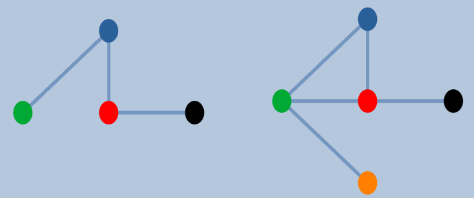
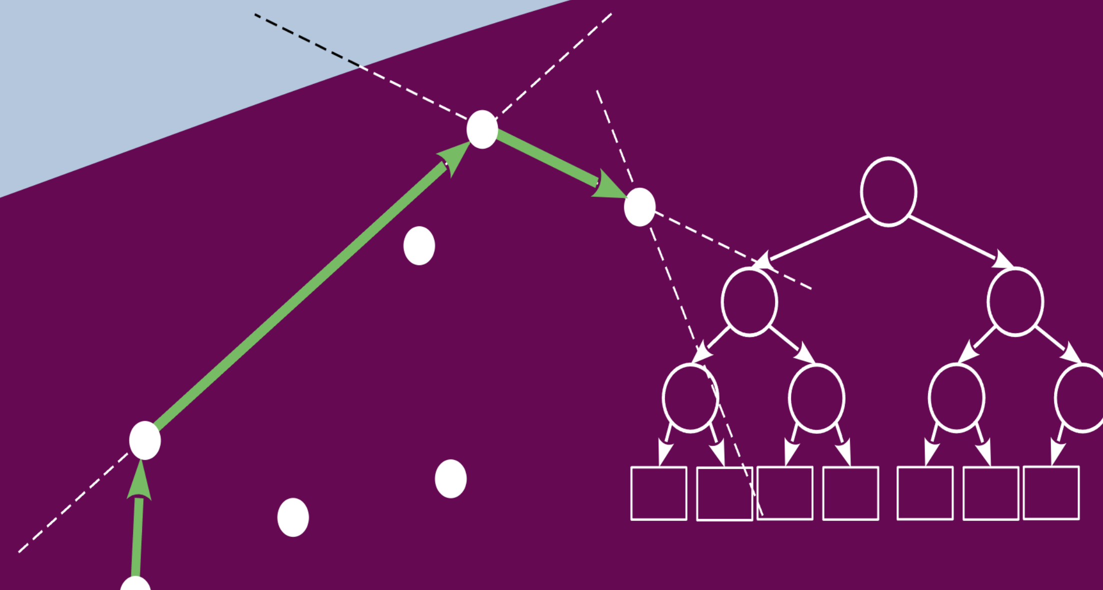
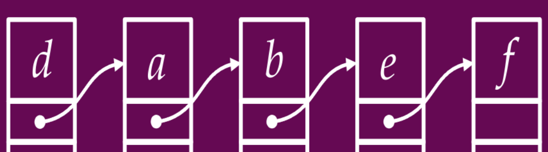
Advanced algorithms and data structures

Week 1: Advanced data structures







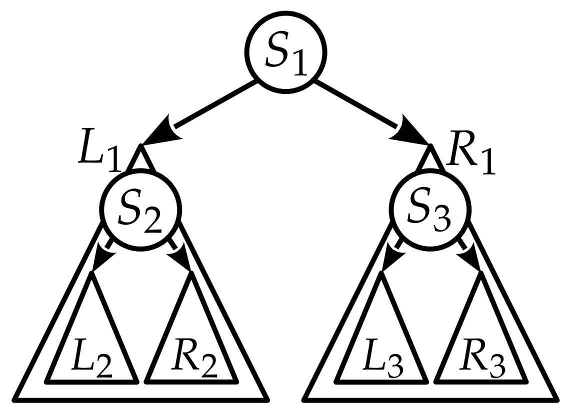
Creative Commons

Binary trees (1)

- Trees are directed acyclic graphs
- Binary trees are specific - each node can have a maximum of two children
  - A binary tree can be described by an ordered triple

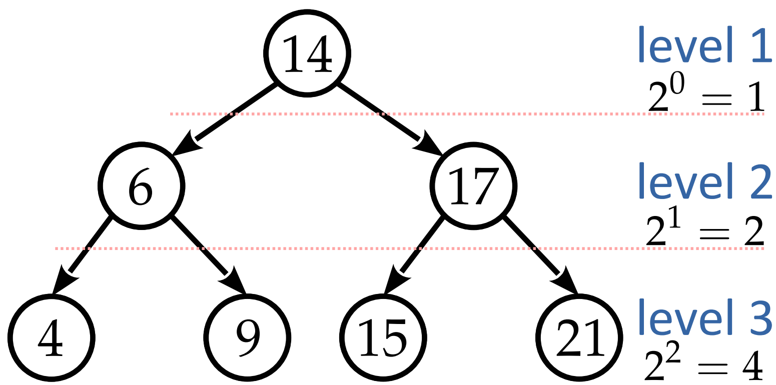
    = (, ,)

    - where L is the left subtree, S is the root of the binary tree B, and R is the right subtree
  - Recursive definition

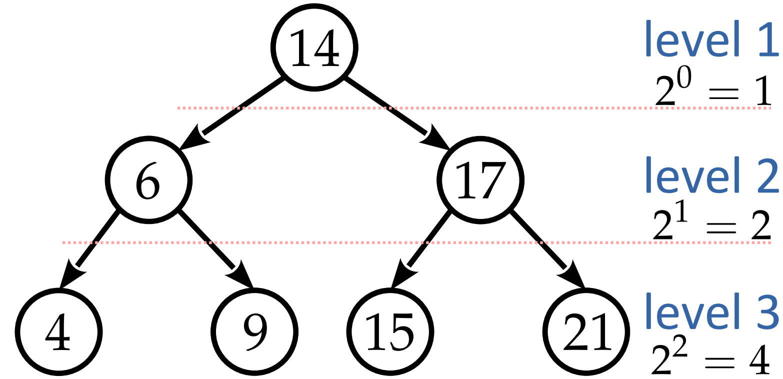


Binary trees (2)

- To describe the relationship between the nodes of a binary tree, we use genealogical terms
  - Parent, child, twins
  - Expressions such as: great-grandfather (parent's parent), uncle (twin of parents) can be used.
- **A perfect binary tree**(*perfect*)
  - A binary tree in which all levels are completely filled with nodes



Binary trees (3)



- Properties of a perfect binary tree
  - Number of nodes = 2!- 1
  - Number of sheets = 2!"#
  - Number of internal nodes = 2!"#- 1
  - Heighth = $ (+ 1 =) $2!
- **Complete binary tree**(*complete*) – A binary tree that has all but the lowest levels completely filled with nodes. In the lowest level, the sheets are filled from the left.
  - Number of nodes ≤ 2!- 1
  - Number of sheets ≤ 2!"#
  - Number of internal nodes ≤ 2!"#- 1

Binary trees (4)

- The height of the complete binary tree - directly related to the complexity of the search

$$h = \lceil \,!(\quad + 1) \qquad\qquad \rceil$$

  - It is also valid

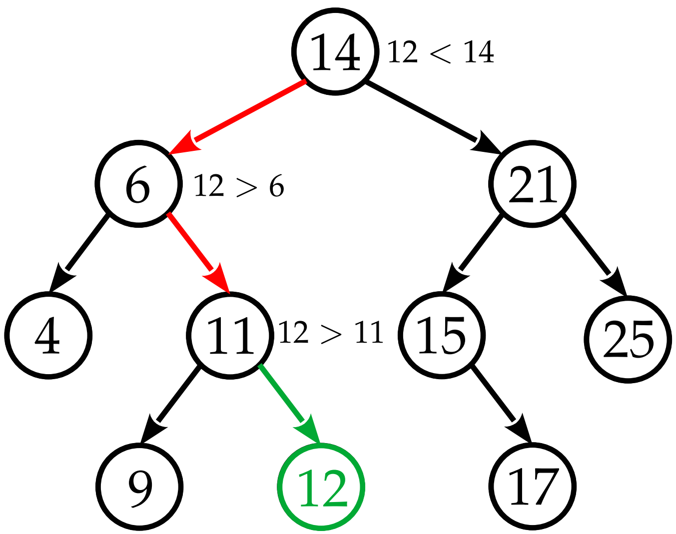
$$= \quad + \quad \leq 2^{!} - 1$$

- **Full binary tree**(*full*) – A binary tree whose internal nodes have exactly two children.
- Organization**sorted**binary tree $= (\; , \; , \;)$
  - No duplicates $\big(\; (\;\;) \big) < \;\; (<) \qquad \big(\; (\;\;) \big)$
  - With duplicates

$$\big(\; (\;\;) \big) \leq \;\; (<) \quad < \big(\; (\leq \;\;) \big)$$
$$\big(\; (\;\;) \big) \qquad (\;\;) \qquad \big(\; (\;\;) \big)$$

Binary tree operations (1)

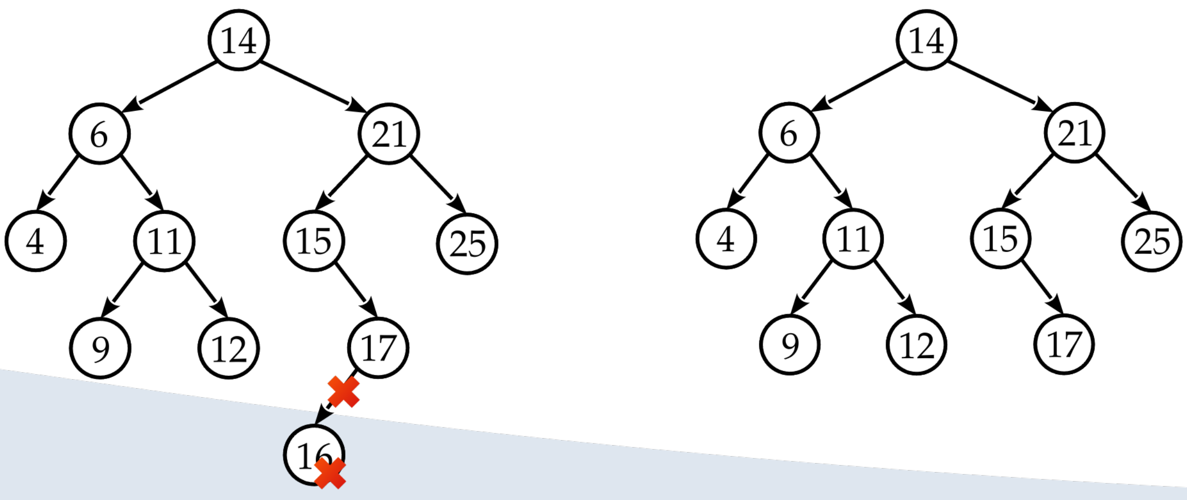
- **Let's repeat** adding values to a binary tree
  - Adding is preceded by a search
  - Upon encountering a free place, we add a new node according to the organization of the binary tree:
    - Left if smaller
    - To the right if it's bigger



Deleting Nodes (1)

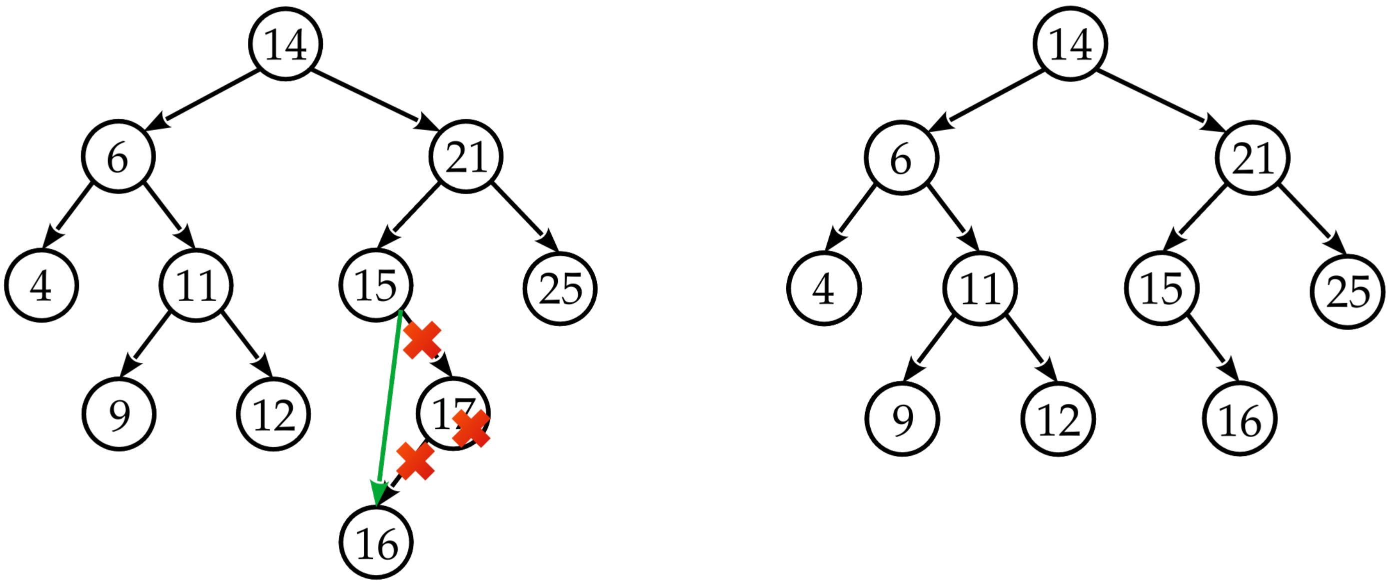
- First we find the node we are deleting
- Three cases:
    1. The node to be deleted is a leaf
    2. The node being deleted has one child
    3. The node being deleted has both children

1. The node to be deleted is a leaf – we just delete the node



Deleting nodes (2)

2. The node has one child – That child becomes the new child of the parent of the node being deleted



Deleting nodes (3)

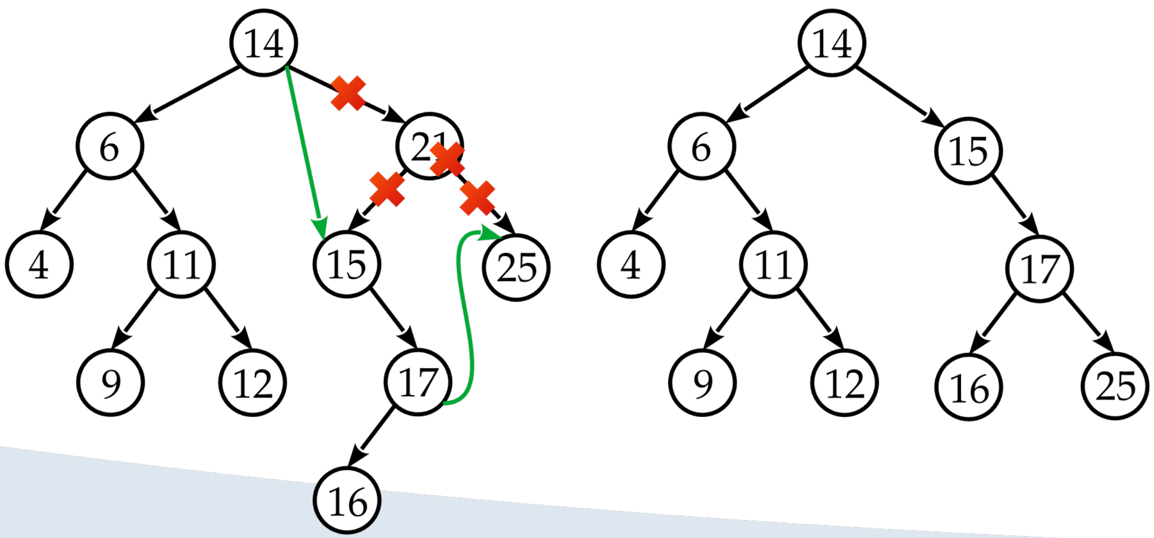
3. The node has two children - options

- Delete by merging – significantly changes the structure
- Delete by copy – the structure changes minimally

- Deletion by union
  - Find the node being deleted and its parent (if the root node is not being deleted)
  - Let's determine on which side the node that is being deleted is in relation to its parent
    - If the root node is in question, then it does not matter which subtree we take as the union subtree (*merging subtree*)
    - Otherwise we have two options:
      - If the node to be deleted is in**the left**subtree of the parent, then the union tree is its right subtree
      - If the node to be deleted is in**with the right**subtree of the parent, then the union tree is its left subtree

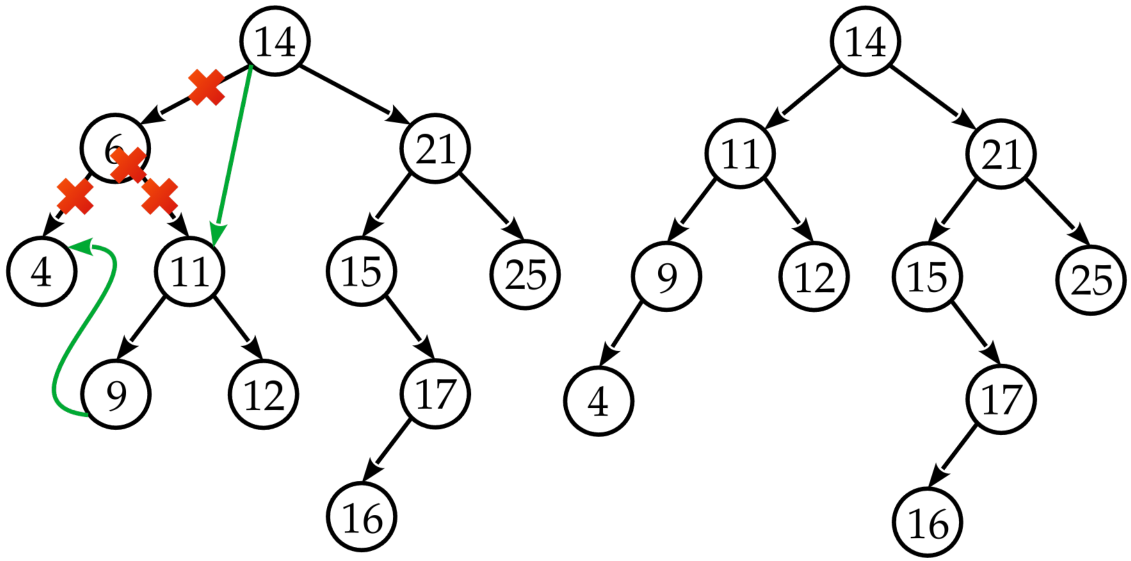
Deleting nodes (4)

- Deleting node A by union
  - We connect the twin node to the root node of the union subtree at
    - Knot**follower**if the subtree of the union was the right subtree of node A
    - Knot**predecessor**if the subtree of the union was the left subtree of node A
  - We connect the root of the union subtree with the parent of the deleted node
    - Except in the case when the root node is deleted



Deleting nodes (5)

- Deletion by union

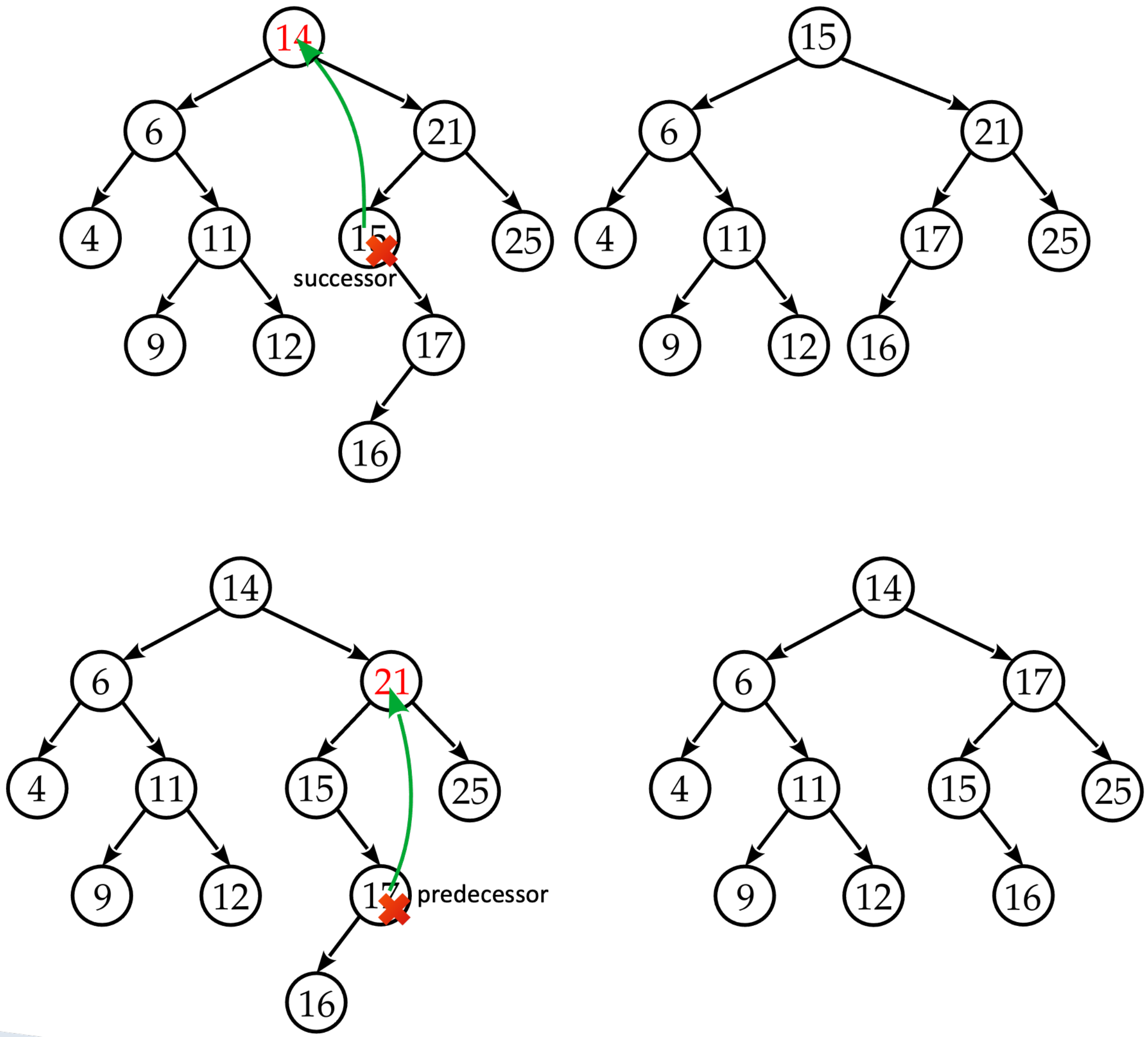


Deleting nodes (6)

- Delete by copying
  - It boils down to deleting a node without children or with one child
  - A replacement node is found and deleted - the binary tree structure is minimally changed

1. We find the node A that we are deleting
2. We find the node X that contains a direct predecessor or successor – a replacement node
   - The predecessor is the rightmost node in the left subtree of A
   - A follower is the leftmost node in the right subtree of A
3. We copy the value of the replacement node X to the node A that is being deleted
4. The replacement node X is removed

Deleting nodes (7)



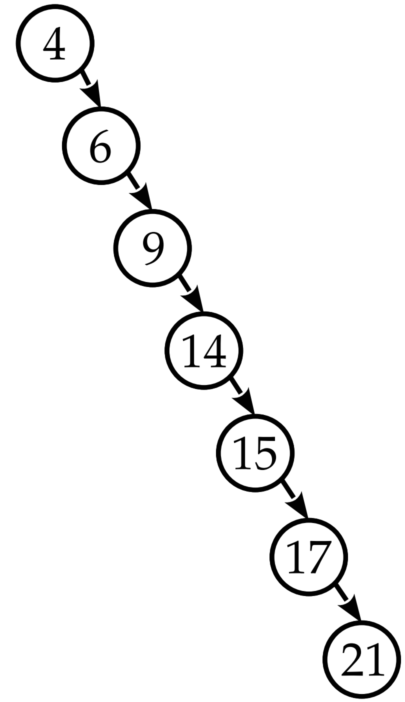
Balanced Binary Tree (1)

- Why is a balanced binary tree interesting to us?

- We want to achieve a search complexity of

  (!)

- The other extreme -**degenerate (oblique) binary tree** (*degenerates*) – example on the right
  - The complexity is ( )

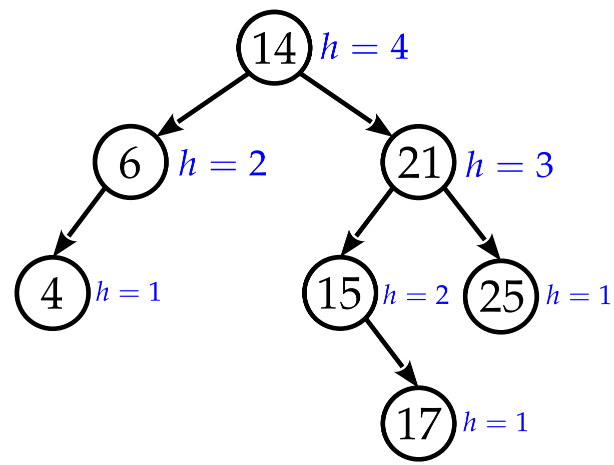


Balanced Binary Tree (2)

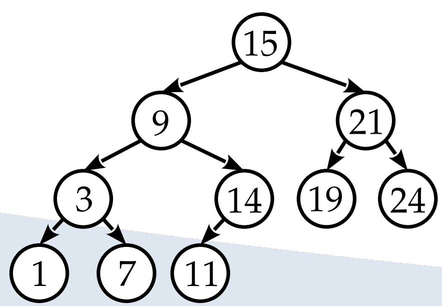
- For a binary tree $= \ (, \ , \ )$ the definition of a balanced tree is

$$\forall \quad |h(\ ) - h(\ ) \leq| 1$$

  - The difference in the height of the left and right subtrees **of each node** can be at most 1



- **A perfectly balanced binary tree** (*perfectly balanced*) – balanced and complete
  - All levels except the last one are completely filled with nodes



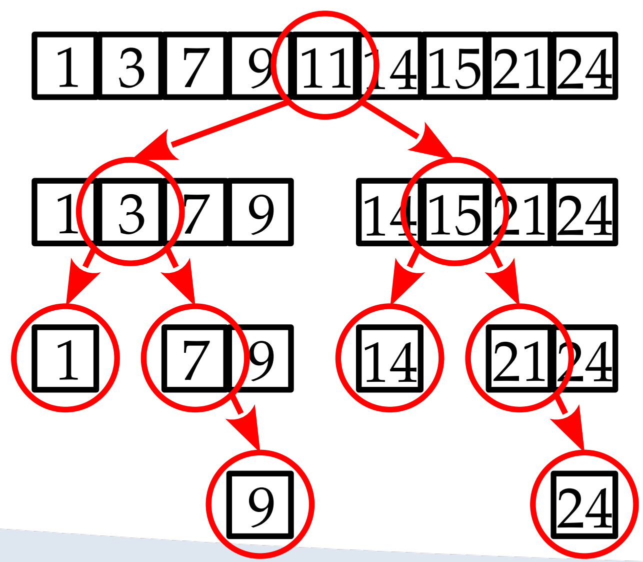
Creating a binary tree (1)
(from a sorted array of values)

- We have a sorted field available (*array*) values

$V_s = \langle 1,3,7,9,11,14,15,21,24 \rangle$

1. We find the positional mean value *c* in the field
2. Let's create a root value node *c* of the current binary tree
3. For the subfield to the left of *c* the left subtree is recursively created
4. For the subfield to the right of *c* the right subtree is recursively created
5. We repeat until we can create a left or right subtree

function CREATEBALANCEDTREE(V_s)
    $n \leftarrow |S|$
    if $n > 0$ **then**
        $i \leftarrow (n \div 2) + (n \,\%\, 2)$
        $root \leftarrow$ create node having value $V_s[i-1]$
        $leftChild(root) \leftarrow$ CREATEBALANCEDTREE($V_s[0, i-1]$)
        $rightChild(root) \leftarrow$ CREATEBALANCEDTREE($V_s[i, n]$)
        return $root$
    else
        return nil



Creating a binary tree (2)
(from a sorted array of values)

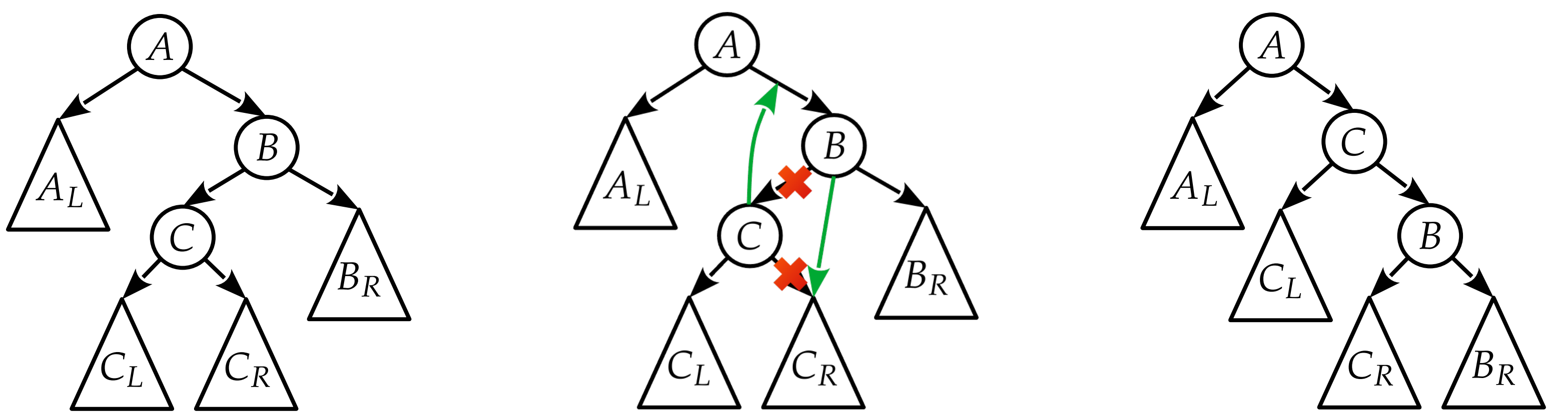
- This algorithm has complexity (! +)
  - Field sorting + traversal of all field elements
- The algorithm can only be used in special situations
  - When we have a field of values and we want to create a balanced binary tree from it
  - It is used relatively often, concrete examples will be given later in the lectures
- The binary tree created in this way is balanced

Rotations in the tree (1) - right

- To balance trees we need two operations (rotations) on binary trees (pulley analogy)

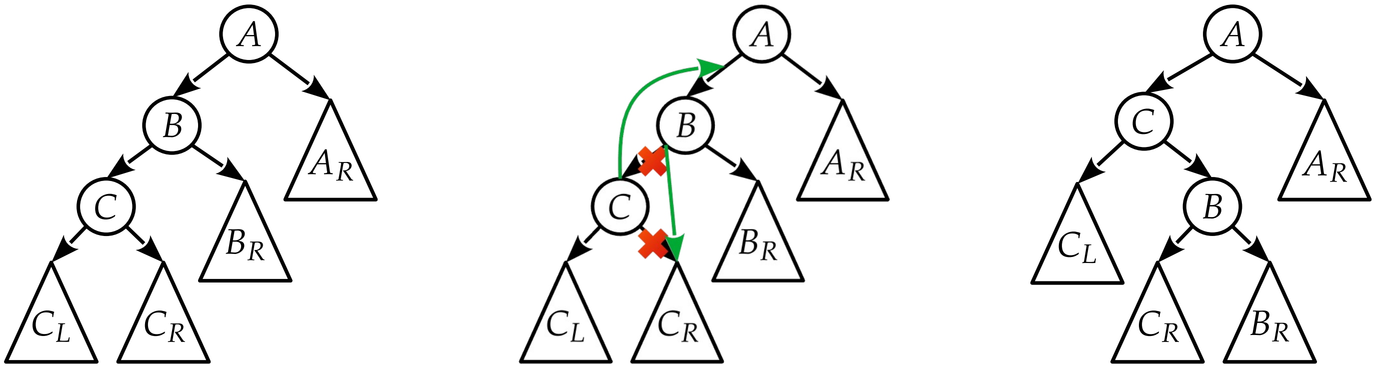
- Right rotation of C about B
  - How to rotate the tree so that C is between A and B, while preserving the order

#(<)< #(<) < ($<) < (($)) ( ) ( )

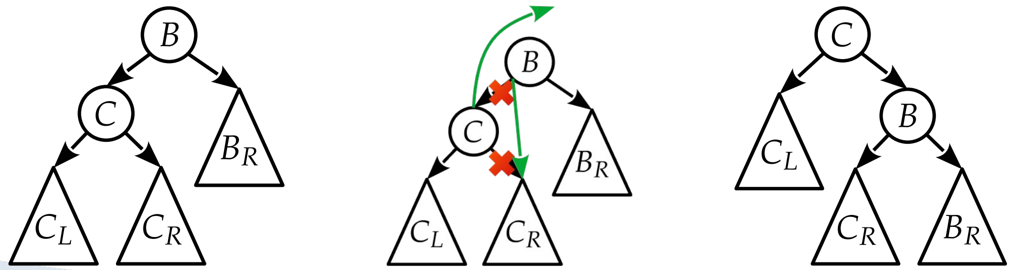


Rotations in the tree (2) - right

- Another right rotation of C around B



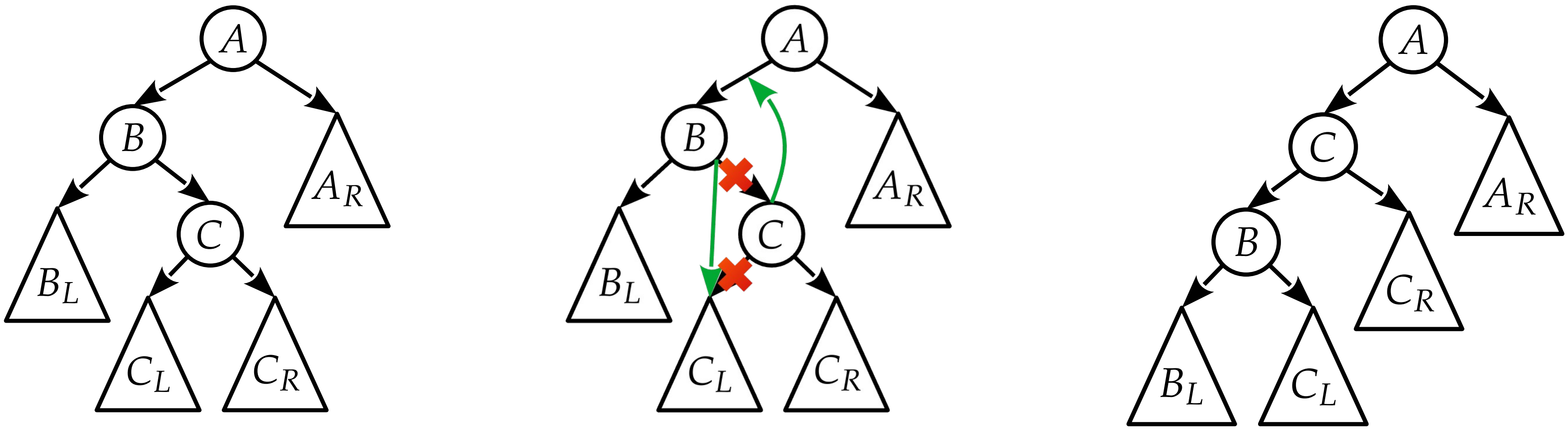
  - The right child of C becomes the left child of B
  - B becomes the right child of C
  - C becomes a child of the former parent of node B (if it exists)



Rotations in the tree (3) - left

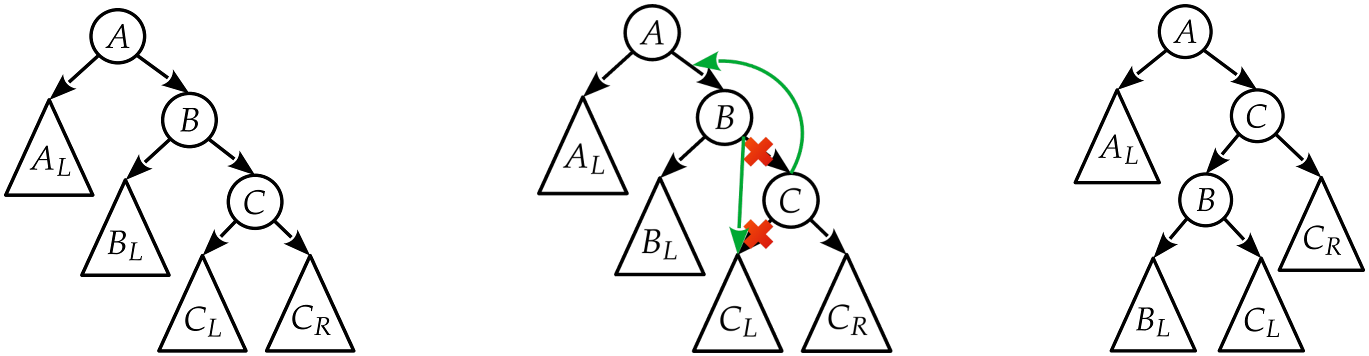
- Left rotation of C around B
  - How to rotate the tree so that C is between A and B, while preserving the order

#(<)< #(<) < ($<) < (($)) ( ) ( )

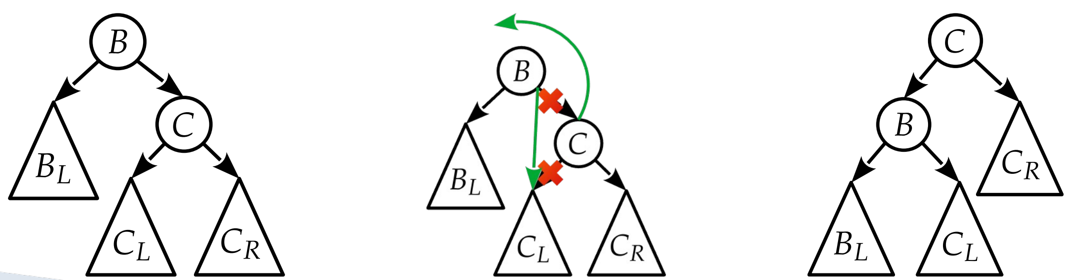


Rotations in the tree (4) - left

- Another left rotation of C around B



  - The left child of C becomes the right child of B
  - B becomes the left child of C
  - C becomes a child of the former parent of node B (if it exists)



Day-Stout-Warren algorithm (DSW)

- Two phases of the algorithm

  1. Making the spine (oblique tree)

  2. Recursively breaking the spine back into a complete tree

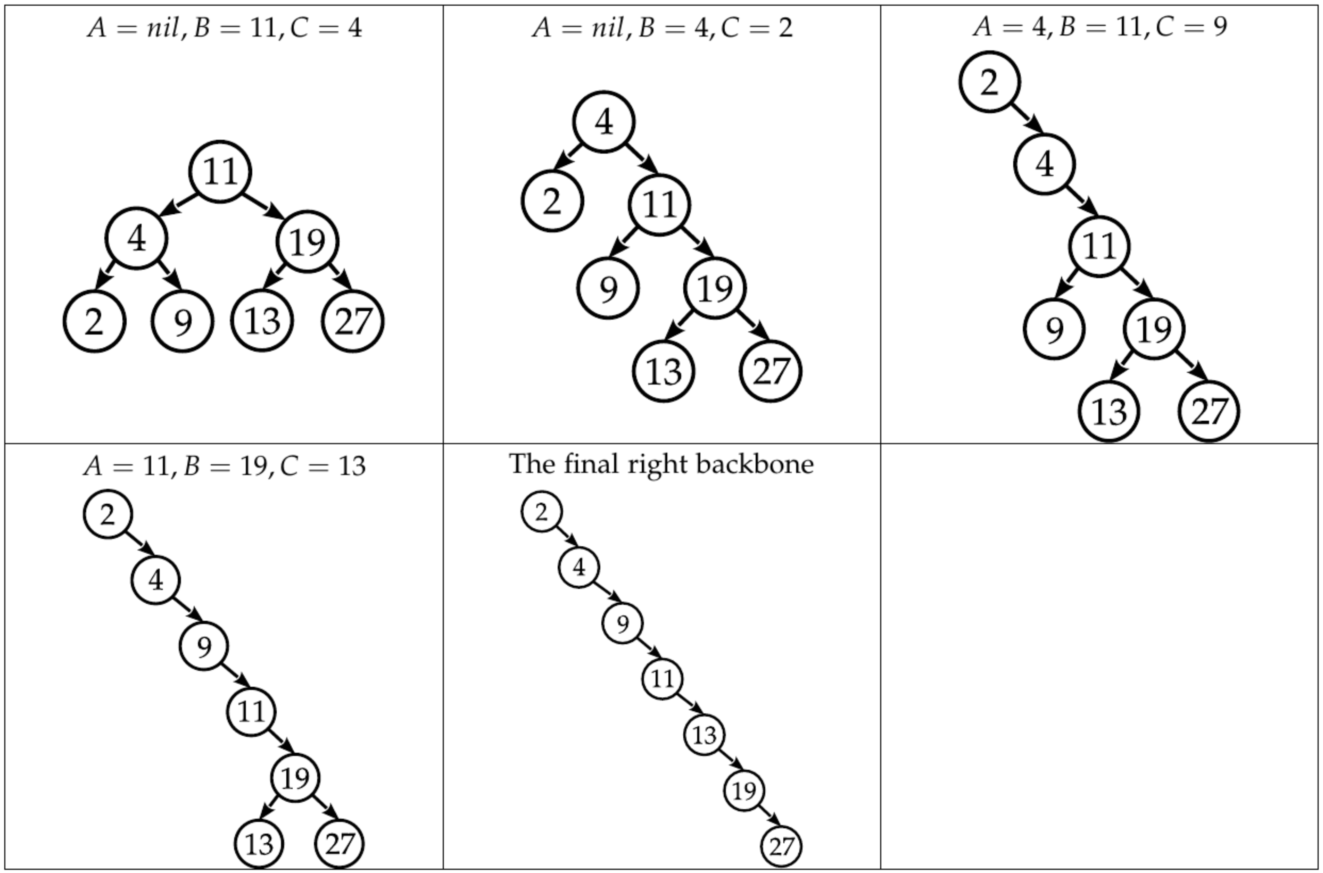
DSW – spine manufacturing

```
procedure RightBackbone(root)
    B ← root
    A ← nil
    while B ≠ nil do
        C ← leftChild(B)
        if C ≠ nil then
            RightRotate(A, B)
            if A = nil then
                root ← C
            B ← C
        else
            ▷ Descending right to the first node that has the left child
            A ← B
            B ← rightChild(B)
```

- The first step is to create a backbone from a binary tree - eg right spine
- We rotate the left children of the nodes to the right
- We repeat right rotations until there are no left children

DSW – spine manufacturing

- Example



DSW - breaking

- Strategically positioned left rotations for a perfectly balanced binary tree

procedure DSW($tree, n$)
  $h \leftarrow \lceil \log_2(n+1) \rceil$
  $i \leftarrow 2^{h-1} - 1$
  perform $n - i$ rotations of every second node from the root
  while $i > 1$ **do**
    $i \leftarrow \lfloor i/2 \rfloor$
    perform i rotations of every second node from the root

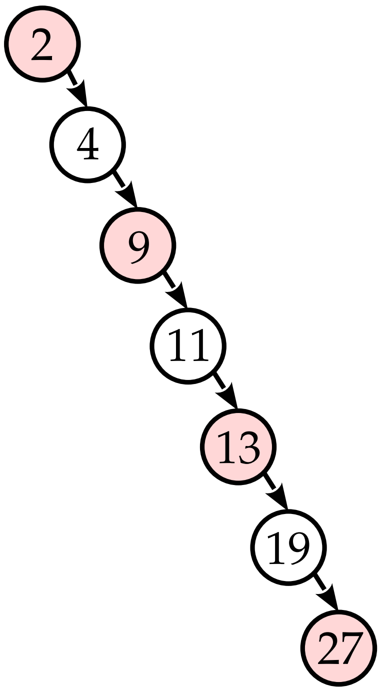
  - h – height of the binary tree for n nodes
  - i – number of internal nodes
  - For the right spine, we do left rotations to bring the nodes back into the binary tree structure

DSW – breaking, example

= 7

$h = 3$

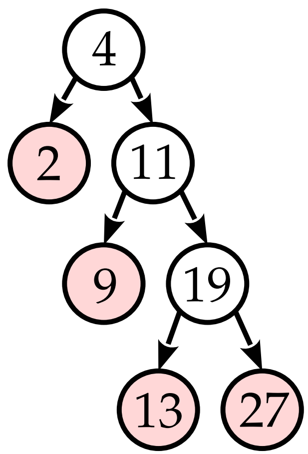
= 3



$- \quad = 4$

left rotations

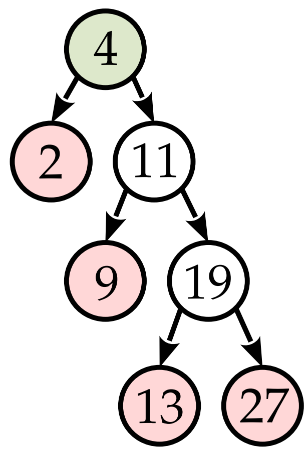
for leaves



$\lfloor /2 \rfloor = \lfloor 3/2 \rfloor = 1$

of left rotations for

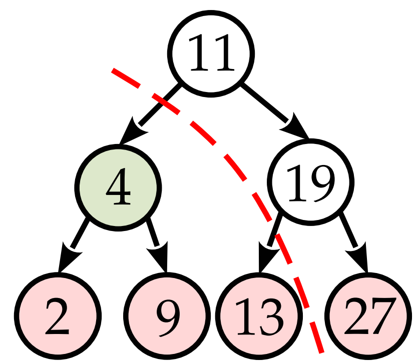
internal nodes



it is obtained

perfectly binary

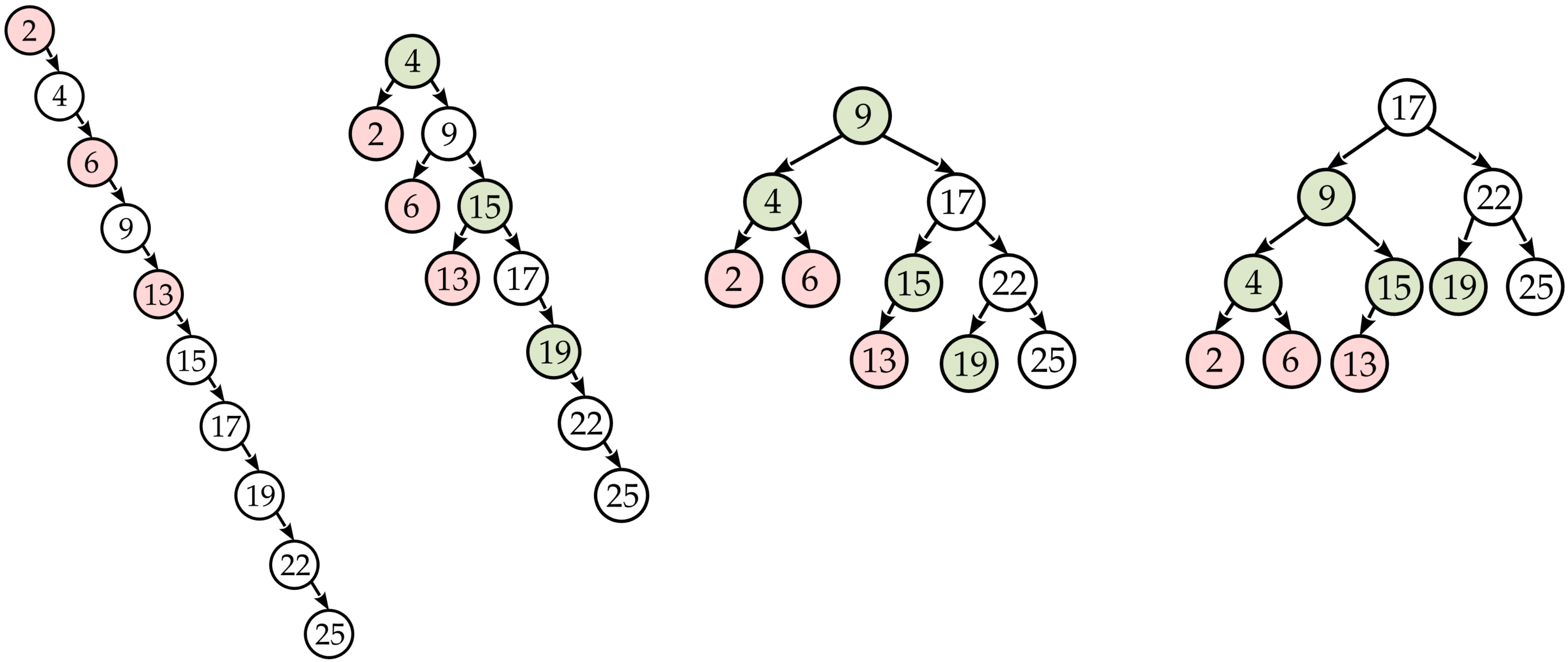
tree



DSW – breaking, example

- Example
  - $= 10$
  - It is $h = 4$ levels
  - The total number of internal nodes is $= 7$
  - Nodes in the lowest level is $10 - 7 = 3$ – first we do 3 left rotations of every other node of the right spine, starting from the root node
  - Then we work $\lfloor 7/2 \rfloor = 3$ left rotations of every other node of the rest of the right spine, starting from the root node, which gives us the lowest level of internal nodes
  - Then we work $\lfloor 3/2 \rfloor = 1$ left rotations of every other node of the rest of the right spine, starting from the root node, which gives us the lowest level of internal nodes

DSW – breaking, example



- This is global binary tree balancing
  - We first destroy the structure of the tree to balance it
  - The complexity of the DSW algorithm is ( )

Adelson-Velski-Landis binary tree (AVL)

- Previous examples are **offline** balancing

- **Online** balancing – adding new values
  - Check if the binary tree is balanced?
  - If not, balance that tree with minimal intervention in its structure

- This is called local balancing

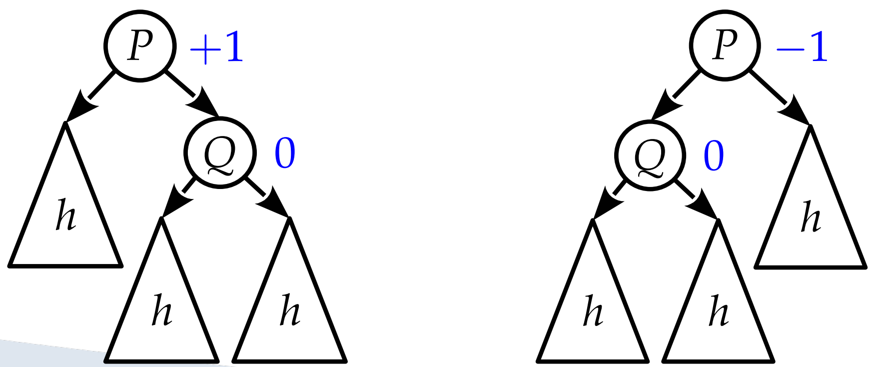
AVL (2)

- By adding a new leaf, we can move along the path to the root node, and check the balance in each node

- For a binary tree $= (, ,)$ we define the balance factor (*balance factor*) as

$$= h(\ ) - h(\ )$$

- For all nodes that have $-1 \le (\ ) \le 1$ we consider their subtree to be balanced



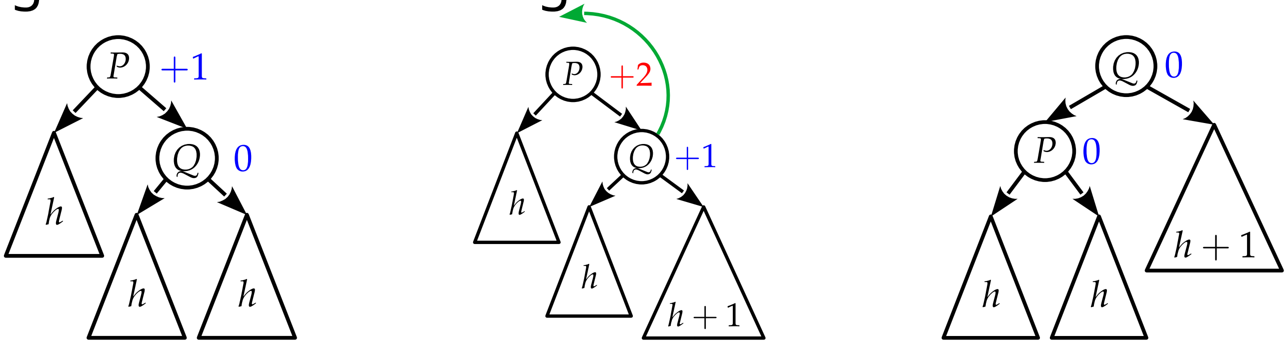
AVL (3)

- After adding, we update the balance factors on the vertical path. If there is a node S with $( ) = -2$or $( ) = 2$,local balancing is required

- Two cases – a node and its child (depending on the sign of the node):
  - **Level up**case, identical signs BF:
    - The balance factor of the node is +2 and the right child has a balance factor of 0 or +1 – the right flattened case
    - The balance factor of a node is -2 and the left child has a balance factor of 0 or -1 – the left aligned case
  - **Broken**case, strictly opposite signs BF:
    - The balance factor of the node is +2 and the right child has a balance factor of -1 – the right broken case
    - The balance factor of the node is -2 and the left child has a balance factor of +1 – the left broken case

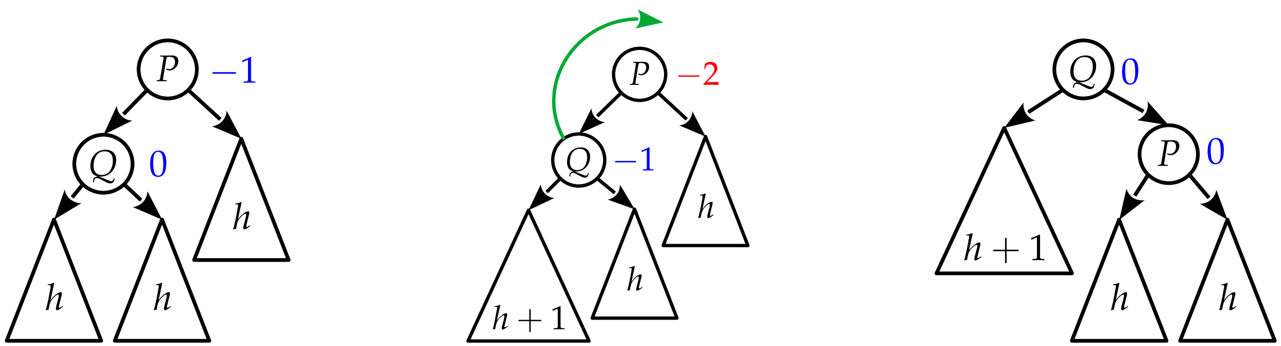
AVL (4)

- Right aligned case: +2 and right child 0 or +1



  - We add the new value to the right subtree of node Q
  - A left rotation of Q around P is performed

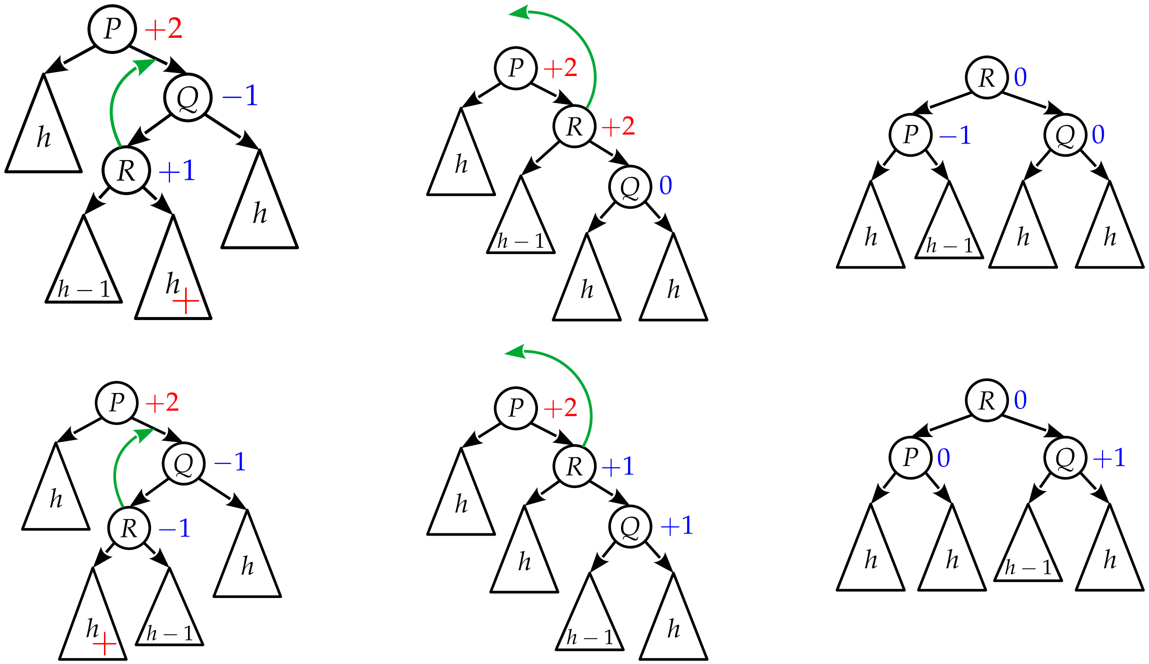
- Left aligned case: -2 and left child 0 or -1



  - A right rotation of Q around P is performed

AVL (5)

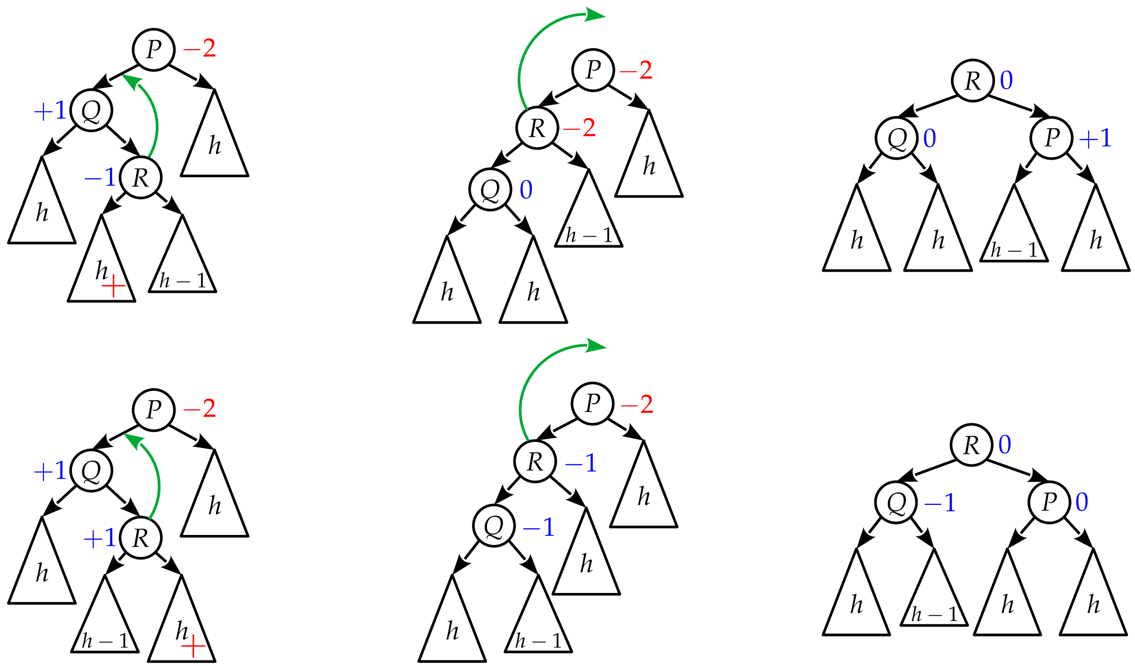
- Right fractured case: +2 and right child -1



  - First the right rotation of R about Q
  - Then a left rotation of R around P

AVL (6)

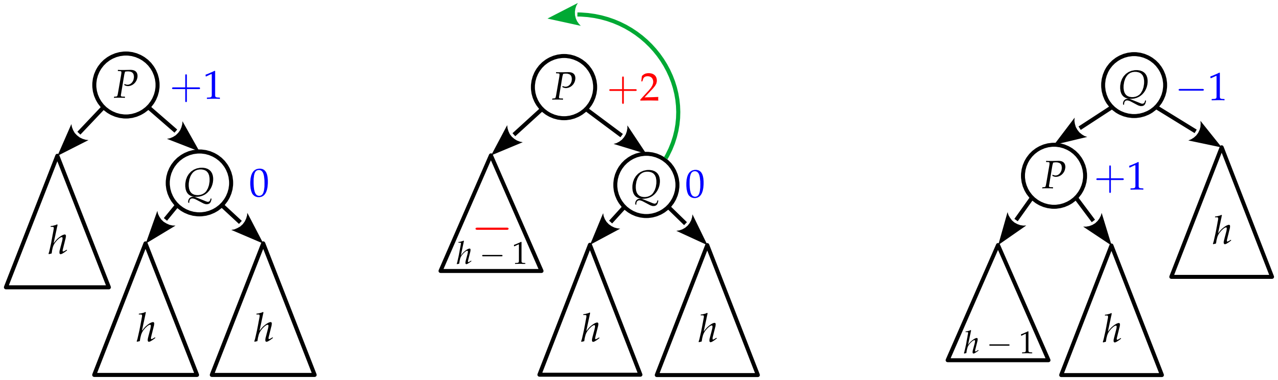
- Left broken case: -2 and left child +1



  - First a left rotation of R about Q
  - Then a right rotation of R around P

AVL (7)

- Deleting values - always delete by copying
  - With such deletion, the height of one of the subtrees can be reduced



  - We can see that deleting a node in the left subtree of node P caused an imbalance
  - It is the right flat case

AVL (8)

```
function AVLDetectRotate(n)
    if balanceFactor(n) is +2 then
        n1 ← rightChild(n)
        if balanceFactor(n1) is 0 or +1 then
            left rotate n1 around n
        if balanceFactor(n1) is −1 then
            n2 ← leftChild(n1)
            right rotate n2 around n1
            left rotate n2 around n
    else
        n1 ← leftChild(n)
        if balanceFactor(n1) is 0 or −1 then
            right rotate n1 around n
        if balanceFactor(n1) is +1 then
            n2 ← rightChild(n1)
            left rotate n2 around n1
            right rotate n2 around n

procedure AVLBalance(n)
    p ← parent(n)
    if balanceFactor(n) is −2 or +2 then
        AVLDetectRotate(n)
    if p is not nil then
        AVLBalance(p)
```

- The complexity of the search is the same as for a classic binary tree (!)
- When writing or deleting values, we return along the vertical path back to the root node, which gives complexity (2 !)
- The theoretical height of the AVL tree is
  !(+ 1) ≤ *h* ≤ . !(+ 2) − 0.328
  - Proof in Drozdek

Questions?