# CSCI 340 Data Structures and Algorithm Analysis

## (Height) Balanced Binary Search Tree

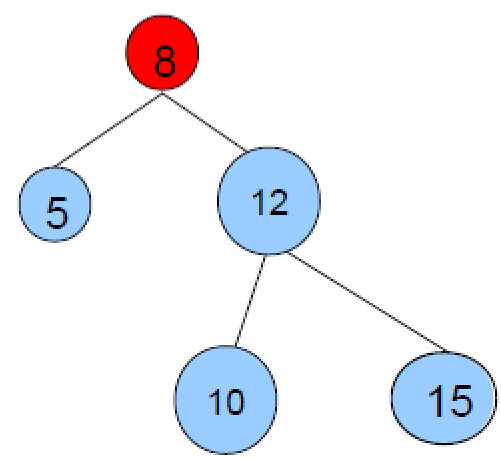
## and

## Balancing a Binary Search Tree

Minmei Hou, Ph.D., Dept. Computer Science, NIU

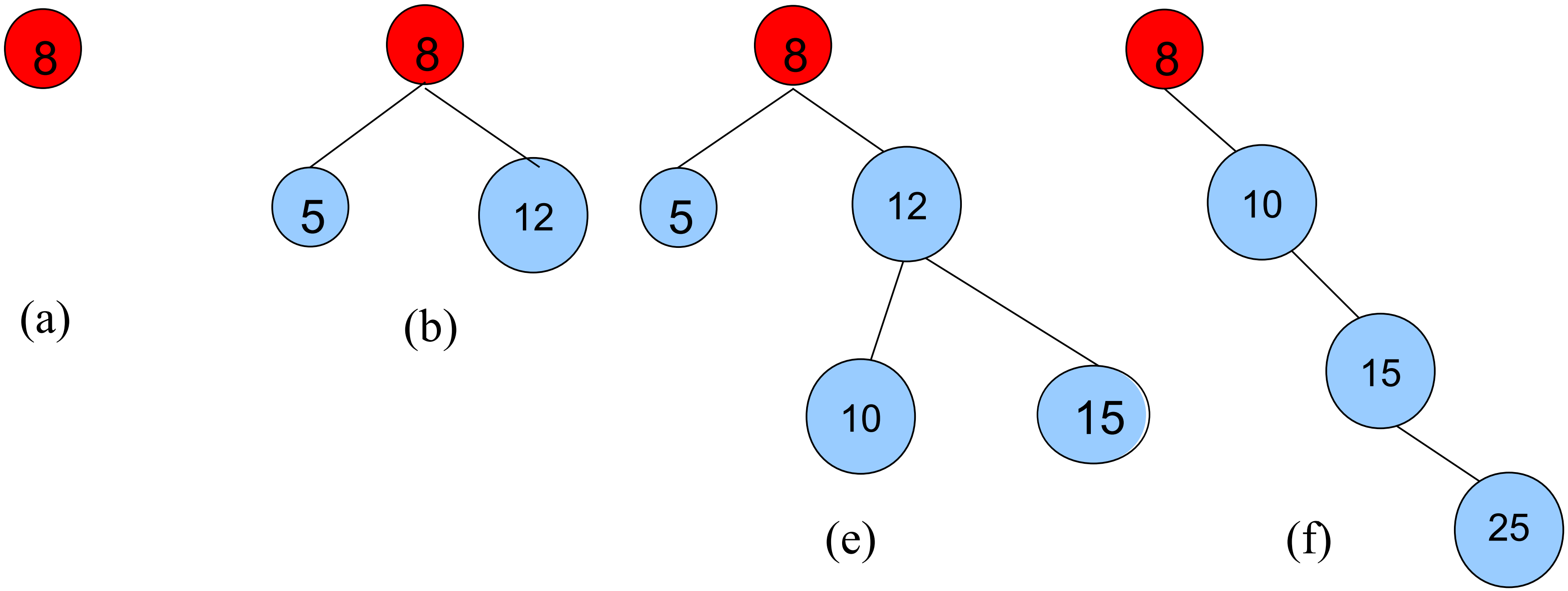
# What is height (of a tree)

- Depth of a node
  - The number of edges from the root to the node
- Height of a node
  - The number of edges from the node to the deepest leaf
- Level of a node
  - 1+depth of the node



# Height (of a tree)

- The height of tree is the height of the root
  - Maximum depth of a tree
  - An empty tree has height -1
  - A tree of a single node has height 0



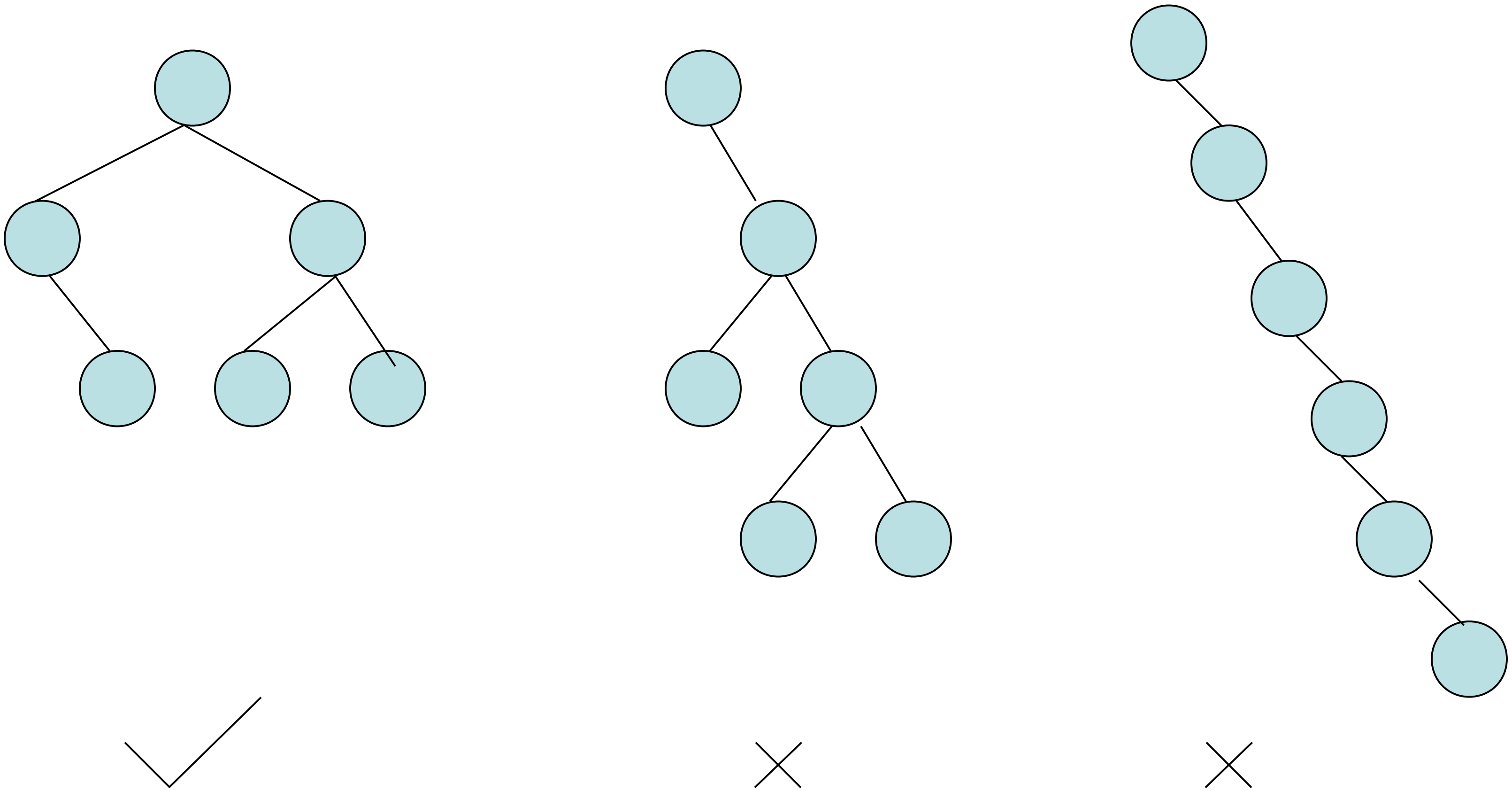

(a) (b) (e) (f)

# (height) balanced B.S.T.

- A binary tree is (height-)balanced if the difference in heights of both subtrees of any node in the tree is 0, -1, or 1.
- A balanced B.S.T. is a B.S.T. that is (height) balanced.
- The height of a balanced binary tree is O(lg$n$) of the size of the tree, where $n$ is size

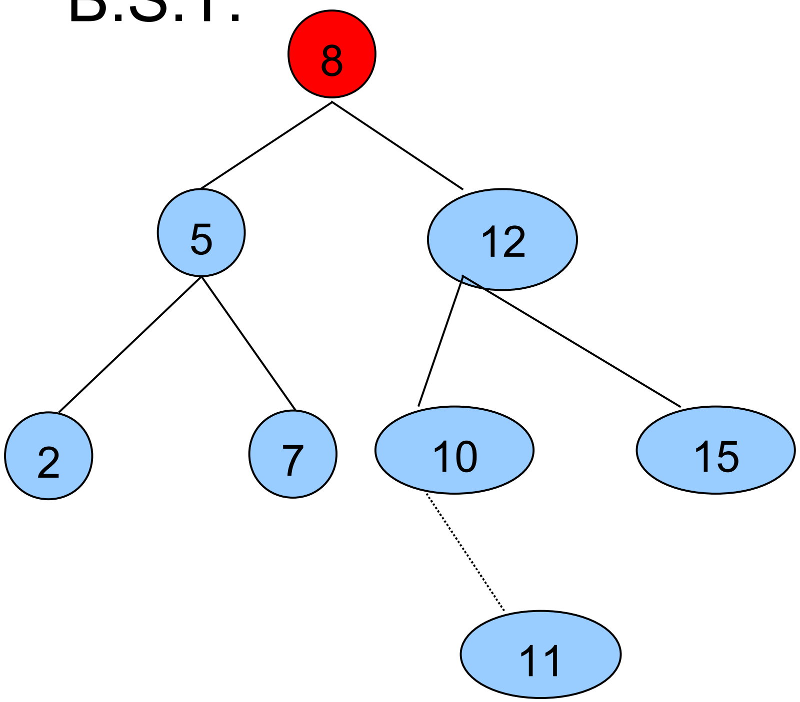
==> The search in balanced B.S.T. is also O(lg$n$)

# Examples

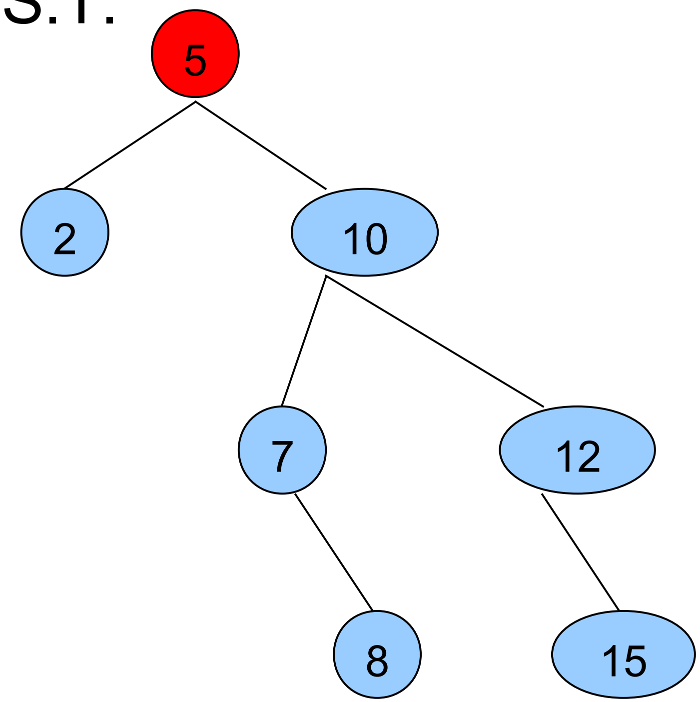


# More examples

- Height balanced B.S.T.

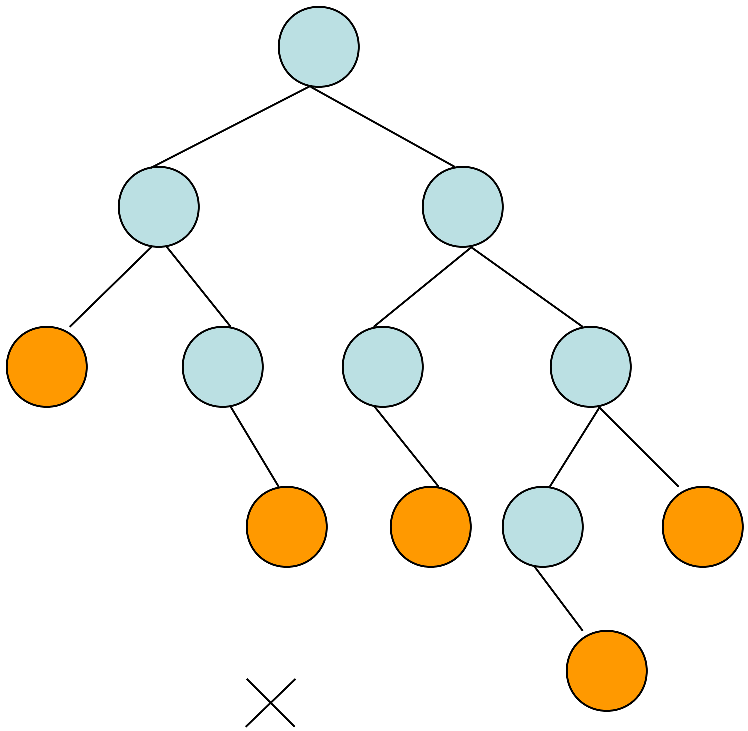


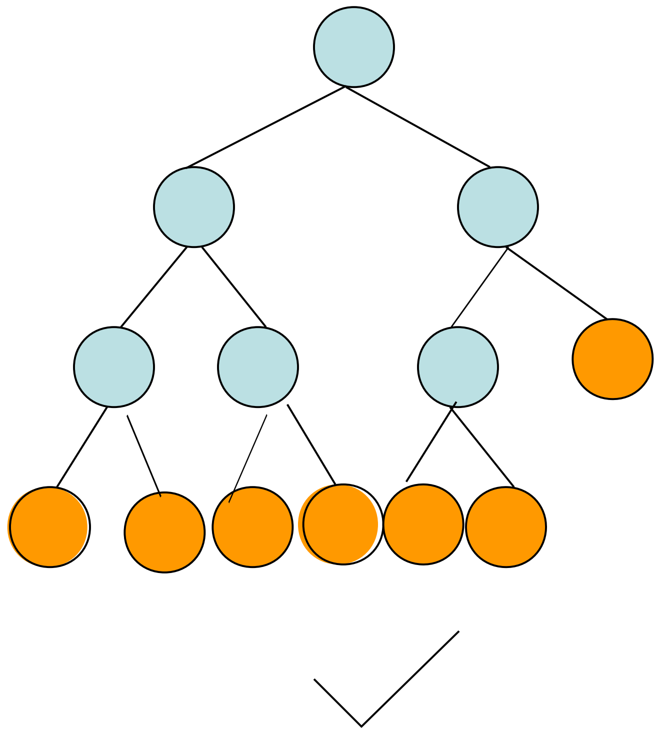

- Non-height balanced B.S.T.



# Perfectly balanced B.S.T.

- A tree is perfectly balanced if
  - It is (height-)balanced, and
  - All leaves are to be found on one or two levels





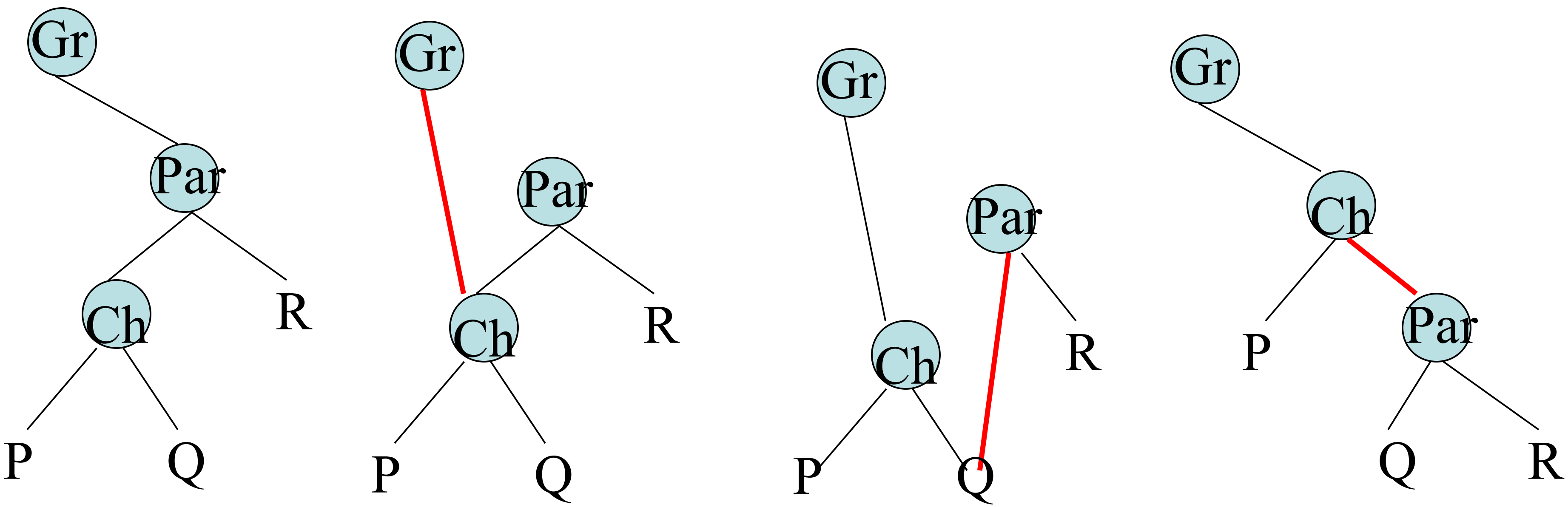
Minmei Hou, Ph.D., Dept. Computer Science, NIU

# The DSW Algorithm

- Devised and improved by Colin Day, Quentin F. Stout, and Bette Warren.
- Transform from any arbitrary B.S.T. to a perfectly balanced B.S.T.
- Rebalance the tree globally
- Two basic operations:
  - Right rotation
  - Left rotation
  - They are symmetric

# Rotations

- Right rotation of node *Ch* about its parent *Par*




*Gr*: Grand parent of *Ch*.

*R*, *P*, *Q* are subtrees.

If *Par* is not root

*Gr* becomes parent of *Ch*.

*Q* becomes left subtree of *Par*.

*Ch* acquires *Par* as its right child.

# Rotations

- Right rotation

  If *par* is not the *root* of the tree

    Grandparent *Gr* of child *Ch* becomes *Ch*'s parent

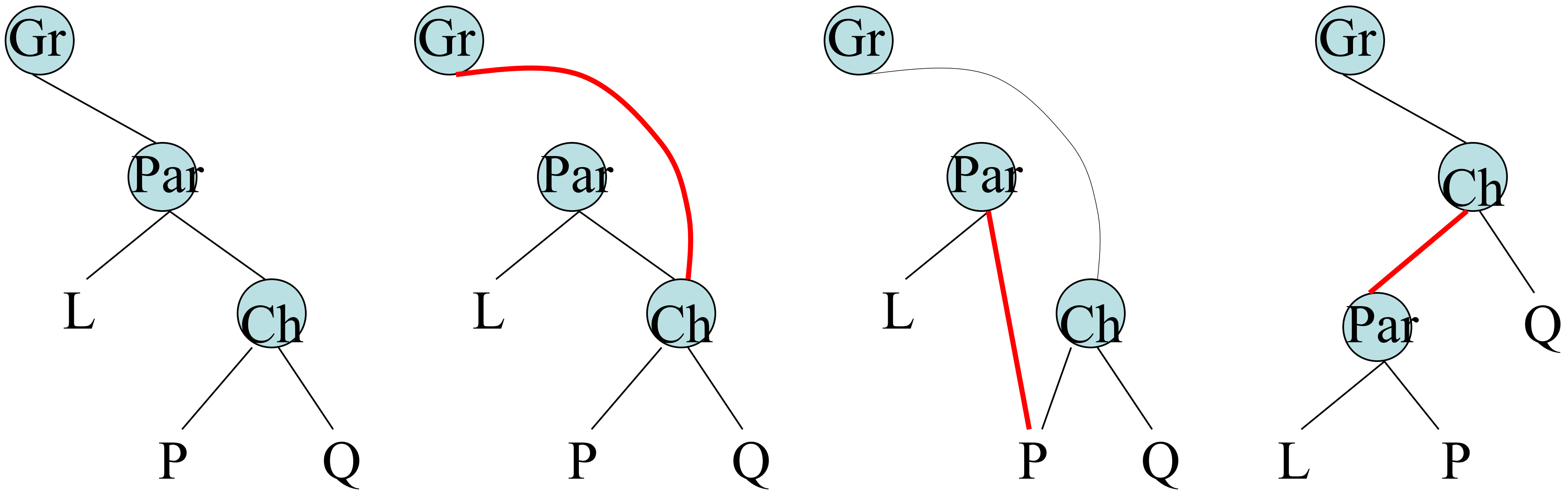
  Right subtree of *Ch* becomes left subtree of *Ch*'s parent *Par*

  Node *Ch* acquires *Par* as its right child

  – (Old parent becomes new right child of Ch.)

# Rotations

- Left rotation of node *Ch* about its parent *Par*




*Gr*: Grand parent of *Ch*.

*L*, *P*, *Q* are subtrees.

If *Par* is not root

*Gr* becomes parent of *Ch*.

*P* becomes right subtree of *Par*.

*Ch* acquires *Par* as its left child.

Minmei Hou, Ph.D., Dept. Computer Science, NIU

# Rotations

- Left rotation

If *par* is not the *root* of the tree

Grandparent *Gr* of child *Ch* becomes *Ch*'s parent

Left subtree of *Ch* becomes right subtree of *Ch*'s parent *Par*

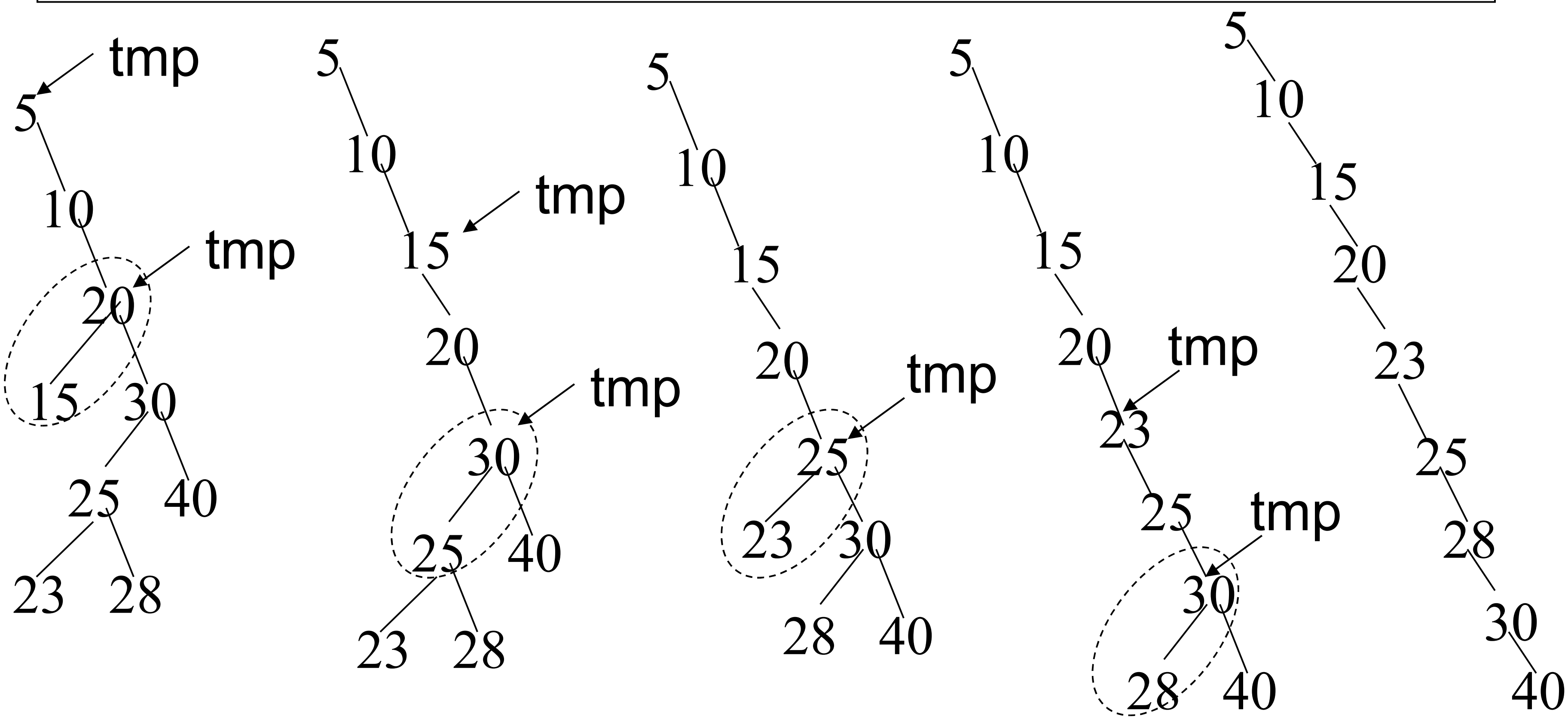
Node *Ch* acquires *Par* as its left child

– (Old parent becomes new left child of Ch.)

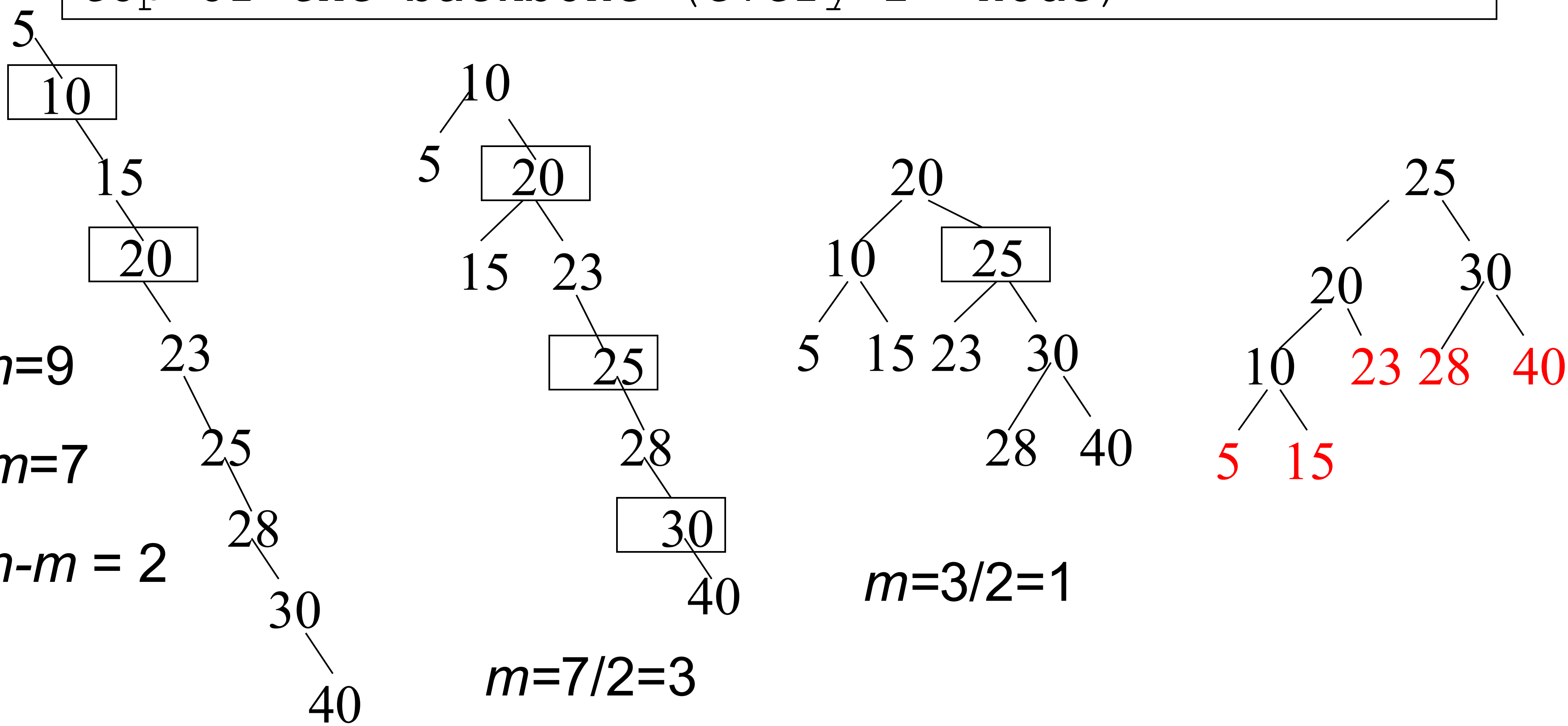
# DSW algorithm

- Two major steps
  - Transform an arbitrary B.S.T. into a linked-list-like tree – backbone (very skewed)
    - CreateBackBone ( root )
  - This backbone is then transformed into a perfectly balanced B.S.T.
    - CreatePerfectlyBalancedTree (n )

```
CreateBackbone ( root )
   tmp <-- root
   while tmp is not empty
      if tmp has a left child Ch
         right rotate Ch about tmp
         tmp <-- Ch
      else
         tmp <-- tmp’s right child
```



```
CreatePerfectlyBalancedTree ( n )
   m <-- 2^floor(lg(n+1)) - 1
   make n-m left rotations starting from the
top of the backbone (every 2nd node)
   while ( m is greater than 1)
      m <-- m/2
      make m left rotations starting from the
top of the backbone (every 2nd node)
```



$n$=9

$m$=7

$n$-$m$ = 2

$m$=7/2=3

$m$=3/2=1

Minmei Hou, Ph.D., Dept. Computer Science, NIU

# The DSW algorithm

- Consider the time complexity
  - The first step: O(*n*)
  - The second step: O(*n*)
  - Overall: O(*n*)

# Balancing algorithms

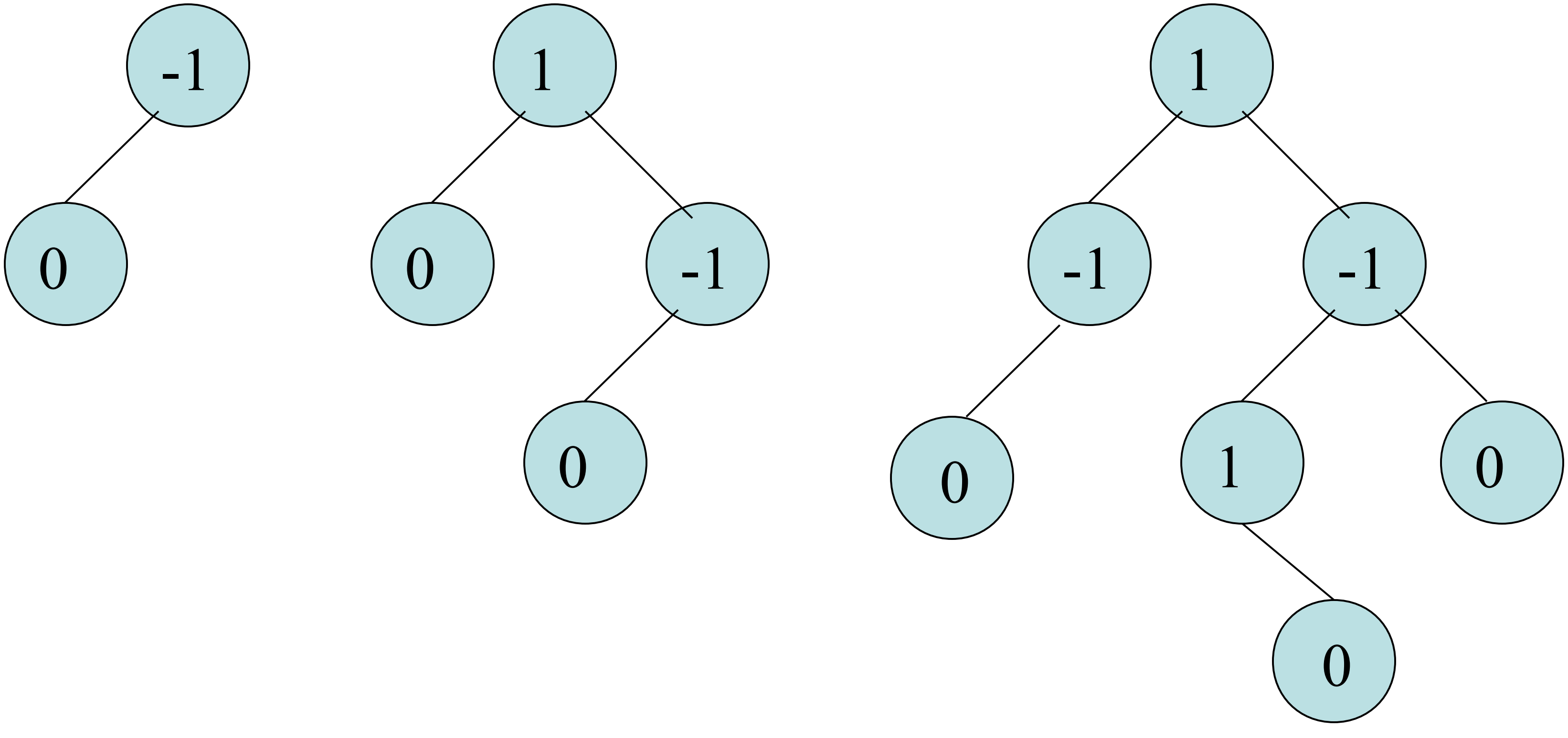
- DSW algorithm rebalance the tree globally
  - Every node could involve in rebalancing
- AVL-trees
  - Rebalancing is done locally
  - Only portion of the tree is affected when a node is inserted or deleted from the tree

# AVL-tree

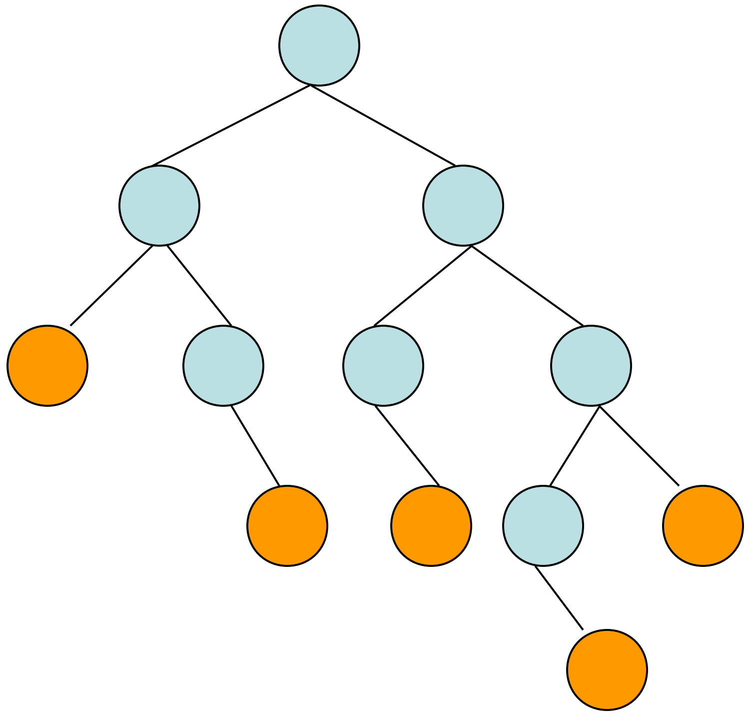
- Proposed by Adel'son-Vel'skii and Landis
- Definition of AVL-tree:
  - The height of left and right subtrees of every node differ by at most one.
  - Balance factor
    - = height(right-subtree) – height(left-subtree)
  - AVL-tree's balance factors must be -1, 0, or 1
- Note that AVL-tree is not necessarily a perfectly balanced tree

# AVL-tree

- Examples



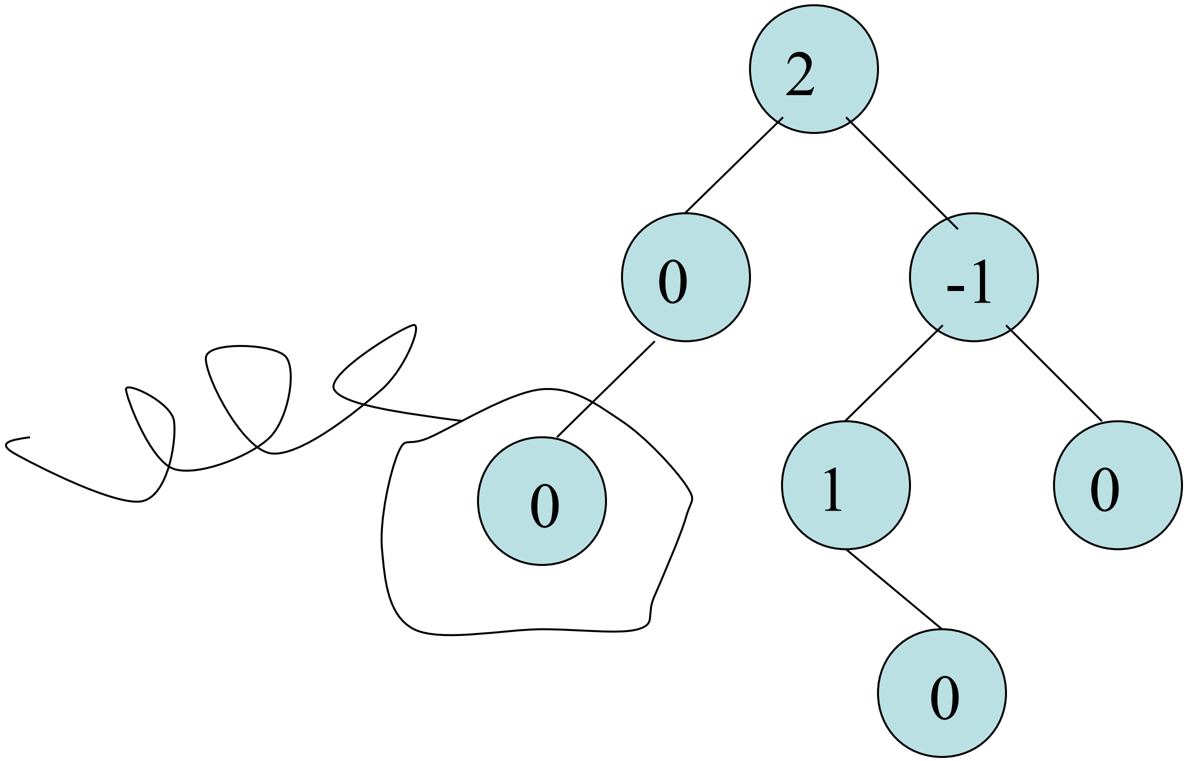
# AVL-tree



Above is an AVL-tree, but not a perfectly balanced tree.

# AVL-tree

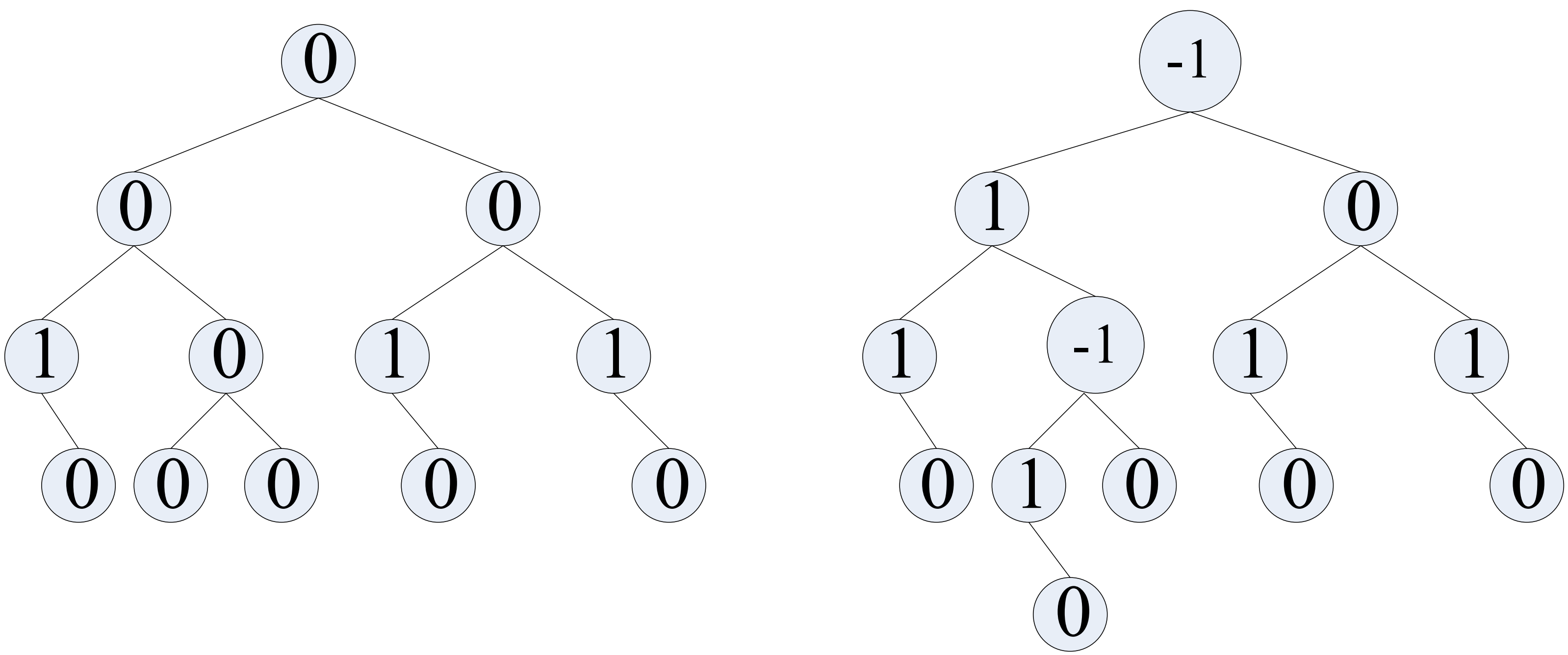
- If the balance factor of any node in an AVL-tree becomes less than -1 or greater than 1, the tree has to be rebalanced




After deletion, the balance factor of root becomes 2.

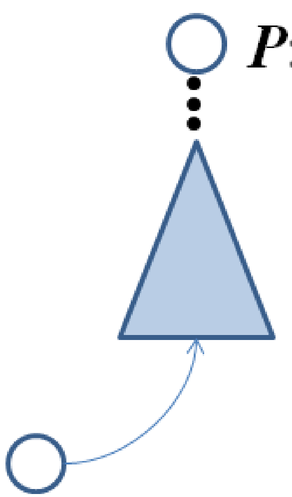
# Insertions

- In some cases, an insertion requires no height rebalancing.



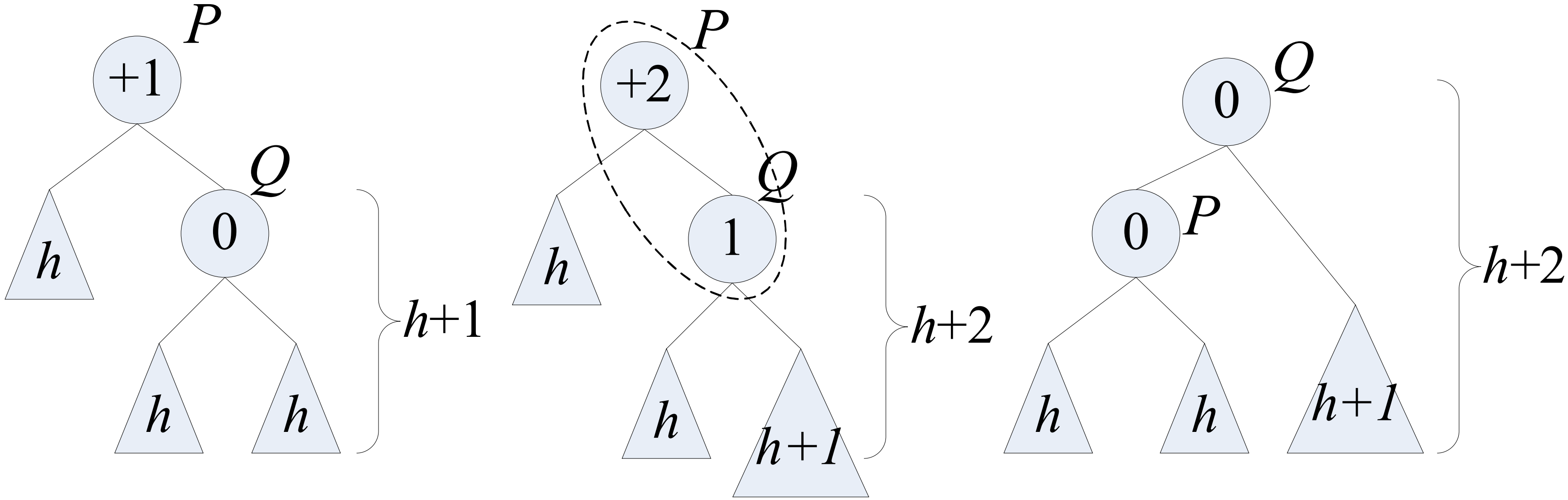
# Insertions

- There are 4 cases when an AVL-tree may become out of balance after insertion
  - Two are symmetric
  - We only discuss two of them
  - Assume the operations start with an AVL-tree



*P*: the nearest ancestor node (of the newly inserted node) whose balance factor will become 2 after the insertion.

# Insertion case I




A node is to be inserted into the right subtree of *Q*.

Note: circles are nodes; triangles are subtrees.

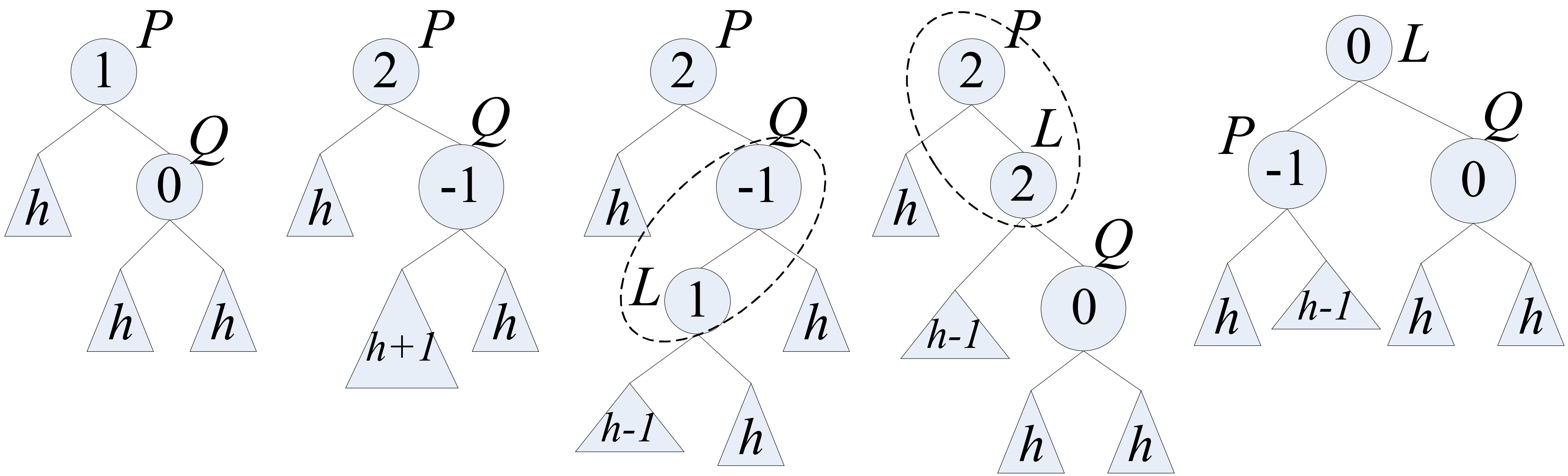
*P* is the nearest ancestor node which becomes unbalanced.

A rotation is necessary here.

After rotation, this portion of tree becomes balanced. Height is also the same as before.

Minmei Hou, Ph.D., Dept. Computer Science, NIU

# Insertion case II




A node is to be inserted into the left subtree of *Q*.

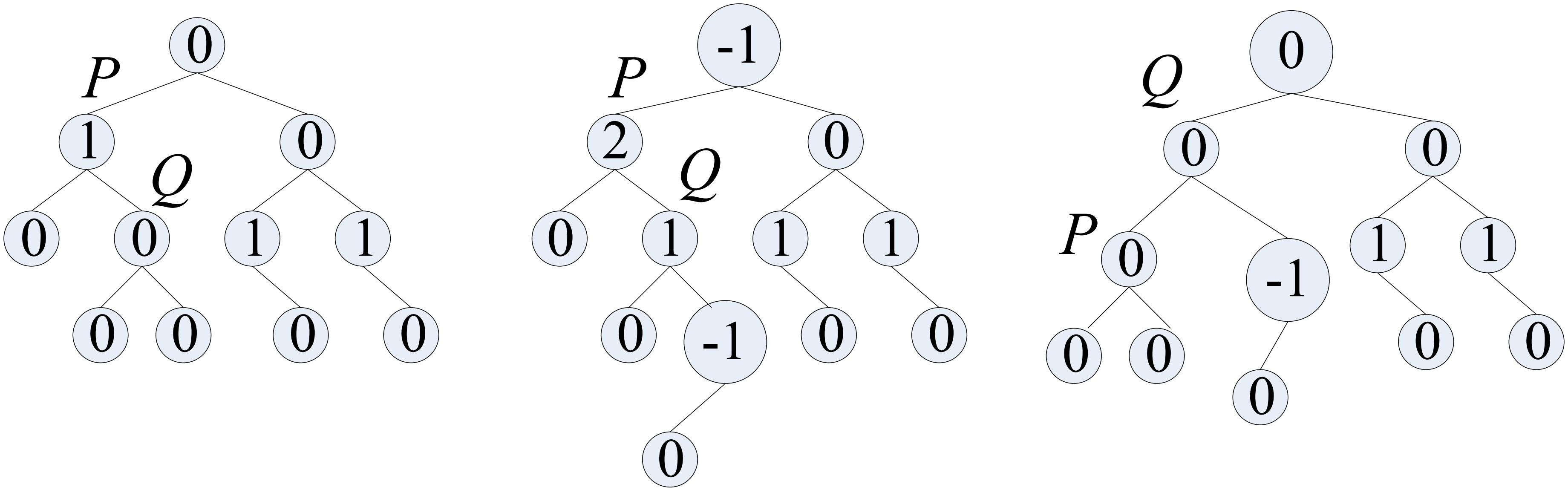
*P* is the nearest ancestor node which becomes unbalanced.

Double rotations are necessary.

After rotation, this portion of tree becomes balanced. Height is also the same as before.

# Insertion

- An example where *P* is not at the root



# Insertion

Observation:

Height of the (sub)tree after rebalancing is the same as the height before insertion.

==>The balance factors of the nodes in other portions of the tree, including ancestors, are not affected.

==>Once  the node that is out of balance is rebalanced, the entire AVL-tree is rebalanced.

# Insertion

Rebalancing algorithm:

For node *n* from the newly inserted node up to the root

*f* <-- update *n*'s balance factor

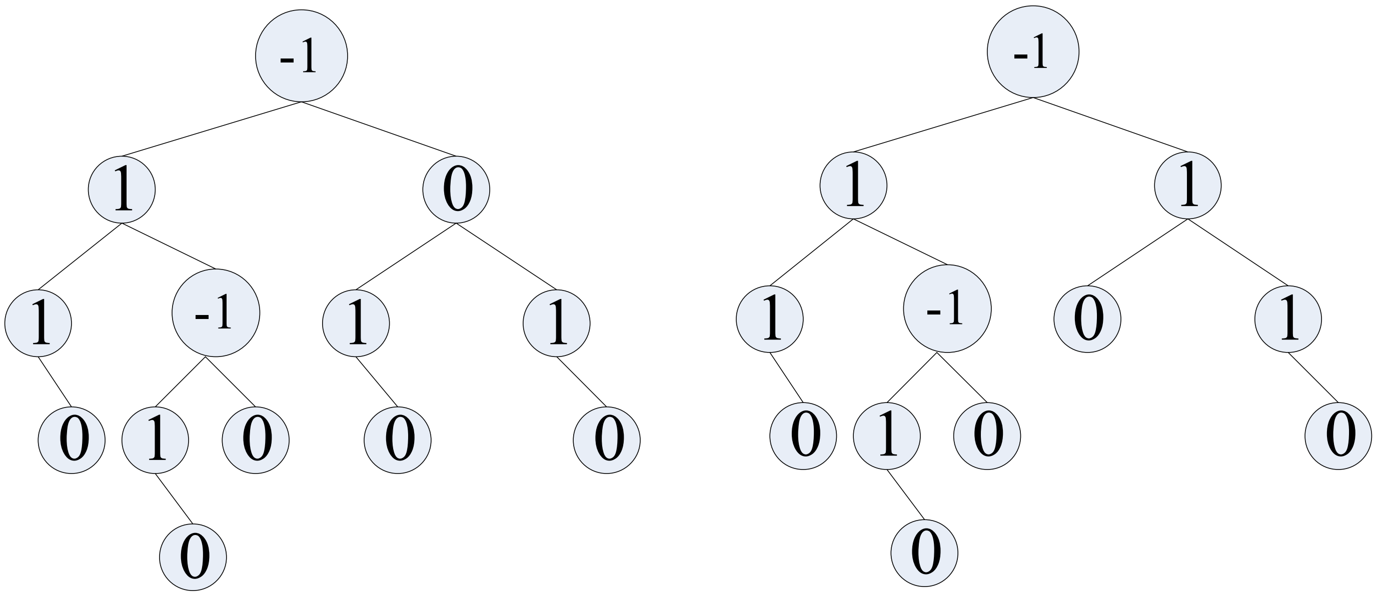
if *f* is 2 or -2

rebalance *n*

stop

# Deletion

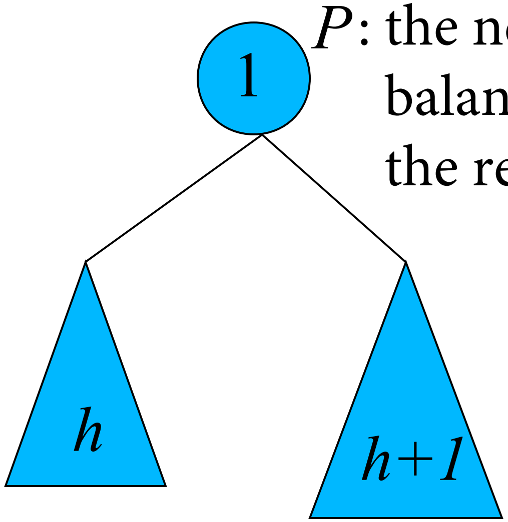
- In some cases, a deletion requires no height rebalancing.



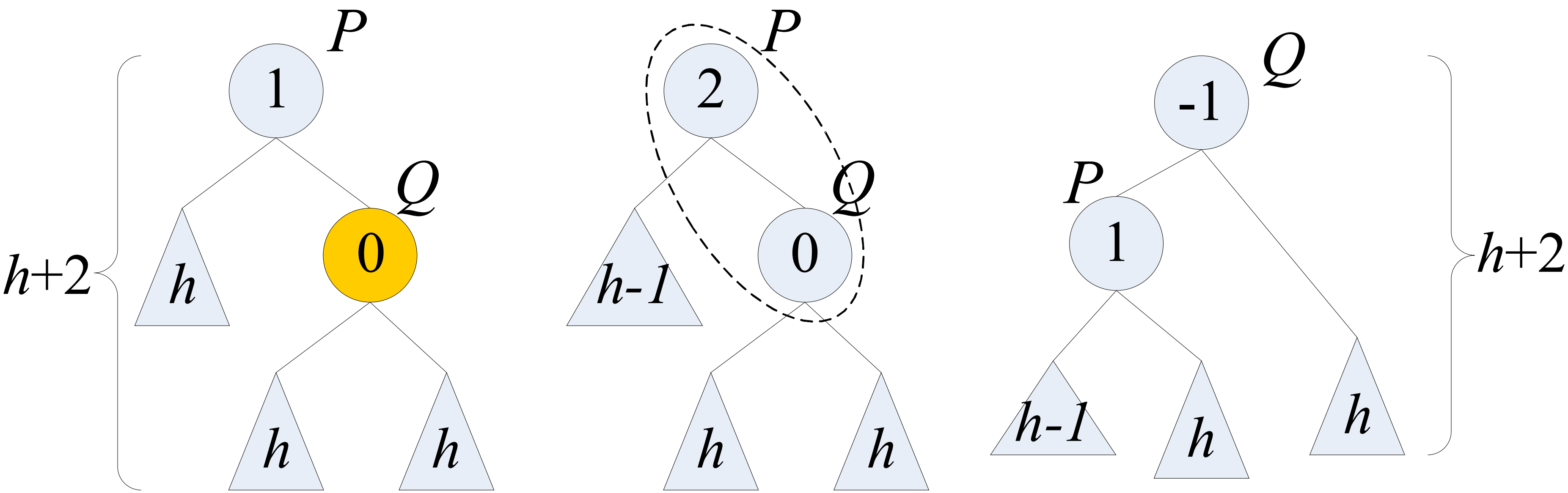
# Deletions

- Assume DeleteByCopying
- 4 cases ( there are 4 other symmetric - cases)
- Assume a node in the left subtree of *P* will be removed.

*P*: the nearest ancestor node (whose balance factor will become 2) of the removed node.



# Deletion case I




The b.f. of *Q* is 0. A node in left subtree of *P* will be deleted.

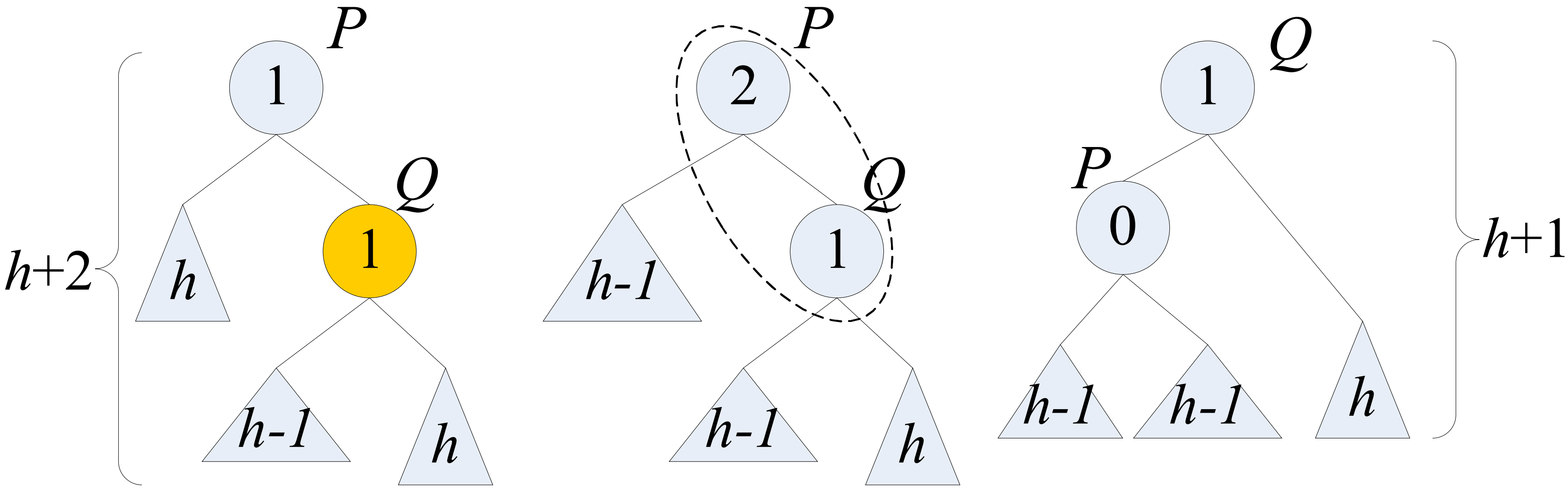
*P* needs rebalancing.

A rotation is necessary.

After rebalancing, the height of this portion of the tree is not changed.

Minmei Hou, Ph.D., Dept. Computer Science, NIU

# Deletion case II



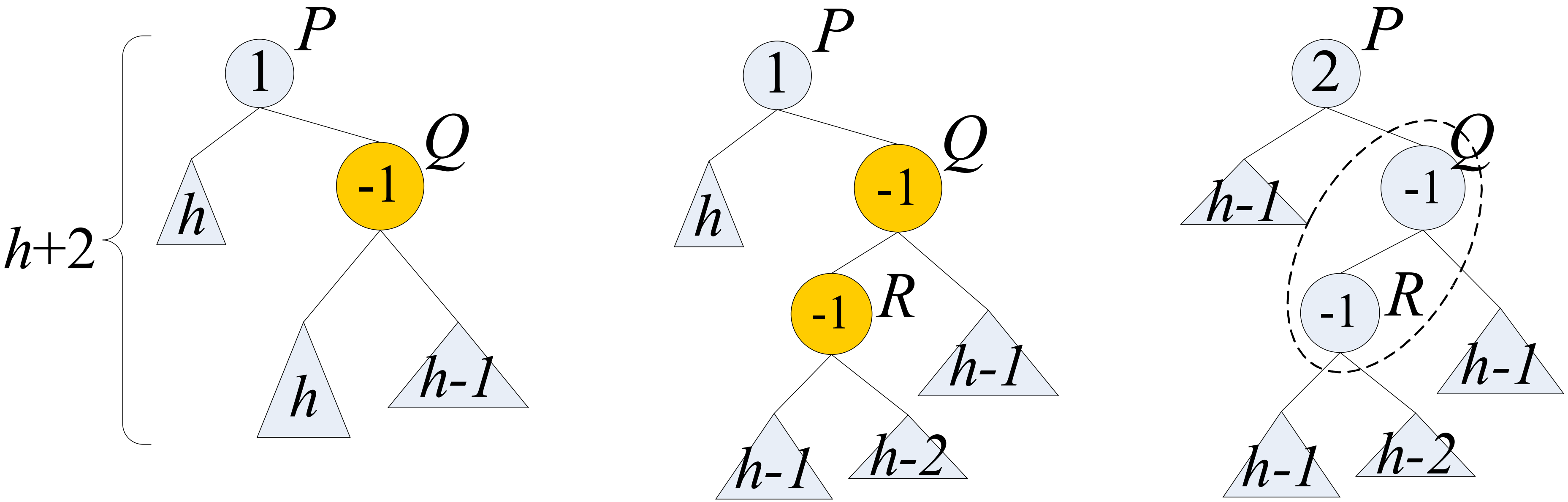

The b.f. of *Q* is 1. A node in left subtree of *P* will be deleted.

*P* needs rebalancing.

A rotation is necessary.

Note that the height of this portion of the tree is reduced.

# Deletion case III




The b.f. of *Q* is -1. A node in left subtree of *P* will be deleted.

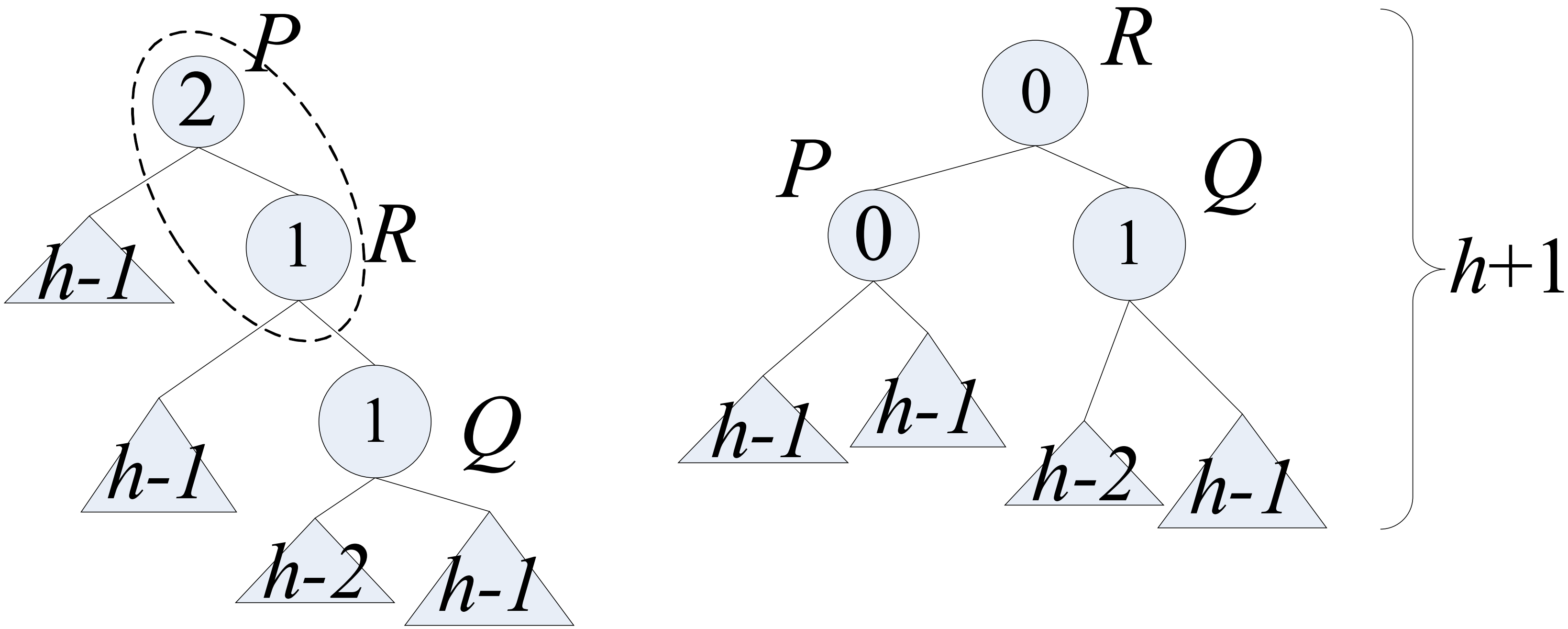
For case III, the b.f. of *Q*'s left child is -1.

After a node in left subtree of *P* is deleted, *P* needs rebalancing.

Two rotations are necessary.

Minmei Hou, Ph.D., Dept. Computer Science, NIU

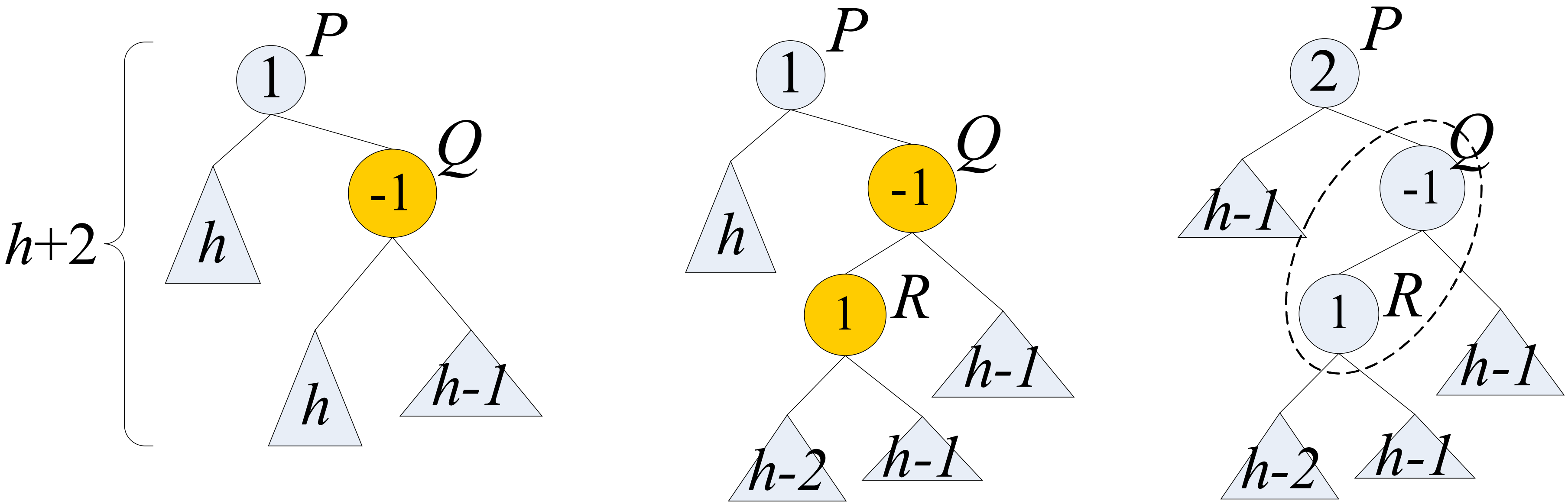
# Deletion case III (cont.)




The 2nd rotation.

Note that the height of this portion of the tree is reduced.

# Deletion case IV



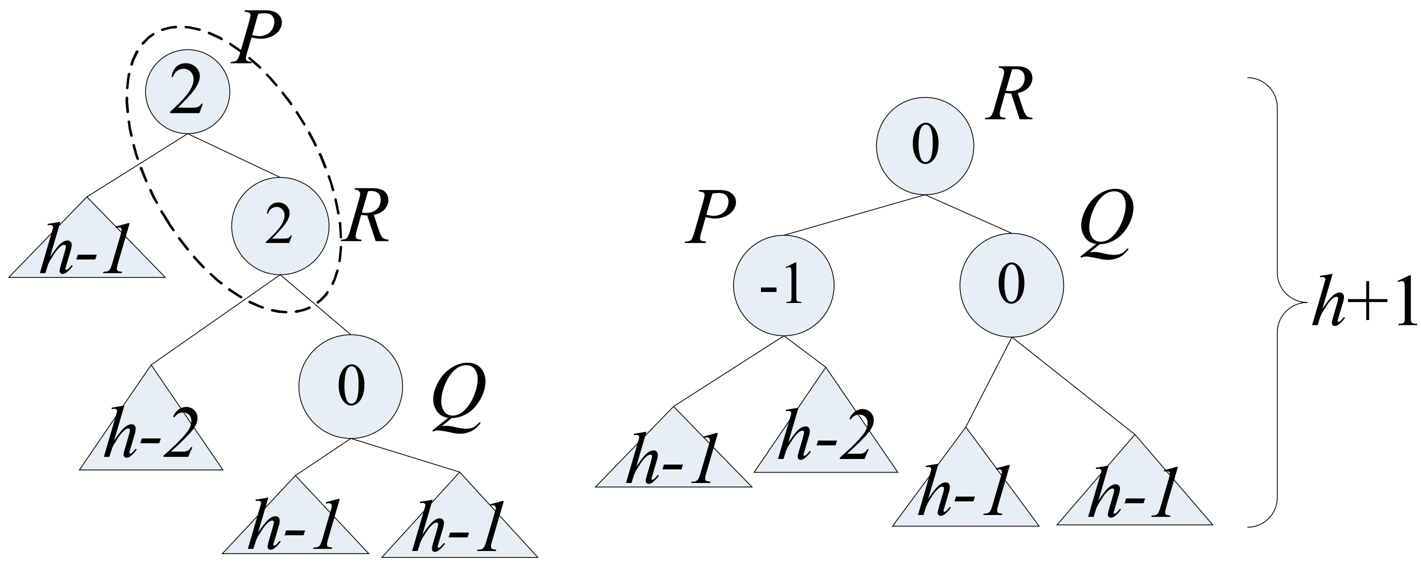

The b.f. of *Q* is 0. A node in left subtree of *P* will be deleted.

For case IV, the b.f. of *Q*'s left child is 1.

After a node in left subtree of *P* is deleted, *P* needs rebalancing.

Two rotations are necessary.

# Deletion case IV (cont.)




The 2nd rotation.

Note that the height of this portion of the tree is reduced.

Minmei Hou, Ph.D., Dept. Computer Science, NIU

# Deletion

Observation:

Height of the (sub)tree after rebalancing may be different from the height before deletion.

==>The balance factors of the ancestors may be affected.

==>It may be necessary to rebalance all ancestors of the parent node of the deleted node.

# Deletion

Rebalancing algorithm:

For node *n* from the parent of the removed node up to the root

*f* <-- update *n*'s balance factor

if *f* is 2 or -2

rebalance *n*

# AVL-tree

- Consider the time complexity.
- Note that the height (h) of an AVL-tree:

  $\lg(n+1) \le h < 1.44 \lg(n+2) - 0.328$
- Insertion ~ O(lgn)
- Deletion ~ O(lgn)
- Search ~ O(lgn)

# Another global technique to create balanced tree

Given a sorted array data[ ]:

```
balance ( data, first, last )
    if first is less than last
        middle <-- (first+last)/2
        b.s.t.-insert ( data[ middle ] )
        balance (data, first, middle – 1)
        balance (data, middle+1, last)
```