Data Structures

October 5

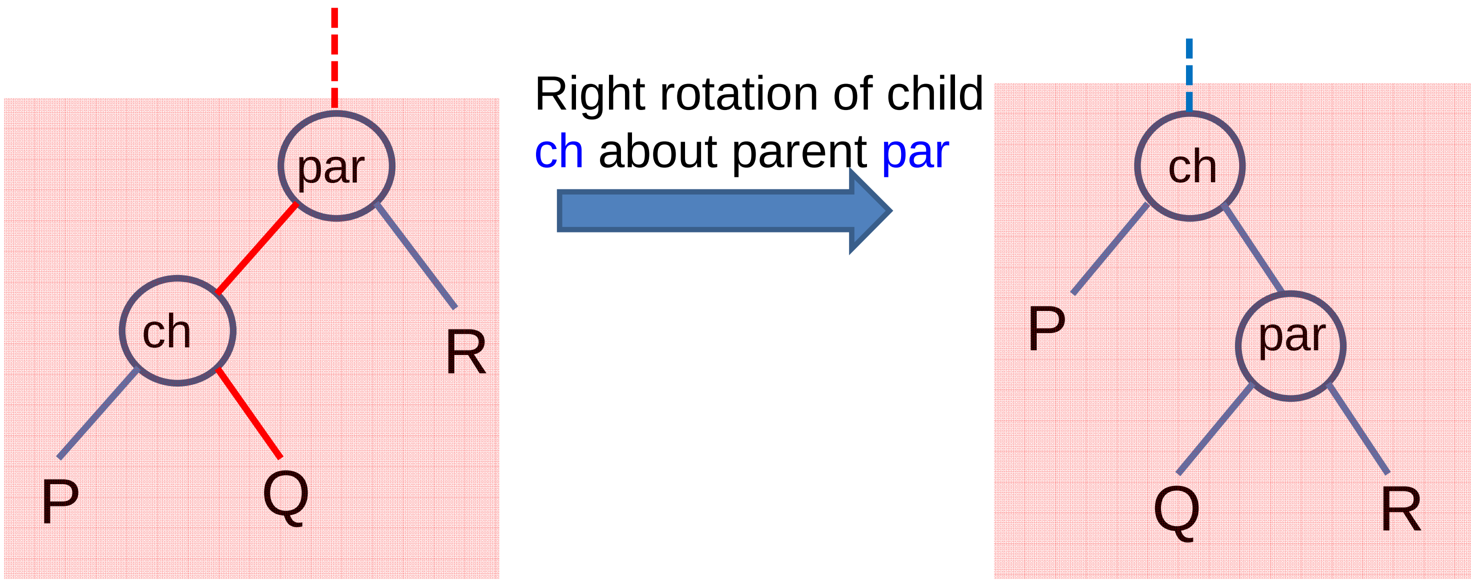
Topics

- Global balancing of binary (search) trees
- Local balancing of binary (search) trees
- Self-organizing (binary search) trees

Recall Definitions

- *Height-balanced* (aka *balanced*) binary tree: for any node in the tree the height of the node's subtrees differ by at most 1.
- Page 251 Drozdek: a *perfectly balanced* tree, is a balanced tree such that all leaves are on one level or on two levels. (Exercise: construct a balanced tree such that the leaves are in more than two levels.)
- A binary tree has the *Symmetric Search property*, if
  - Each node contains a key
  - Different nodes will have different keys
  - There is an ordering on the keys
  - For each node in the tree, the key of the node is strictly bigger than *any* key in the left subtree and strictly smaller than *any* key in the right subtree

Binary Tree Transformations: Rotations

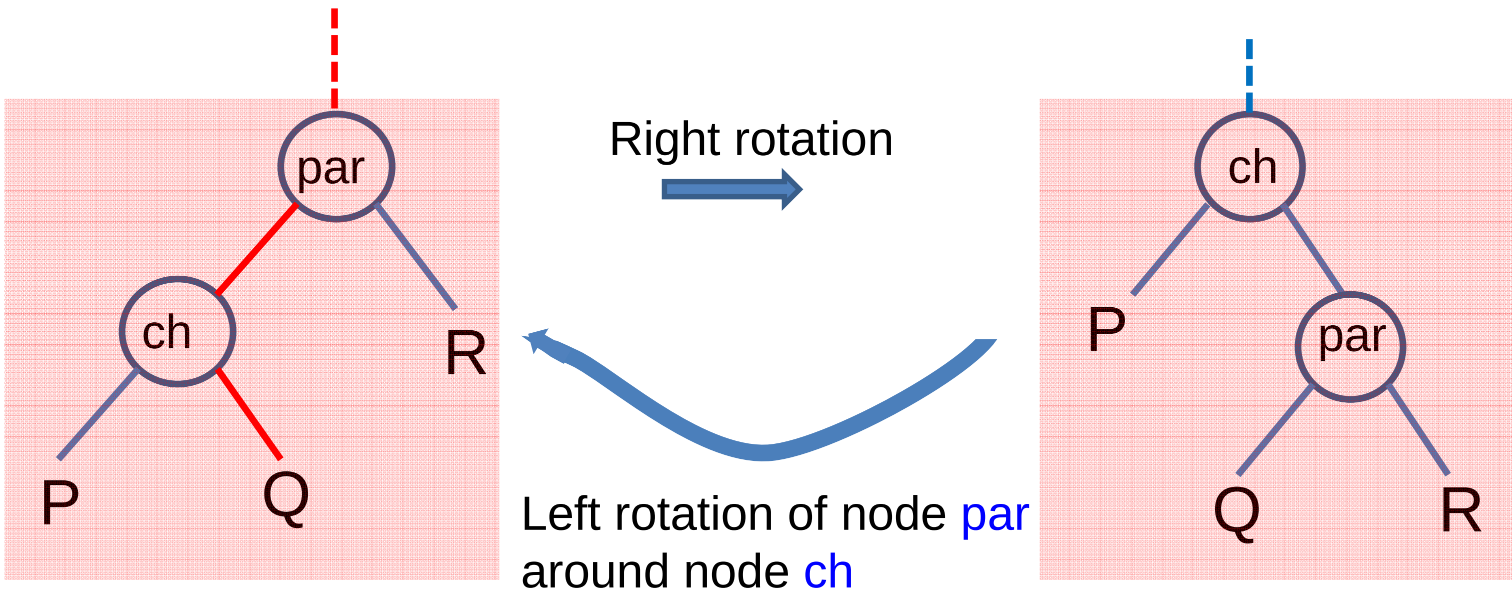


!! This transformation preserves symmetric Binary Search Tree property !!

Some details:
P, Q, and R are subtrees, possibly empty
Dashed means: does not matter whether present or not.
Vertical line means that it does not matter whether the node is a left or a right child of somebody.
Red lines on the left mean that on the lower level specifications (for instance on the implementation level of the tree, say in pointer based, the value of the red links will change)

Right rotation of child ch about parent par (aka right rotation on node par)

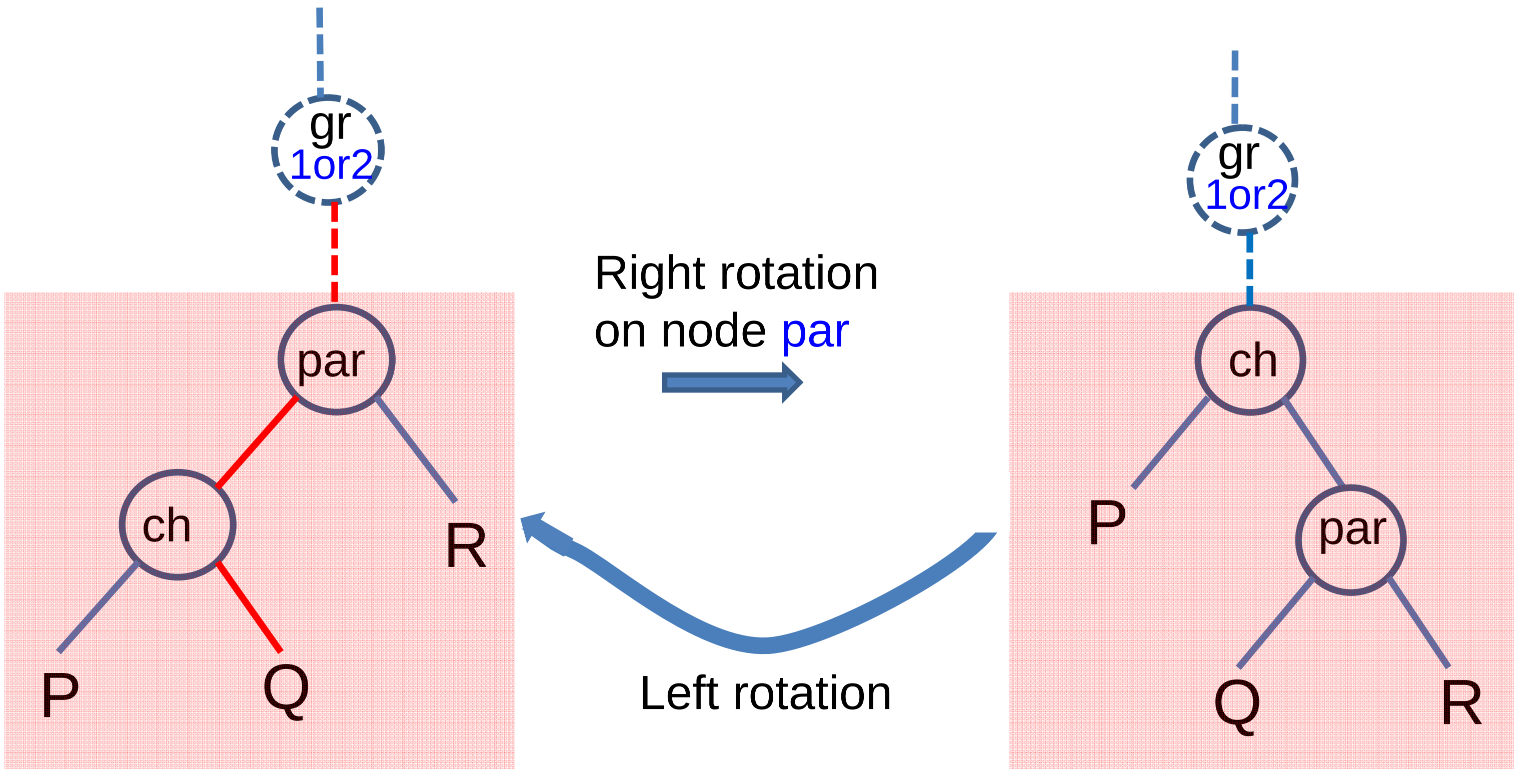
Binary Tree Transformations: Rotations



Of course, starting with the tree on the right, we say that the tree on the left is the left rotation of node par around node ch.
(The right and left rotations are inverses of each other.)
If the tree happens to be a seach tree, then a rotation will Preserve the search property.

These are the only rotations needed; As *double* rotations are built up from single rotations.

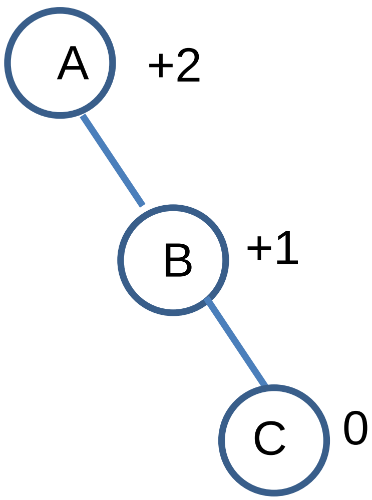
Binary Tree Transformations: Rotations



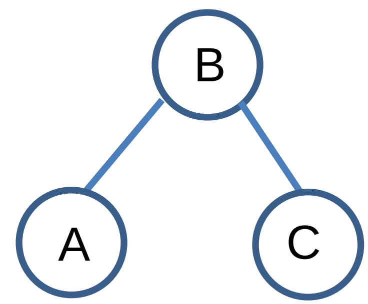
Node par can have parent; node par can be either a left or a right child of gramps – it does not matter; by the same token in the right tree, the node ch can have a parent and it can either be a left or right child of gramps; (of course, in the right tree, ch and par are just names, and are not intended to suggest parent-child relationship)

Motivation for the Left and Right Rotation (continued)

We can motivate the left rotation, on a *very simple unbalanced* BST:



Carry out left rotation on A:

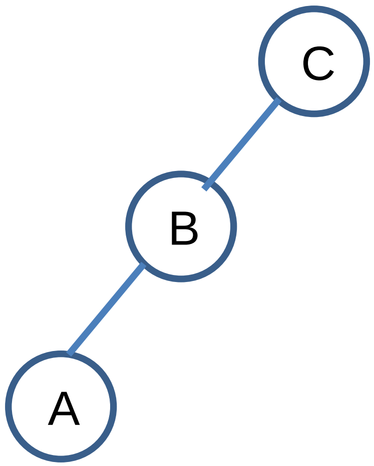


Balanced !

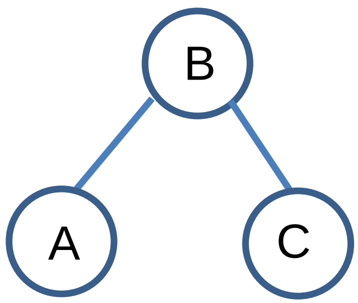
Numbers next to the nodes are the *balance factors* of the nodes (balance factor of a node = height of right subtree – height of left subtree)

Motivation for the Left and Right Rotation (continued)

Similarly we can motivate the right rotation, again on a *very simple unbalanced* BST:



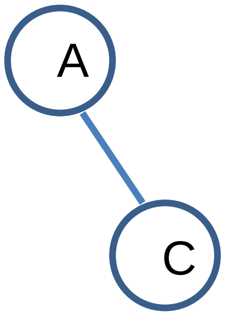
Carry out right rotation on C:



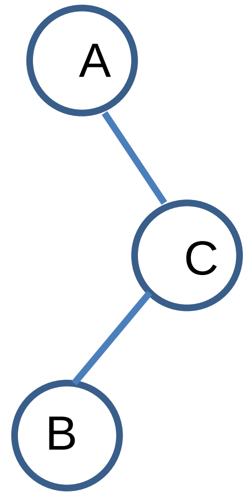
Balanced !

Left-right

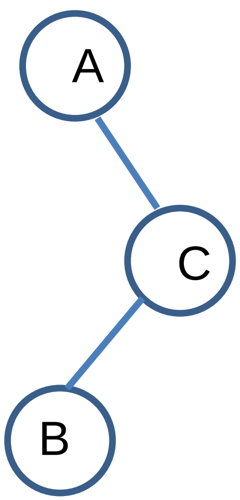
Left and right rotations are not enough for balancing:



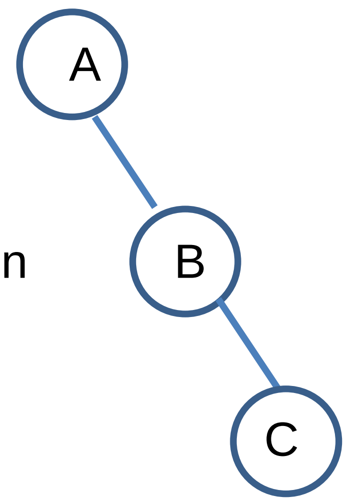
Consider the simple balanced BST on the left. Suppose an insertion with key B takes place, we then get:



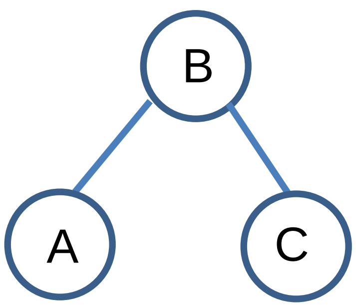
The resulting tree on the right is clearly unbalanced. Moreover it cannot be balanced by a single rotation (left or right) and on any of the nodes A, B or C. (Exercise)



Right Rotation on C:

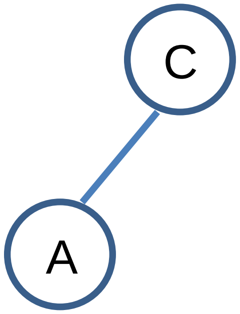


Left Rotation on A:

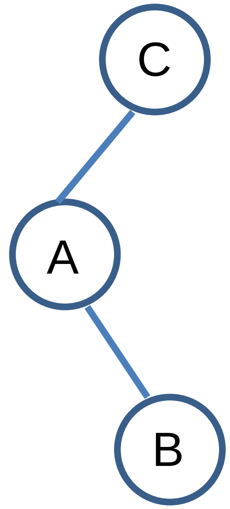


Balanced !

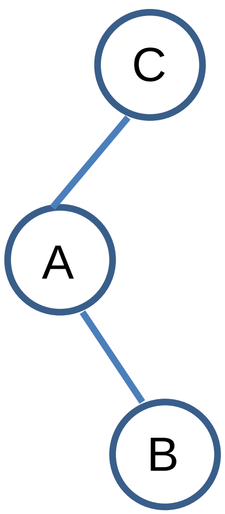
Left and right rotations are not enough for balancing (symm. case):



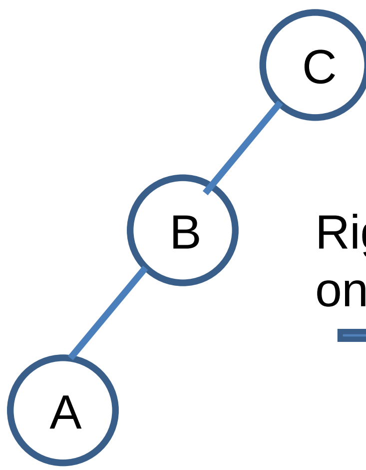
Consider the simple balanced BST on the left. Suppose an insertion with key B takes place, we then get:



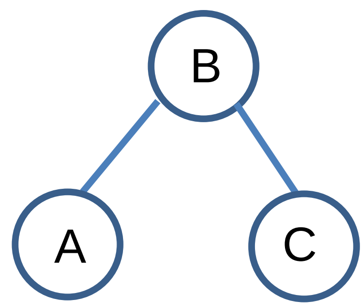
The resulting tree on the right is clearly unbalanced. Moreover it cannot be balanced by a single rotation (left or right) and on any of the nodes A, B or C. (Exercise)



left Rotation on A:

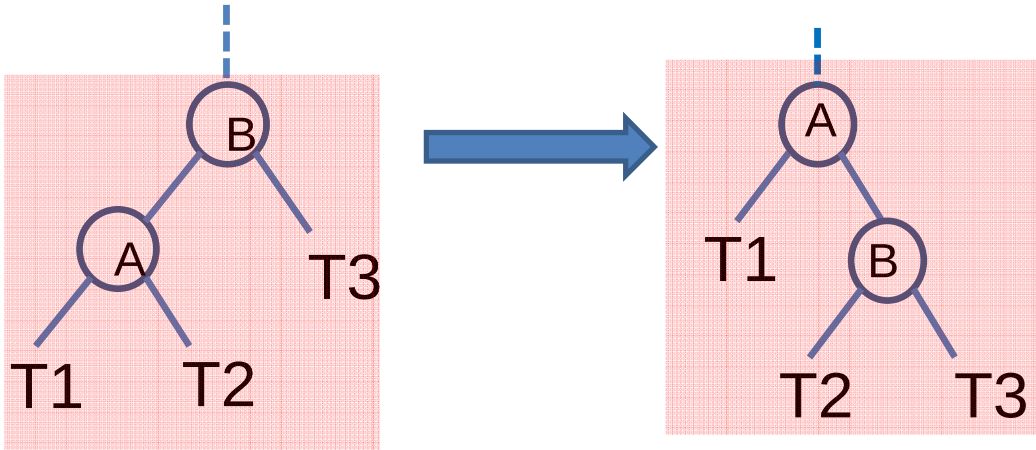


Right Rotation on C:

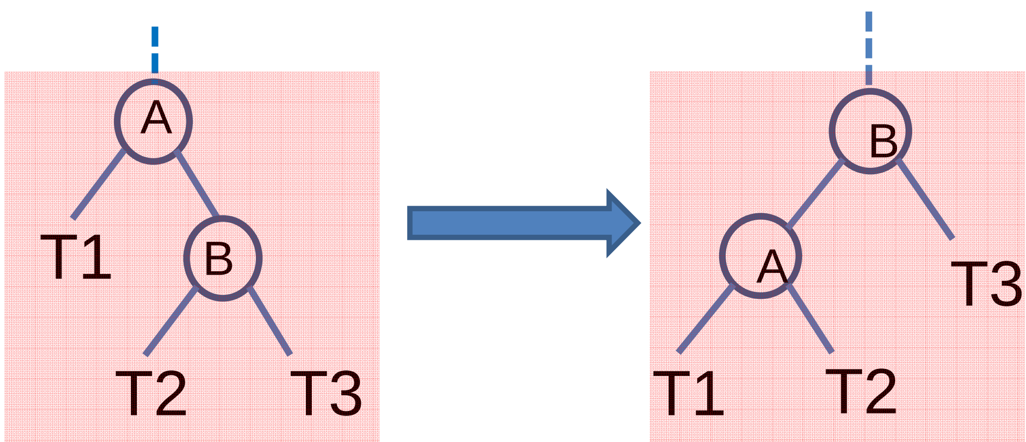


Balanced !

Summary of Single Rotations

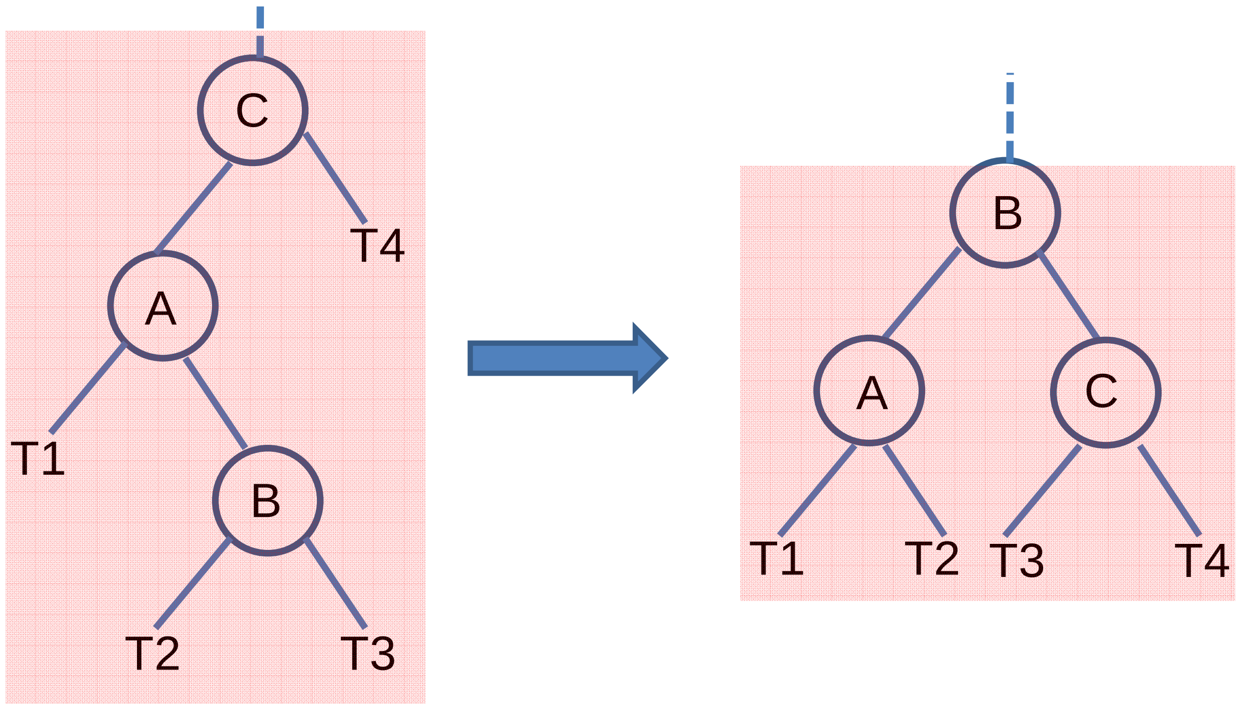


Single right rotation



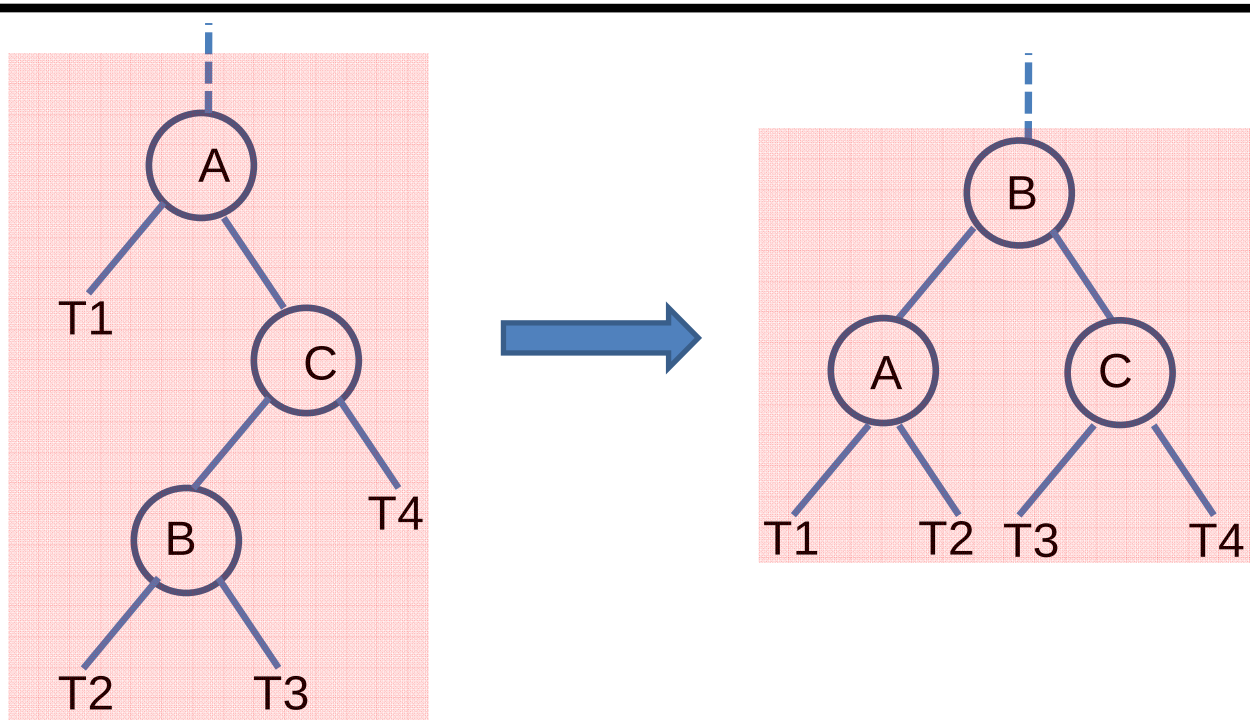
Single left rotation

Summary of Double Rotations



Double rotation to the right

Composition of
left on A
Followed by right on C



Double rotation to the left

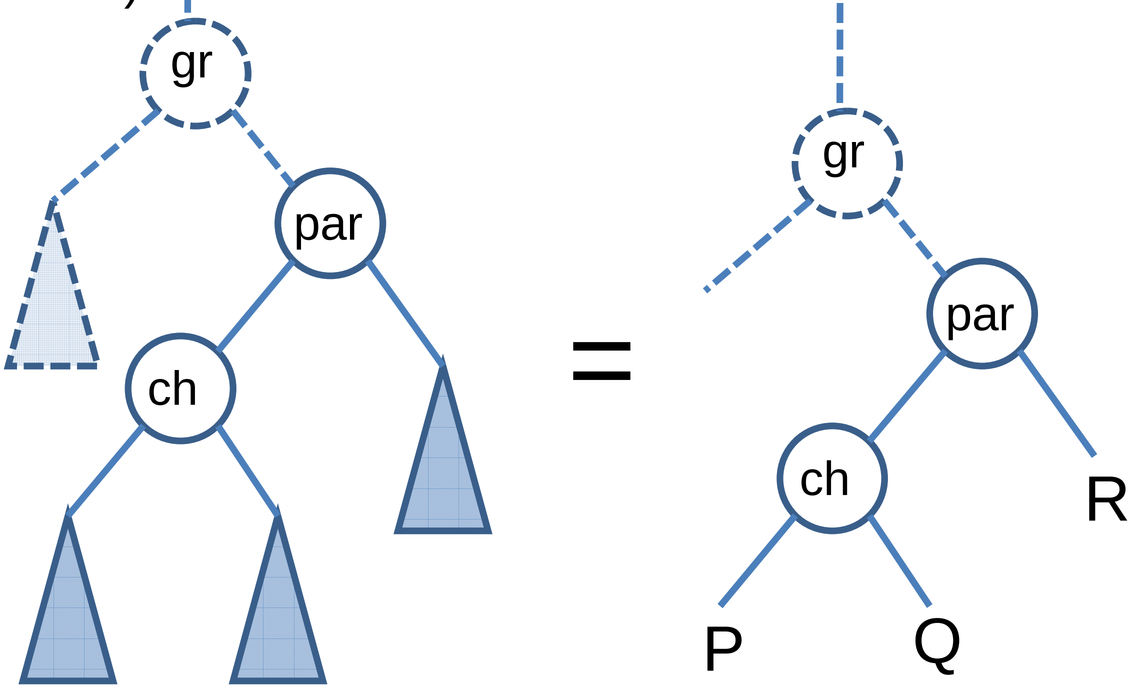
Composition of
Right on C
Followed by left on A

Digression: Rigidity

- In the following we verify that, if we require the rotation transformation of the tree to be symmetric search property preserving, than the rotation is completely determined (is unique) – or in simpler terms: we know where nodes ch, par and subtrees P, Q, and R go in the right transformation and we know where nodes par, ch and subtrees P, Q, and R go in the left transformation
- Sometimes life is easy on your memory ☺

Binary Tree Transformations: Rotations; The Anatomy

The dashed lines and dashed subtrees indicate that we don't care whether these links and subtrees are present or not. Verticality means we don´t care whether gr is a left or right descendant of its parent. We represent the same situation with each of the two pictures below. We are going to consider rotations (right and left which are inverses of each other). All or some of the direct descendants P, Q, and R can be absent. But we do care about them! That is we need to tell explicitly where they end up after the rotation. A better way of saying this is that P, Q, or R could be the empty subtree and show their position in the rotated tree (whether they are empty or not).

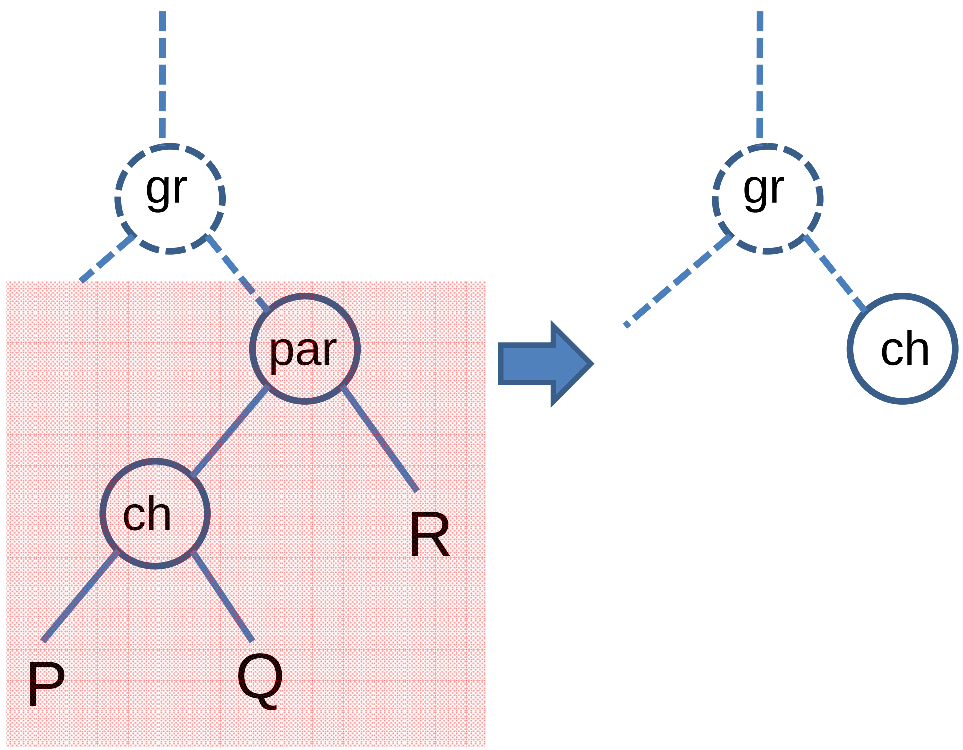


In what follows, the node par could also be the left child of gramps; in order to fix our thoughts we let the node par be the right descendant of gramps

Bin Trees Transfo: Rotations

The rigidity of a right rotation of node ch around its parent par.

The rotation can be done in any binary tree but in order to rediscover what the rotation should be, we let us guide by the **binary symmetric search tree property.** In case the rotation is applied to a BST we require the BST property to be invariant (i.e., the BST property, if present before the rotation, should be present also after the rotation.)

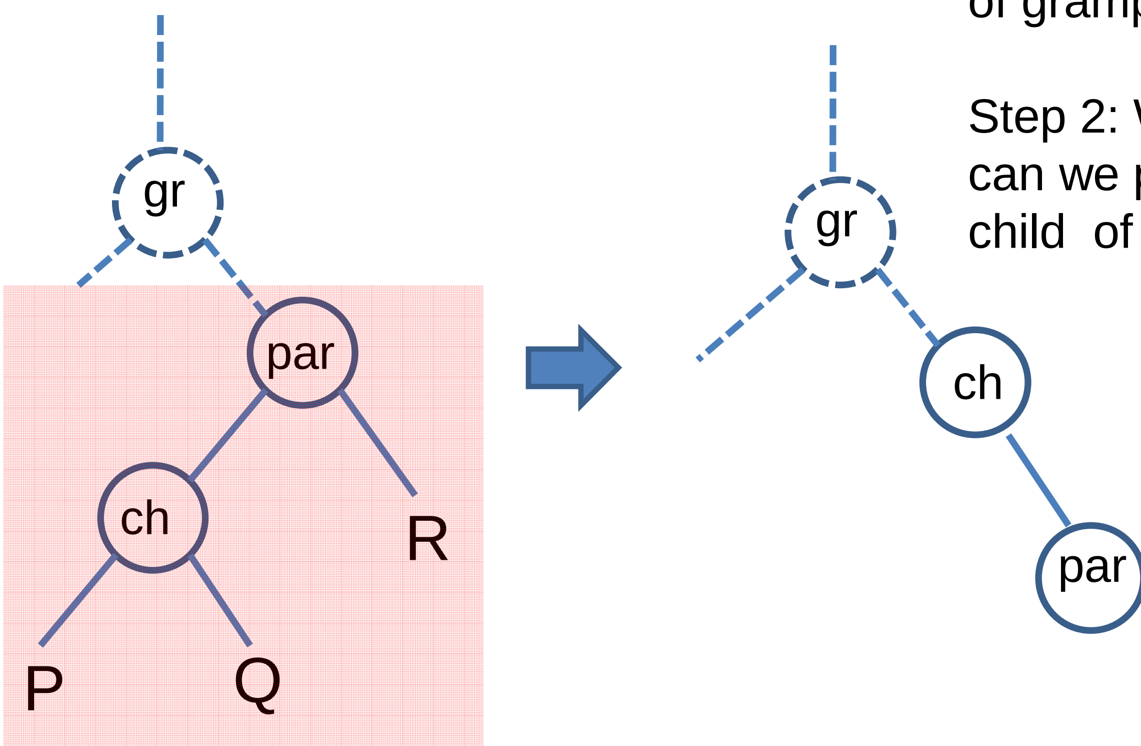


Step 1: the ch takes the placs of par
a)BST prop. still holds
b)With respect to gramps, ch can *only* be put down as a right descendant of gramps.

What about parent par ? Where can we put it?

Bin Tree Transfo: Rotations

The rigidity of a right rotation of node ch around its parent par.



Step 1: the ch takes the placs of par
a)BST prop. still holds
b)With respect to gramps, ch can *only* be put down as a right descendant of gramps.

Step 2: What about parent par ? Where can we put it? It can *only* be the right child of ch:

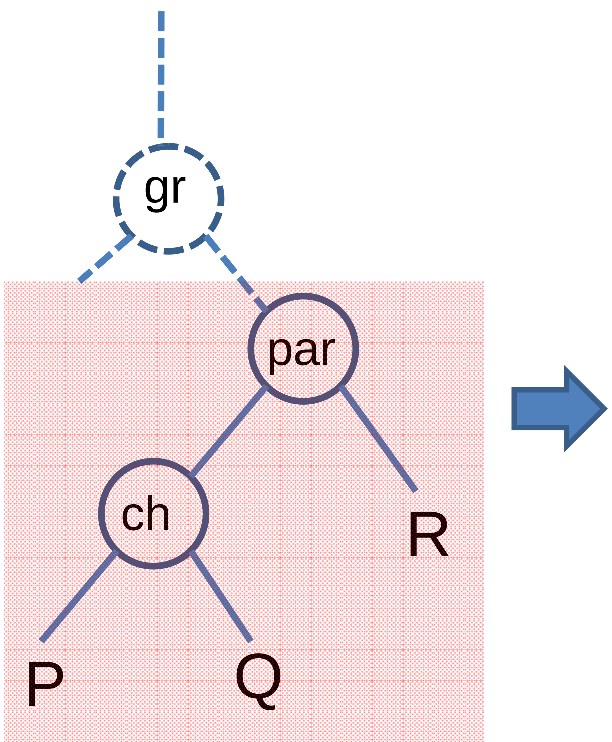
Next: where can we put P, Q, and R?

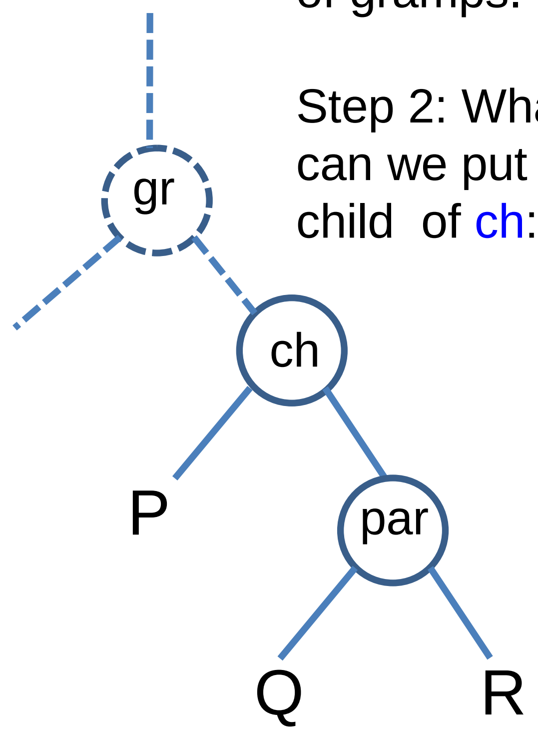
Bin Tree Transfo: Rotations

The rigidity of a right rotation of node ch around its parent par.

Step 1: the ch takes the placs of par
a)BST prop. still holds
b)With respect to gramps, ch can *only* be put down as a right descendant of gramps.

Step 2: What about parent par ? Where can we put it? It can *only* be the right child of ch:

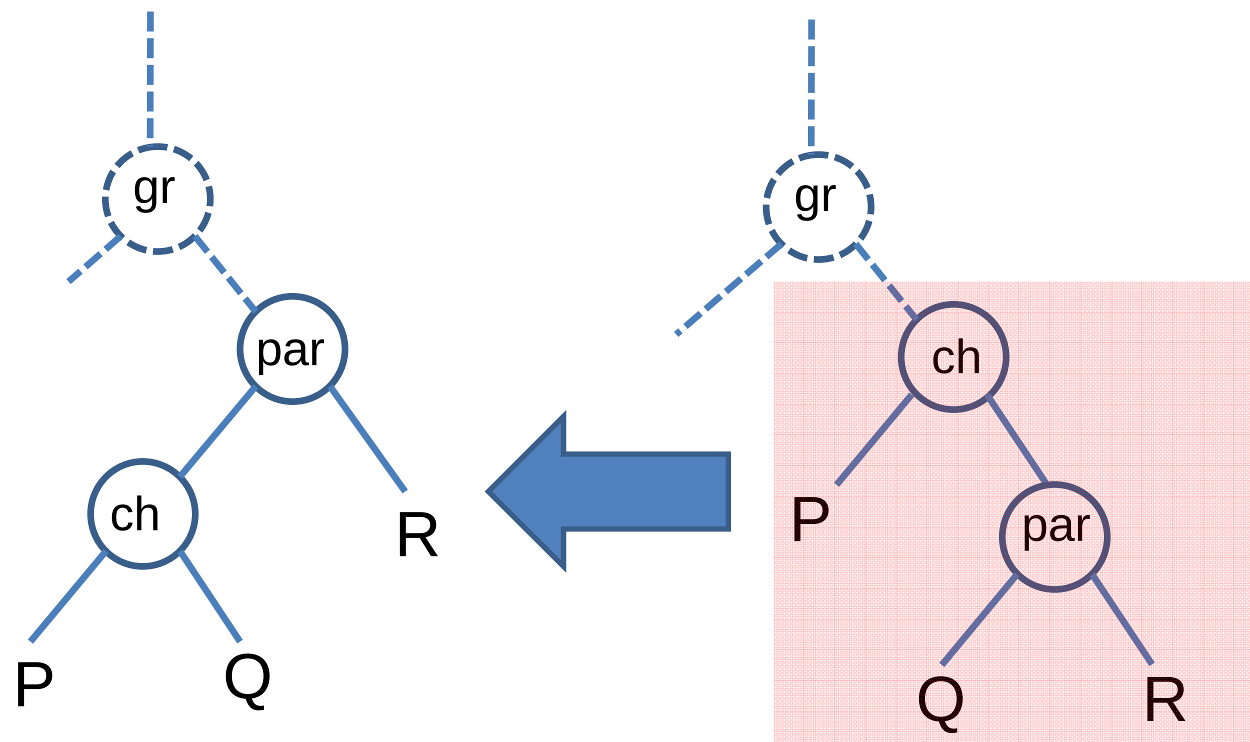




Next: where can we put P, Q, and R?
Out of the 3! = 6 possibilities for placing P, Q , and R only one preserves the BST prop.

Bin Tree Transfo: Rotations

By the same token we have rigidity of a left rotation of node par around its parent node ch starting from the tree on the right



Digression on Rigidity

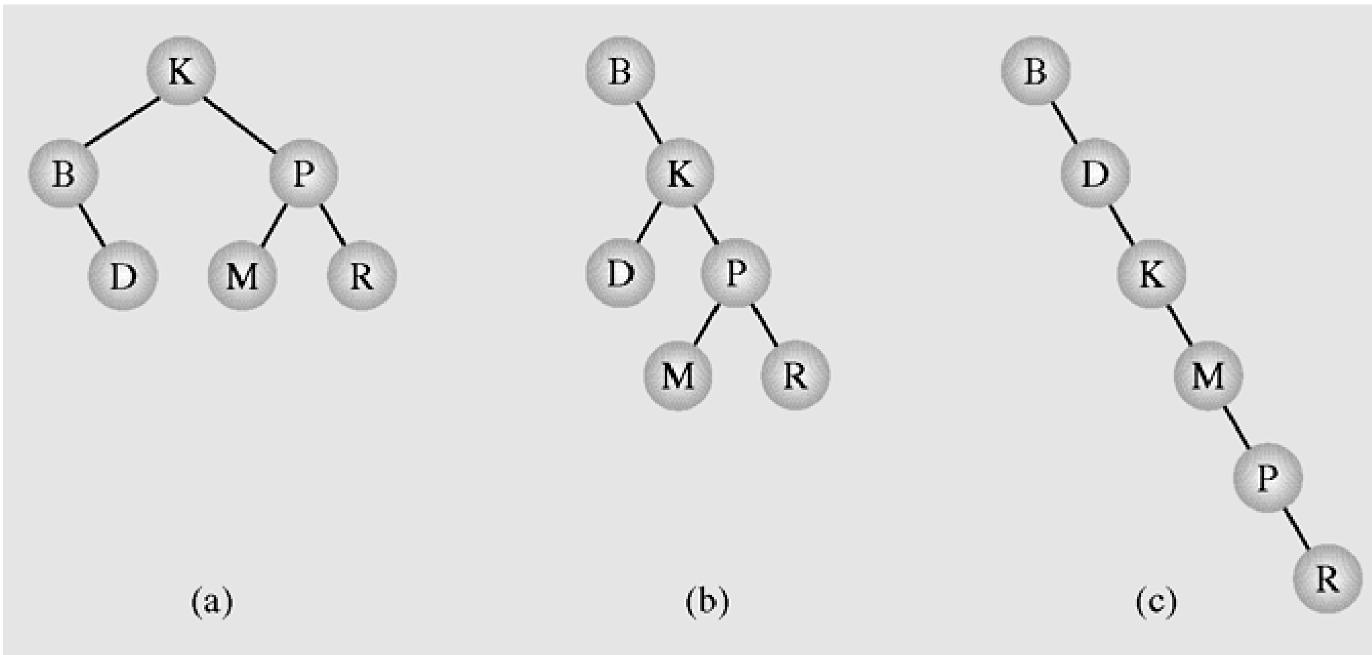
Rigidity also applies to the double rotations

End of Rigidity discussion

Balancing a Tree

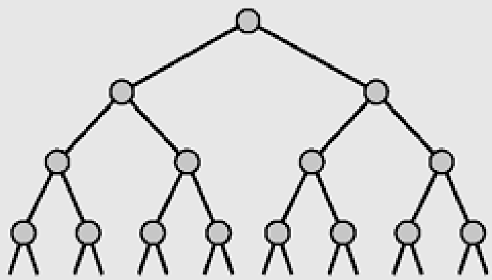
- A binary tree is **height-balanced** or **balanced** if the difference in height of both subtrees of any node in the tree is either zero or one
- A tree is considered **perfectly balanced** if it is balanced and all leaves are to be found on one level or two levels

Balancing a Tree (continued)



Different binary search trees with the same information

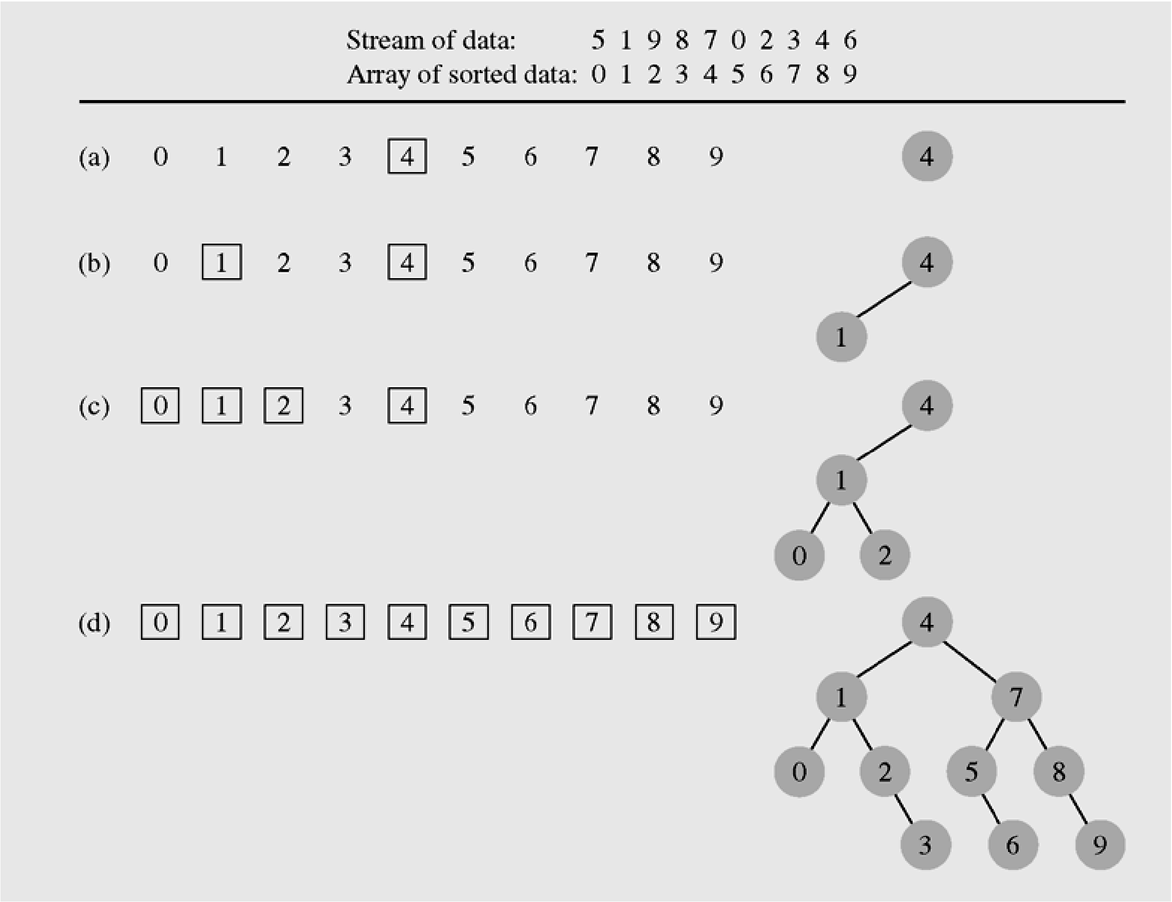
Balancing a Tree (continued)



Height	*Nodes at One Level*	*Nodes at All Levels*
1	$2^0 = 1$	$1 = 2^1 - 1$
2	$2^1 = 2$	$3 = 2^2 - 1$
3	$2^2 = 4$	$7 = 2^3 - 1$
4	$2^3 = 8$	$15 = 2^4 - 1$
⋮		
11	$2^{10} = 1{,}024$	$2{,}047 = 2^{11} - 1$
⋮		
14	$2^{13} = 8{,}192$	$16{,}383 = 2^{14} - 1$
⋮		
h	2^{h-1}	$n = 2^h - 1$
⋮		

Maximum number of nodes in binary trees of different heights

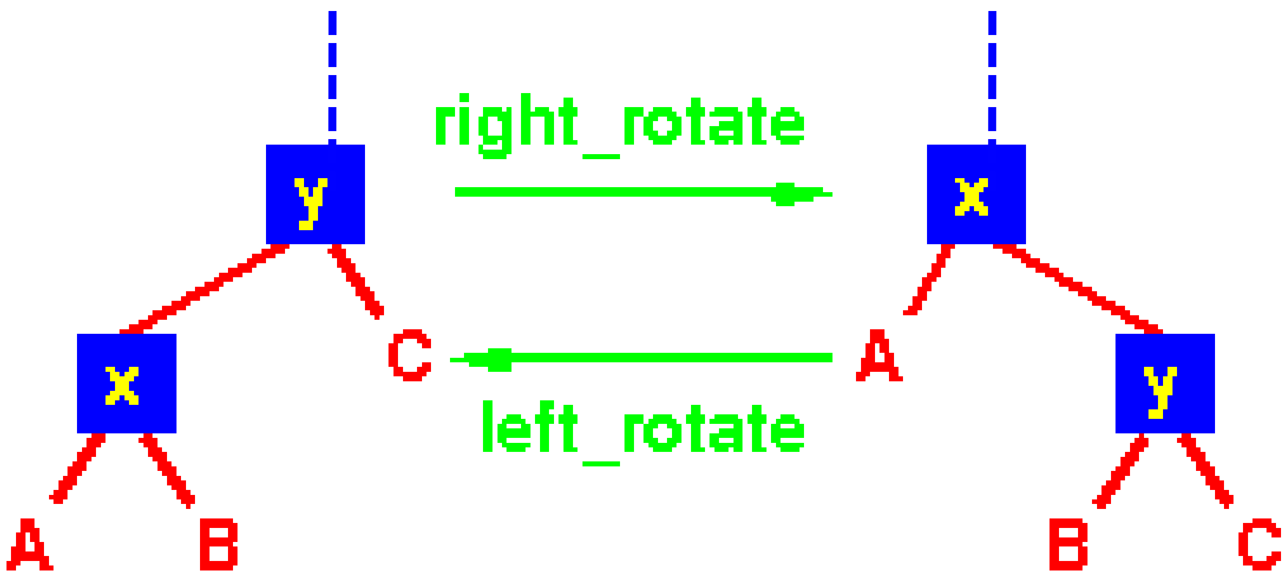
Balancing a Tree (continued)



Creating a binary search tree from an ordered array

The DSW Algorithm

- The building block for tree transformations in this algorithm is the **rotation**
- There are two types of rotation, left and right, which are symmetrical to one another as discussed earlier:



The DSW Algorithm (continued)

`rotateRight` `(Gr, Par, Ch)`

`if` `Par` ***is not the root of the tree*** // i.e., if `Gr` is not null

`{`

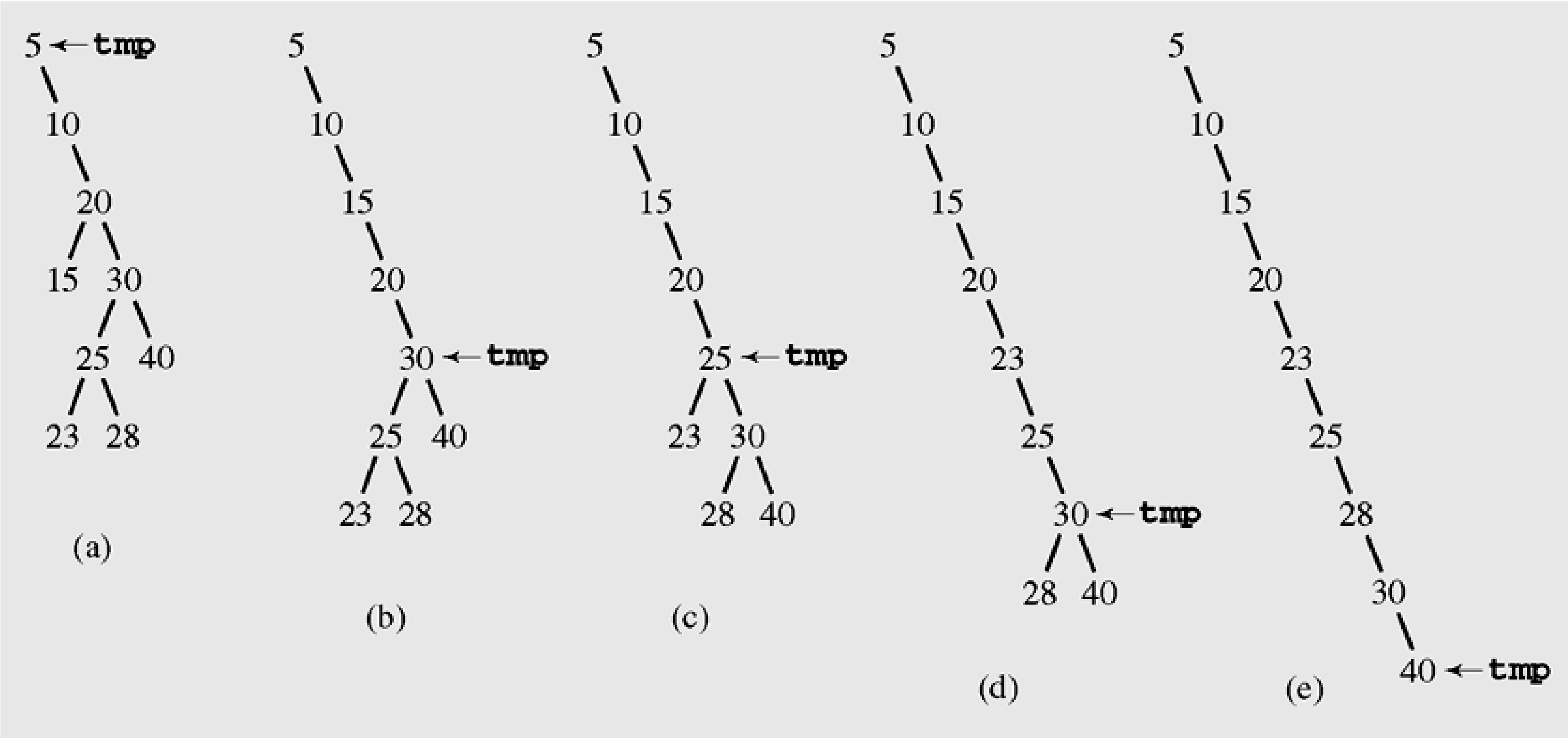
grandparent `Gr` ***of child*** `Ch` ***becomes*** `Ch’s` ***parent;***

`}`

right subtree of `Ch` ***becomes left subtree*** *of* `Ch’s` ***parent*** `Par;`

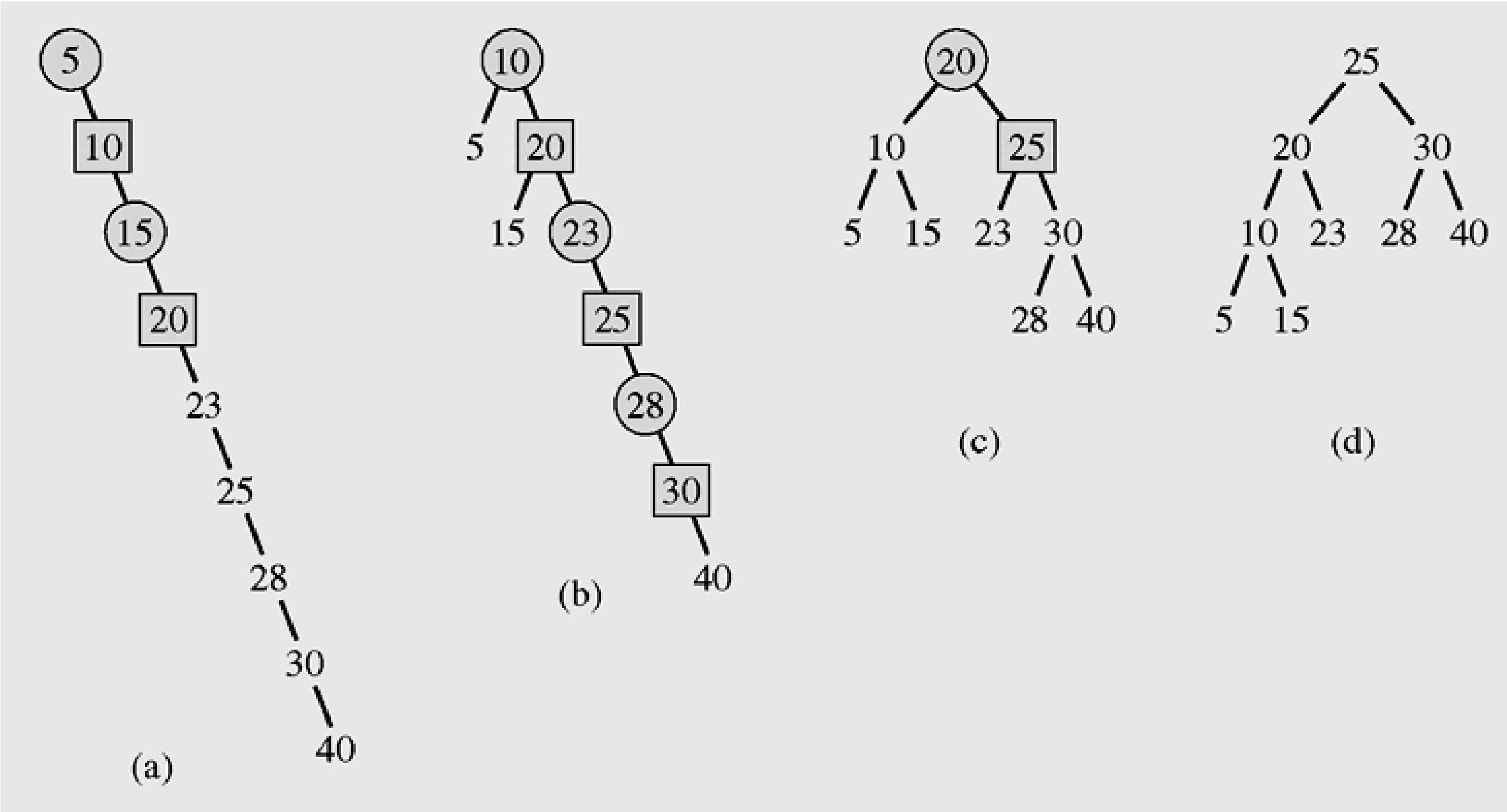
node `Ch` ***acquires*** `Par` ***as its right child***;

The DSW Algorithm (continued)



Transforming a binary search tree into a backbone

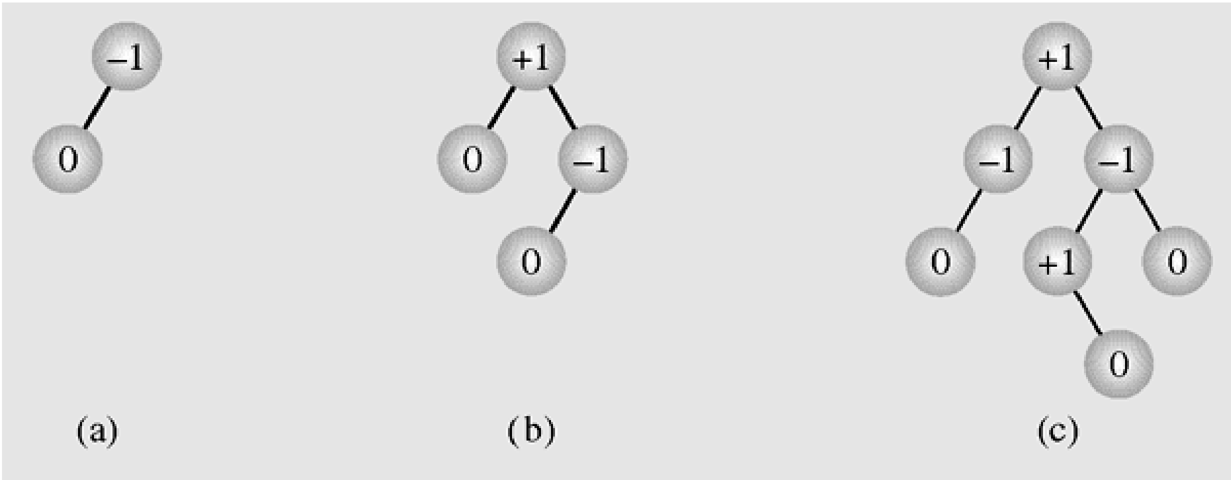
The DSW Algorithm (continued)



Transforming a backbone into a perfectly balanced tree

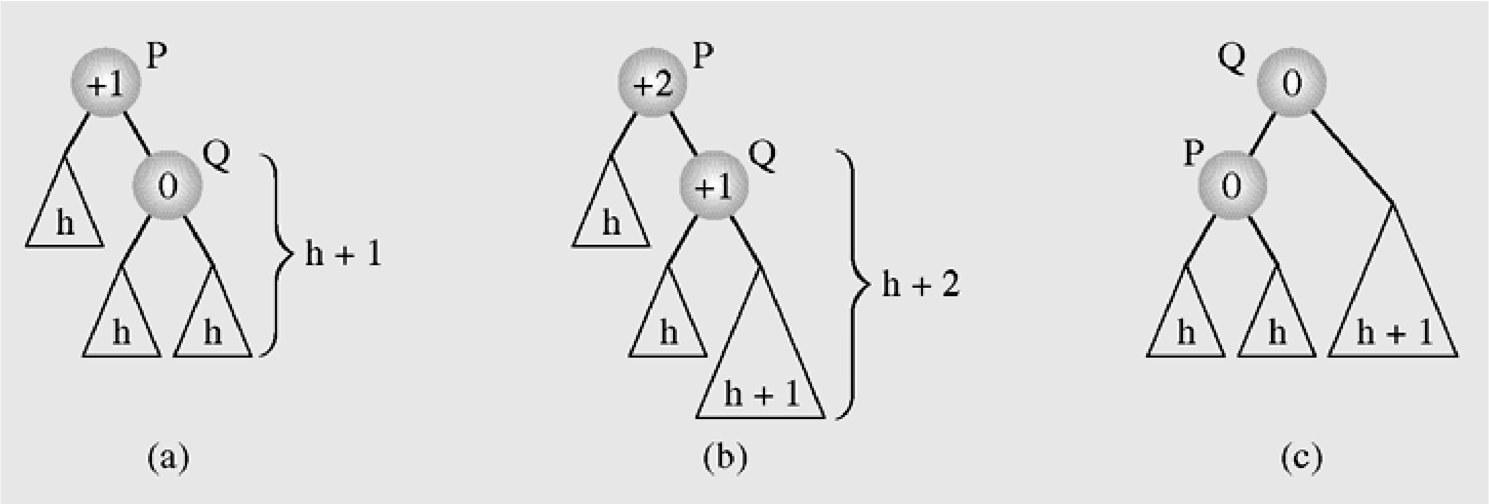
AVL Trees

- An **AVL tree** is one in which the height of the left and right subtrees of every node differ by at most one



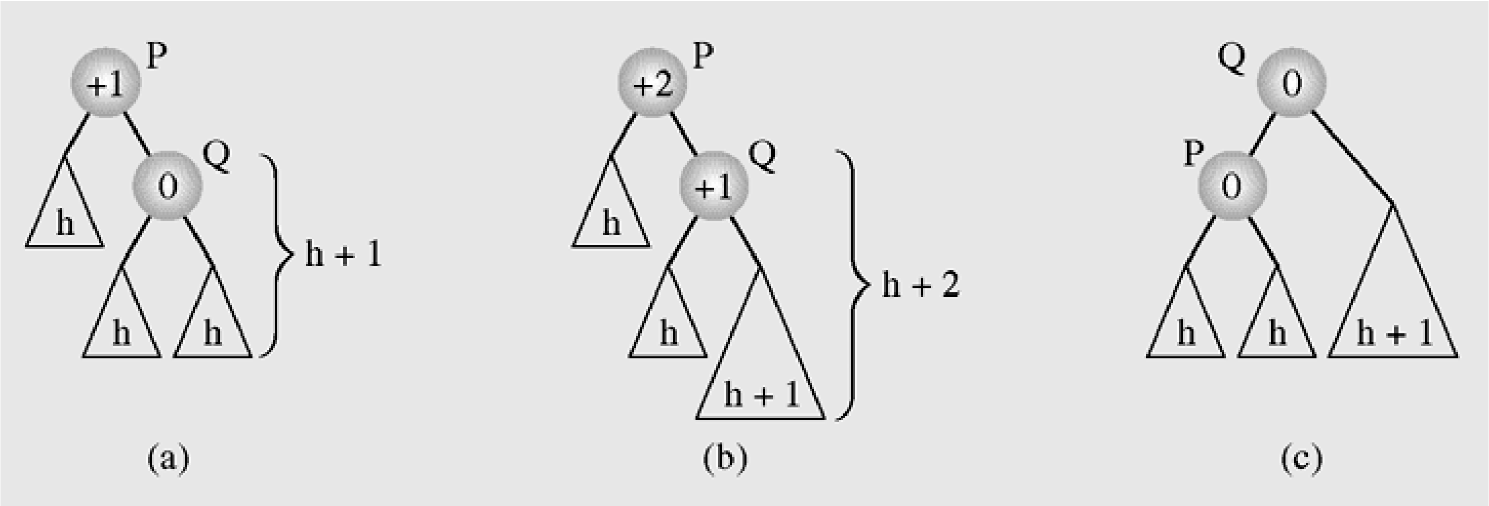
Examples of AVL trees

AVL Trees (continued)



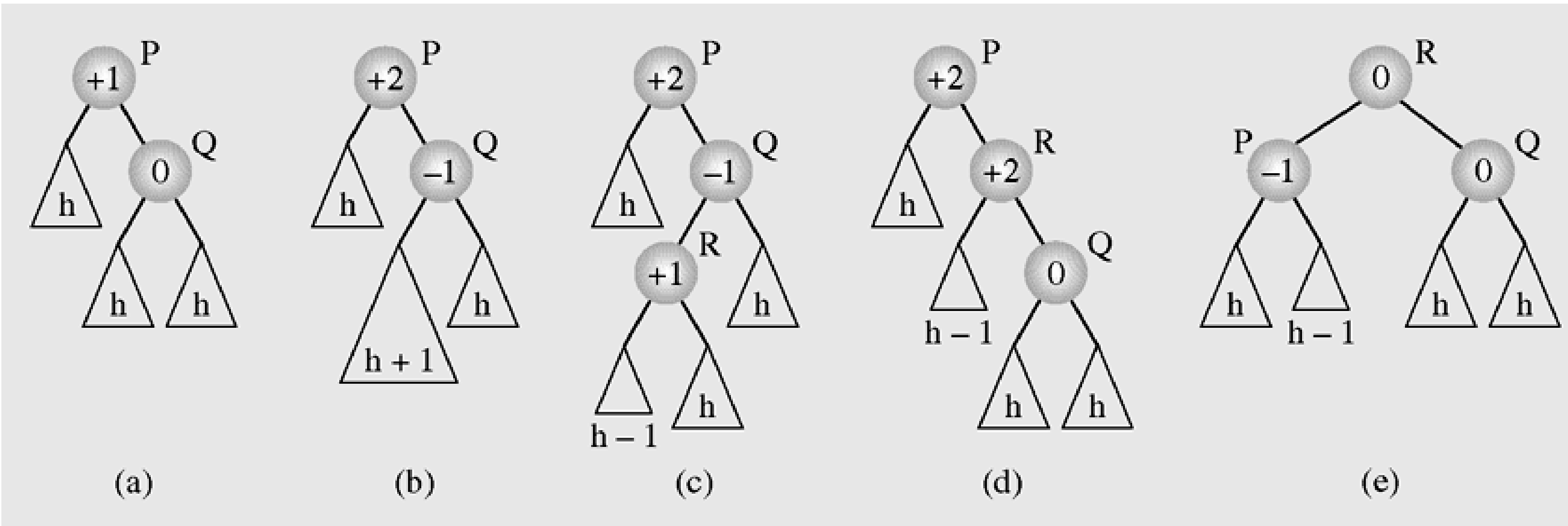
Balancing a tree after insertion of a node in the right subtree of node *Q*

AVL Trees (continued)



Balancing a tree after insertion of a node in the right subtree of node *Q*

AVL Trees (continued)



Balancing a tree after insertion **of a node in the left subtree of node *Q***

To be continued