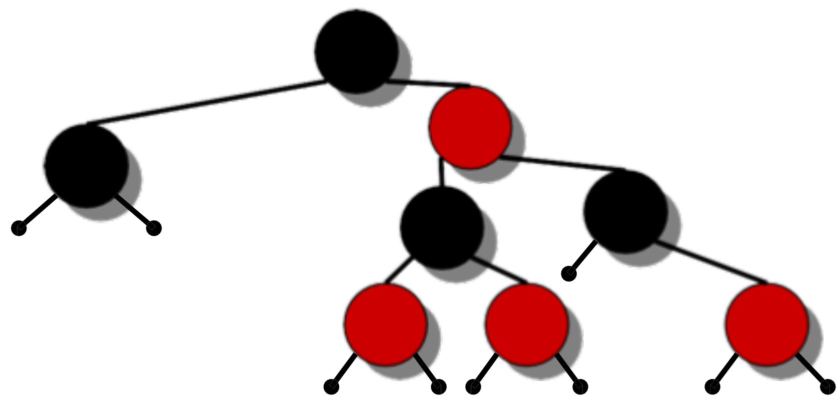
Theorem. A red-black tree with n keys has height

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

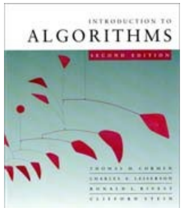
Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.



Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Height of a red-black tree

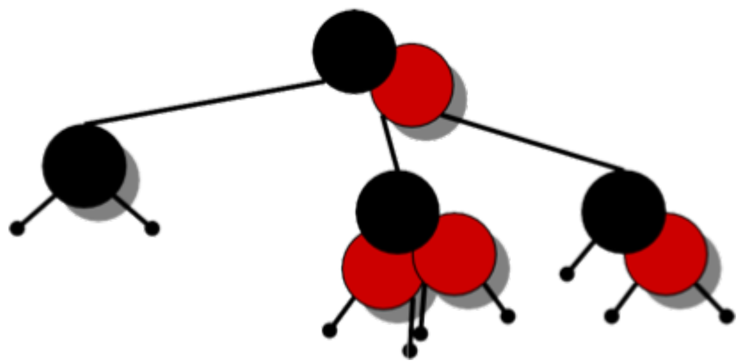
Theorem. A red-black tree with n keys has height

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

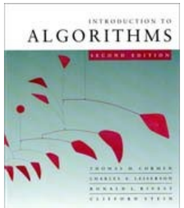
Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.



 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Height of a red-black tree

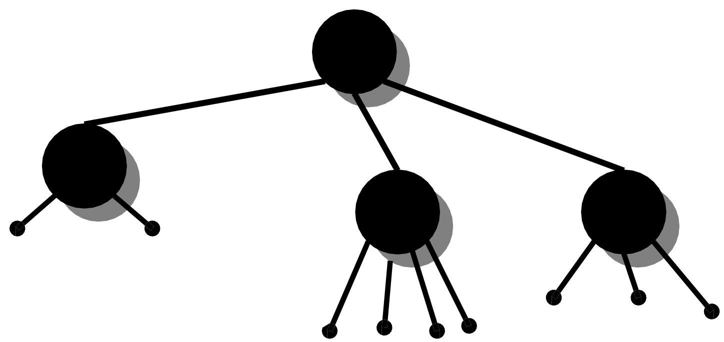
Theorem. A red-black tree with n keys has height

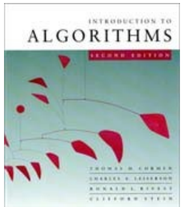
$$h \leq 2\lg(n+1).$$

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

INTUITION:

- Merge red nodes into their black parents.





Height of a red-black tree

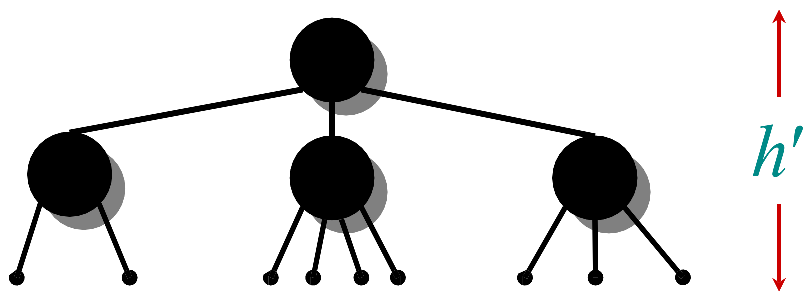
Theorem. A red-black tree with n keys has height

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

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

INTUITION:

- Merge red nodes into their black parents.

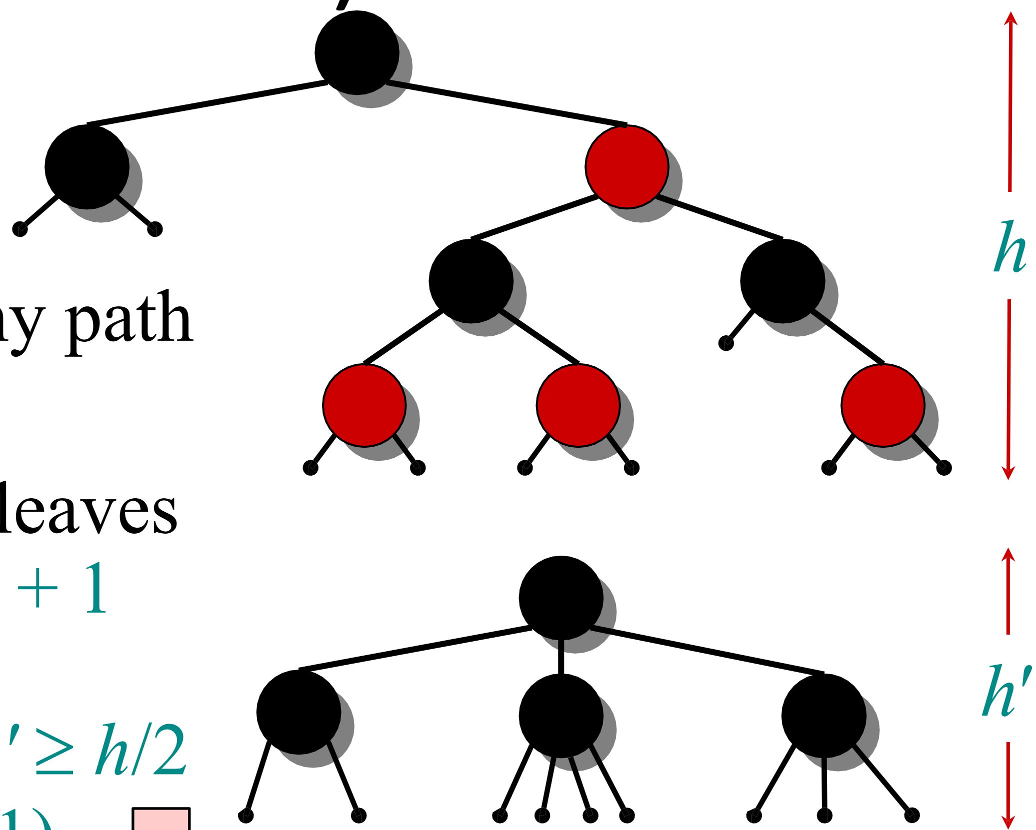


- This process produces a tree in which each node has 2, 3, or 4 children.
- The 2-3-4 tree has uniform depth h' of leaves.

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Proof (continued)

- 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 $n + 1$
  $\Rightarrow n + 1 \geq 2^{h'}$
  $\Rightarrow \lg(n + 1) \geq h' \geq h/2$
  $\Rightarrow h \leq 2 \lg(n + 1)$. $\square$



Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

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.

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

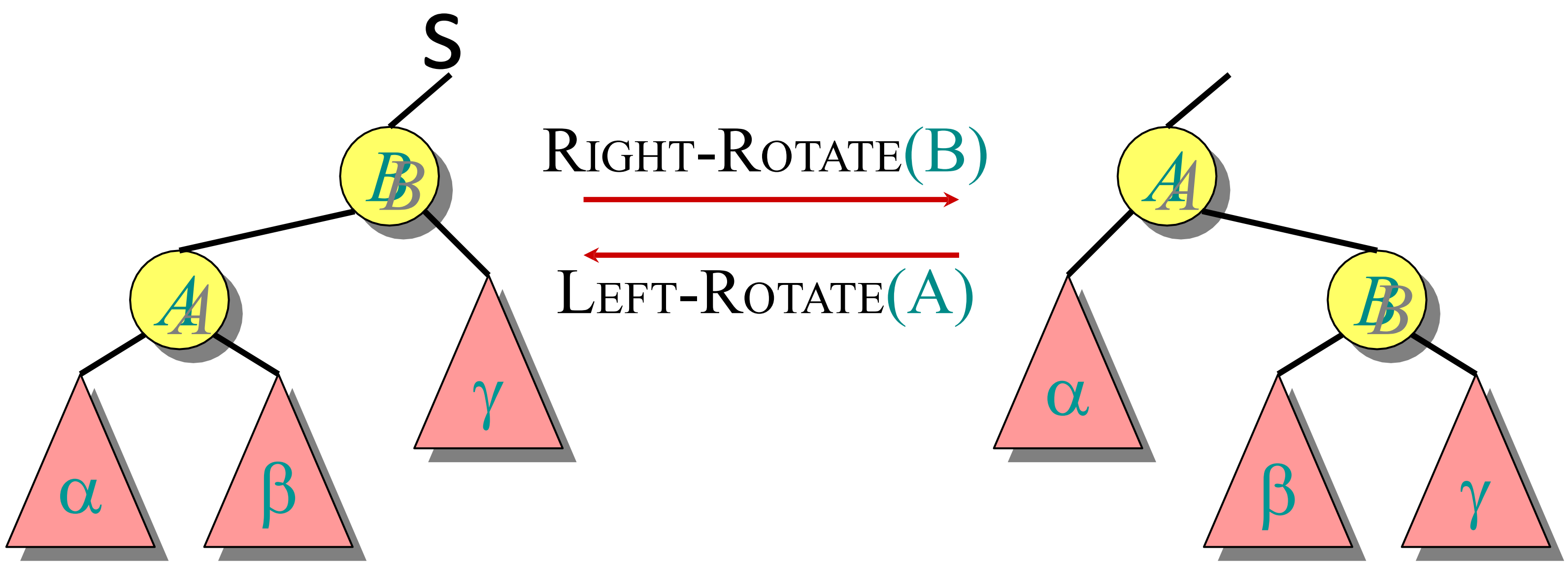
Modifying operations

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"***.

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Rotation

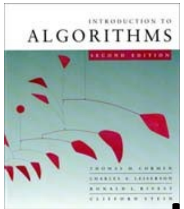


Rotations maintain the inorder ordering of keys:

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

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

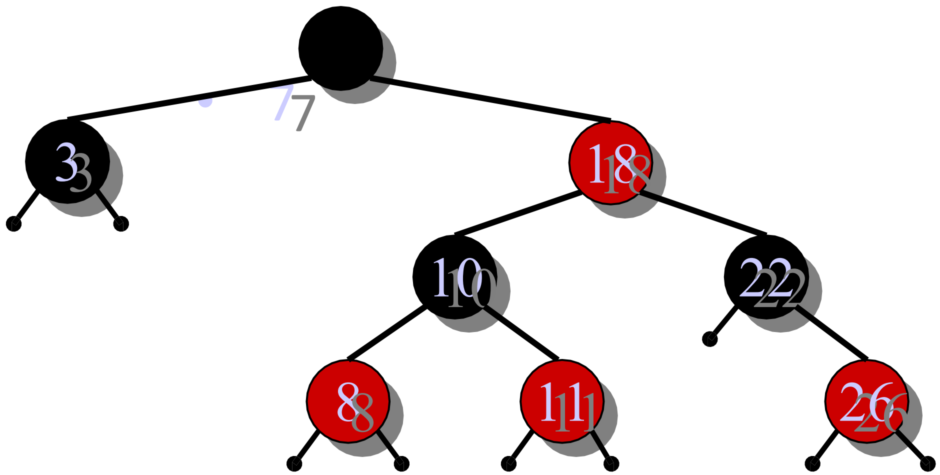
 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



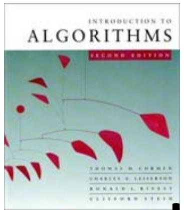
Insertion into a red-black tree

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

- **Example:**



 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

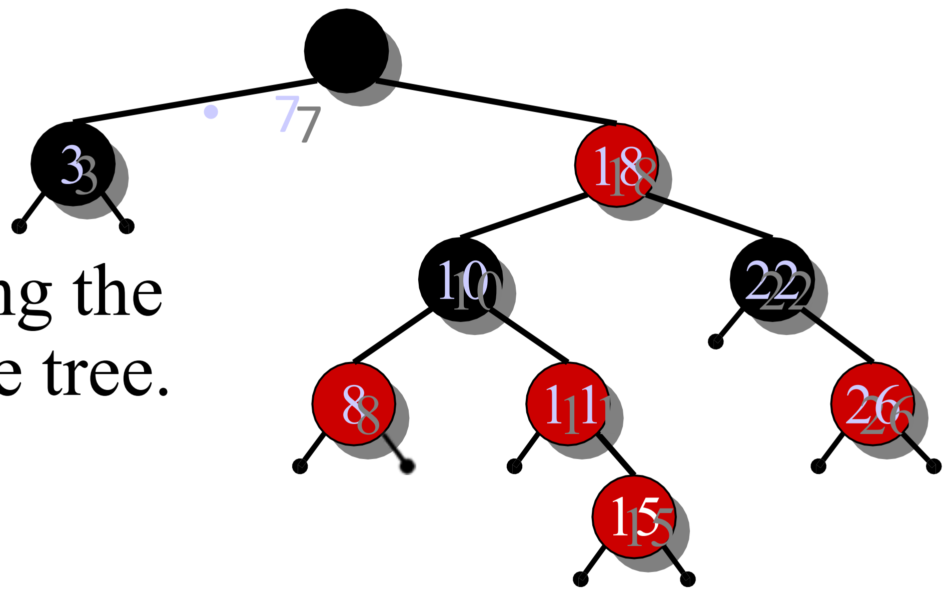


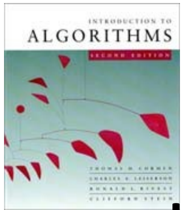
Insertion into a red-black tree

- **IDEA:** Insert x in tree. Color x red. Only red-black 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.



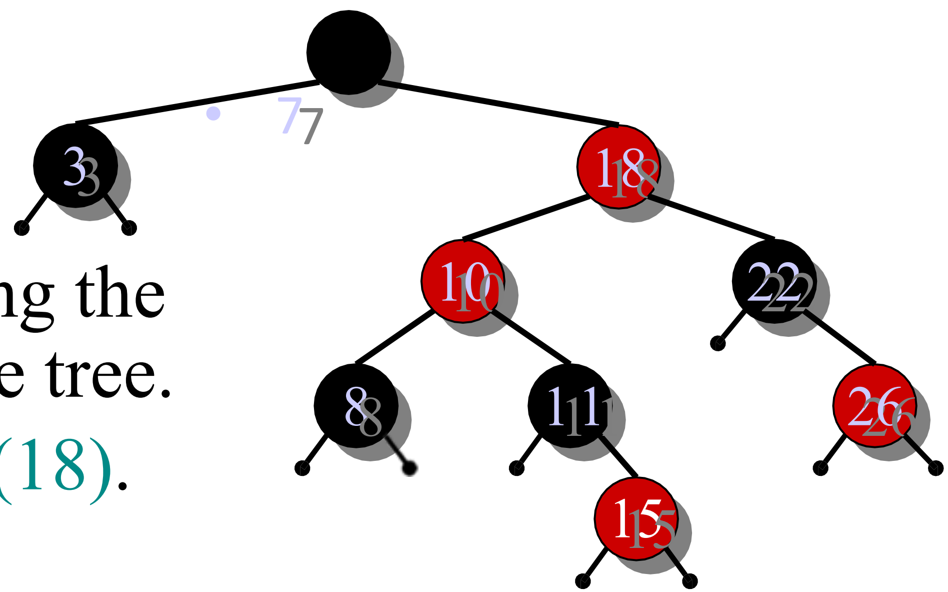


Insertion into a red-black tree

• **IDEA:** Insert *x* in tree. Color *x* red. Only red-black 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).



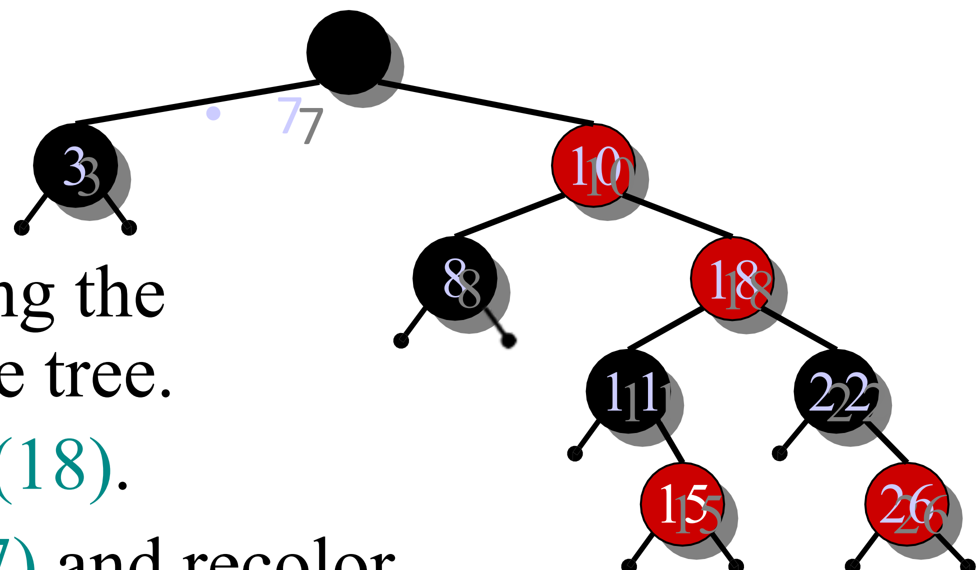
 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Insertion into a red-black tree

•**IDEA:** Insert *x* in tree. Color *x* red. Only red-black 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.

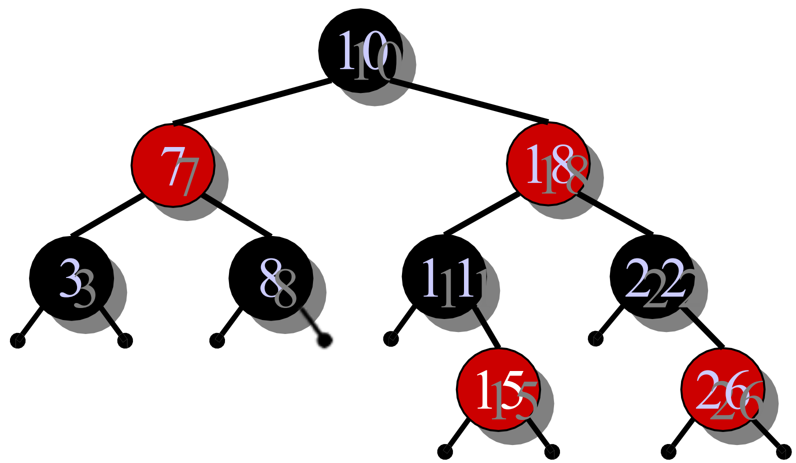


Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black 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.

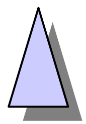


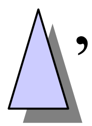
Pseudocode

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

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

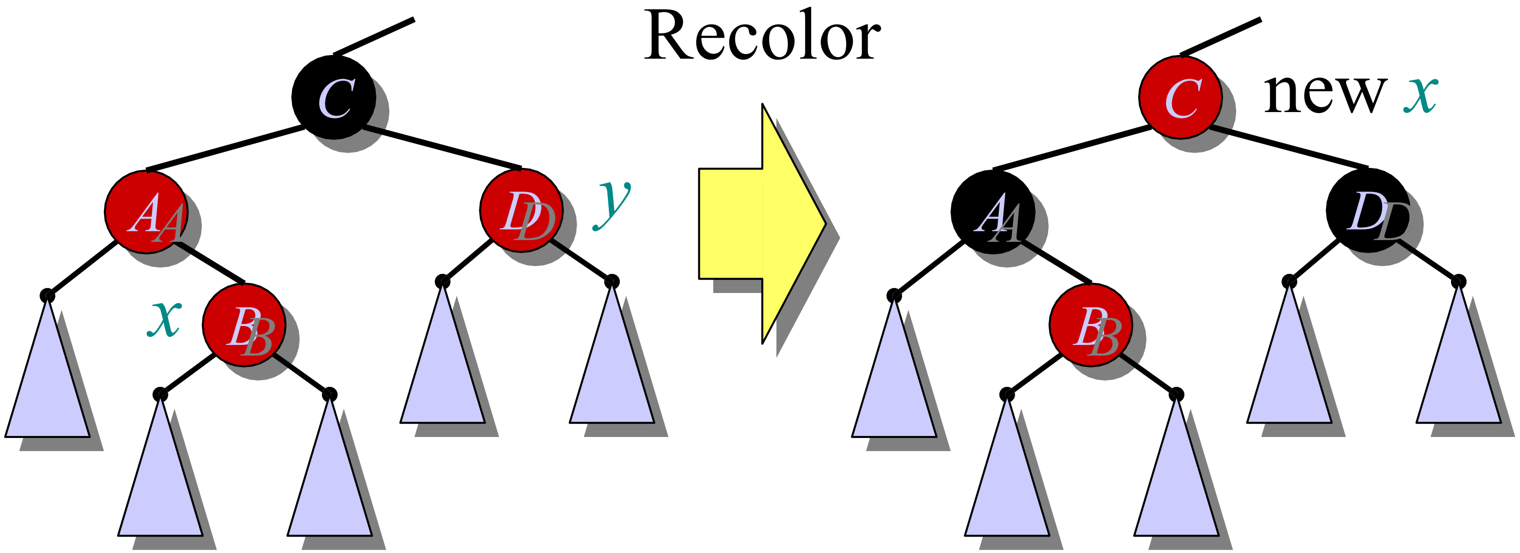
Graphical notation

Let  denote a subtree with a black root.

All ’s have the same black-height.

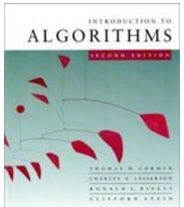
Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Case 1

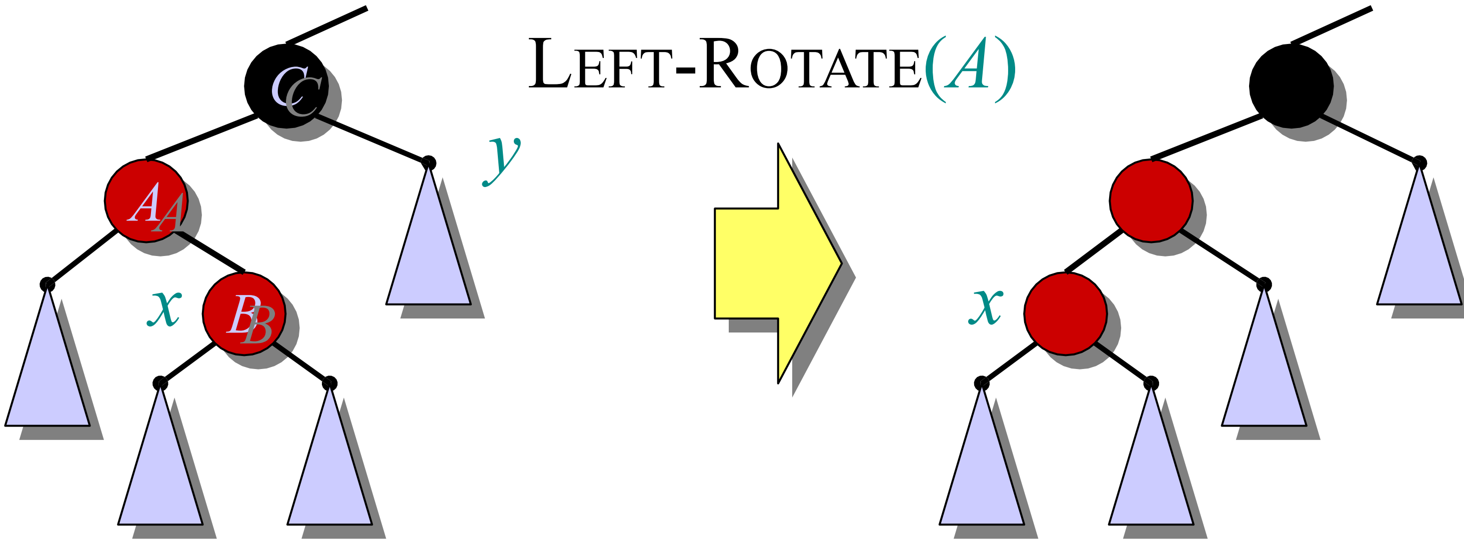


(Or, children of *A* are swapped.)

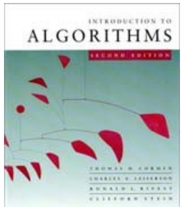
Push *C*'s black onto *A* and *D*, and recurse, since *C*'s parent may be red.



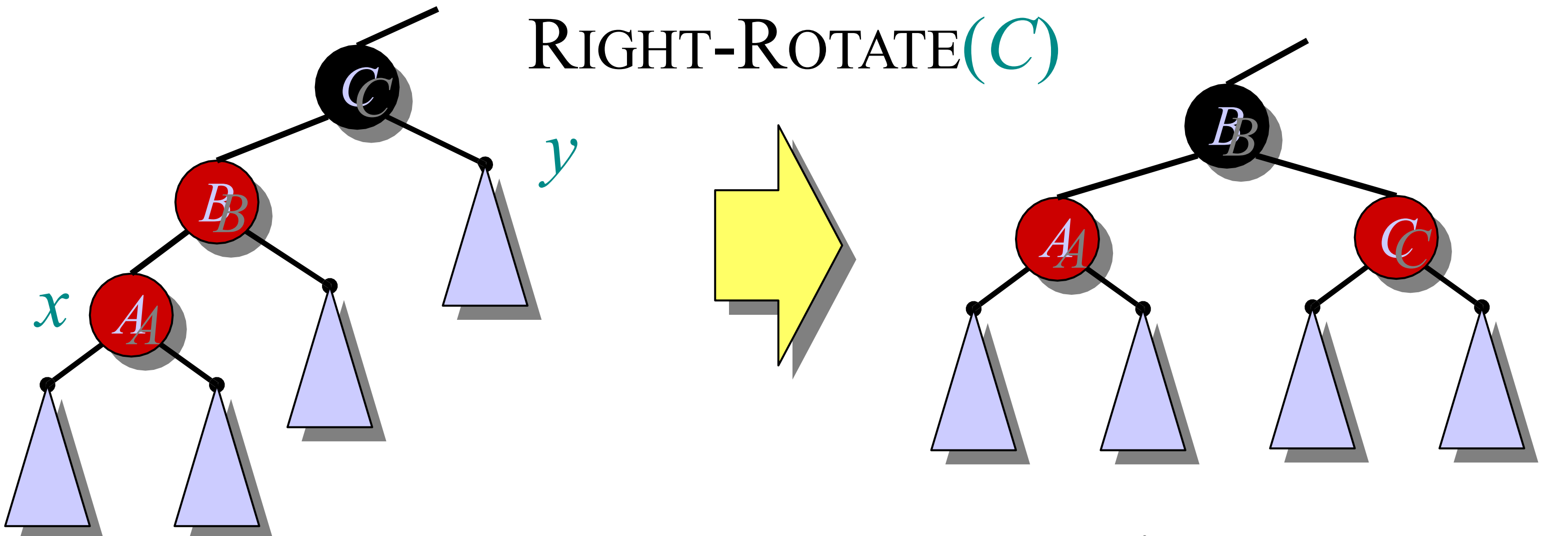
Case 2



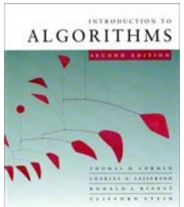
Transform to Case 3.



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).

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

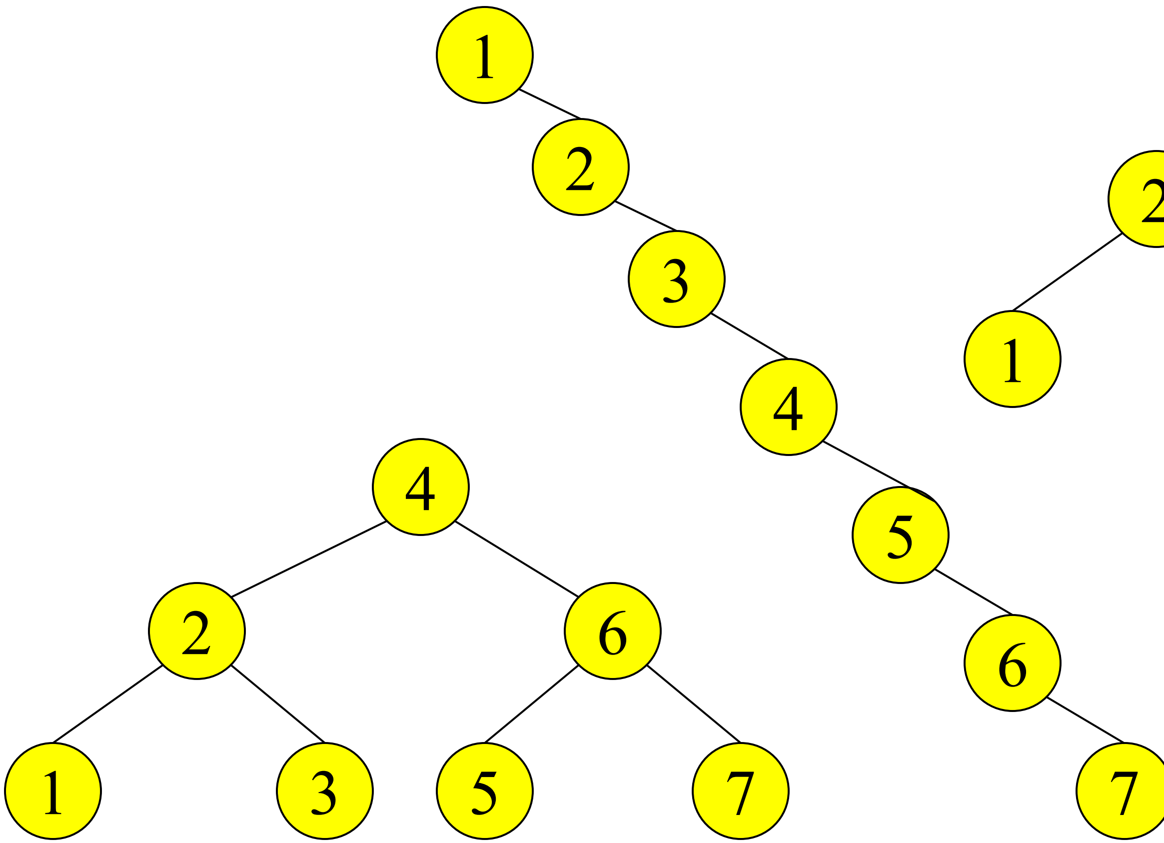
Binary Search Tree - Best Time

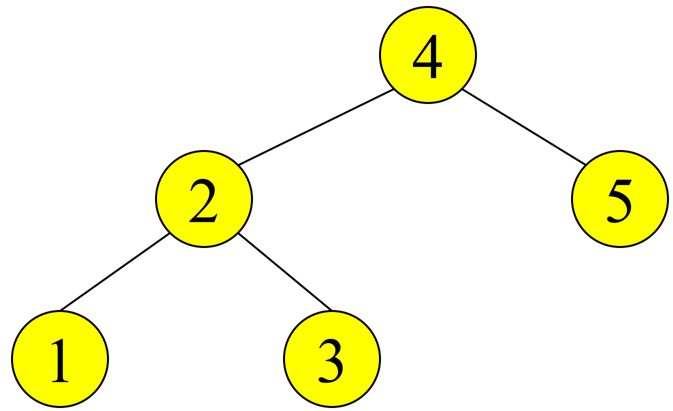
- All BST operations are O(d), where d is tree depth
- minimum d is $d = \lfloor \log_2 N \rfloor$ for a binary tree with N nodes
  - › What is the best case tree?
  - › What is the worst case tree?
- So, best case running time of BST operations is O(log N)

Binary Search Tree - Worst Time

- Worst case running time is O(N)
  - › What happens when you Insert elements in ascending order?
    - Insert: 2, 4, 6, 8, 10, 12 into an empty BST
  - › Problem: Lack of "balance":
    - compare depths of left and right subtree
  - › Unbalanced degenerate tree

Balanced and unbalanced BST





Is this “balanced”?

Approaches to balancing trees

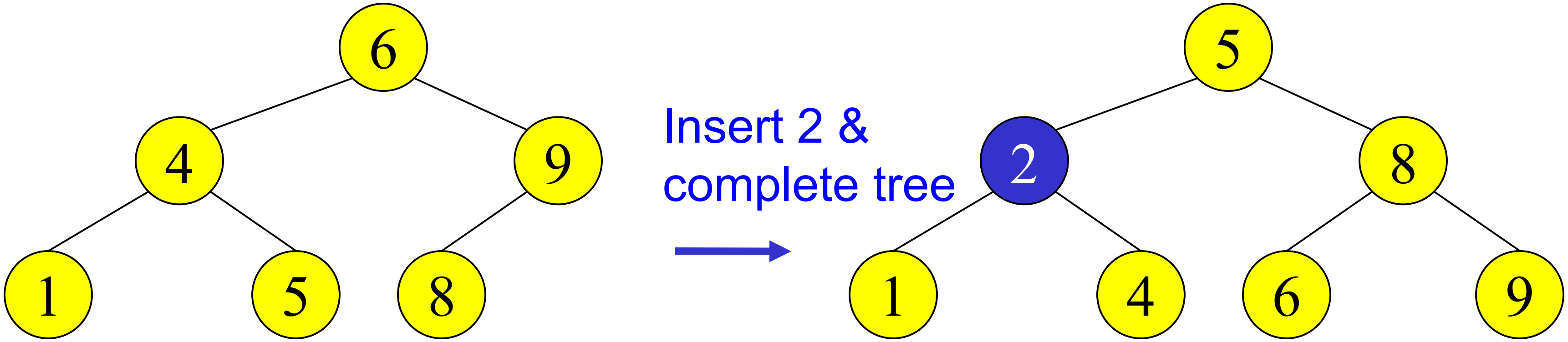
- Don't balance
  - May end up with some nodes very deep
- Strict balance
  - The tree must always be balanced perfectly
- Pretty good balance
  - Only allow a little out of balance
- Adjust on access
  - Self-adjusting

Balancing Binary Search Trees

- Many algorithms exist for keeping binary search trees balanced
  - Adelson-Velskii and Landis (AVL) trees (height-balanced trees)
  - Splay trees and other self-adjusting trees
  - B-trees and other multiway search trees

Perfect Balance

- Want a complete tree after every operation
  - › tree is full except possibly in the lower right
- This is expensive
  - › For example, insert 2 in the tree on the left and then rebuild as a complete tree

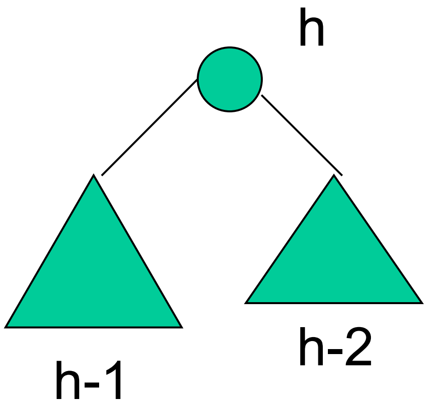


AVL - Good but not Perfect Balance

- AVL trees are height-balanced binary search trees
- Balance factor of a node
  - height(left subtree) - height(right subtree)
- An AVL tree has balance factor calculated at every node
  - For every node, heights of left and right subtree can differ by no more than 1
  - Store current heights in each node

Height of an AVL Tree

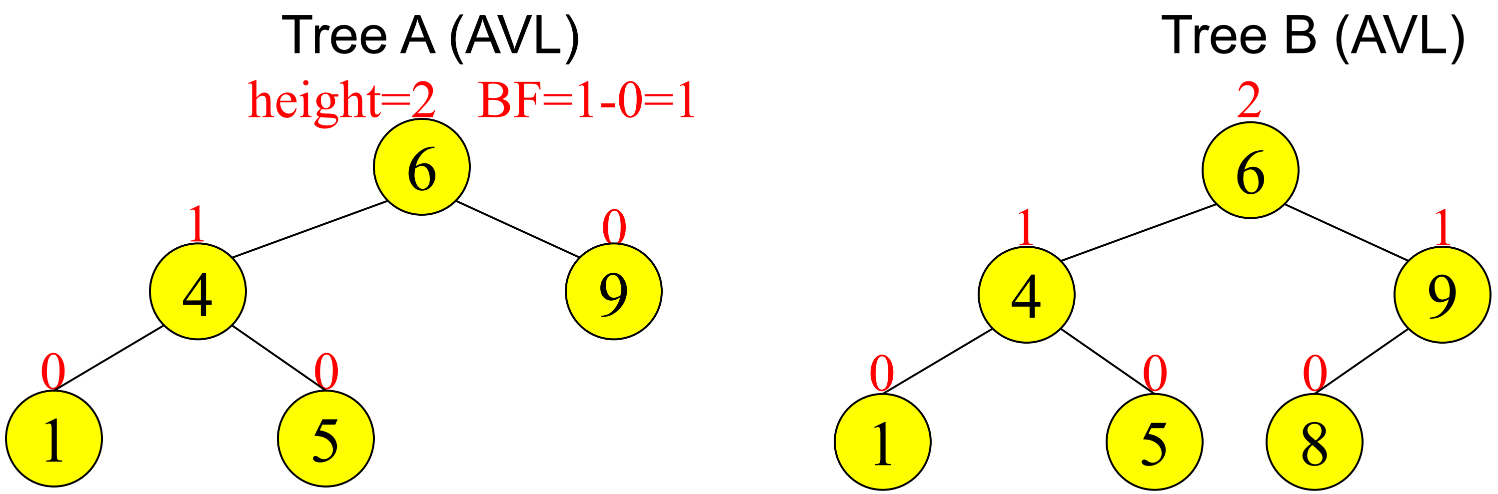
- N(h) = minimum number of nodes in an AVL tree of height h.
- Basis
  - › $N(0) = 1, N(1) = 2$
- Induction
  - › $N(h) = N(h-1) + N(h-2) + 1$
- Solution (recall Fibonacci analysis)
  - › $N(h) \geq \phi^h \quad (\phi \approx 1.62)$



Height of an AVL Tree

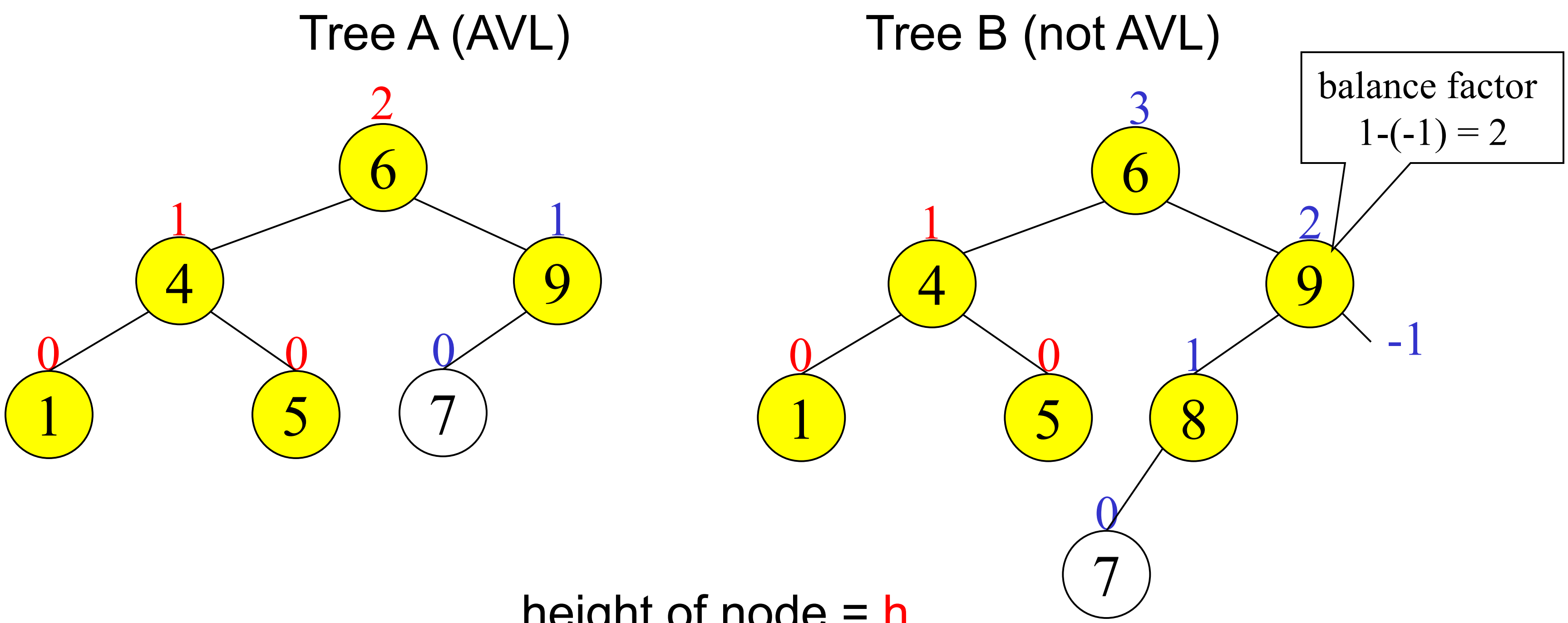
- $N(h) \geq \phi^h$ ($\phi \approx 1.62$)
- Suppose we have n nodes in an AVL tree of height h.
  - $n \geq N(h)$ (because N(h) was the minimum)
  - $n \geq \phi^h$ hence $\log_\phi n \geq h$ (relatively well balanced tree!!)
  - $h \leq 1.44 \log_2 n$ (i.e., Find takes O(logn))

Node Heights



height of node = h
balance factor = h_{left}-h_{right}
empty height = -1

Node Heights after Insert 7



height of node = h

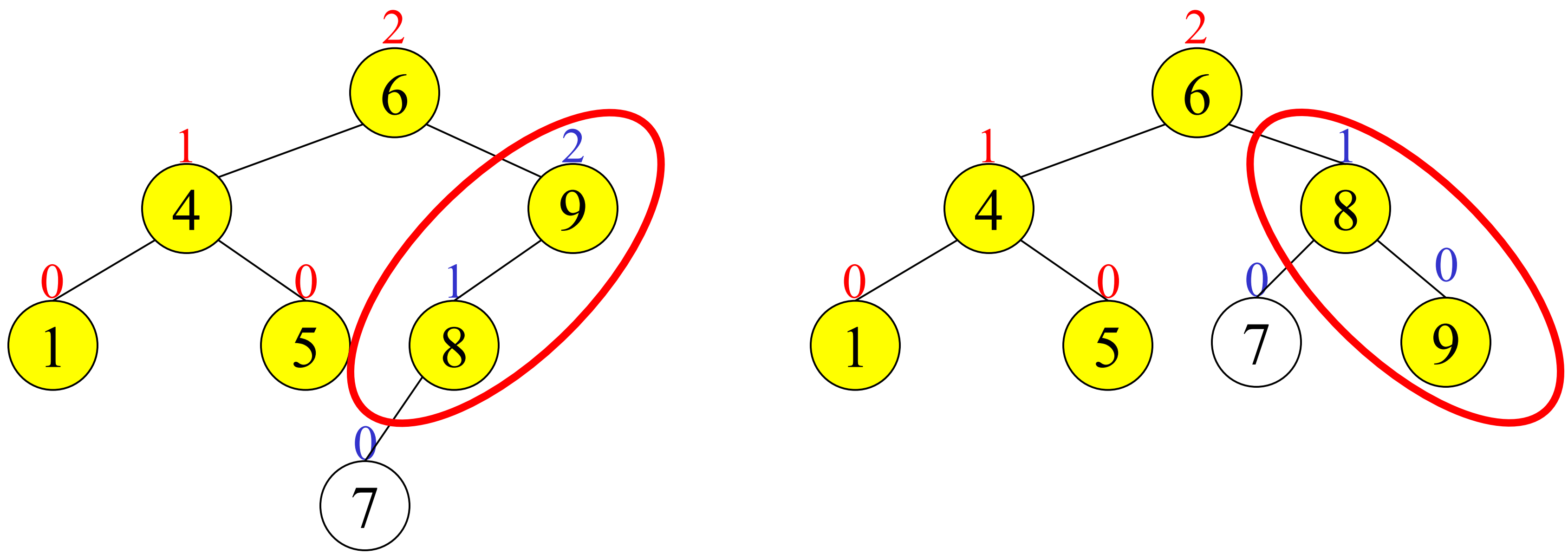
balance factor = h_{left}-h_{right}

empty height = -1

Insert and Rotation in AVL Trees

- Insert operation may cause balance factor to become 2 or –2 for some node
  - only nodes on the path from insertion point to root node have possibly changed in height
  - So after the Insert, go back up to the root node by node, updating heights
  - If a new balance factor (the difference h_{left}-h_{right}) is 2 or –2, adjust tree by *rotation* around the node

Single Rotation in an AVL Tree



Insertions in AVL Trees

Let the node that needs rebalancing be α.

There are 4 cases:

Outside Cases (require single rotation) :

1. Insertion into left subtree of left child of α.
2. Insertion into right subtree of right child of α.

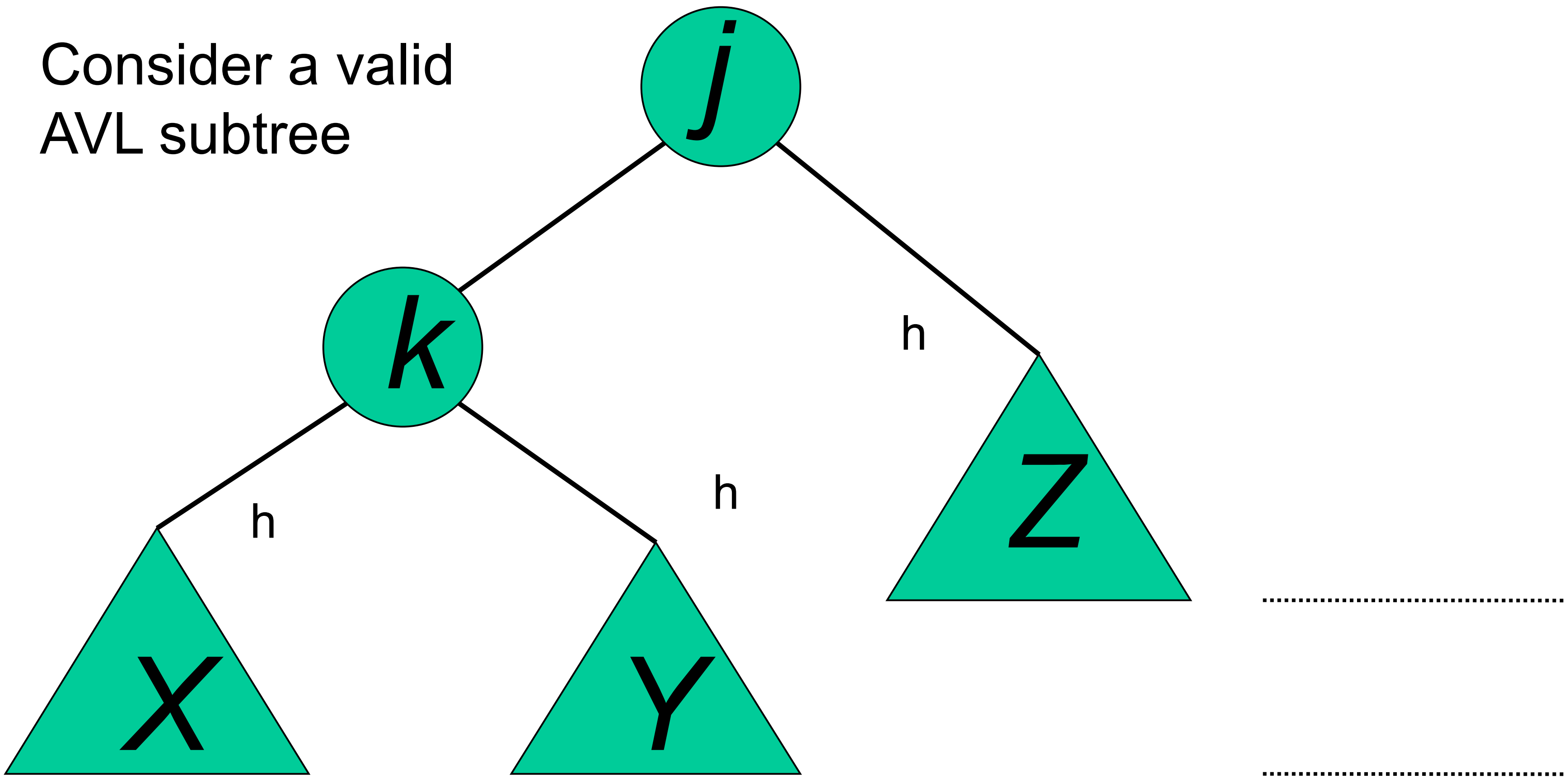
Inside Cases (require double rotation) :

3. Insertion into right subtree of left child of α.
4. Insertion into left subtree of right child of α.

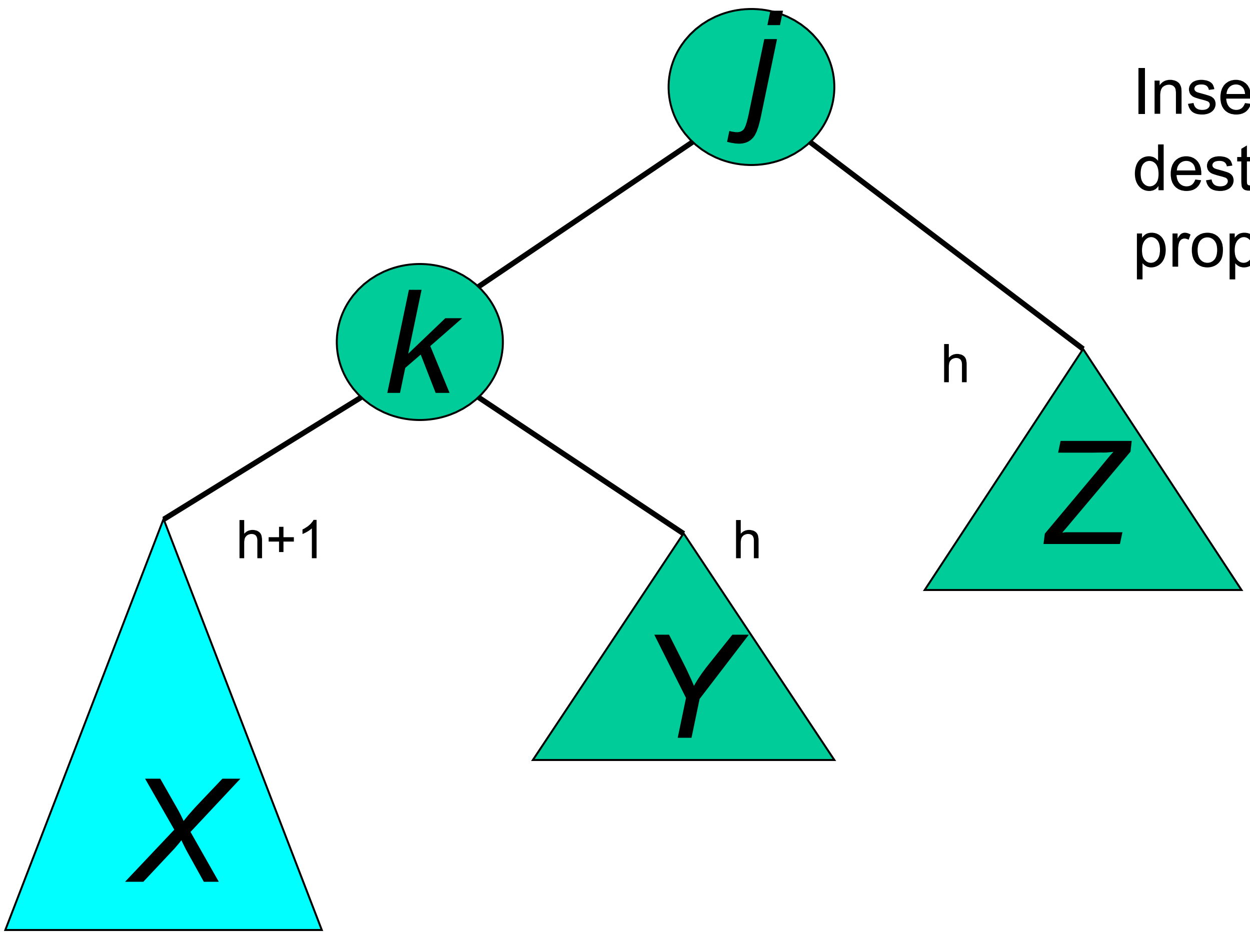
The rebalancing is performed through four separate rotation algorithms.

AVL Insertion: Outside Case

Consider a valid
AVL subtree

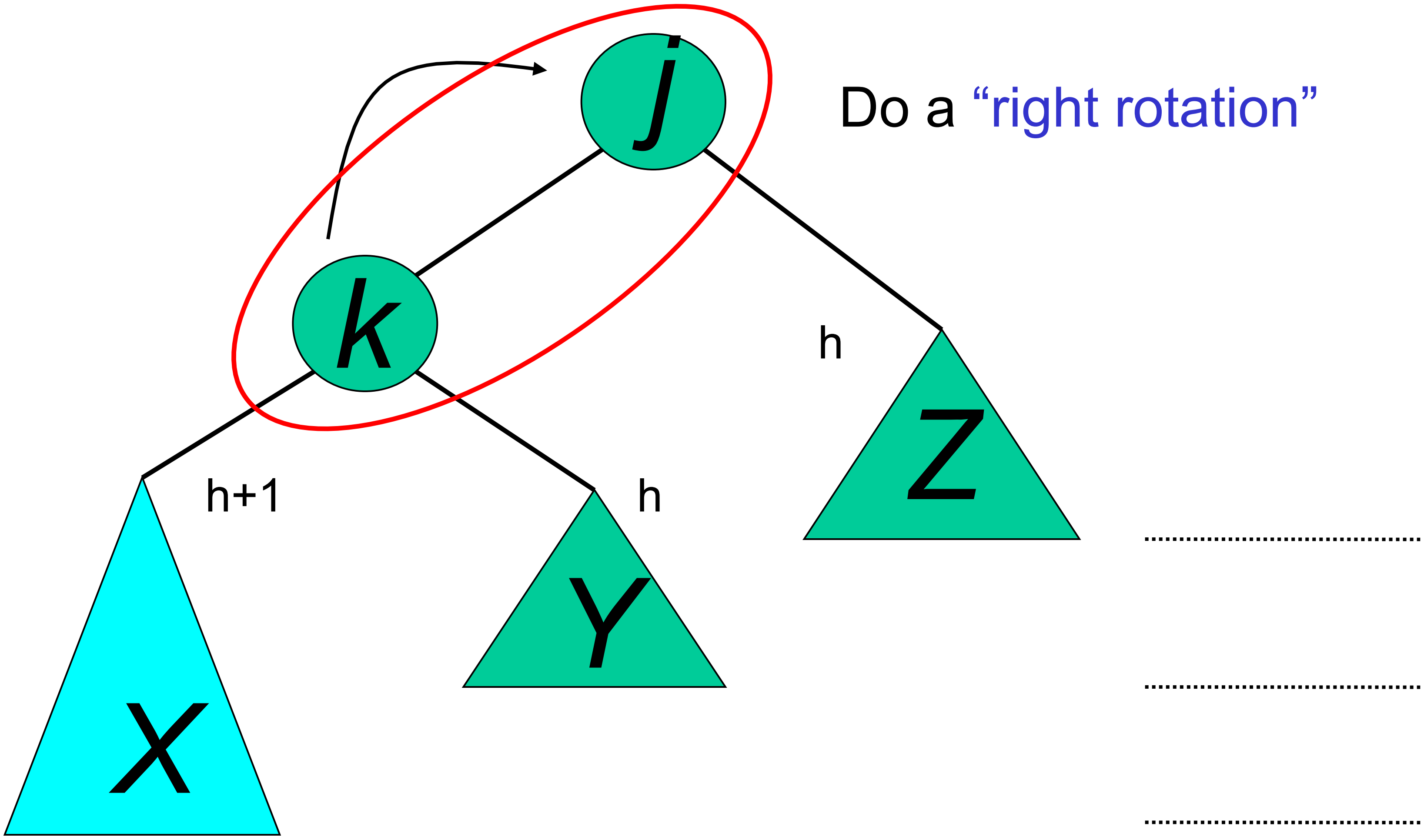


AVL Insertion: Outside Case



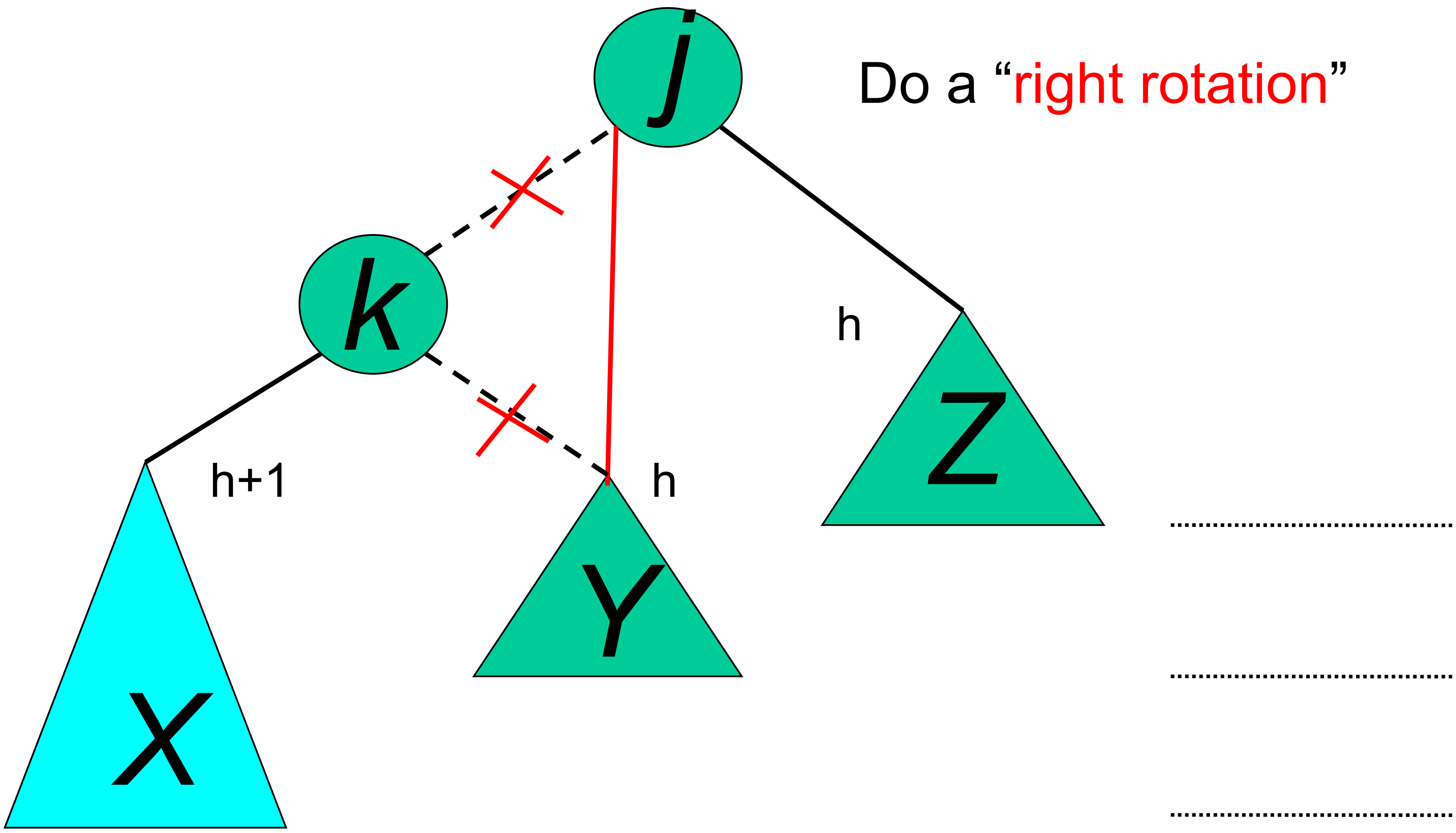
Inserting into X destroys the AVL property at node j

AVL Insertion: Outside Case

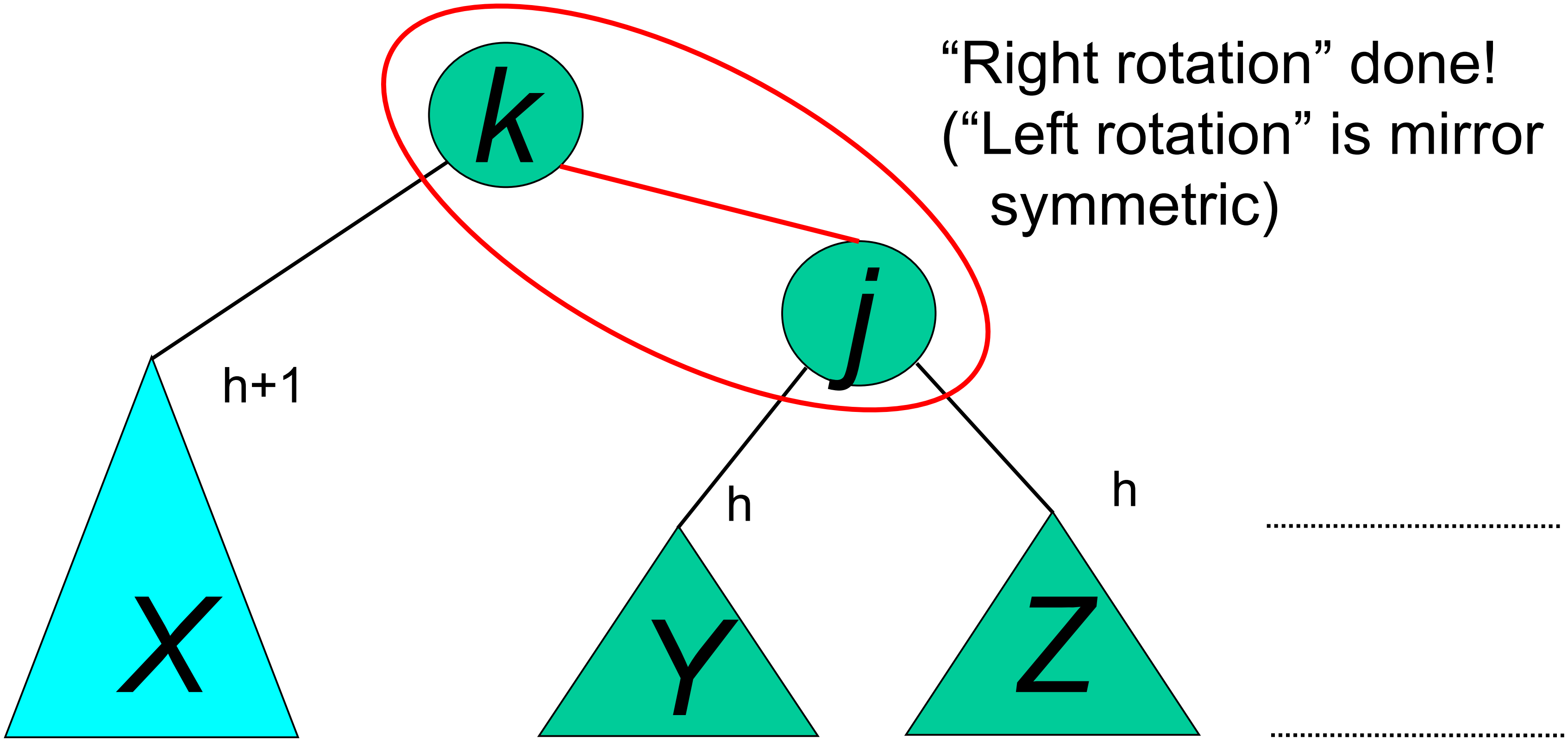


Do a "right rotation"

Single right rotation



Outside Case Completed

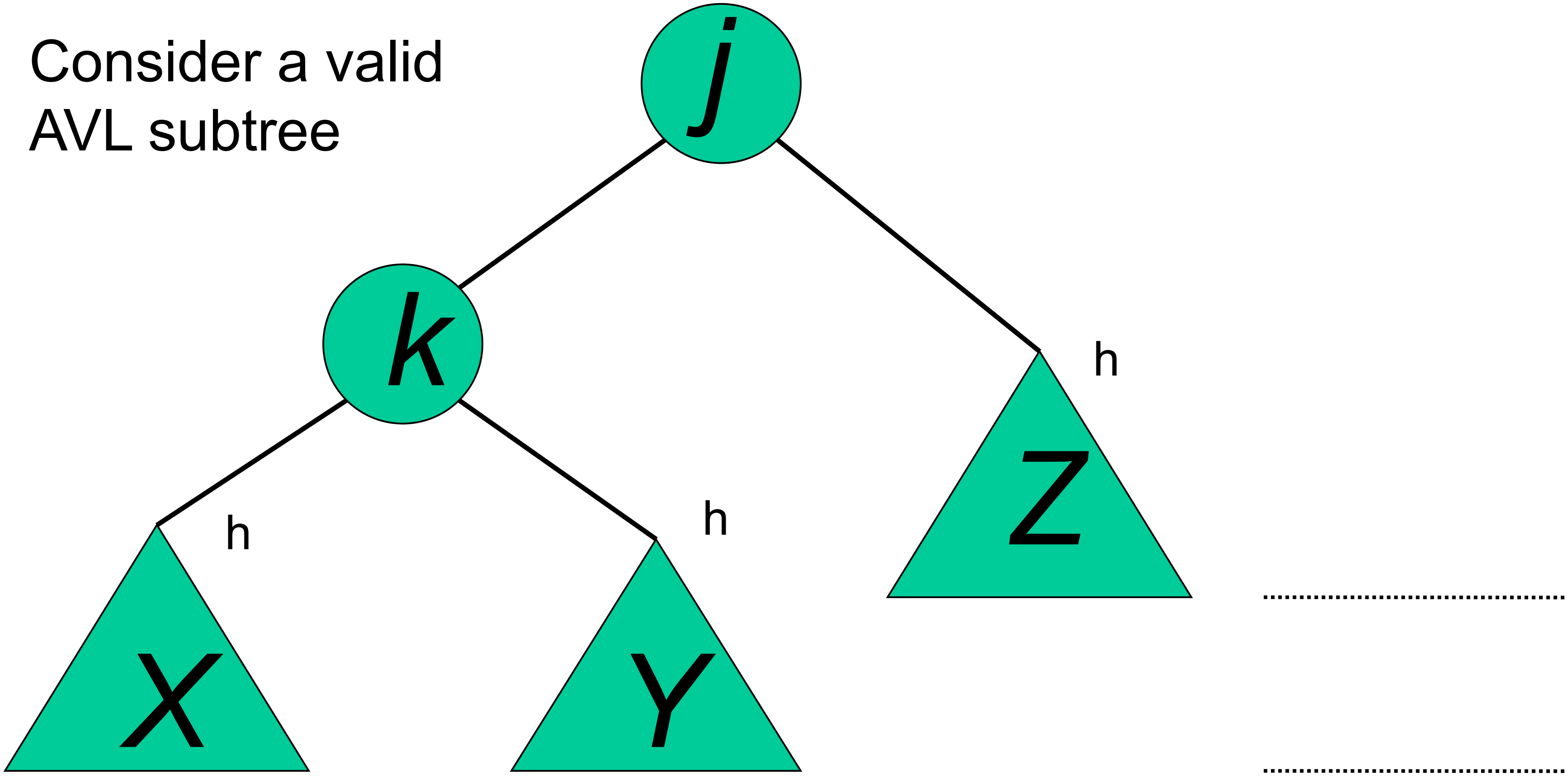


"Right rotation" done!
("Left rotation" is mirror symmetric)

AVL property has been restored!

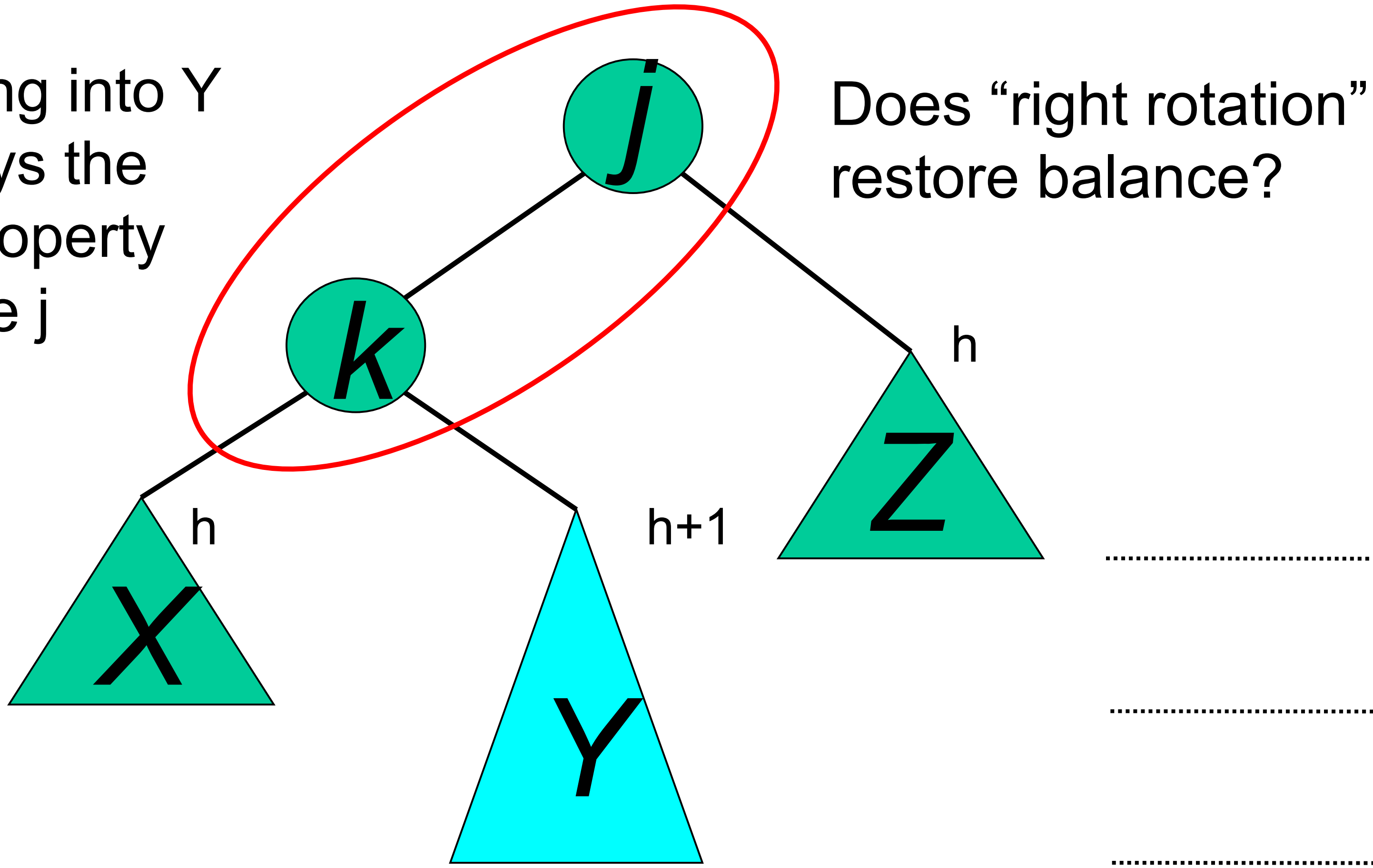
AVL Insertion: Inside Case

Consider a valid
AVL subtree



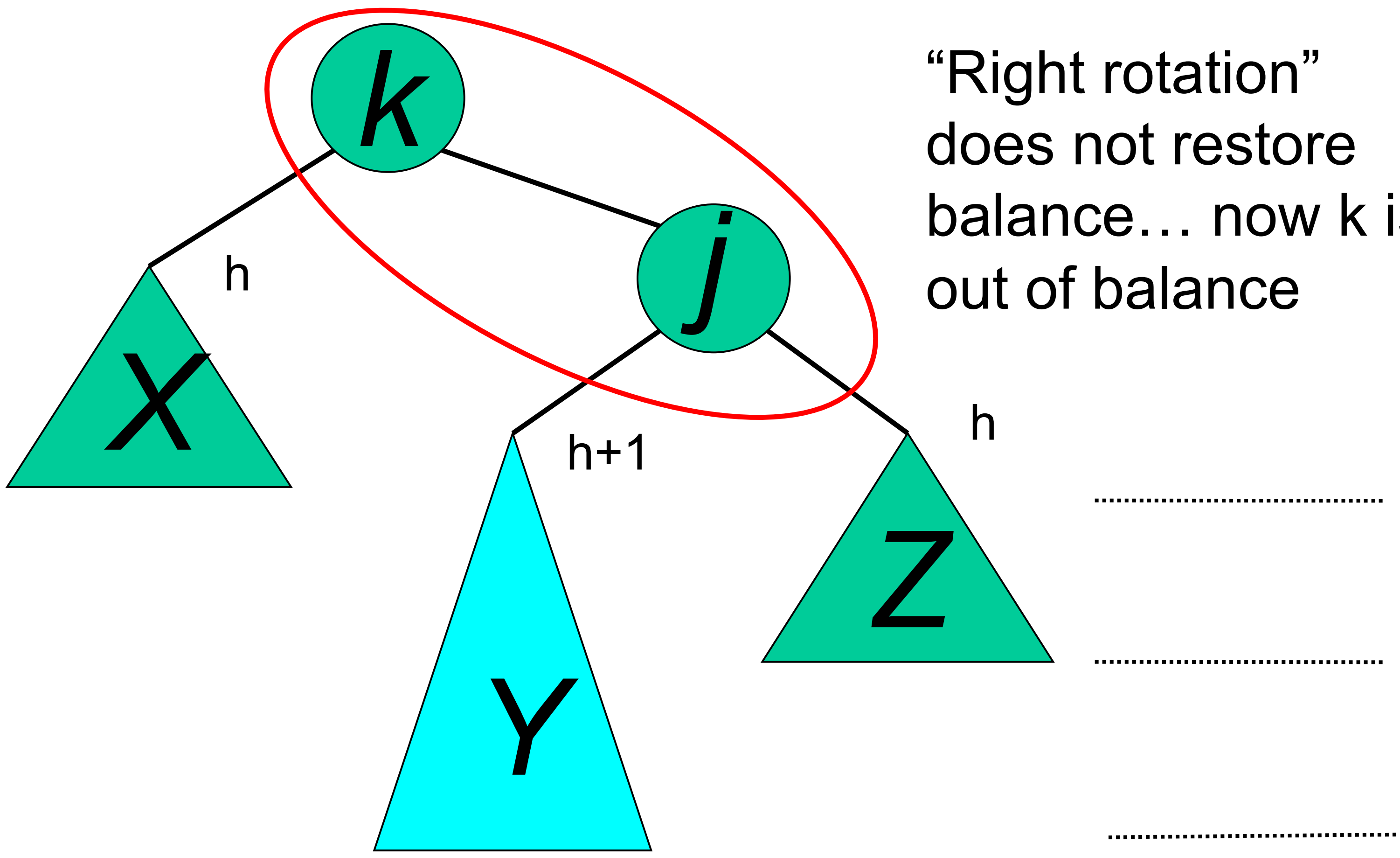
AVL Insertion: Inside Case

Inserting into Y destroys the AVL property at node j



Does "right rotation" restore balance?

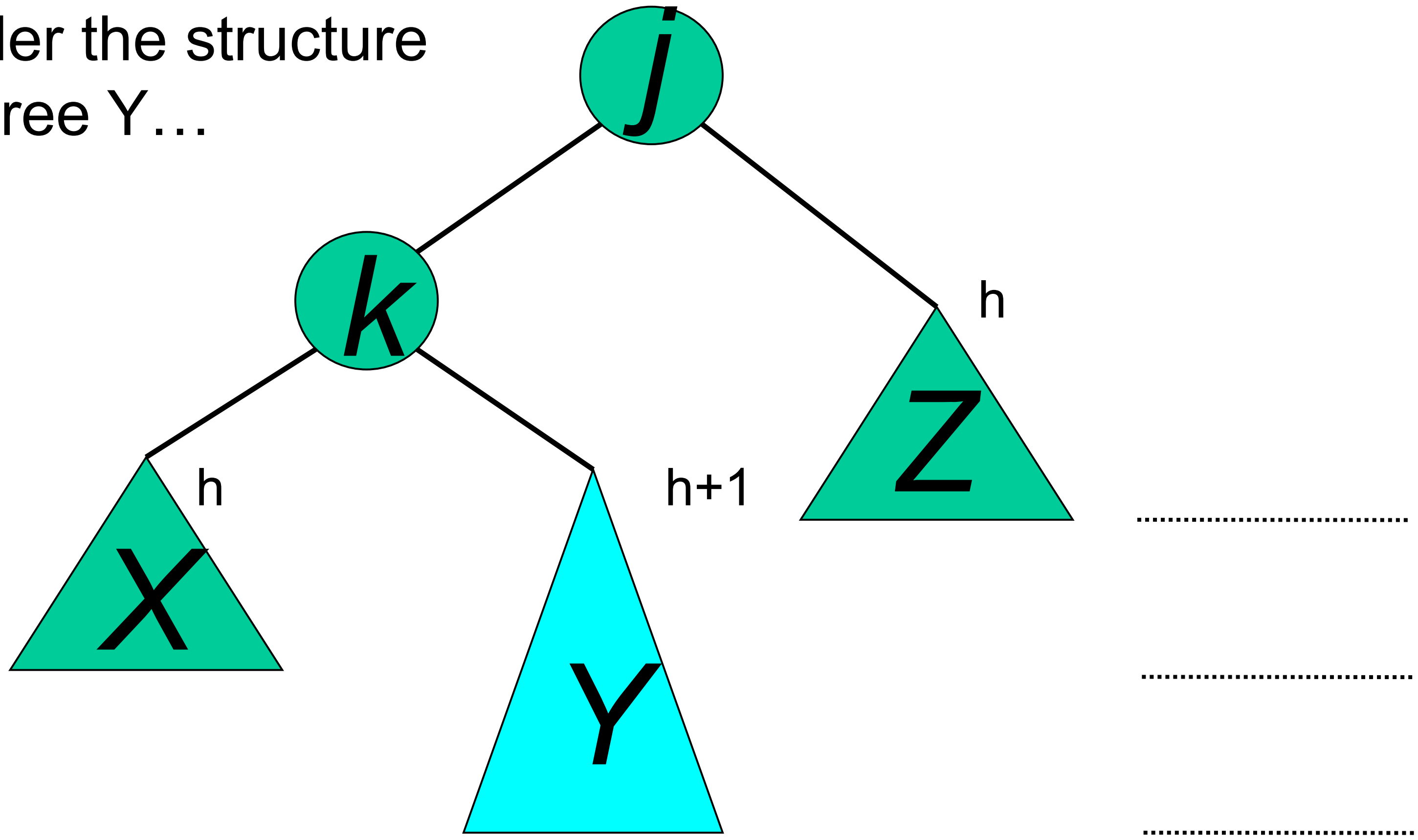
AVL Insertion: Inside Case



"Right rotation" does not restore balance… now k is out of balance

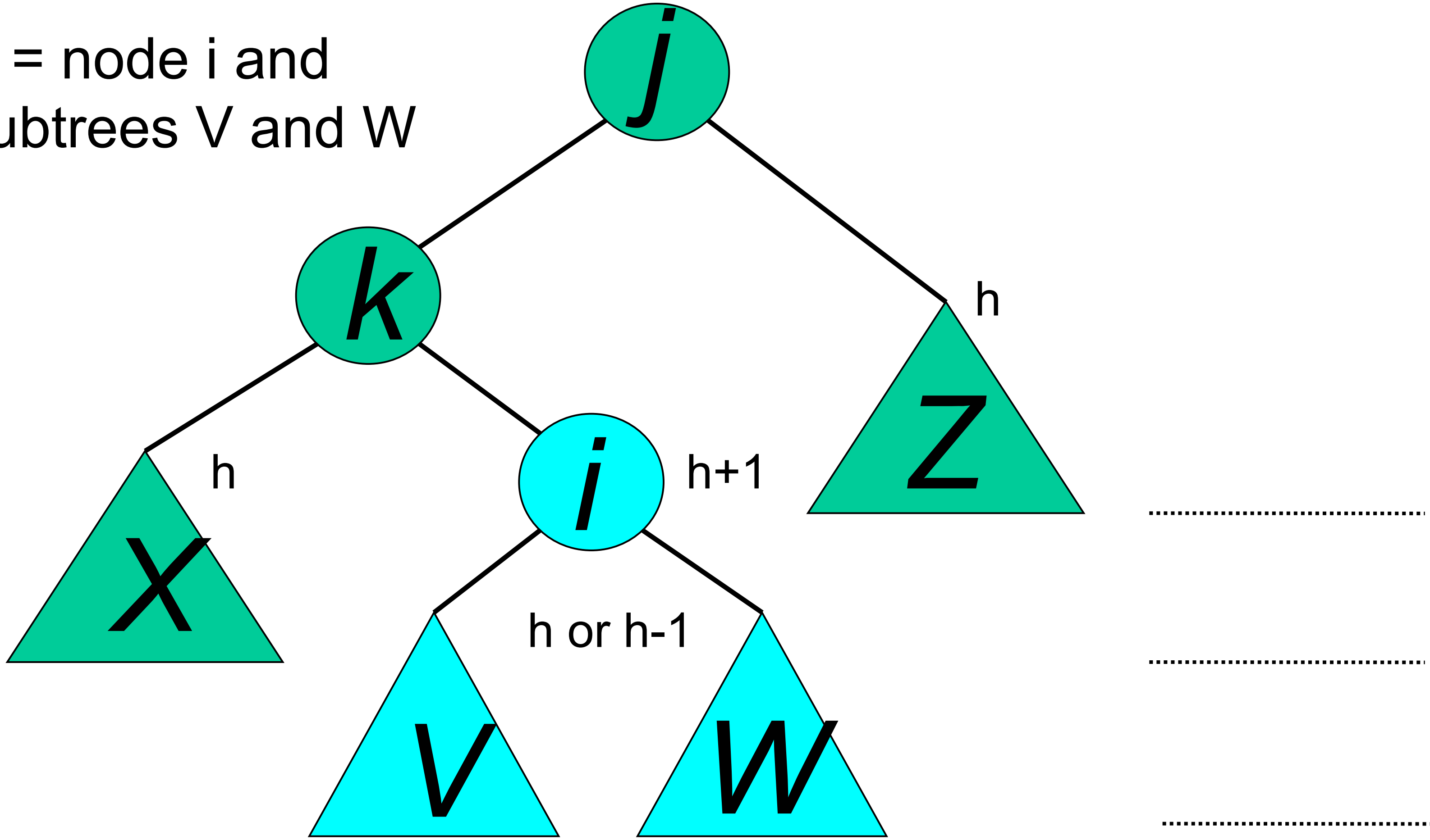
AVL Insertion: Inside Case

Consider the structure of subtree Y…

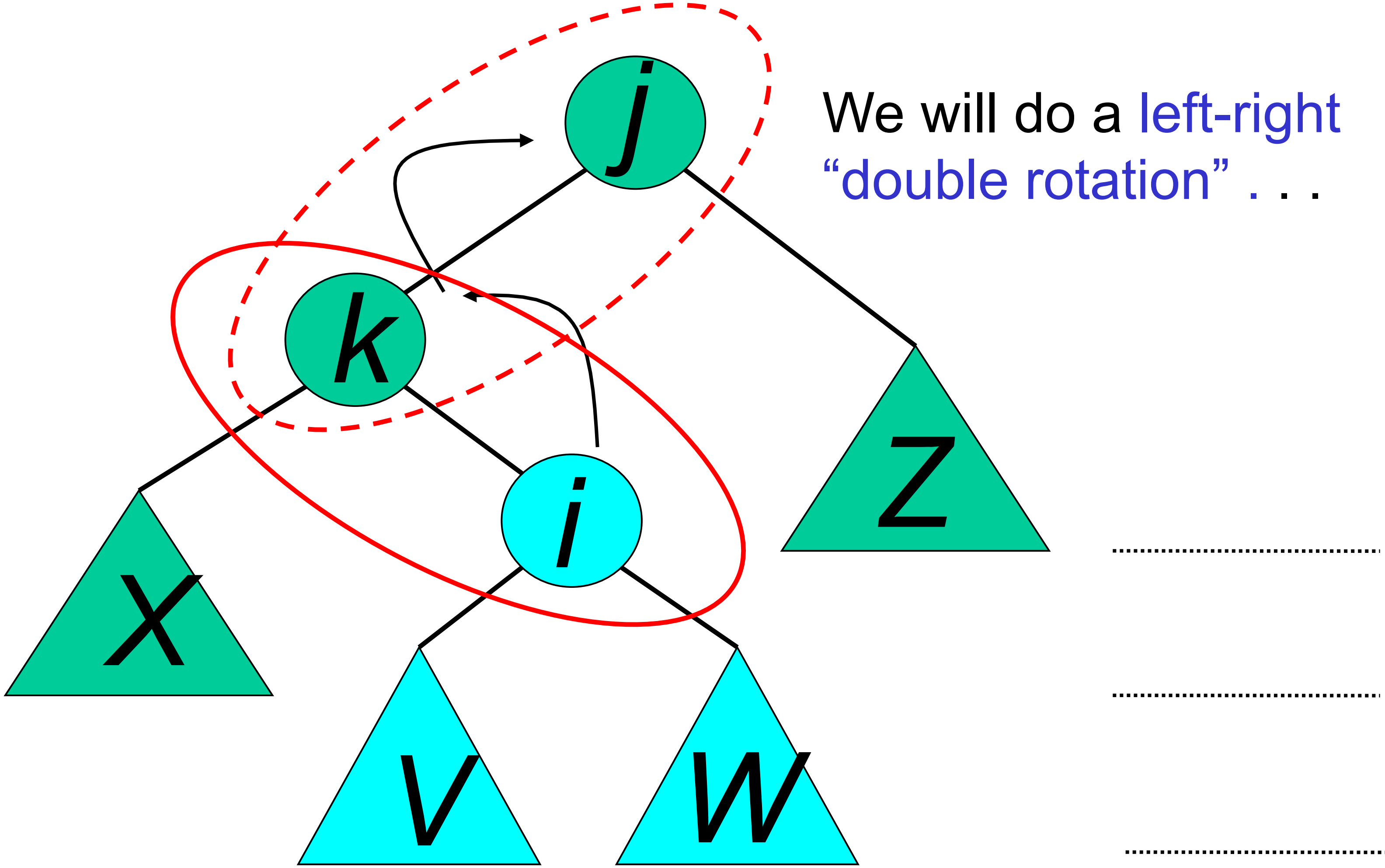


AVL Insertion: Inside Case

Y = node i and subtrees V and W

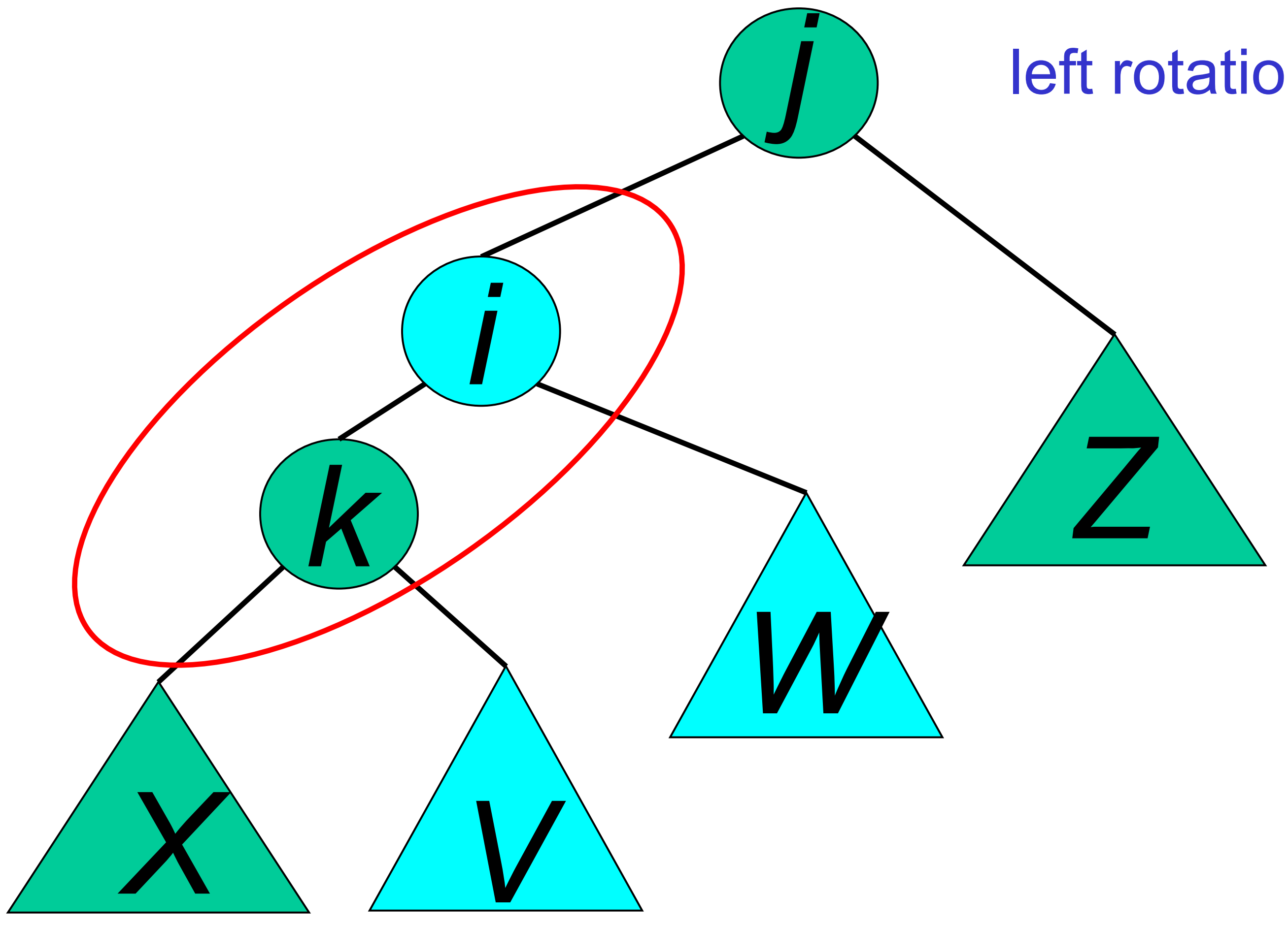


AVL Insertion: Inside Case



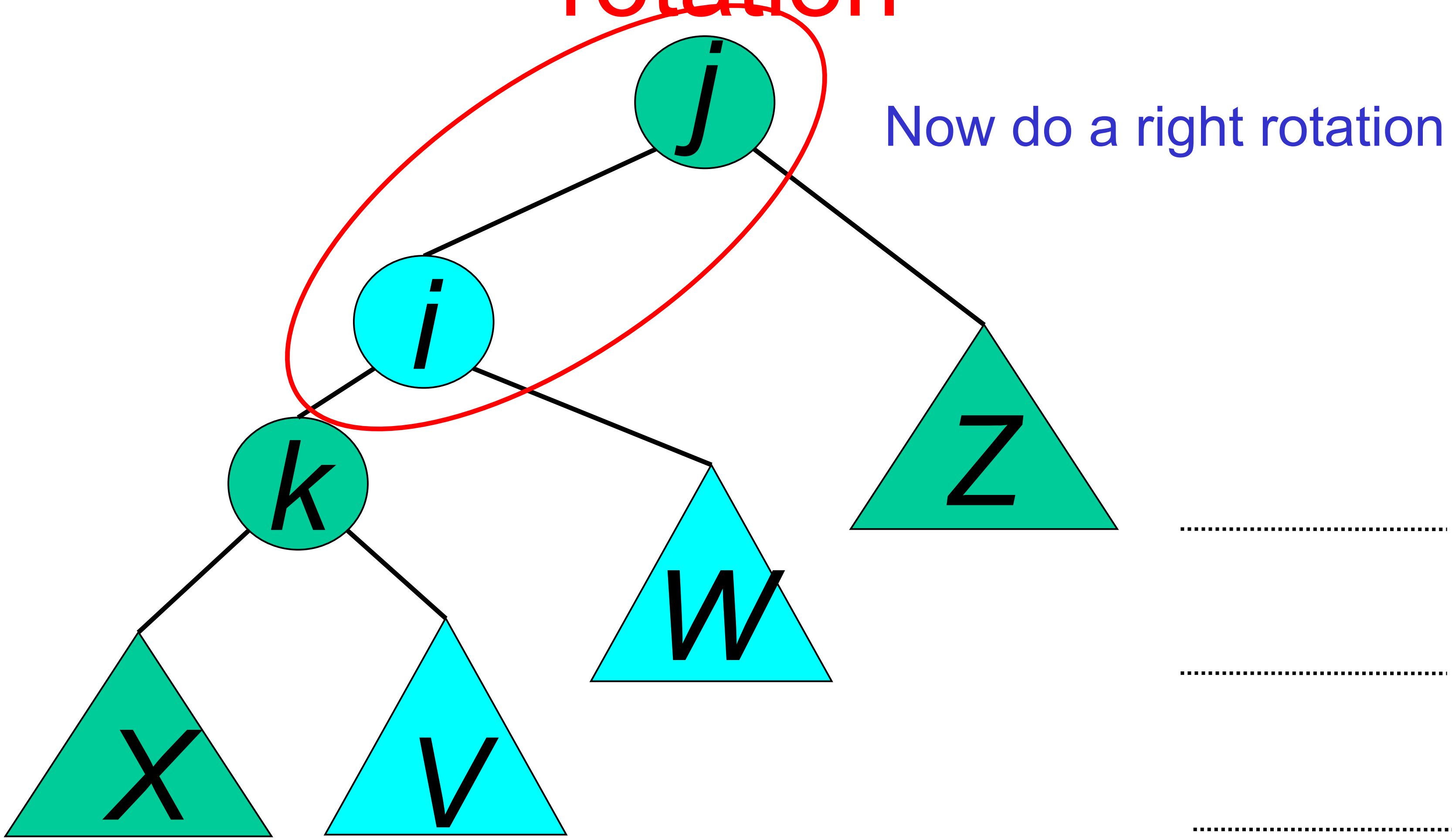
We will do a left-right "double rotation" . . .

Double rotation : first rotation



left rotation complete

Double rotation : second rotation

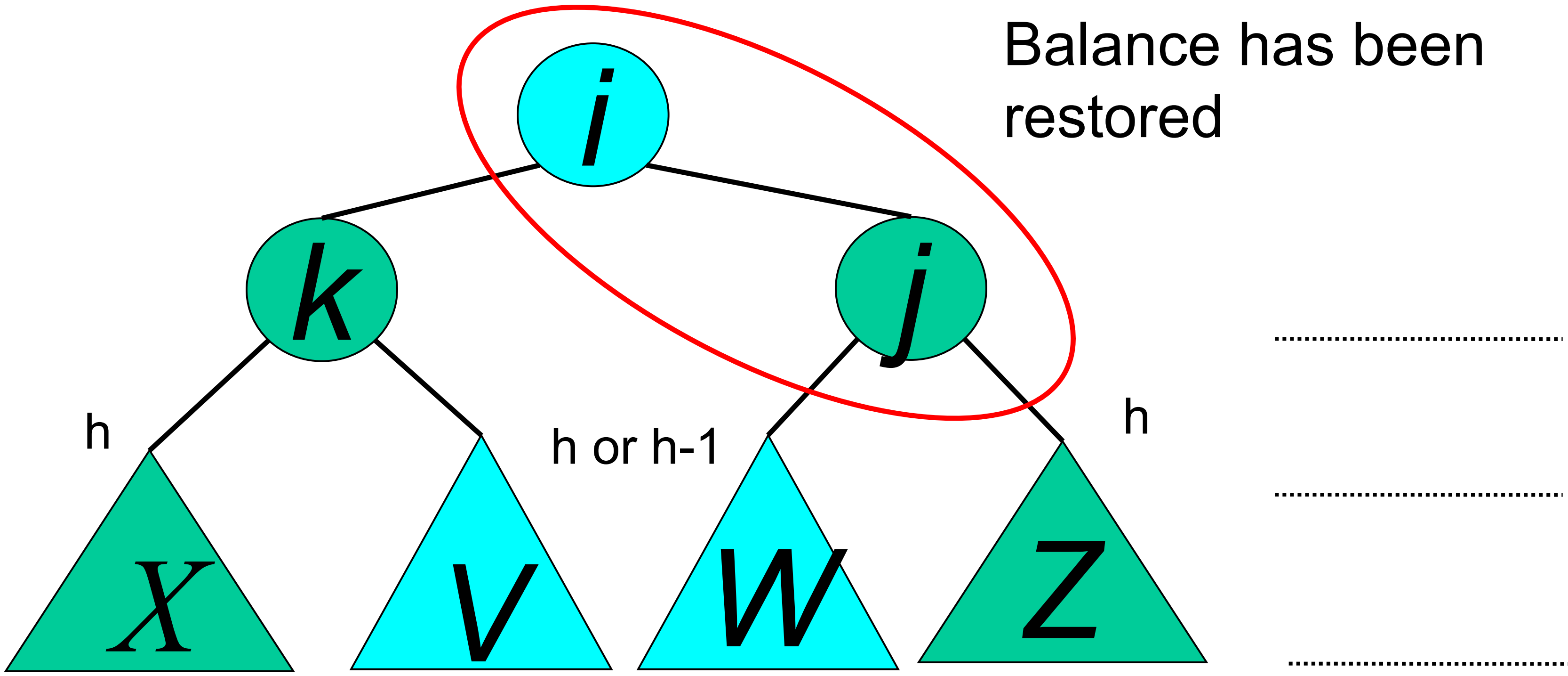


Now do a right rotation

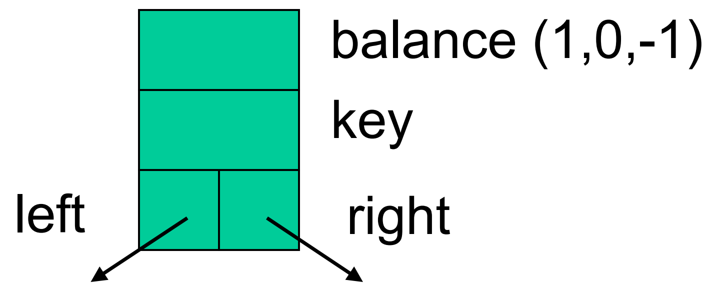
Double rotation : second rotation

right rotation complete

Balance has been restored



Implementation



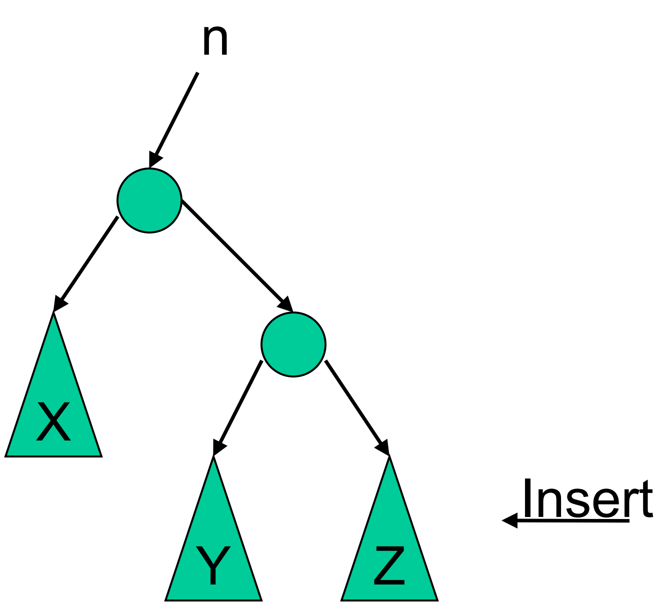
No need to keep the height; just the difference in height, i.e. the balance factor; this has to be modified on the path of insertion even if you don't perform rotations

Once you have performed a rotation (single or double) you won't need to go back up the tree

Single Rotation

```
RotateFromRight(n : reference node pointer) {
p : node pointer;
p := n.right;
n.right := p.left;
p.left := n;
n := p
}
```

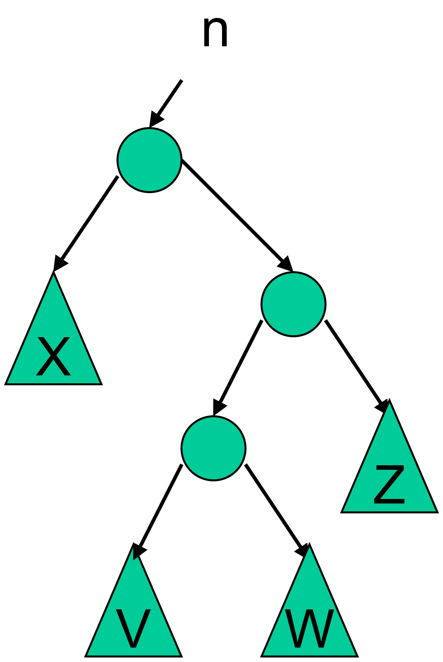
You also need to modify the heights or balance factors of n and p



Double Rotation

- Implement Double Rotation in two lines.

```
DoubleRotateFromRight(n : reference node pointer) {
????
}
```



Insertion in AVL Trees

- Insert at the leaf (as for all BST)
  - only nodes on the path from insertion point to root node have possibly changed in height
  - So after the Insert, go back up to the root node by node, updating heights
  - If a new balance factor (the difference h_{left}-h_{right}) is 2 or –2, adjust tree by *rotation* around the node

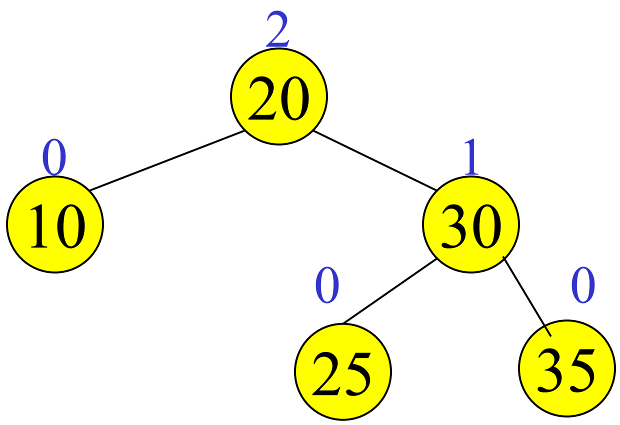
Insert in BST

```
Insert(T : reference tree pointer, x : element) : integer {
if T = null then
  T := new tree; T.data := x; return 1;//the links to
                                 //children are null
case
  T.data = x : return 0; //Duplicate do nothing
  T.data > x : return Insert(T.left, x);
  T.data < x : return Insert(T.right, x);
endcase
}
```

Insert in AVL trees

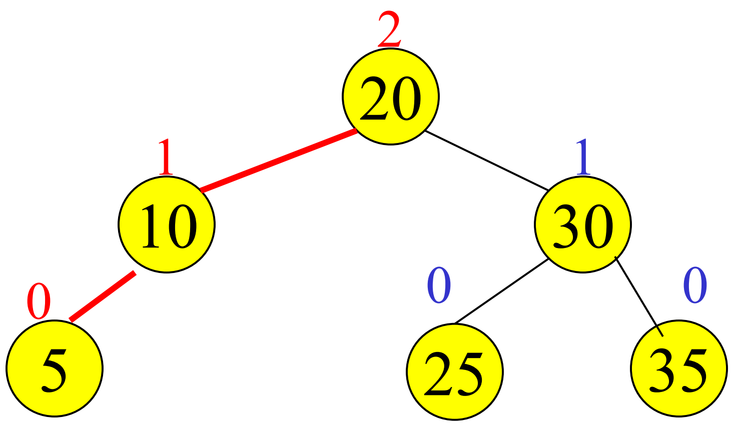
```
Insert(T : reference tree pointer, x : element) : {
if T = null then
  {T := new tree; T.data := x; height := 0; return;}
case
  T.data = x : return ; //Duplicate do nothing
  T.data > x : Insert(T.left, x);
               if ((height(T.left)- height(T.right)) = 2){
                  if (T.left.data > x ) then //outside case
                         T = RotatefromLeft (T);
                  else                       //inside case
                         T = DoubleRotatefromLeft (T);}
  T.data < x :  Insert(T.right, x);
                code similar to the left case
Endcase
  T.height := max(height(T.left),height(T.right)) +1;
  return;
}
```

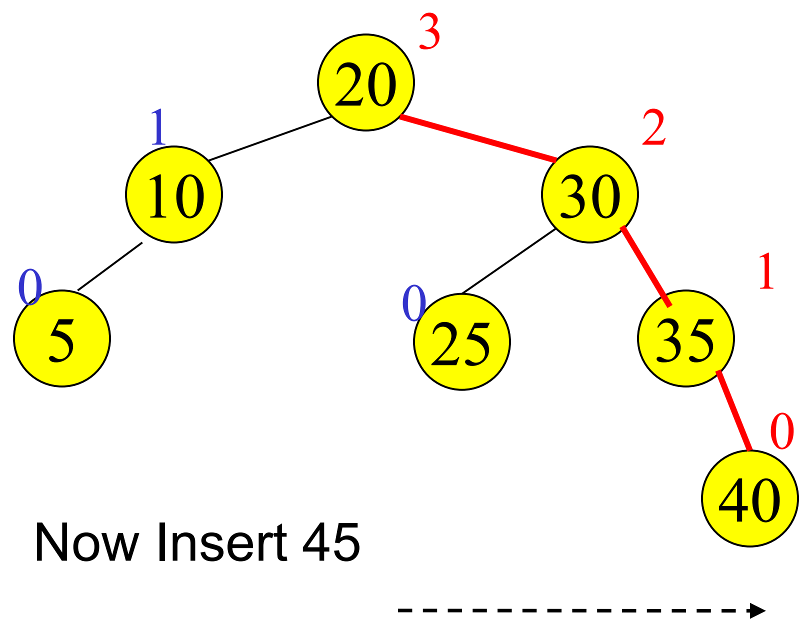
Example of Insertions in an AVL Tree



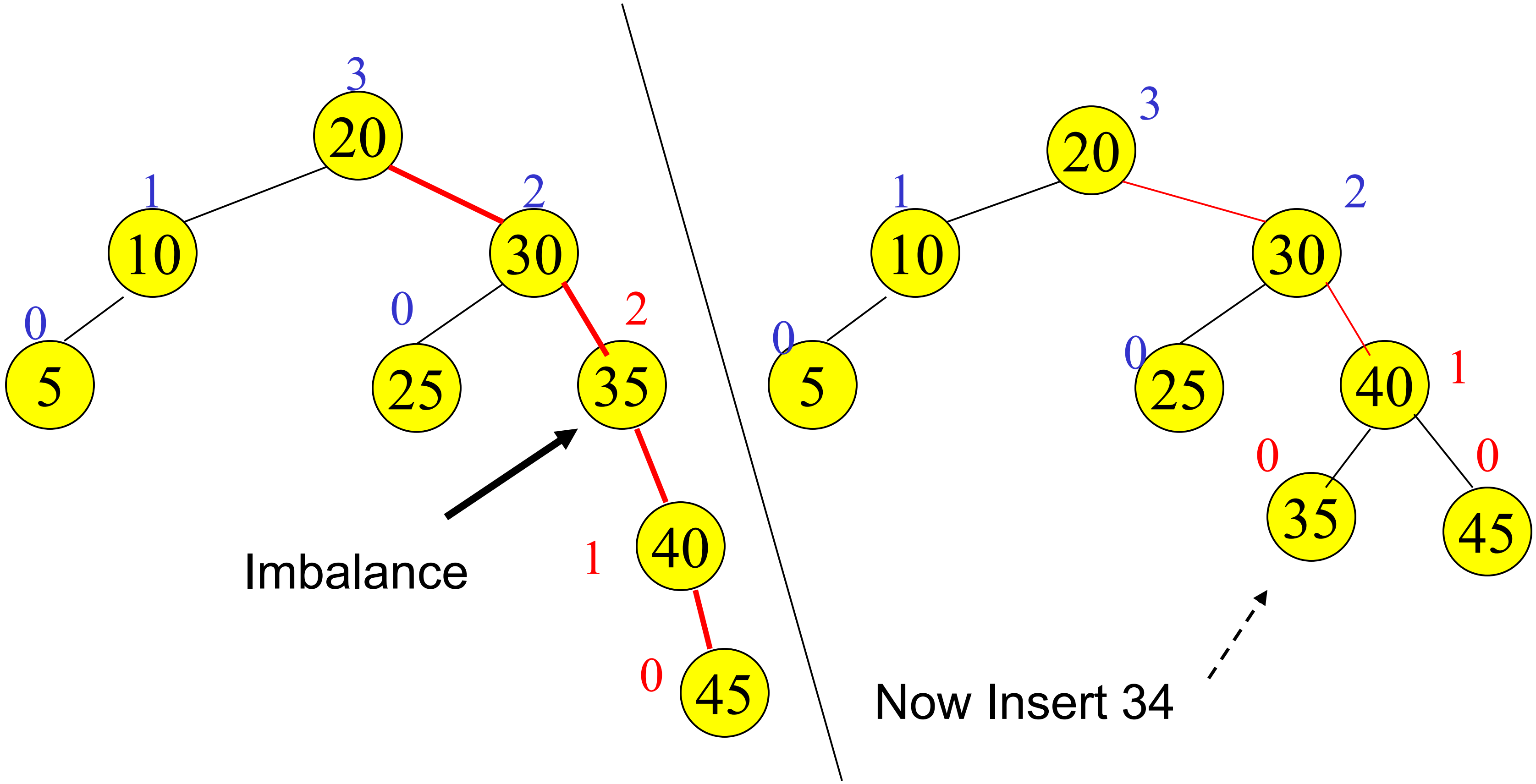
Insert 5, 40

Example of Insertions in an AVL Tree

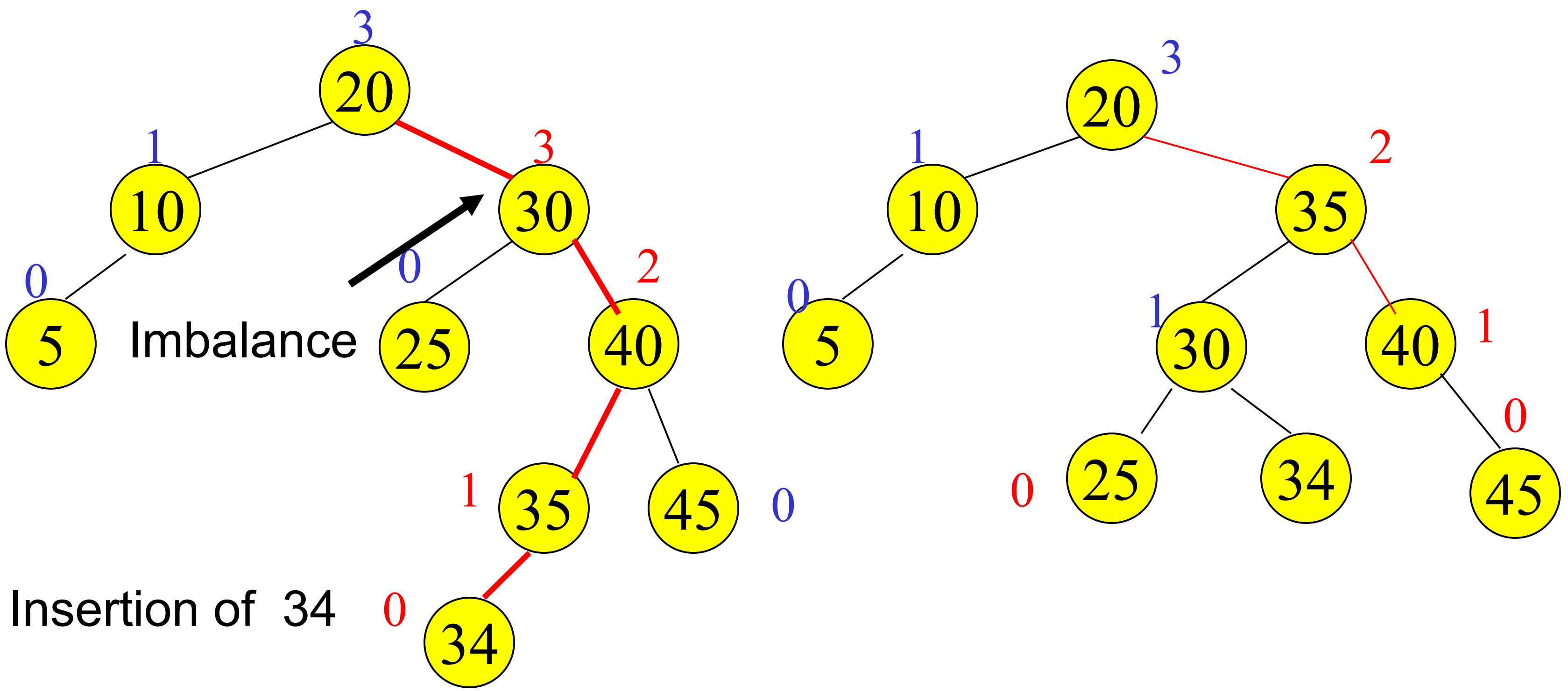




Single rotation (outside case)



Double rotation (inside case)



AVL Tree Deletion

- Similar but more complex than insertion
  - Rotations and double rotations needed to rebalance
  - Imbalance may propagate upward so that many rotations may be needed.

Pros and Cons of AVL Trees

Arguments for AVL trees:

1. Search is O(log N) since AVL trees are always balanced.
2. Insertion and deletions are also O(logn)
3. The height balancing adds no more than a constant factor to the speed of insertion.

Arguments against using AVL trees:
1. Difficult to program & debug; more space for balance factor.
2. Asymptotically faster but rebalancing costs time.
3. Most large searches are done in database systems on disk and use other structures (e.g. B-trees).
4. May be OK to have O(N) for a single operation if total run time for many consecutive operations is fast (e.g. Splay trees).

Double Rotation Solution

```
DoubleRotateFromRight(n : reference node pointer) {
RotateFromLeft(n.right);
RotateFromRight(n);
}
```