Lecture 11

- Binary search tree
- Balanced binary search tree
  - AVL

Lect. PhD. Lupsa Dana
Babes - Bolyai University
Computer Science and Mathematics Faculty
2022 - 2023

Binary search trees

A Binary Search Tree (BST) is a binary tree that satisfies the following property:

if x is a node of the binary search tree then:

- for every node y from the left subtree of x, the information from y is less than or equal to the information from x
- for every node y from the right subtree of x, the information from y is greater than or equal to the information from x

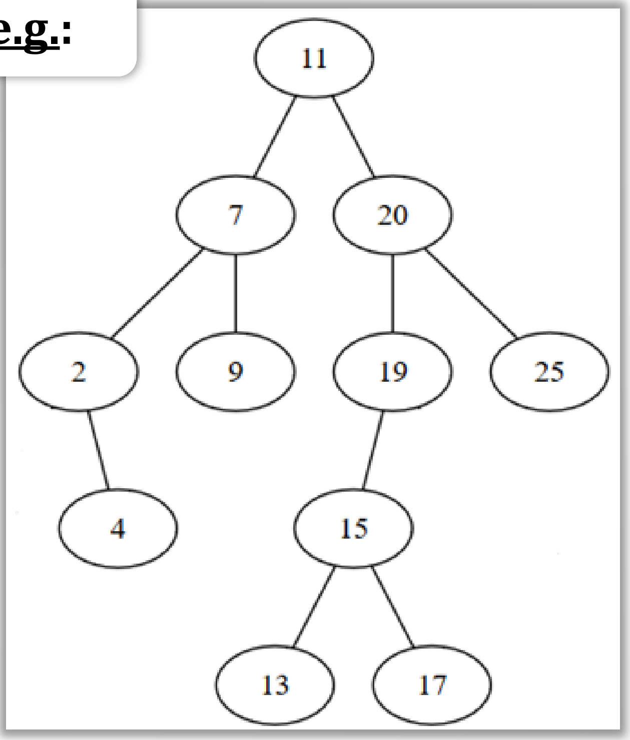
Remarks:

- In order to have a binary search tree, we need to store information in the tree that is of type TComp.
- The relation used to order the nodes can be any relation $\mathcal{R}$.

  We are going to use “<=“ by default.

Binary search tree

e.g.:

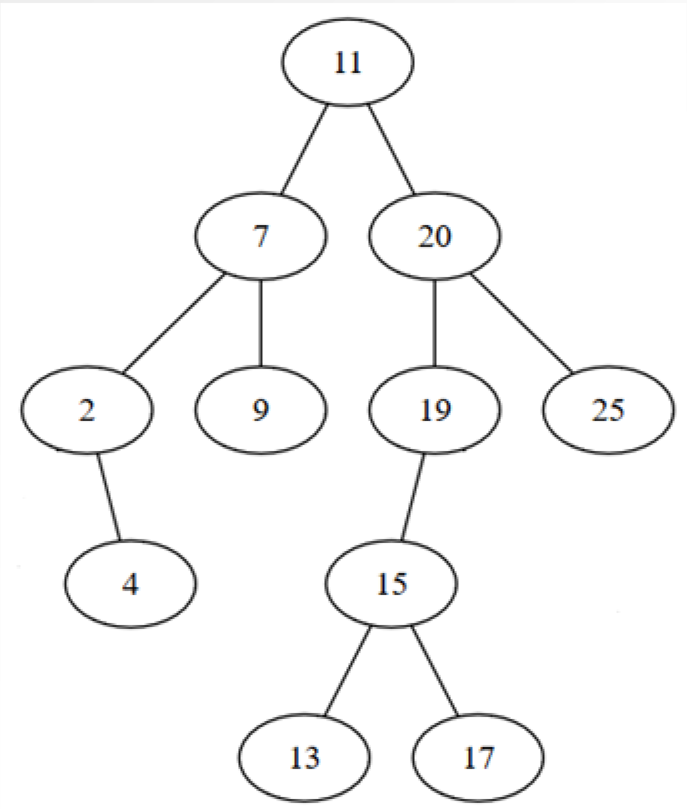


- If we do an inorder traversal of a binary search tree, we will get the elements in increasing order (according to the relation used).
- Binary search trees can be used as representation for sorted containers: sorted maps, sorted multimaps, priority queues, sorted sets, etc.

What about SortedList?

BST: Operations

.



- How can we search for element 15? And for element 14?
- How/Where can we insert element 14?

How can we implement this operations recursively and non-recursively?

- How can we remove the value 25? And value 2? And value 11?

BST - search operation (recursive)

```
BSTNode:
      info:  TComp
      left:   ↑ BSTNode
      right: ↑ BSTNode
BinarySearchTree:
      root: ↑ BSTNode
```

```
function search_rec (node, elem) is:
    if node = NIL then
        search_rec ← false
    else
        if [node].info = elem then
            search_rec ← true
        else if  elem ≤ [node].info then
                search_rec ← search_rec([node].left, elem)
            else
                search_rec ← search_rec([node].right, elem)
        end-if   end-if
    end-if
end-function
```

Complexity of the search algorithm: O(h) (which is O(n))

```
function search (tree, e) is:
    search  search_rec(tree.root, e)
end-function
```

BST - search operation (non-recursive)

BSTNode:
    info: TComp
    left: ↑ BSTNode
    right: ↑ BSTNode
BinarySearchTree:
    root: ↑ BSTNode

```
function search (tree, elem) is:
    currentNode ← tree.root
    found ← false
    while currentNode ≠ NIL and not found execute
        if [currentNode].info = elem then
            found ← true
        else if elem ≤ [currentNode].info then
            currentNode ← [currentNode].left
        else
            currentNode ← [currentNode].right
        end-if … end-if
    end-while
    search ← found
end-function
```

- BC ?
- WC ?

BST - insert operation (recursive)

```
function createNode(e) is:
        allocate(node)
        [node].info  ← e
        [node].left  ← NIL
        [node].right ← NIL
        createNode   ← node
end-function
```

```
function insert_rec(node, e) is:
    if node = NIL then
        node ← createNode(e)
    else if e ≤ [node].info then
            [node].left ← insert rec([node].left, e)
        else
            [node].right ← insert rec([node].right, e)
        end-if
    end-if
    insert_rec ← node
end-function
```

- Like in case of the search operation, we need a wrapper function to call insert rec with the root of the tree.
- How can we implement the insert operation non-recursively?

BST – Other operations

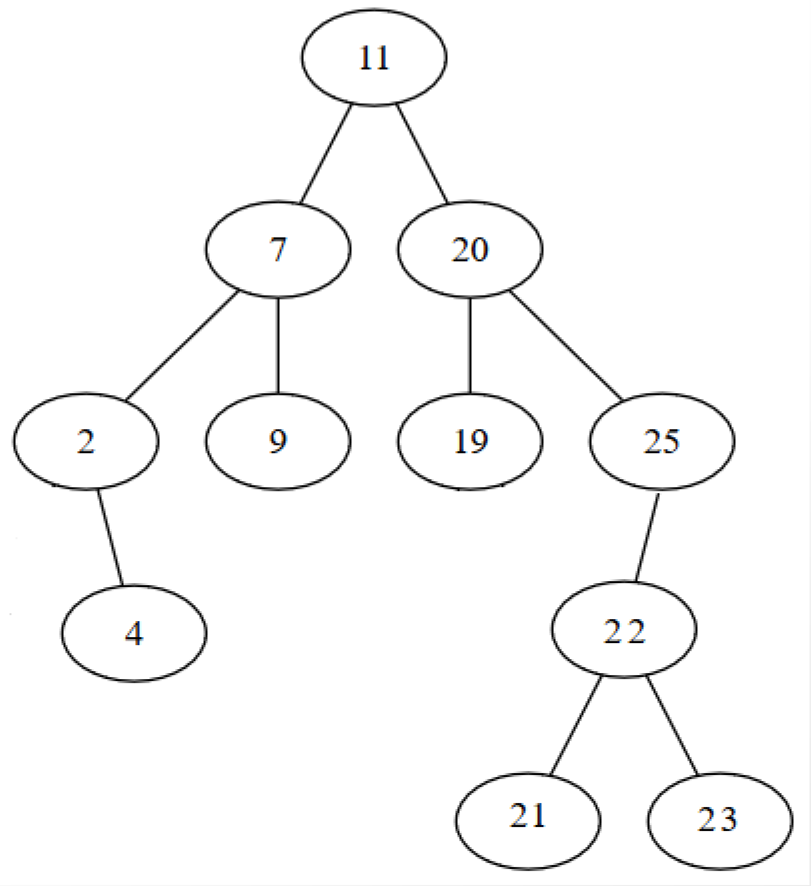
Finding the minimum element

Finding the parent of a node

Finding the successor of a node

… of 11, of 7, of 9

Finding the predecessor of a node



BST - Finding the parent of a node

pre: tree is a BinarySearchTree, node is a pointer, node ≠ NIL
post: returns the parent of node, or NIL if node is the root

```
function parent(tree, node) is:
    c ← tree.root
    if c = node then
        parent ← NIL
    else
        while c ≠ NIL  and  [c].left ≠ node  and  [c].right ≠  node execute
            if  [node].info  ≤  [c].info then
                c ← [c].left
            else
                c ← [c].right
            end-if
        end-while
        parent ← c
    end-if
end-function
```

Complexity: O(h)

BST - Finding the successor of a node

```
//pre: tree is a BinarySearchTree, node is a pointer, node ≠ NIL
//post: returns the node with the next value after the value from node
//                                         or NIL if node is the maximum
function successor(tree, node) is:
    if [node].right ≠ NIL then
        c ← [node].right
        while [c].left ≠ NIL execute
            c ← [c].left
        end-while
        successor ← c
    else
        p ← parent(tree, c)
        while p ≠ NIL and [p].left ≠ c execute
            c ← p
            p ← parent(tree, p)
        end-while
        successor ← p
    end-if
end-function
```

- BC ?
- WC ?

BST - Remove a node

When we want to remove a value (a node containing the value) from a binary search tree we have three cases:

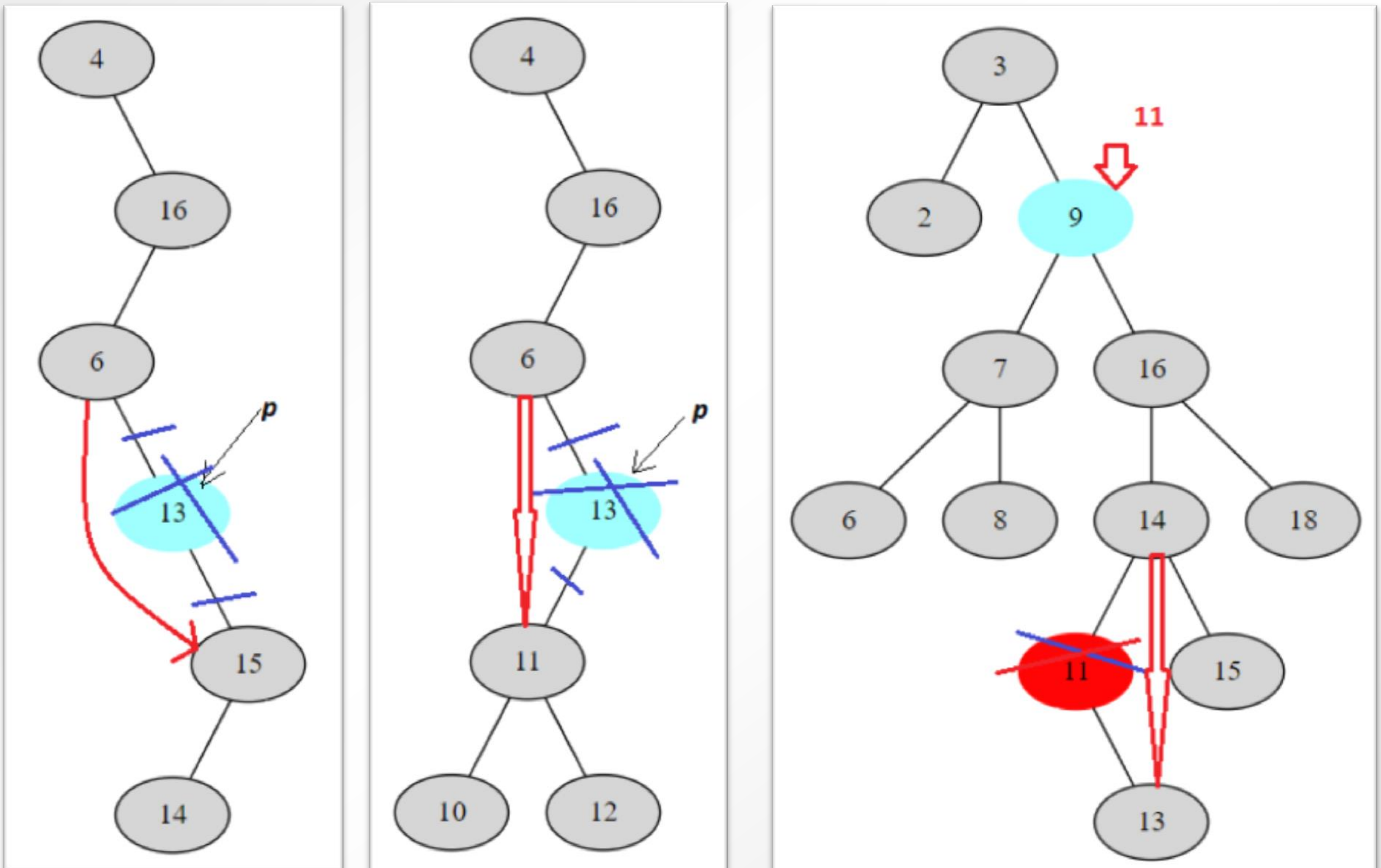
- The node to be removed has no descendant:
  - Set the corresponding child of the parent to NIL
- The node to be removed has one descendant:
  - Set the corresponding child of the parent to the descendant
- The node to be removed has two descendants
  - Find the maximum of the left subtree, move the value to the node to be deleted, and delete the found node (maximum)

  OR

  - Find the minimum of the right subtree, move the value to the node to be deleted, and delete the found node (minimum)

BST - Remove a node

e.g.:



BST

Think about it:

- Is the operation of deletion "commutative" in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y and then x?

  (Give a counterexample)

- Draw binary search trees of height 2, 3, 4, 5, and 6 on the set of keys

  {1, 4, 5, 10, 16, 17, 21}

- Suppose that we have numbers between 1 and 1000 in a binary search tree and want to search for the number 363. Which of the following sequences could not be the sequence of nodes examined?

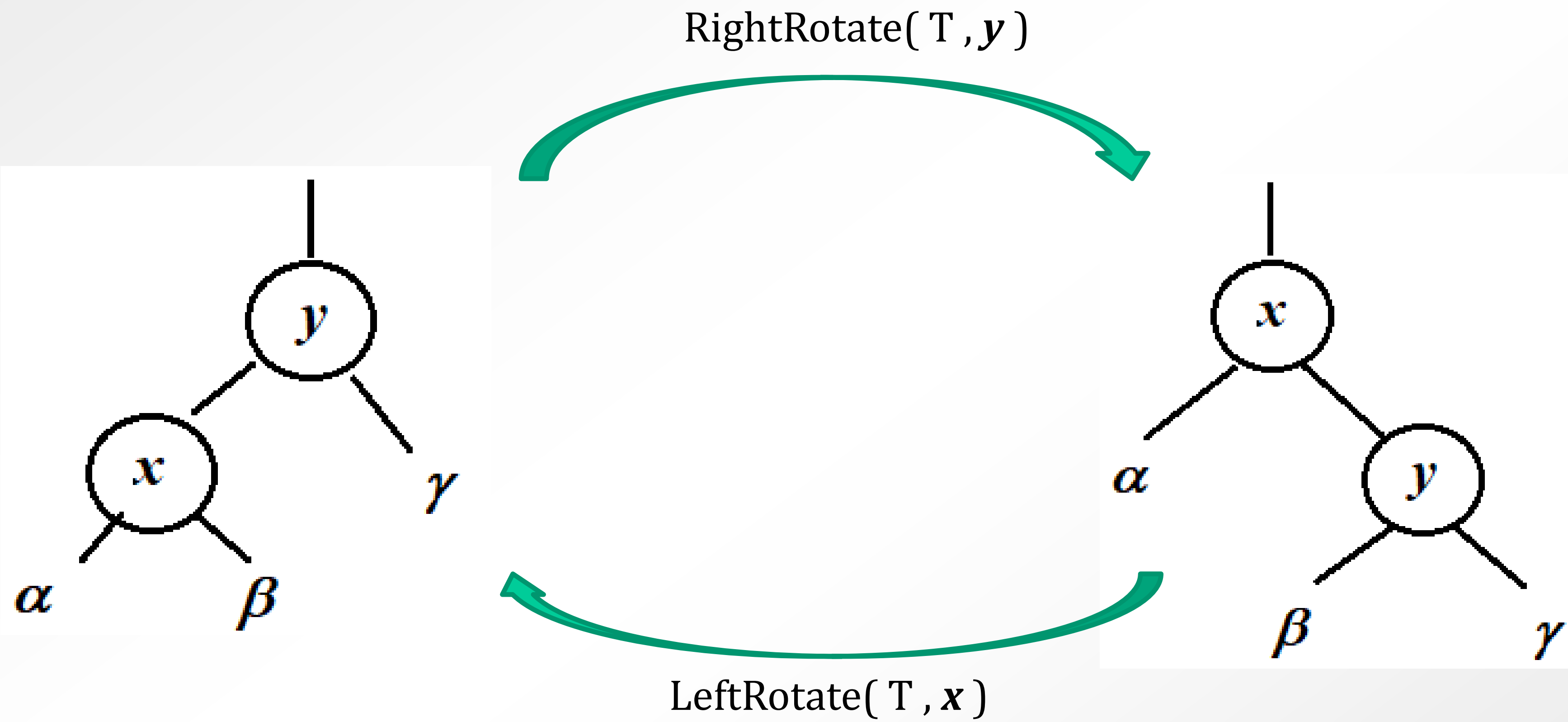
  2, 252, 401, 398, 330, 344, 397, 363

BST

<u>Think about it</u>:

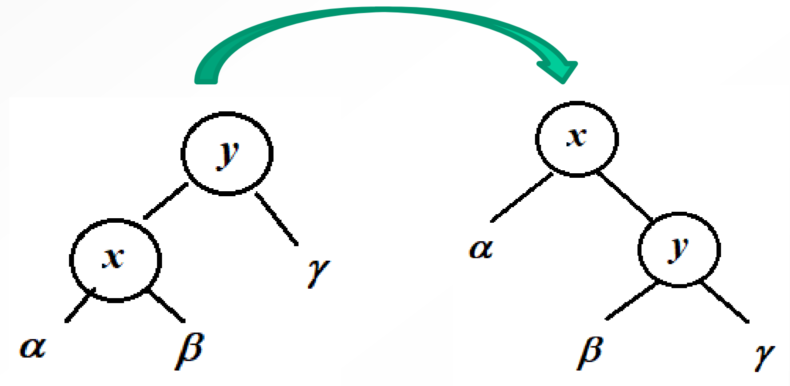
- Give 2 different BSTs that contains the same set of elements
- Given a BST, give 2 different sequences of distinct elements that can create that tree
- BST with repeating values
  - Starting from an initially empty Binary Search Tree and the relation <=, insert into it, in the given order, the following values: 10, 20, 5, 7, 15, 5, 30, 3, 5, 5, 1, 9, 29, 2.
  - How would you count how many times the value 5 is in the tree?

Rotate-left – rotate-right



Resulting tree is still a BST

BST : RightRotate



Function RightRotate (y)

x ← [y].left

[y].left ← [x].right

[x].right ← x

RotateRight ← x

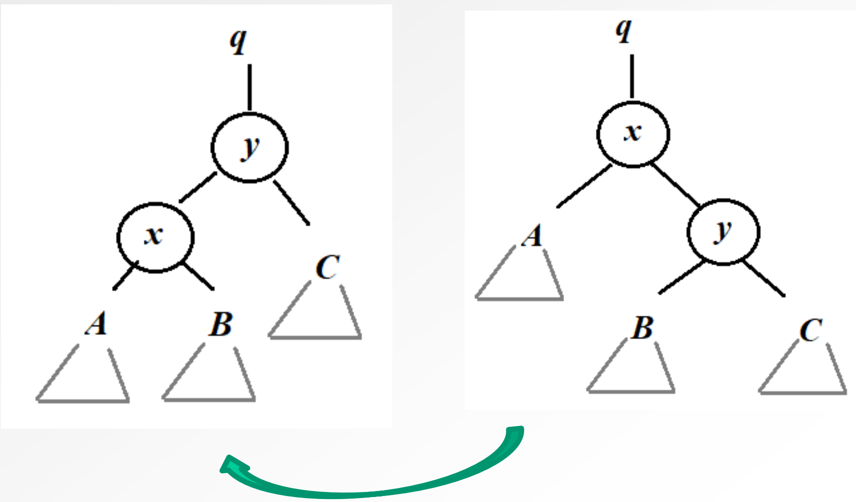
end_function

New root of the subtree

BSTNode:
info: TComp
left: ↑ BSTNode
right: ↑ BSTNode
BinarySearchTree:
root: ↑ BSTNode

- Similar for: LeftRotate

BST : LeftRotate




```
BSTNode:
      info: TComp
      left:   ↑ BSTNode
      right: ↑ BSTNode
      parent: ↑ BSTNode
BinarySearchTree:
      root:  ↑ BSTNode
```

- Similar for: RightRotate


```
Subalg. LeftRotate(T, x)
    y ←  [x].right
    [x].right  ← [y].left
    if [y].left <> NIL then
       [[y].left].parent ← x
    endif
    [y].parent ←  [x].parent
    if [x].parent = NIL then
       T.root ←  y
    else
       if  x= [[x].parent].left then
         [[x].parent].left ← y
       else
         [[x].parent].right ←  y
       endif
    endif
    [y].left ←  x
    [x].parent ←  y
End-subalg.
```

Balanced tree

"Close" definitions found in literature:

1. ***Balanced tree***

   Or: Nearly -- Heigh -- **balanced**

no leaf is much farther away from the root than any other leaf.

*Different balancing schemes allow different definitions of "much farther" and different amounts of work to keep them balanced***.**

2. ***Height-Balanced tree***

A tree whose subtrees differ in height by no more than one and the subtrees are height-balanced, too. An empty tree is height-balanced.

Many times, "balanced" is used also for "height balanced".

Balanced tree

Self-balancing:

- "*automatically*" keeps its balanced in the face of insertions and deletions

Property of all types of binary balanced tree:

- The height of the tree is $O(\log_2 n)$

Popular self-balancing Binary Search Tree

- red-black tree
- AVL tree

BST in Java.util and C++ STL

Java.util

- TreeMap

    a Red-Black tree based implementation

- TreeSet

    implementation based on a TreeMap

C++ STL

- map, multimap

  are typically implemented as binary search trees

  www.cplusplus.com

  maps are usually implemented as red-black trees

  en.cppreference.com/w/cpp/container/map

Binary Search Tree. Operation complexity

- Specific operations for binary trees run in O(h) time, which can be $\Theta(n)$ in worst case
  - *We, in these classes, when we estimate the complexity of an operation of a BST, we can use **h***
- Best case is when a tree is balanced. There, the height of the tree is ($\log_2 n$)

So, to reduce the complexity of algorithms, we want to keep the tree balanced.

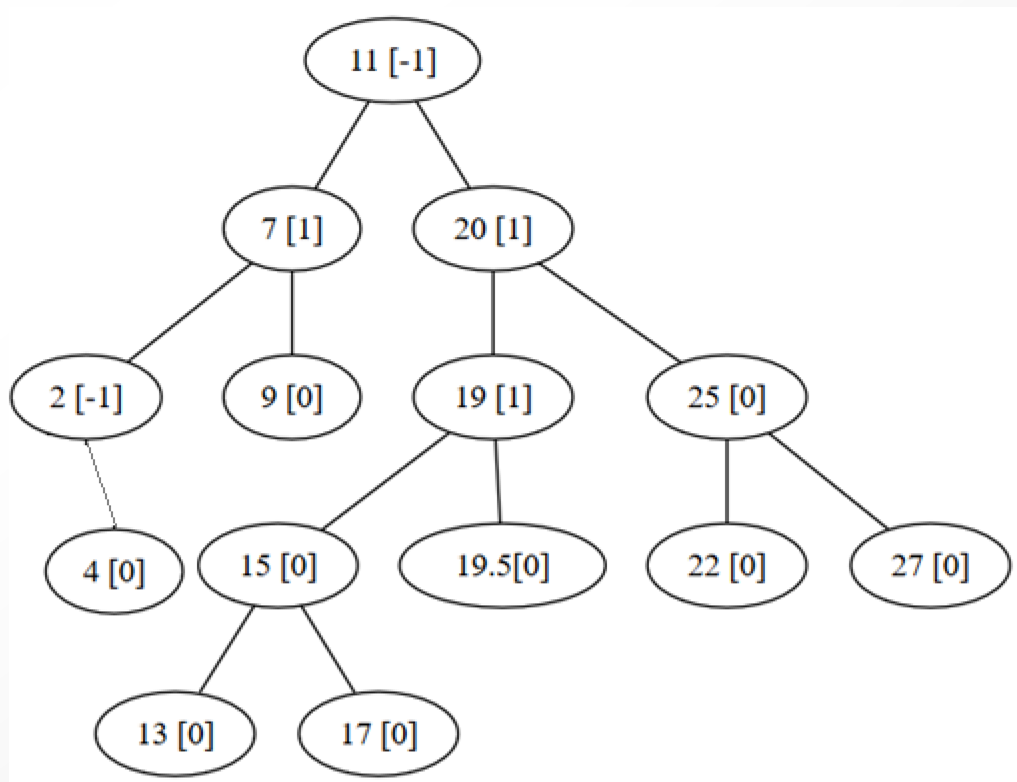
AVL trees

An AVL (Adelson-Velskii Landis) tree is a binary search tree which satisfies the following property (AVL tree property):

- If x is a node of the AVL tree:

  the difference between the height of the left and right subtree of x

  is 0, 1 or -1



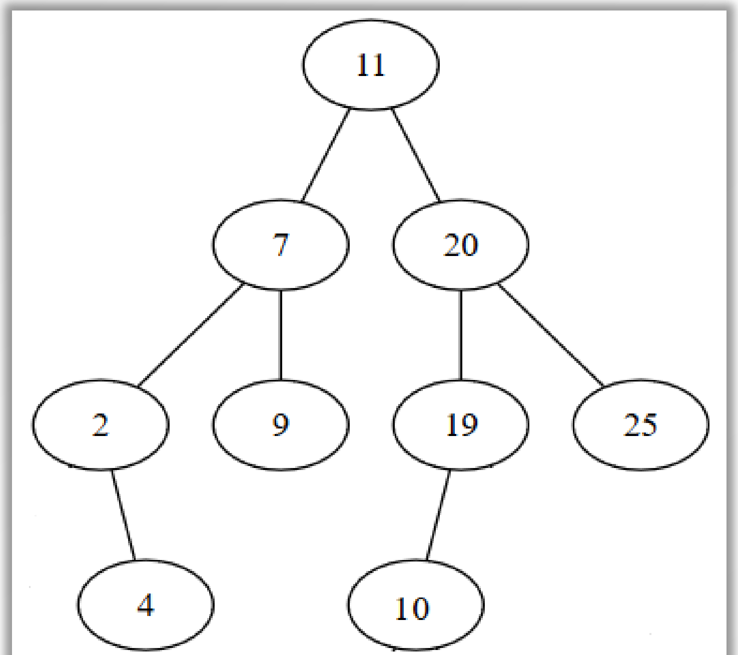
Remarks:

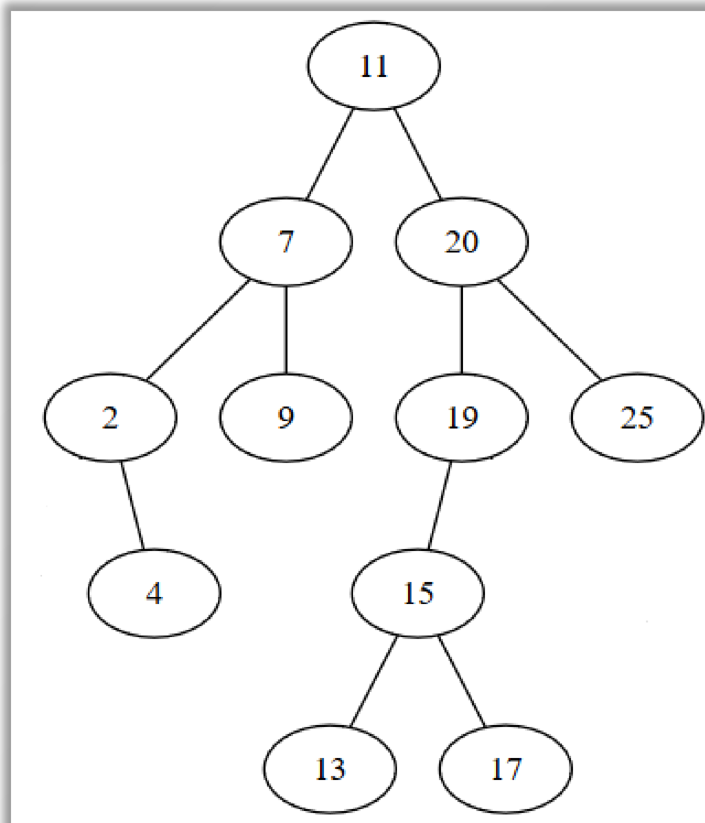
- Height of an empty tree is -1
- Height of a single node is 0

Values in square brackets show the balancing information of a node.

AVL trees

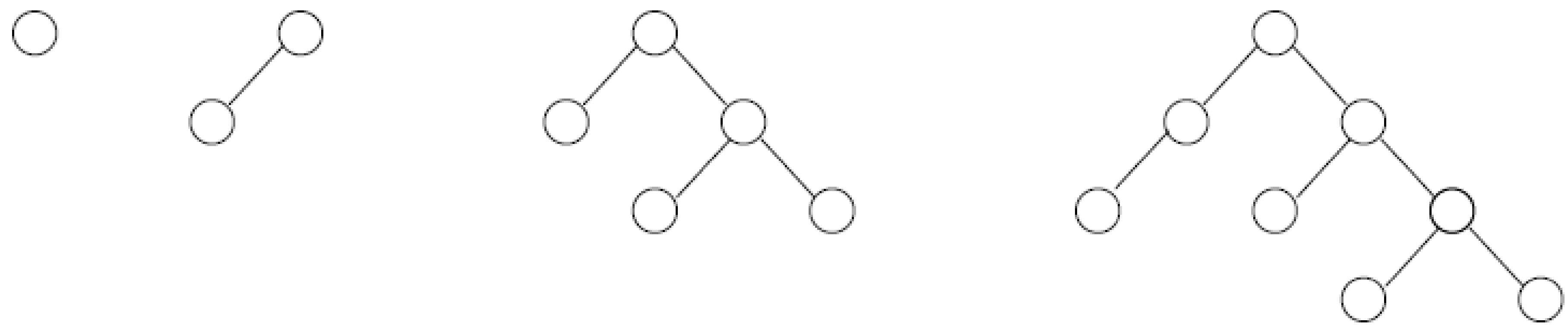
Are these AVL trees?

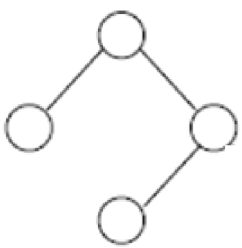


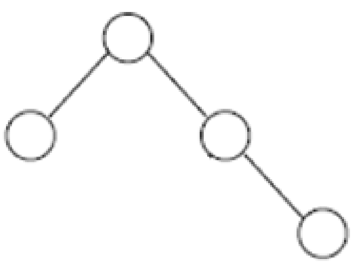


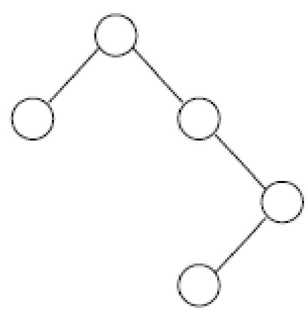
AVL trees

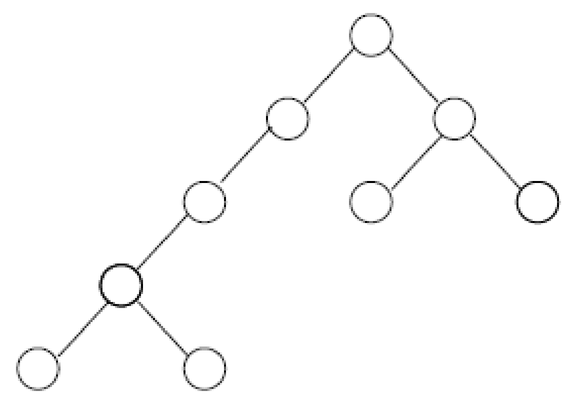
Which of the next binary trees have the shape of an AVL tree?











?

AVL Trees : insert/remove

- Adding or removing a node
  - add/remove them as for an BST

might result in a binary tree that violates the AVL tree property.

In such cases, the property has to be restored

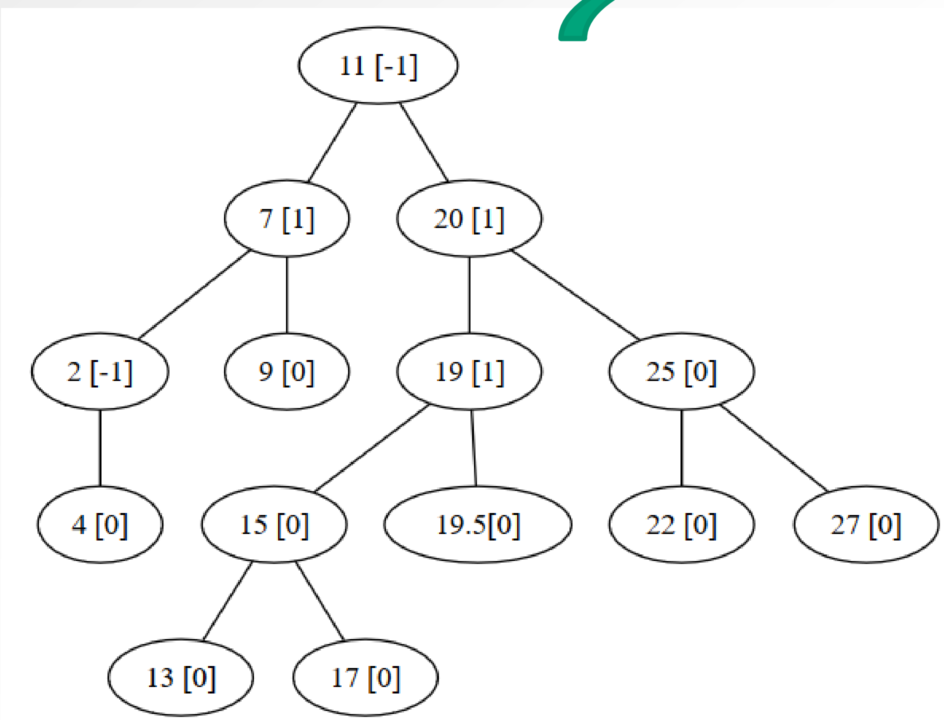
- ➢ Use rotations: they keep the BST property.

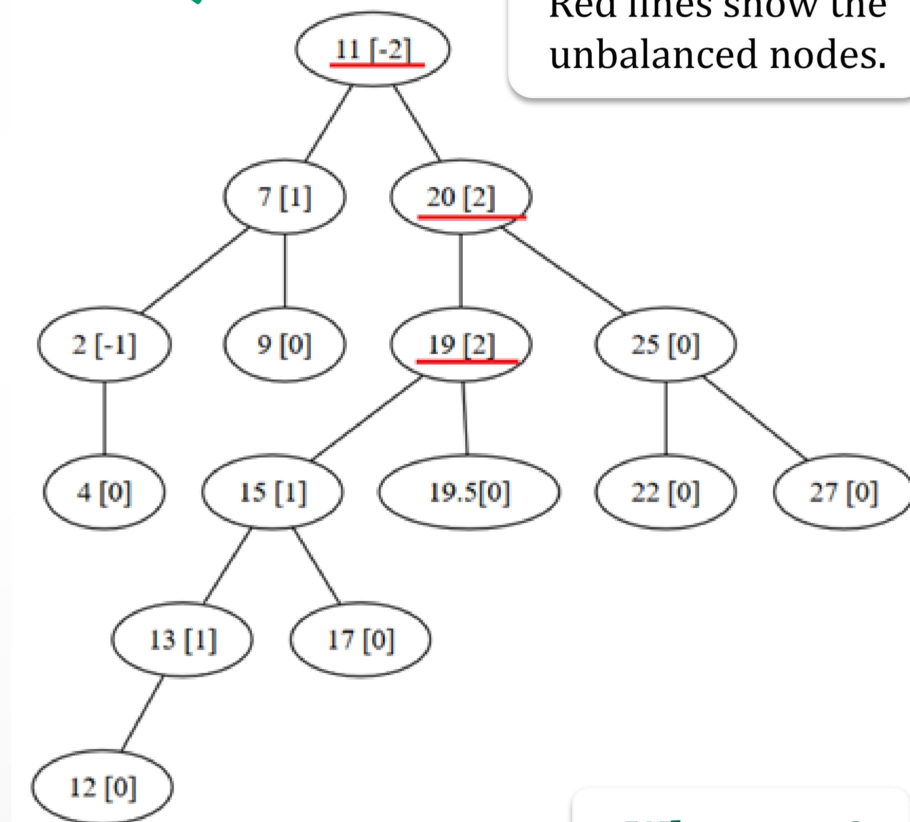
Properties:

- Only the nodes on the path to the modified node can change their height.
- We check the balancing information on the path from the modified node to the root. When we find a node that does not respect the AVL tree property, we perform a suitable rotation to rebalance the (sub)tree.

AVL Tress - insert

we insert element 12 as in BST





Red lines show the unbalanced nodes.

What next?

AVL - insert

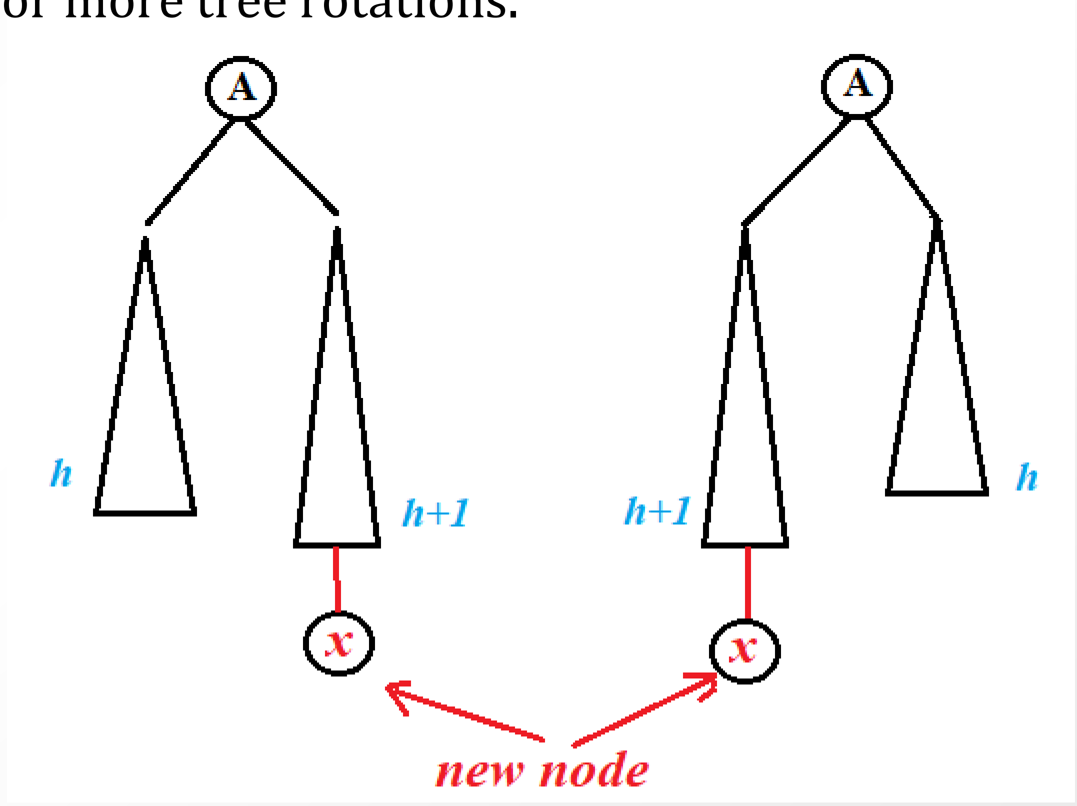
Insertion:

- insert an element like in BST case
- rebalance the tree (if it is the case)

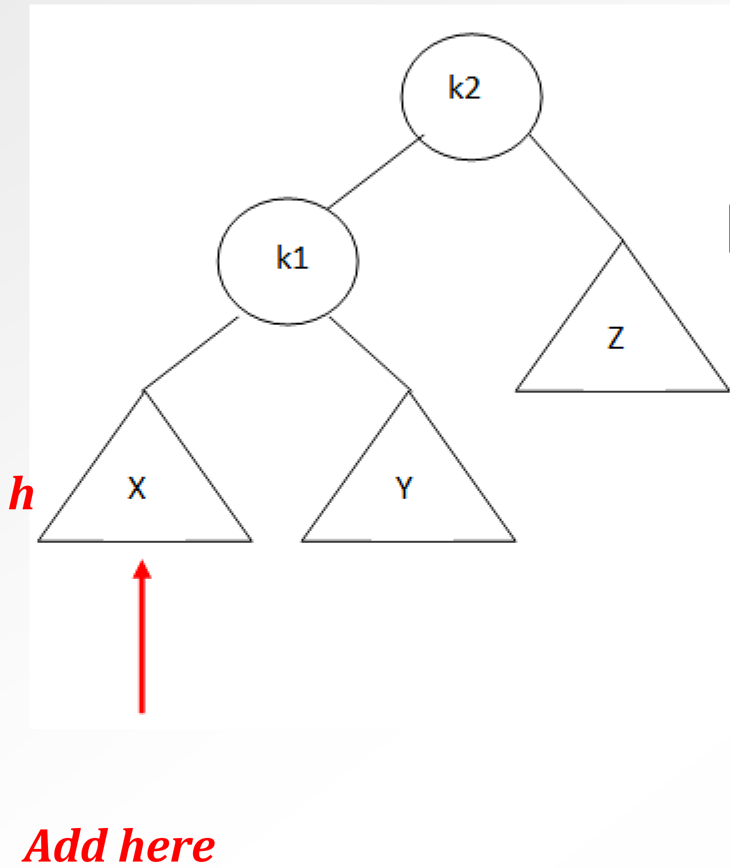
  consider all the ancestors (to the root)

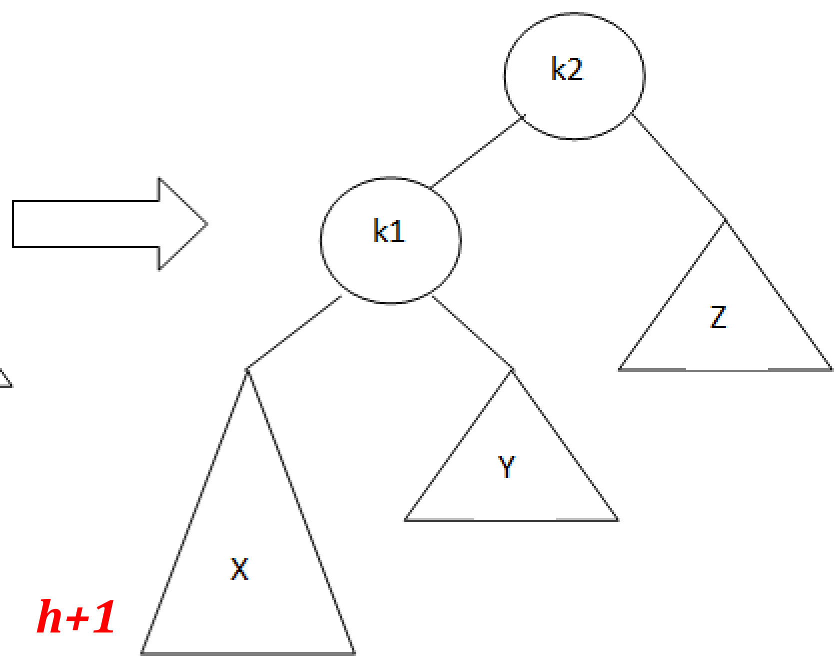
  rebalance → one or more tree rotations.

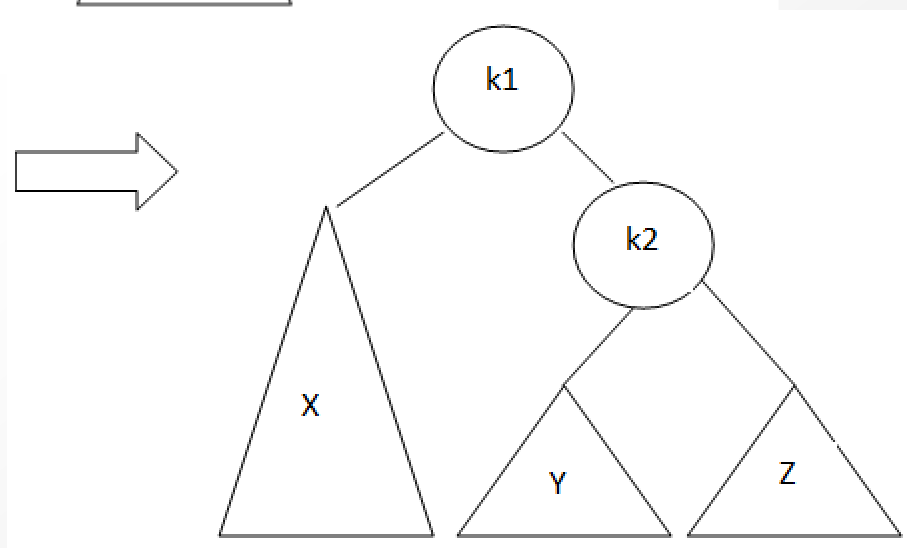
When to rebalance:



AVL Trees – insert : case 1

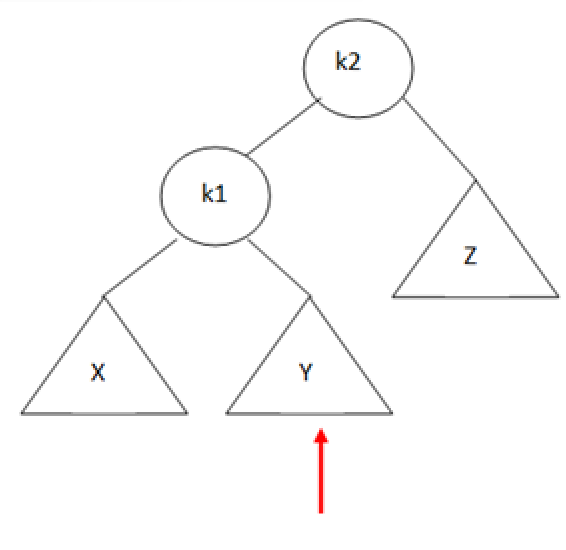




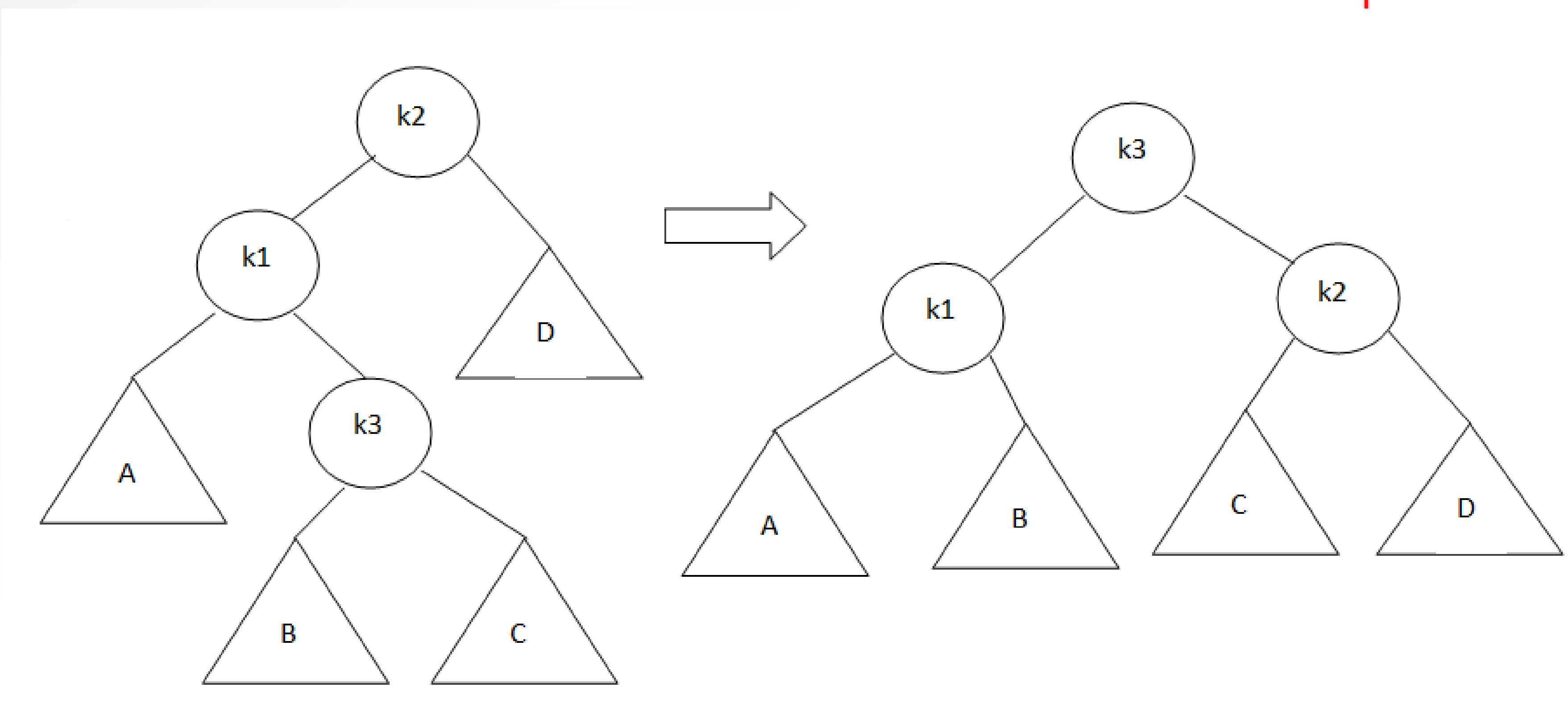


- X, Y and Z represent subtrees with the same height.
- Solution: single rotation to right

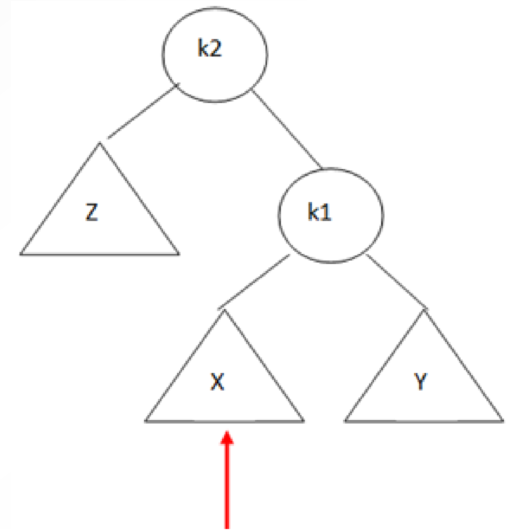
AVL Trees – insert: case 2



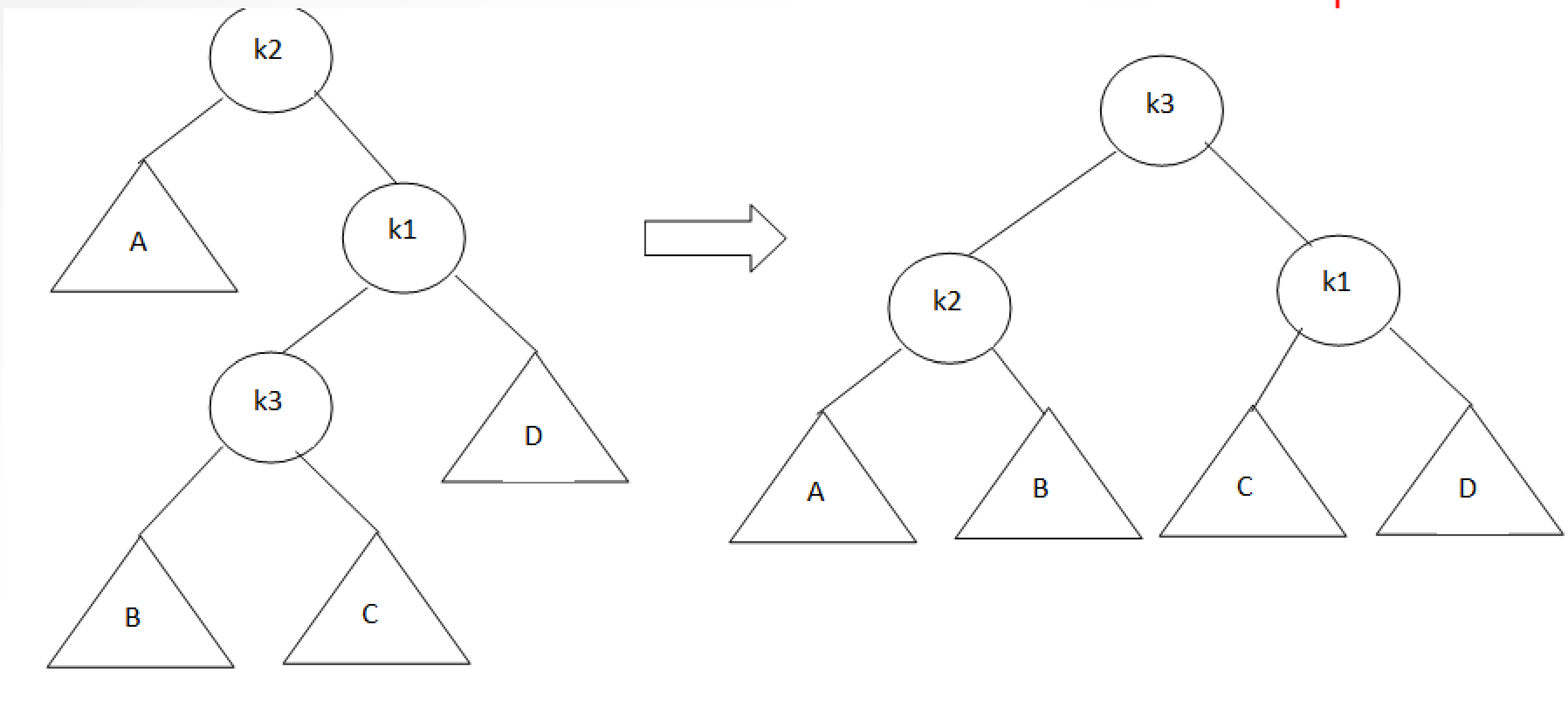
Double rotation to right



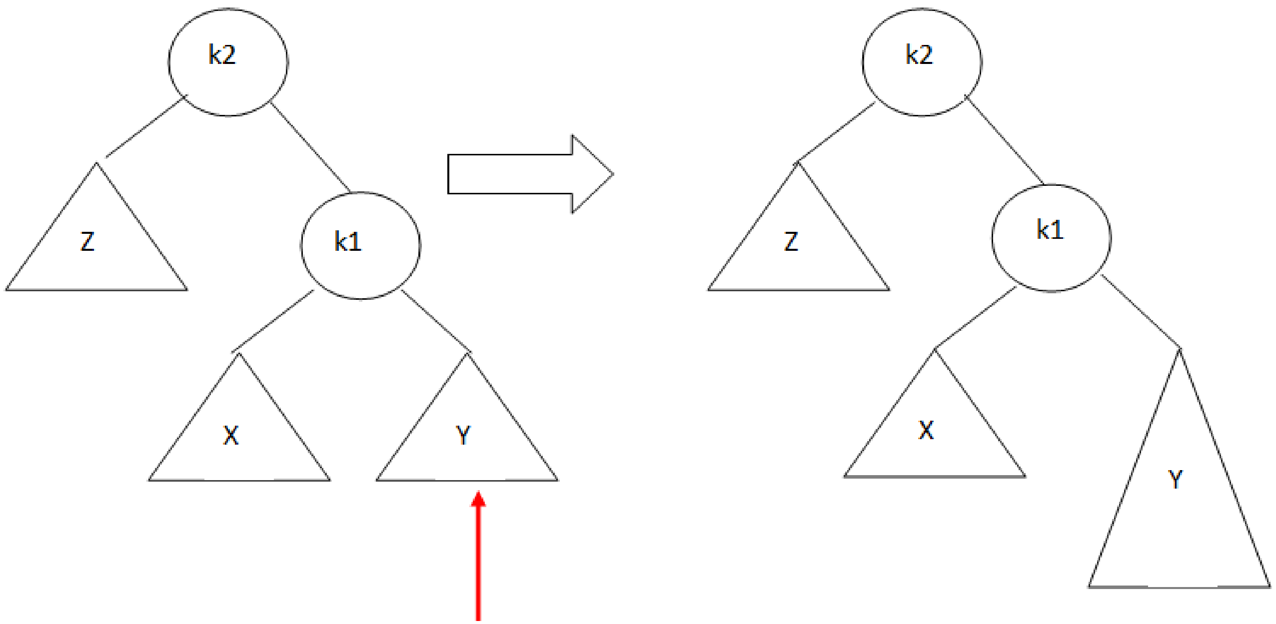
AVL Trees – insert: case 3



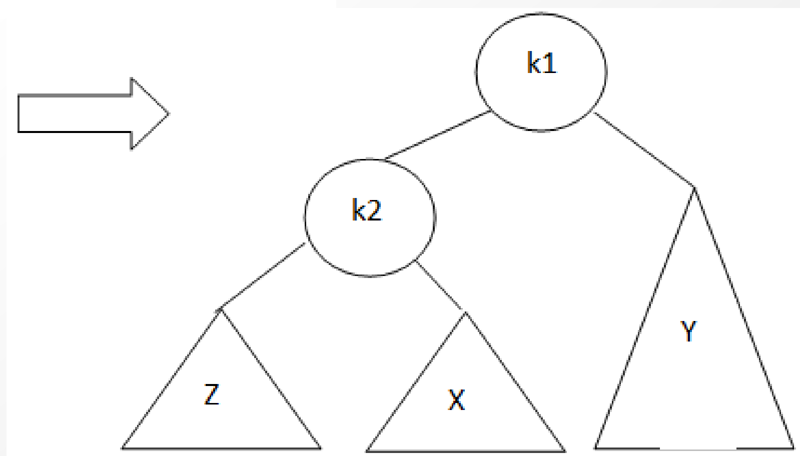
Double rotation to left



AVL Trees – insert: case 4



Single rotation
to left



AVL Trees - insert

Assume that at a given point k2 is the node that needs to be rebalanced.

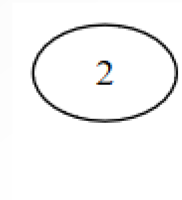
- k2 was balanced before the insertion,
- k2 is not balanced after the insertion

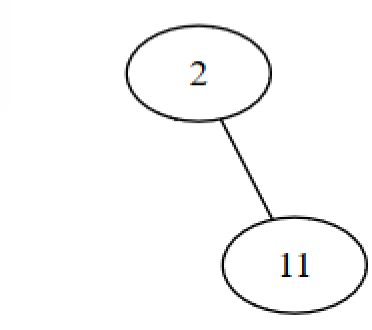
There are four cases in which a violation might occur:

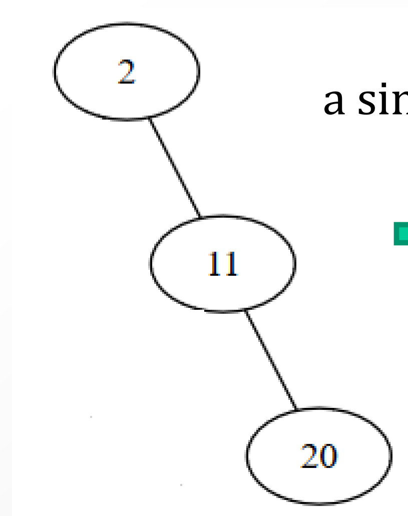
- Insertion into the left subtree of the left child of k2 **(*case 1*)**
- Insertion into the right subtree of the left child of k2 **(*case 2*)**
- Insertion into the left subtree of the right child of k2 **(*case 3*)**
- Insertion into the right subtree of the right child of k2 **(*case 4*)**

AVL insert: example

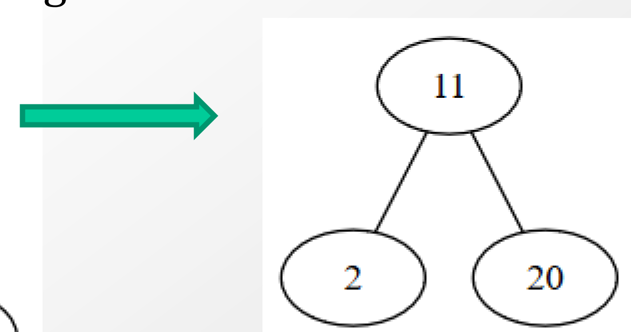
- Start with an empty AVL tree
- Insert 2
- Insert 11
- Insert 20
- Insert 7 ...





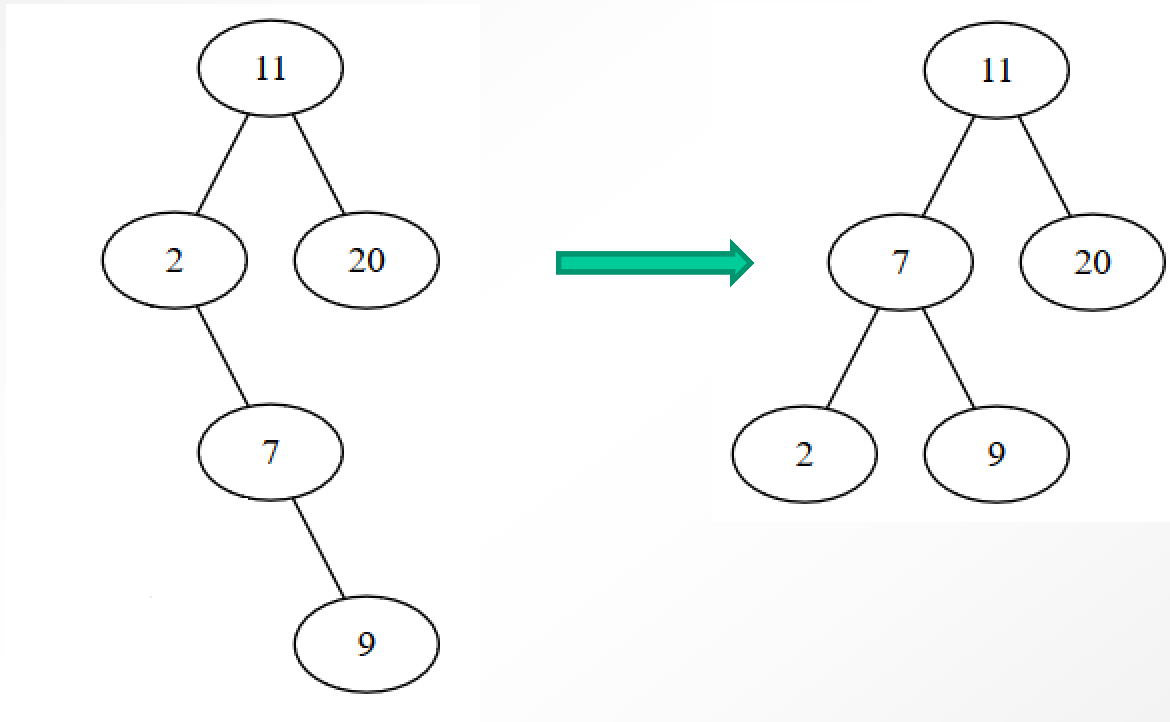


a single left rotation on node 2



AVL insert: example

- Operation:
  Insert 9

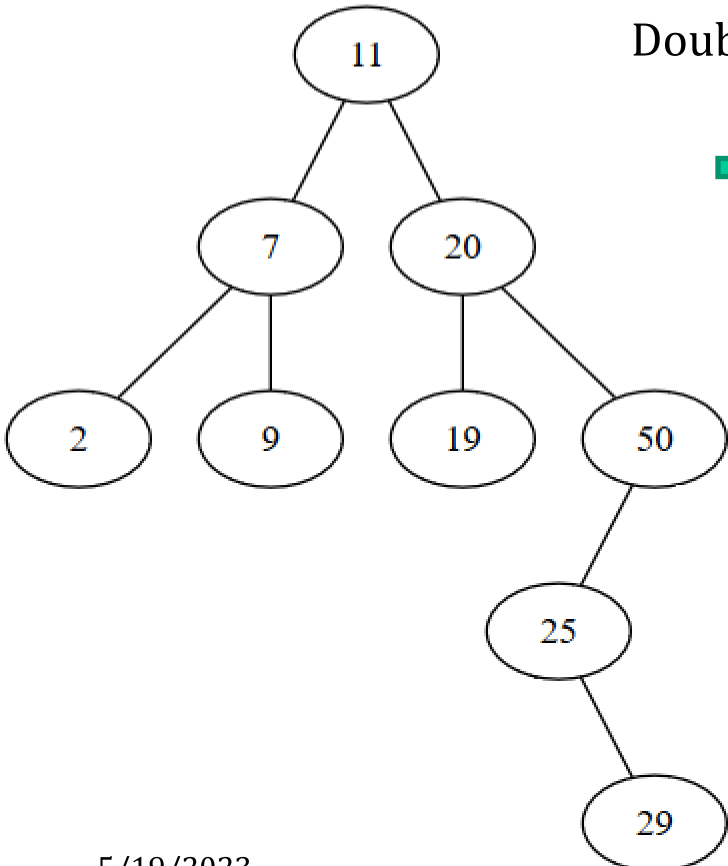


a single left rotation on node 2

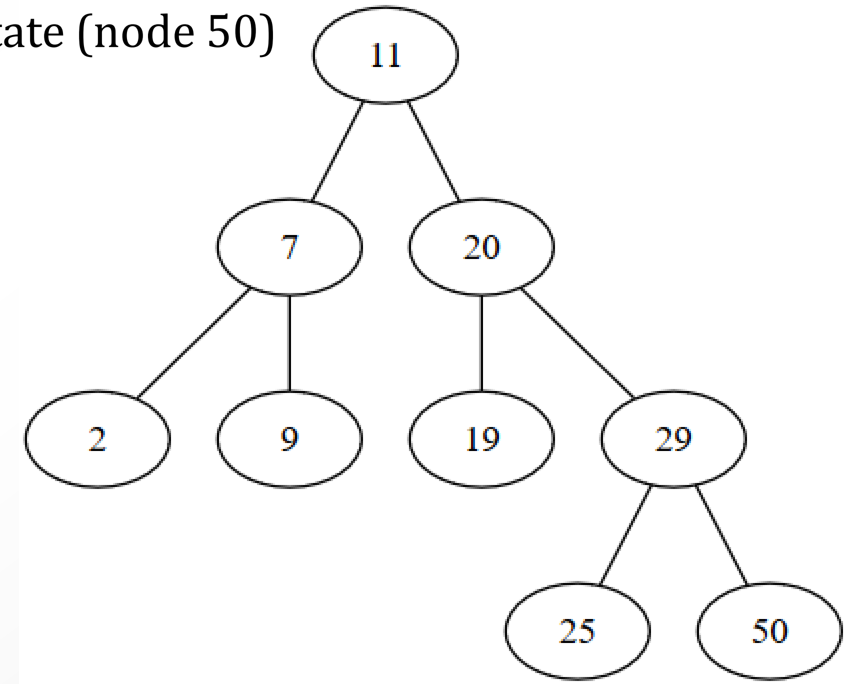
- Insert 50
- Insert 19
- Insert 25
- Insert 29

AVL insert: example

- Operation: insert 29



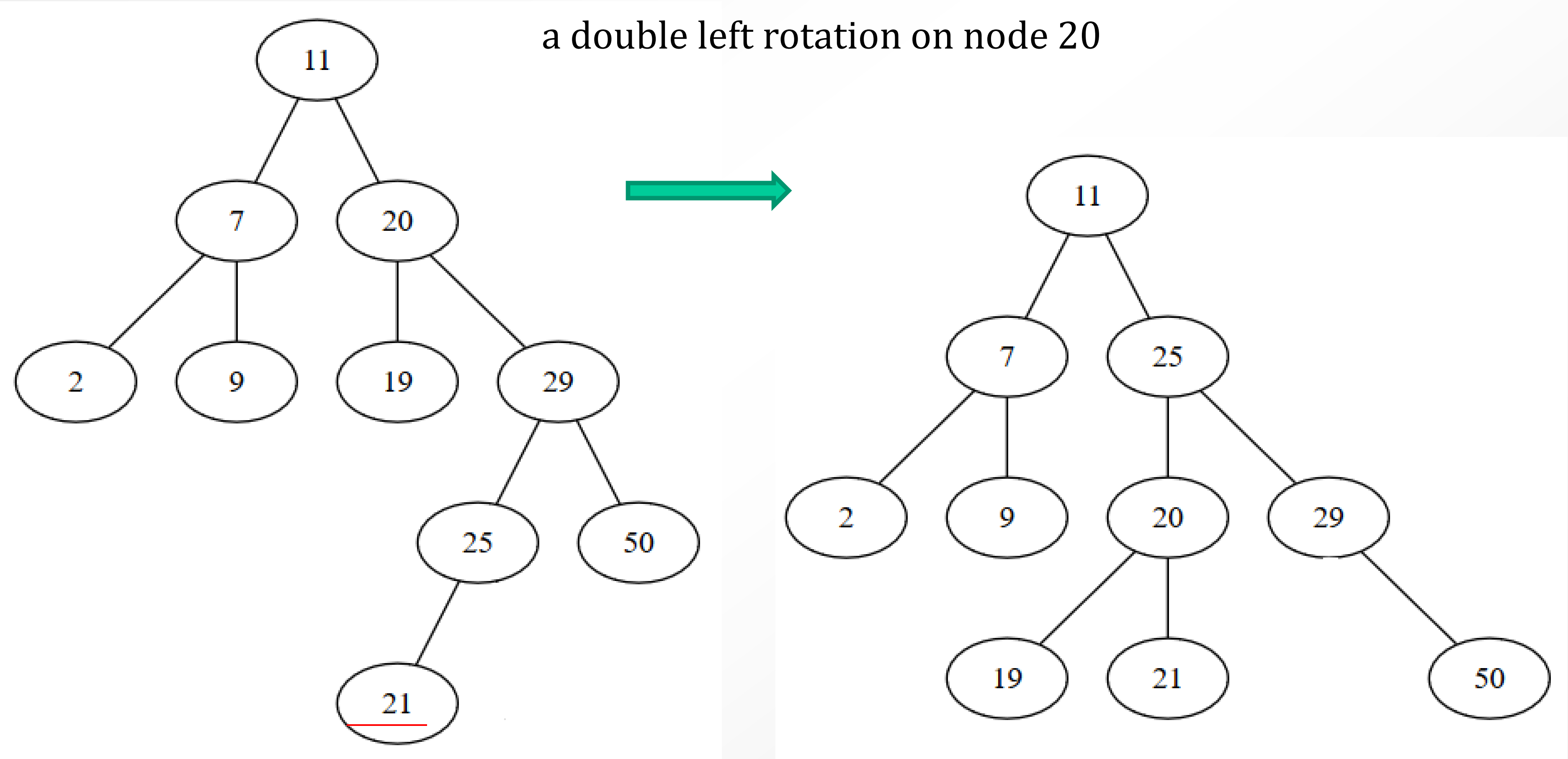
a double right rotation on node 50
DoubleRightRotate (node 50)



AVL insert: example

- Operation: add 21 to the previous tree

a double left rotation on node 20



Pre: el - element, p - AvlTreeNode
Post: return the new p

```
Function insert_rec(p , el)
    if( p = NIL)
        p ← createNode(el)
    else
        if( el <= [p].info)  then
            [p].left ← insert_rec([p].left , el )
            if (Height( [p]. left )  - Height([p]. right)  = 2 )
                if( el <= [[p].left].info)
                    p ← RightRotate ( p )
                else
                    p ← DoubleRightRotate ( p )
                endif
            endif
        else   // el > [p].info
            [p].right ← insert_rec([p].right , el )
            if( Height( [p].right ) - Height( [p].left ) = 2 )
                if( el > [[p].right].info ) then
                    p ← LeftRotate ( p )
                else
                    p ← DoubleLeftRotate ( p );
                endif
            endif
        endif
        [p].h ← Max( Height( [p].left ), Height( [p].right ) ) + 1;
    endif
    insert_rec ← p
End_function
```

```
Function createNode ( el)
    allocate(p)
    [p].info ← el
    [p].h ←  0;
    [p].left ← NIL
    [p].right ←  NIL
    createNode ← p
end_function
```

```
Subalg. insert(T , el)
        p ← T.root
        T.root ← insert_rec(p, el)
end_subalg.
```

Delete(k)

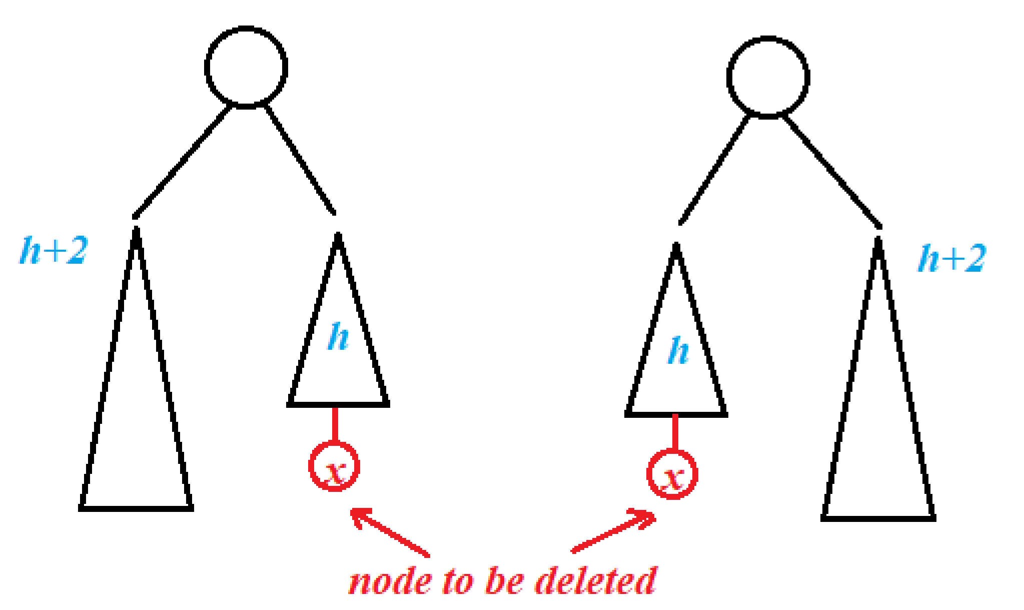
- find the node x where k is stored
- delete the contents of node x ~ similar with BST

  Deleting a node in an AVL tree can be reduced to deleting a leaf

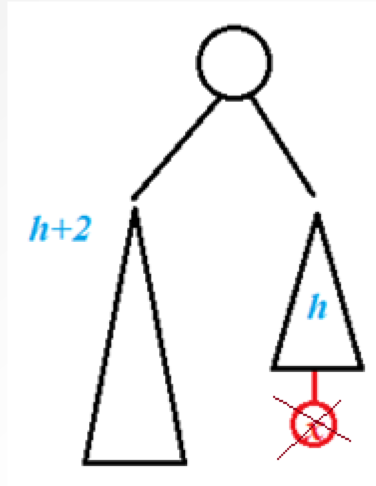
- rebalance:

  go from the deleted leaf towards the root

  and rebalance with rotations if necessary.



AVL Trees – remove cases

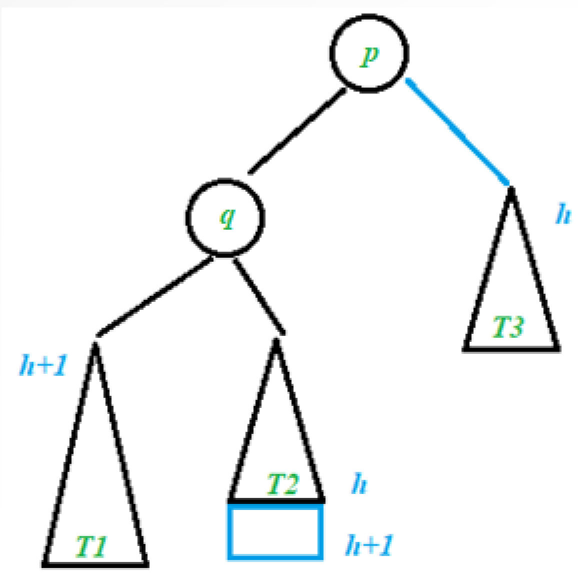


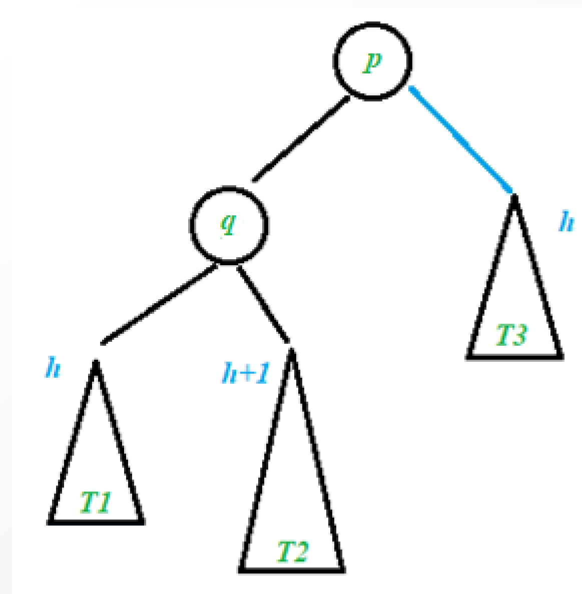
Case 1:

- RightRotate (***p***)

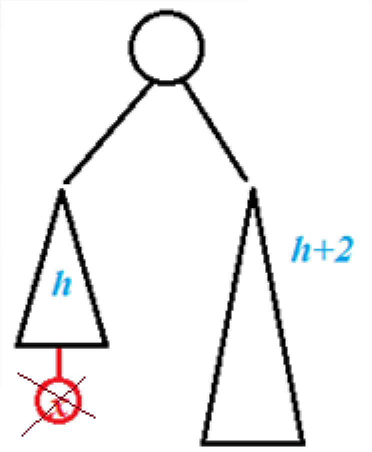
Case 2:

- DoubleRightRotate(***p***)



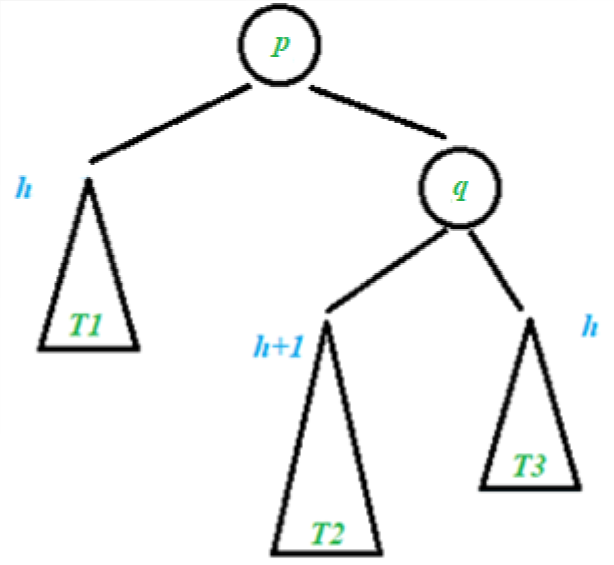


AVL Trees – remove cases



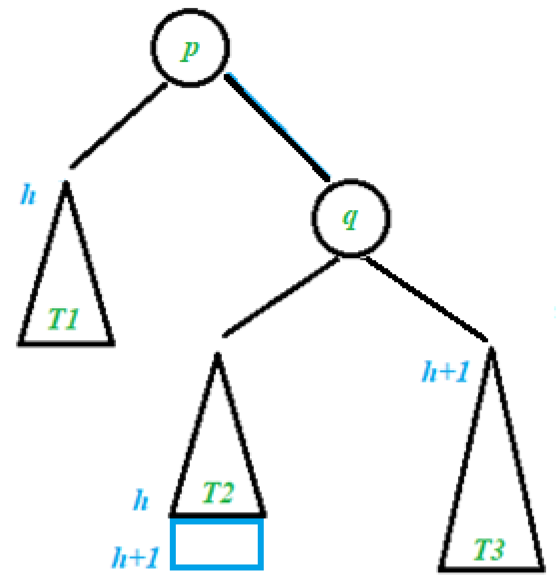
Case 3:

- DoubleLeftRotate($\boldsymbol{p}$)



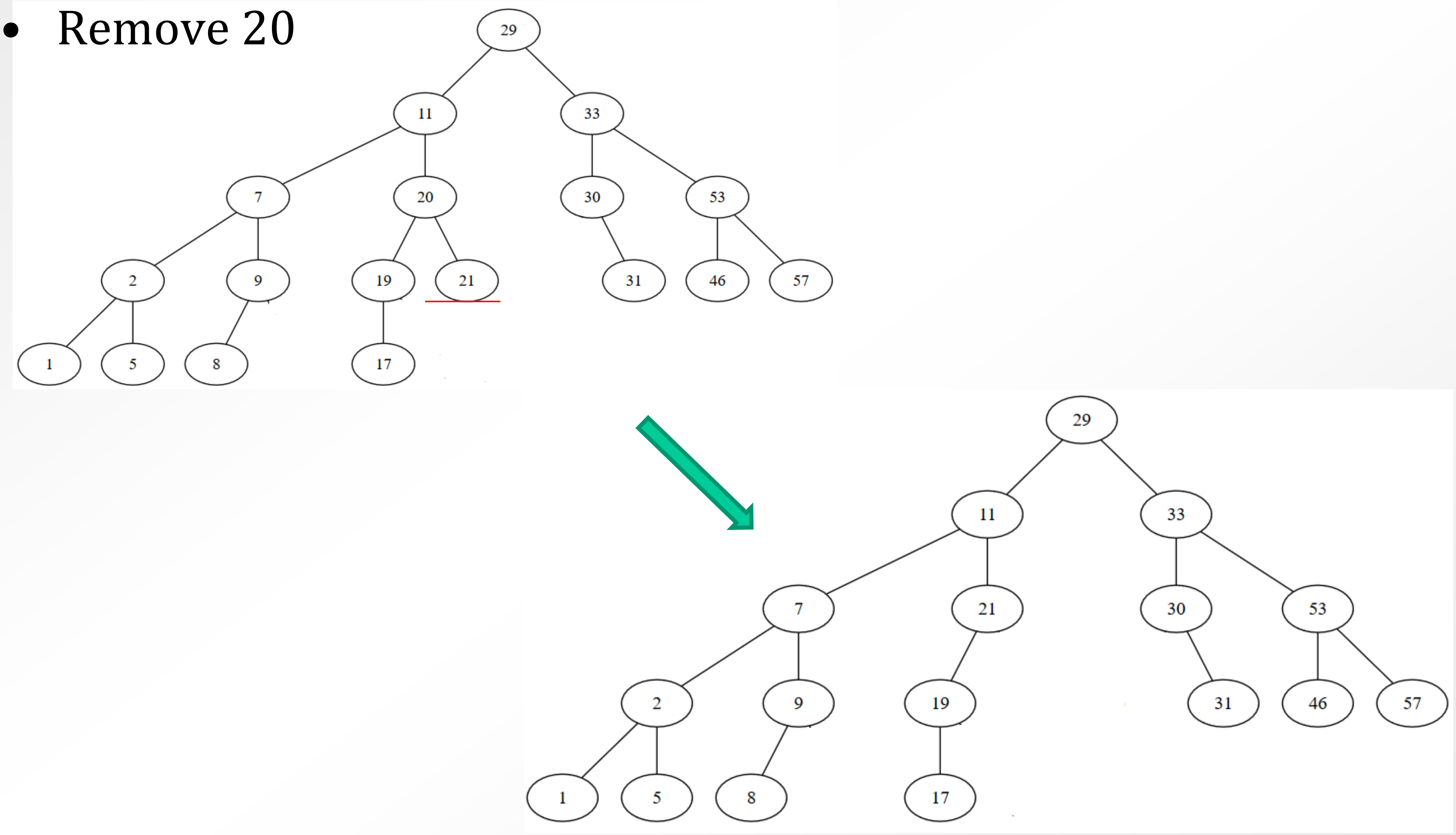
Case 4:

- LeftRotate ($\boldsymbol{p}$)

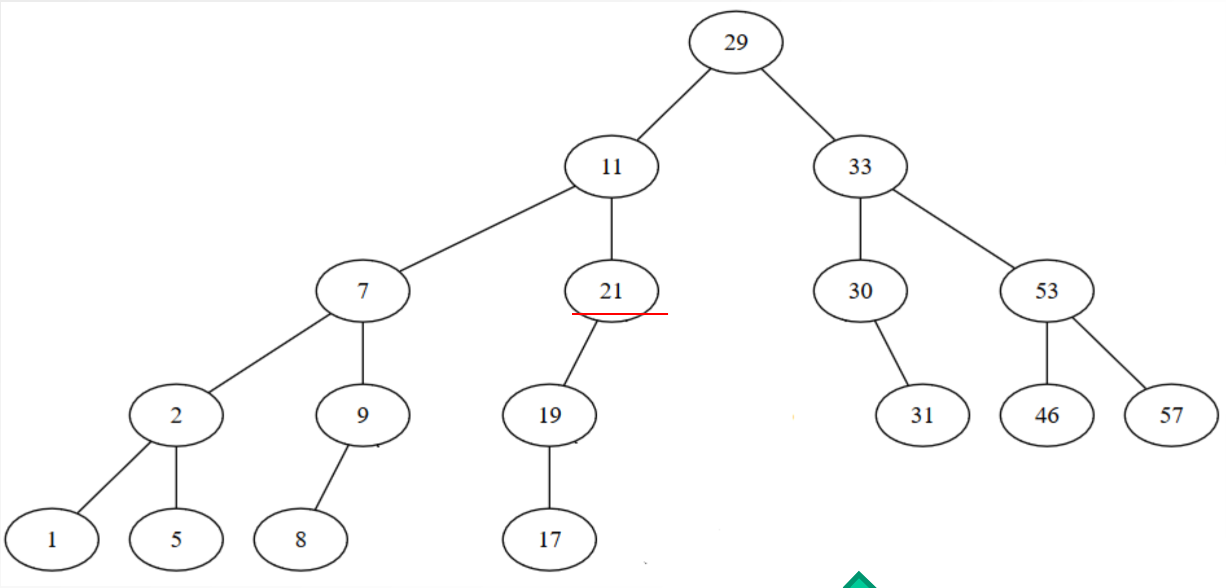


AVL - example of remove

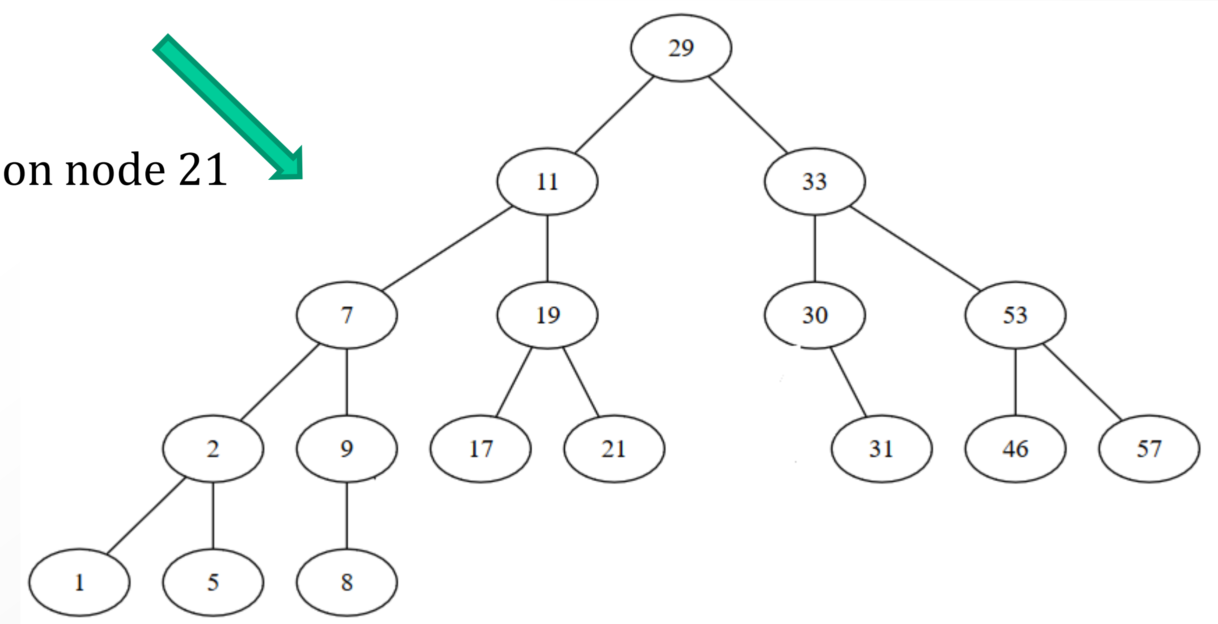
- Remove 20



AVL - example of remove

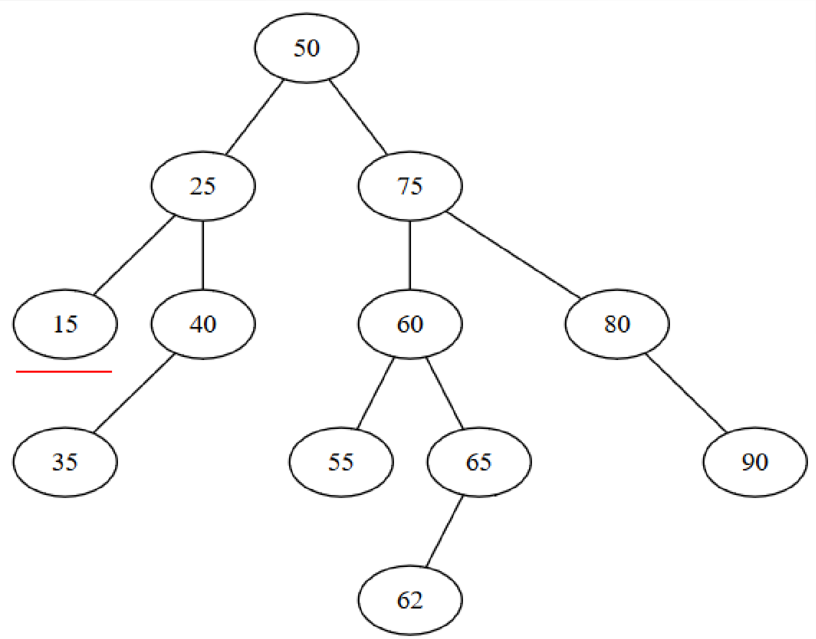


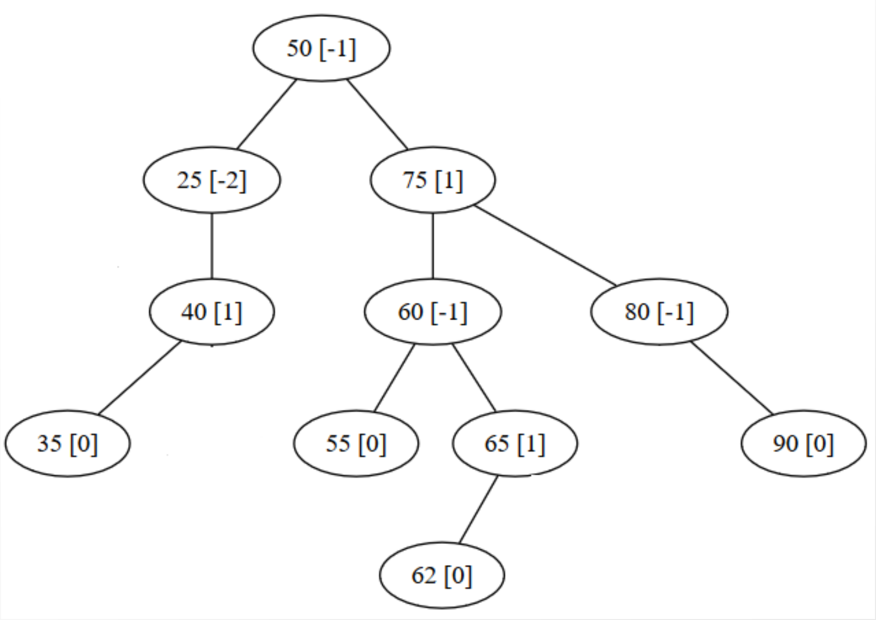
a single right rotation on node 21



AVL - example of remove

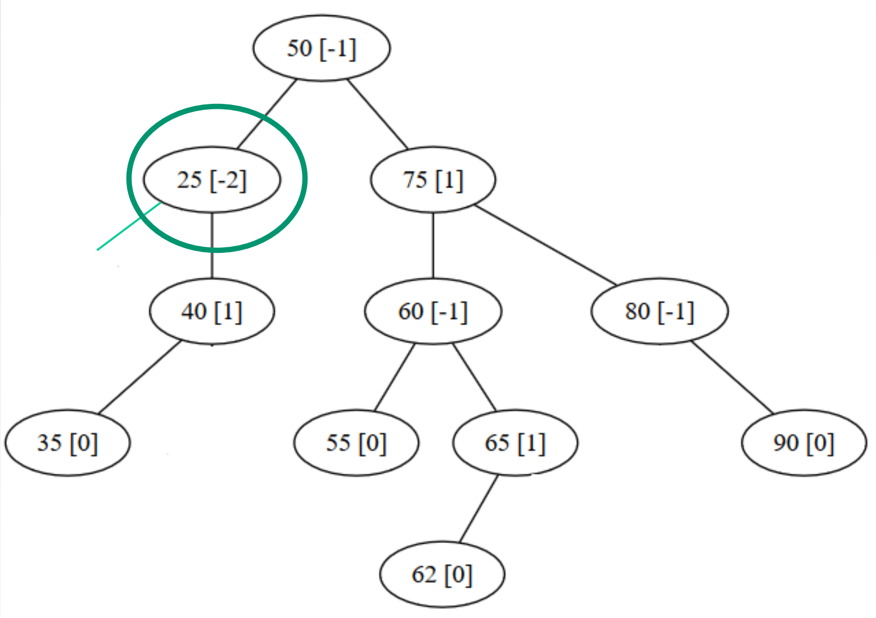
- Remove 15

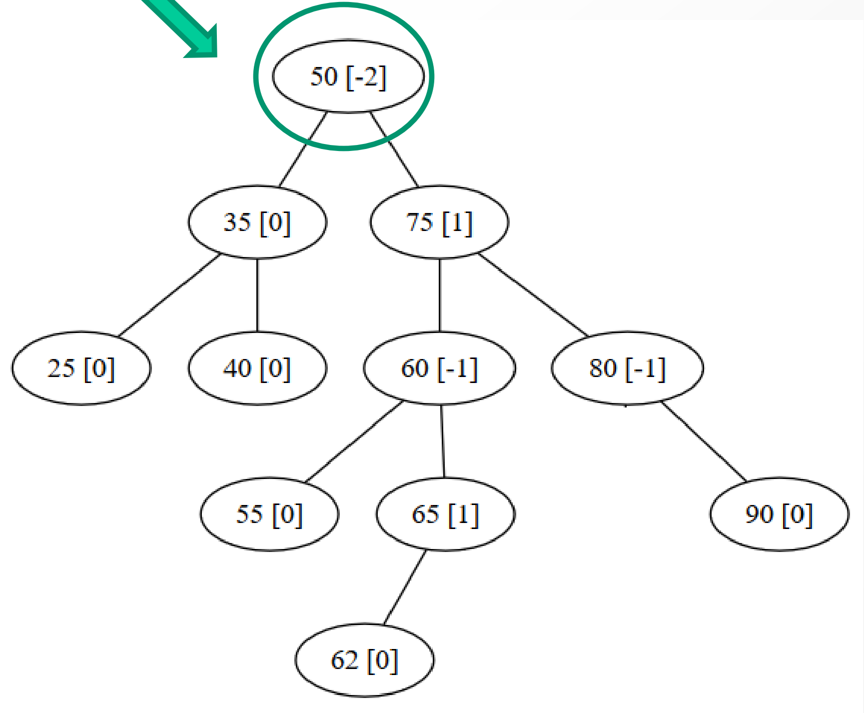




- .

AVL - example of remove





- .

AVL - example of remove