Lecture 11

- Binary search tree
- Balanced binary search tree: AVL

Lect. PhD. Lupsa Dana
Babes - Bolyai University
Computer Science and Mathematics Faculty
2021 - 2022

Some slides borrowed from: Lect. PhD. Onet-Marian Zsuzsanna

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:

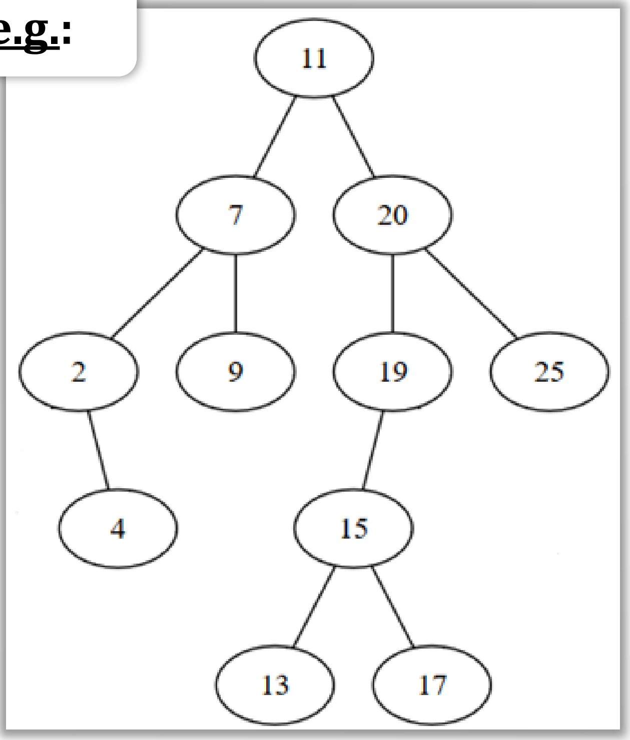
- for every node y from the left subtree of x, the information from y is less than or equal to the information from x
- for every node y from the right subtree of x, the information from y is greater than or equal to the information from x

Remarks:

- In order to have a binary search tree, we need to store information in the tree that is of type TComp.
- Obviously, the relation used to order the nodes can be considered in an abstract way (instead of having “<=" as in the definition).

Binary search tree

e.g.:

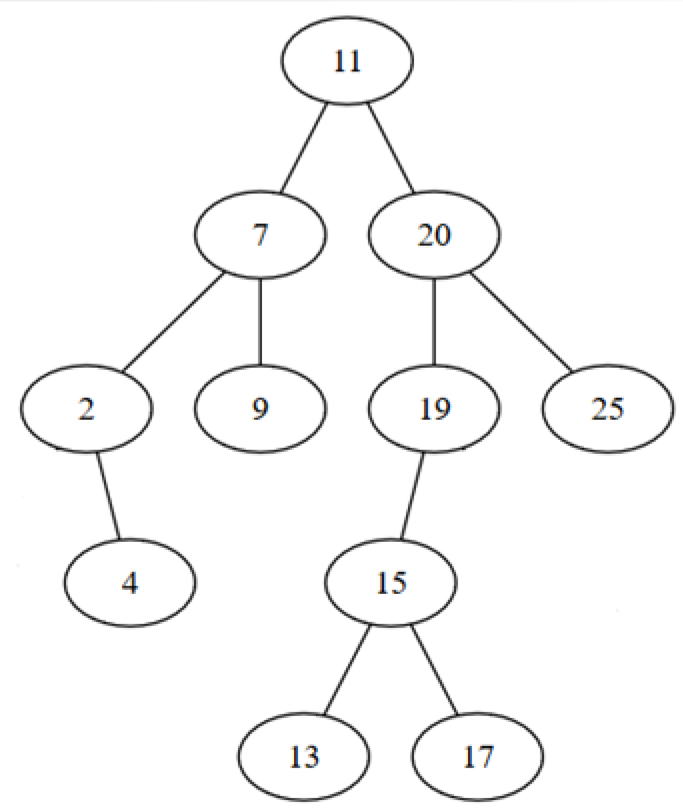


- If we do an inorder traversal of a binary search tree, we will get the elements in increasing order (according to the relation used).
- Binary search trees can be used as representation for sorted containers: sorted maps, sorted multimaps, priority queues, sorted sets, etc.

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-function

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

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

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

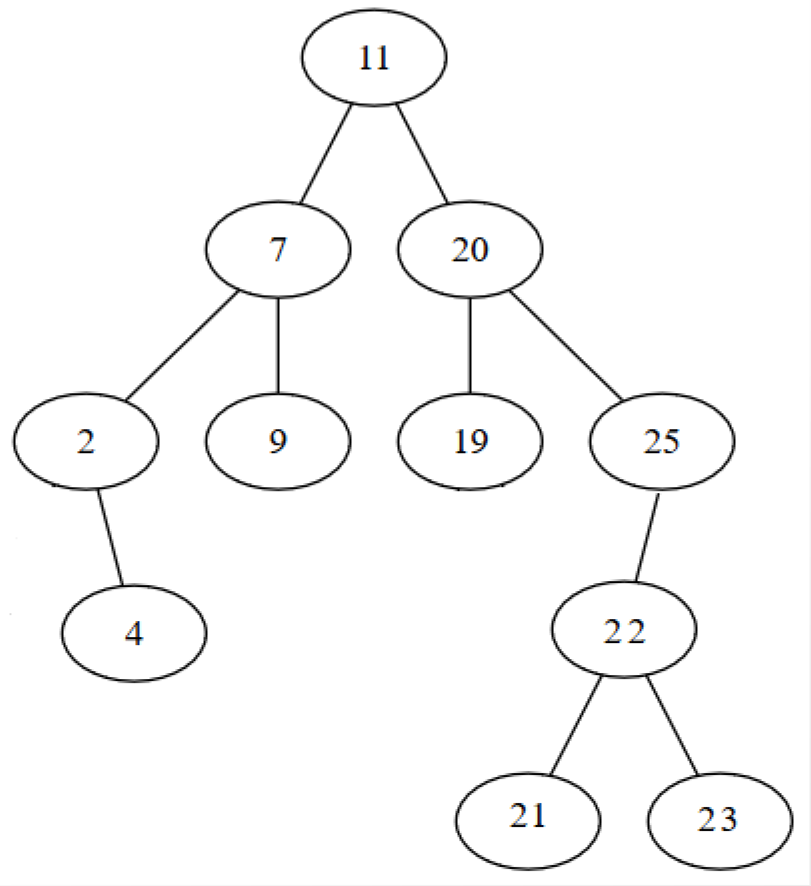
Finding the minimum element

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)

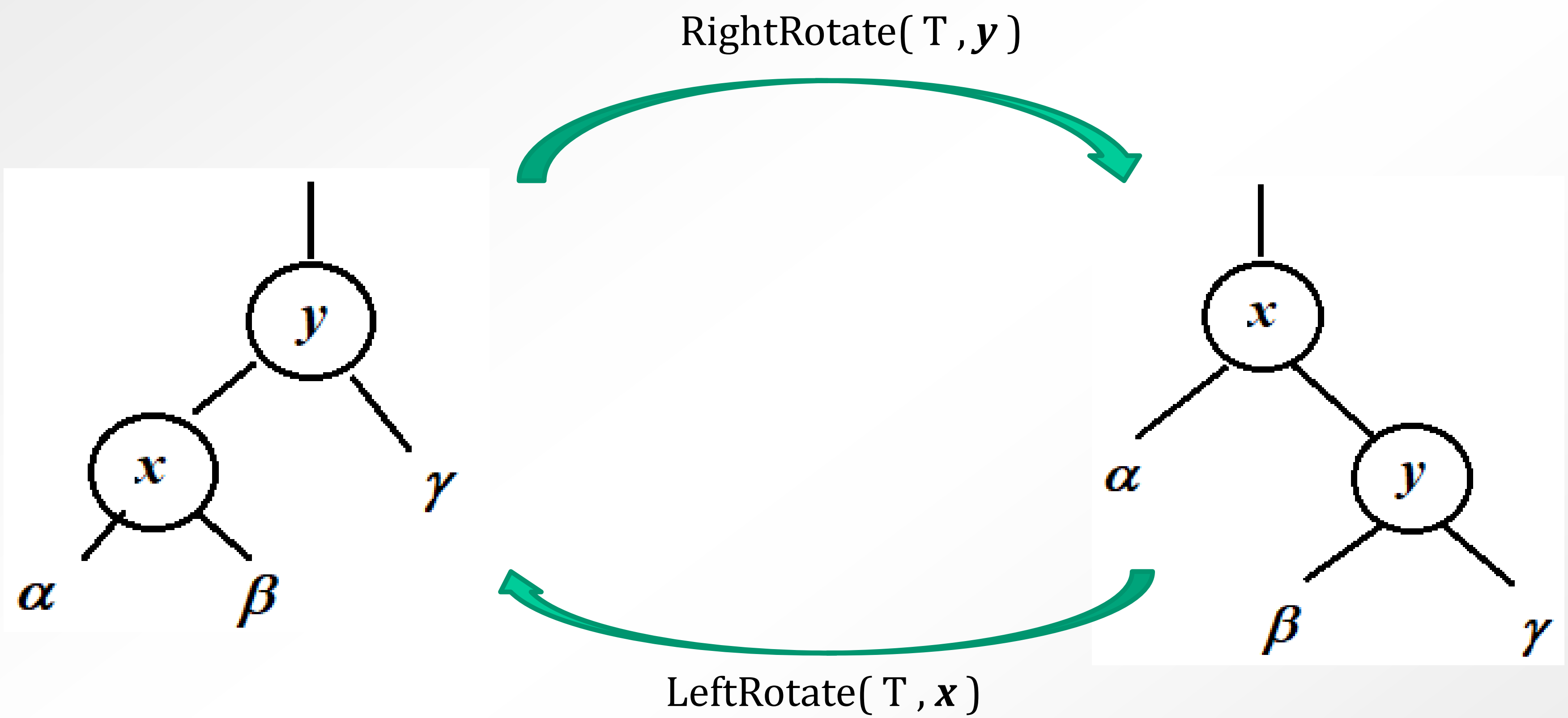
BST

Think about it:

- 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?
- From the tree in the above example, remove 3 (show both options)

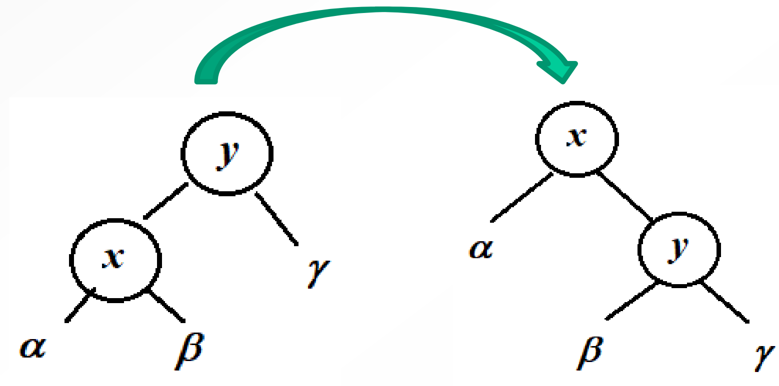
- Give 2 different BSTs that contains the same set of elements
- Given a BST, give 2 different sequences of distinct elements that can create that tree

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

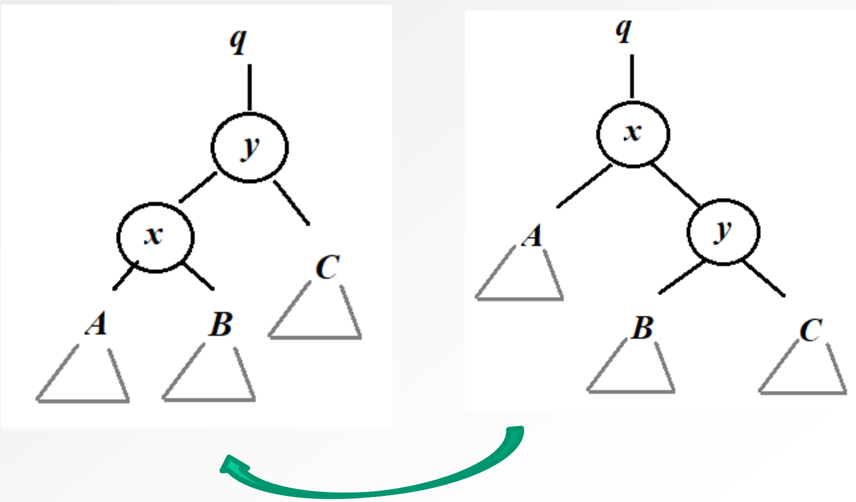
RotateRight ← x ***// New root***

end_RotateRight

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.
```

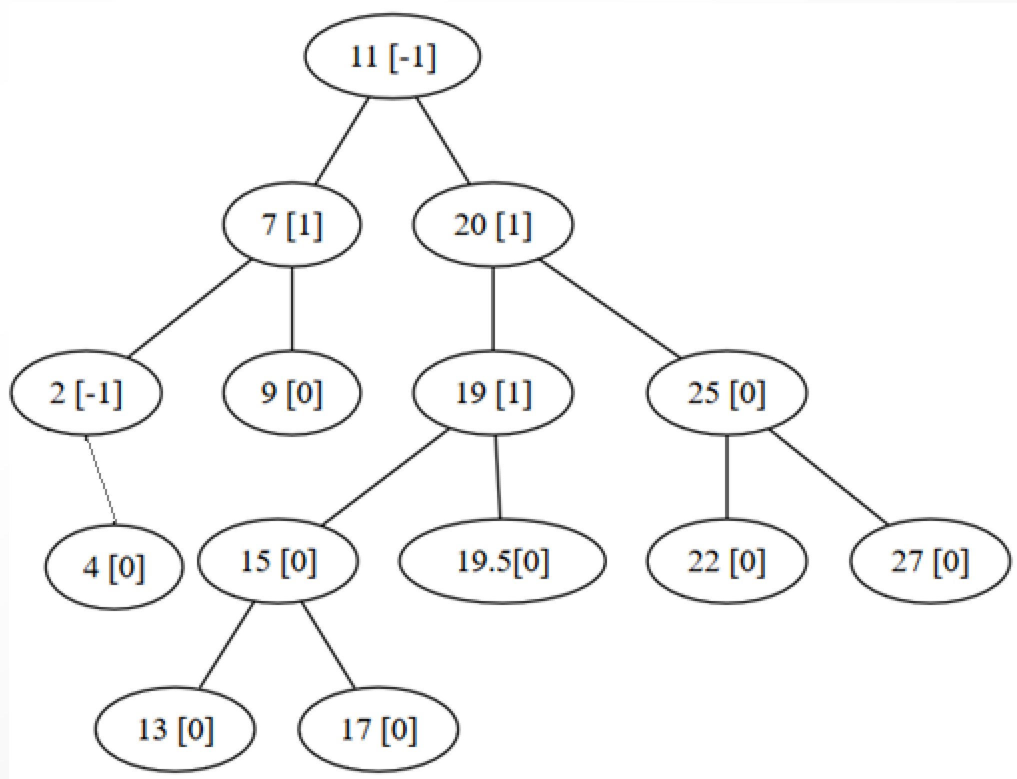
AVL trees

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



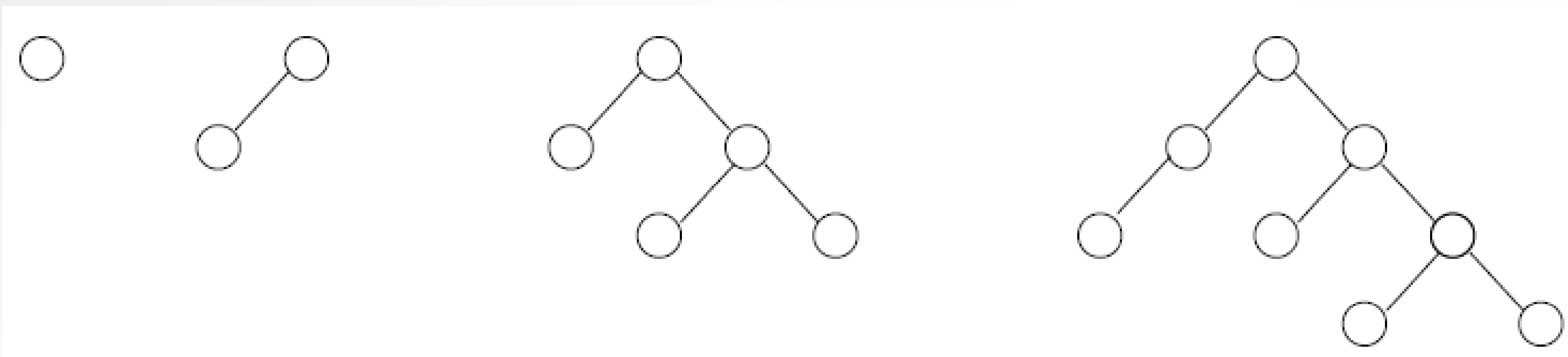
Remarks:

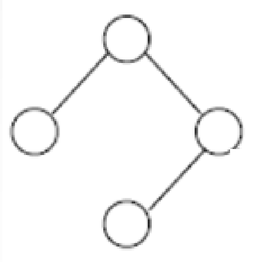
- Height of an empty tree is -1
- Height of a single node is 0

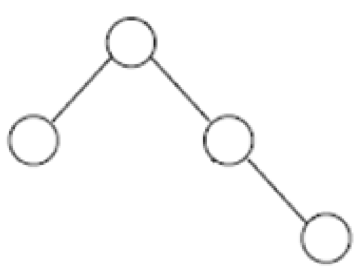
Values in square brackets show the balancing information of a node.

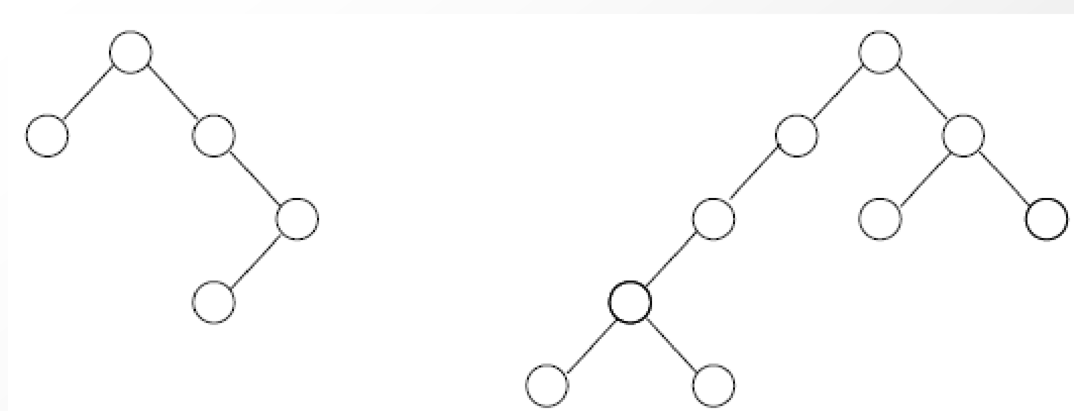
AVL trees

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





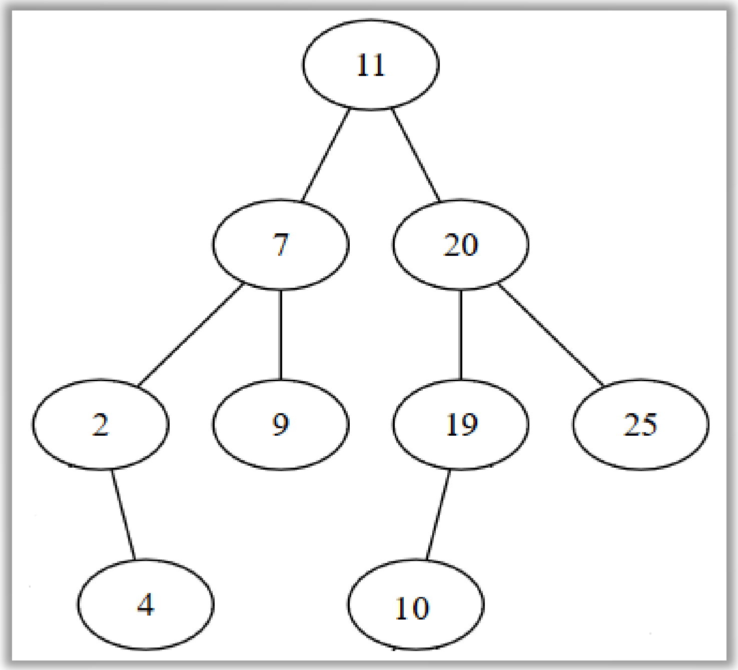


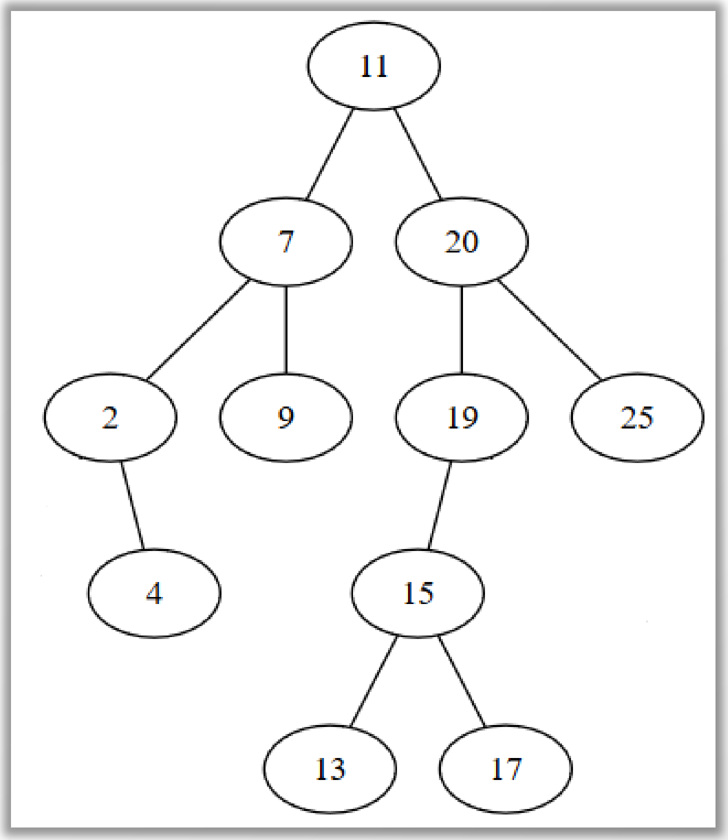


?

AVL trees

Are these AVL trees?





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

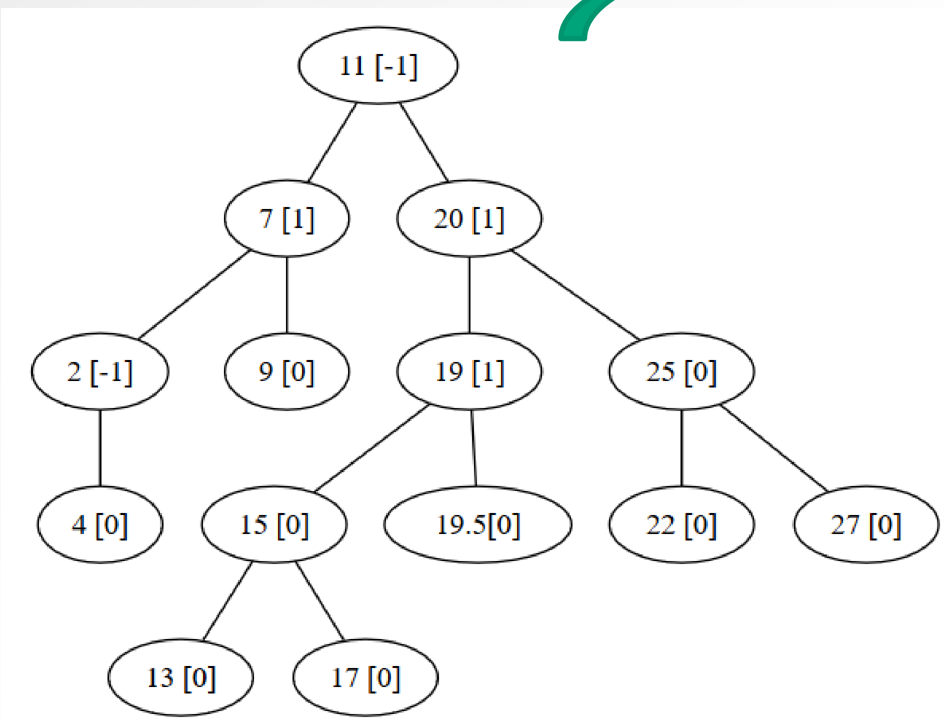
  ➢ Use rotations: they keep the BST property.

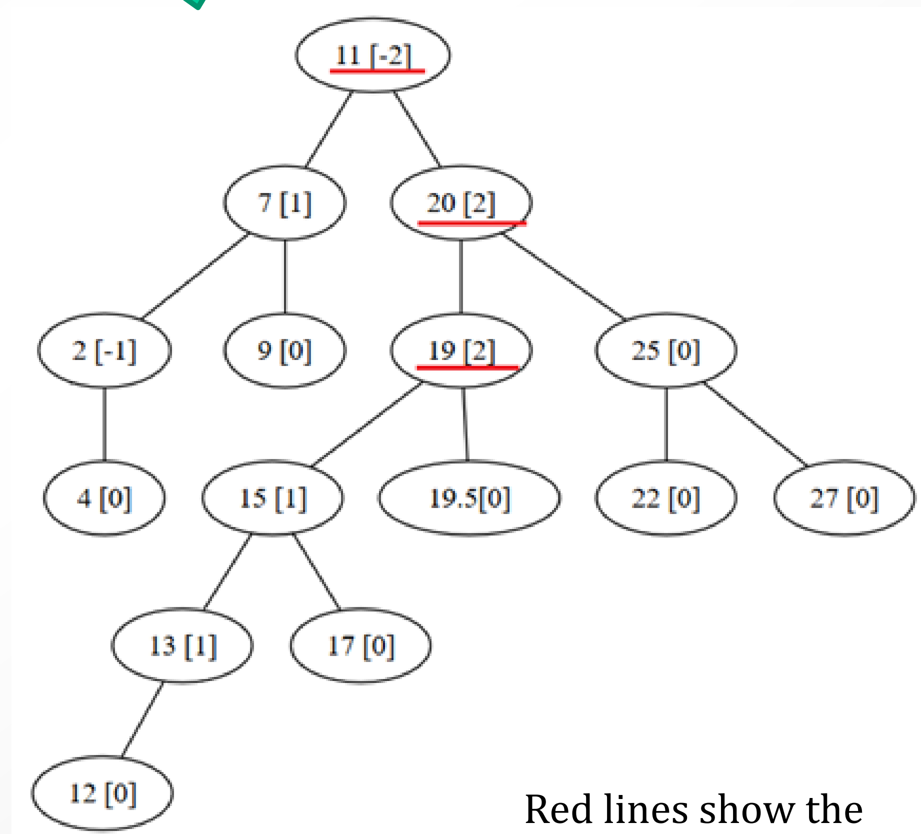
Properties:

- 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





Red lines show the unbalanced nodes.

AVL - insert

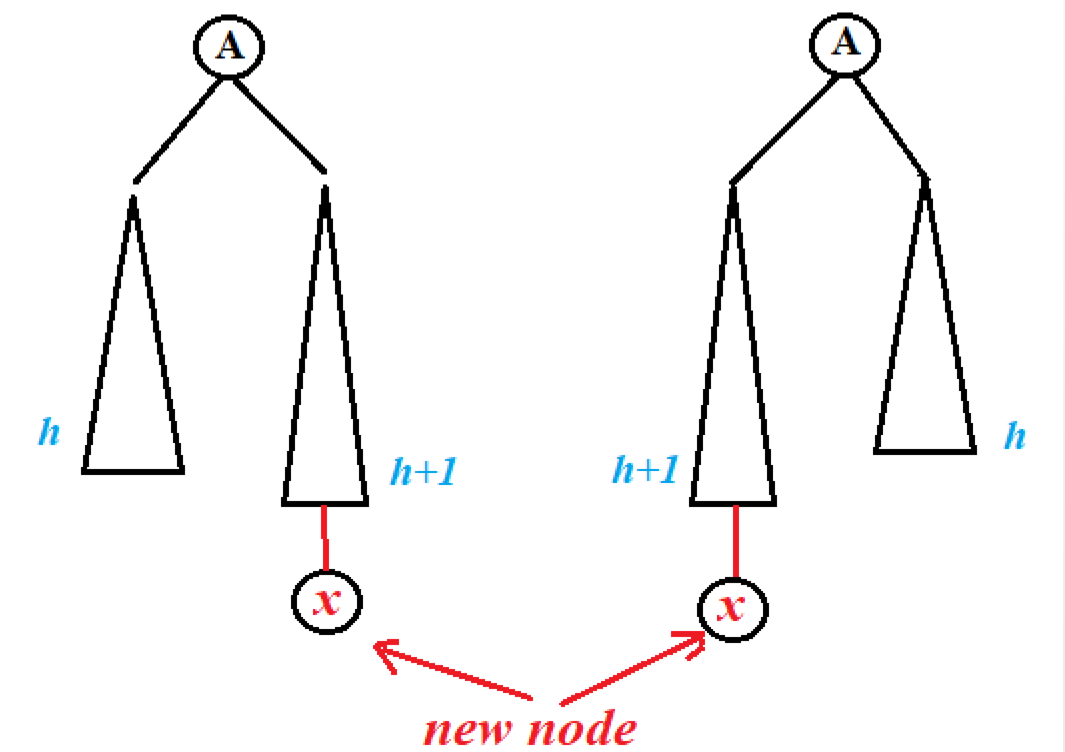
Insertion:

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

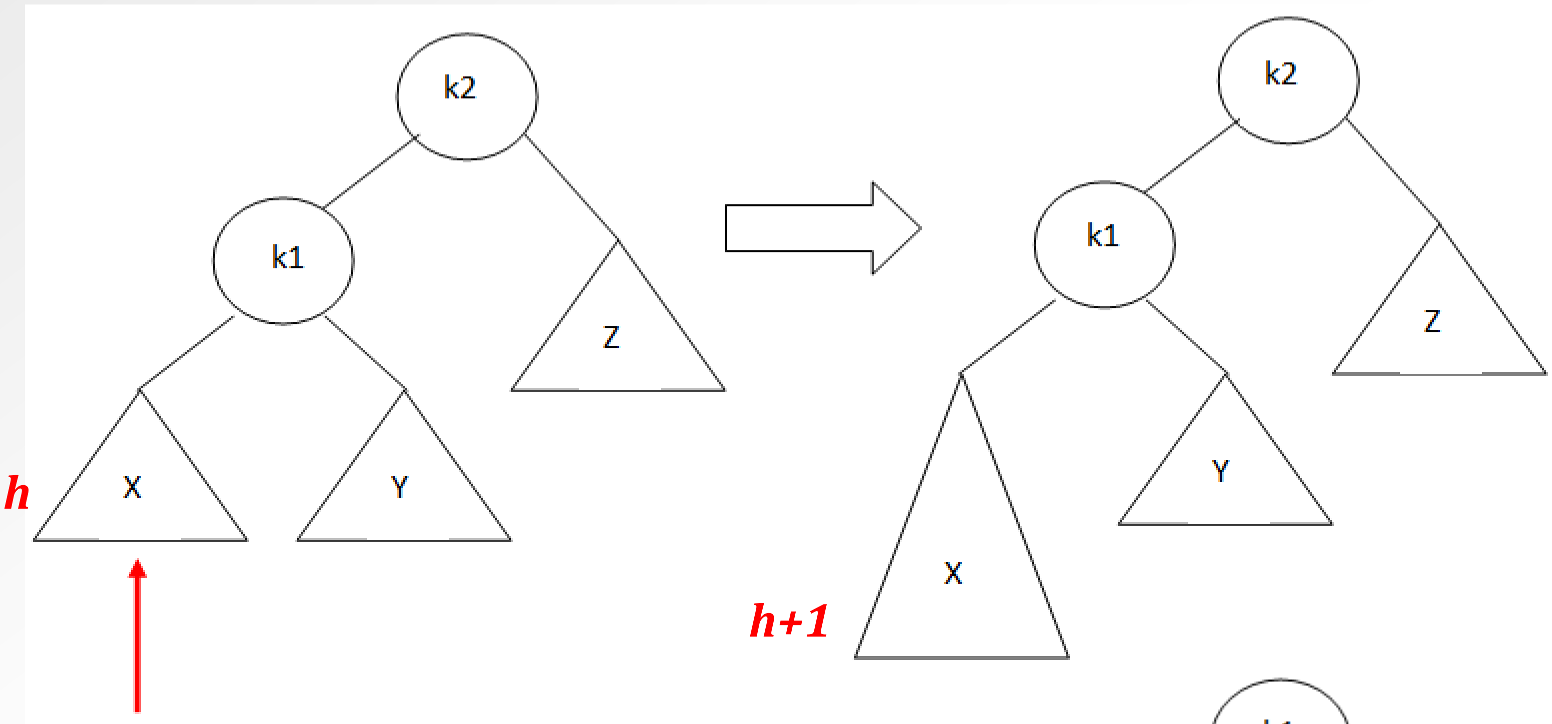
  consider all the ancestors (to the root)

  rebalance → one or more tree rotations.

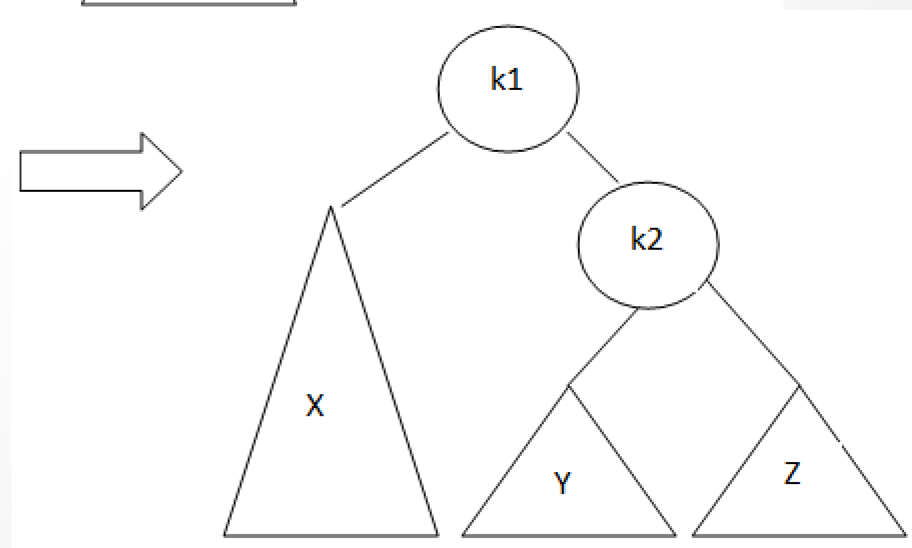
When to rebalance:



AVL Trees – insert : case 1

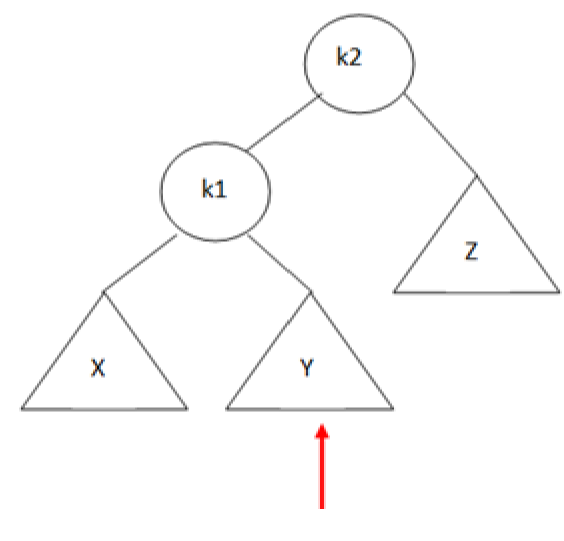


Add here

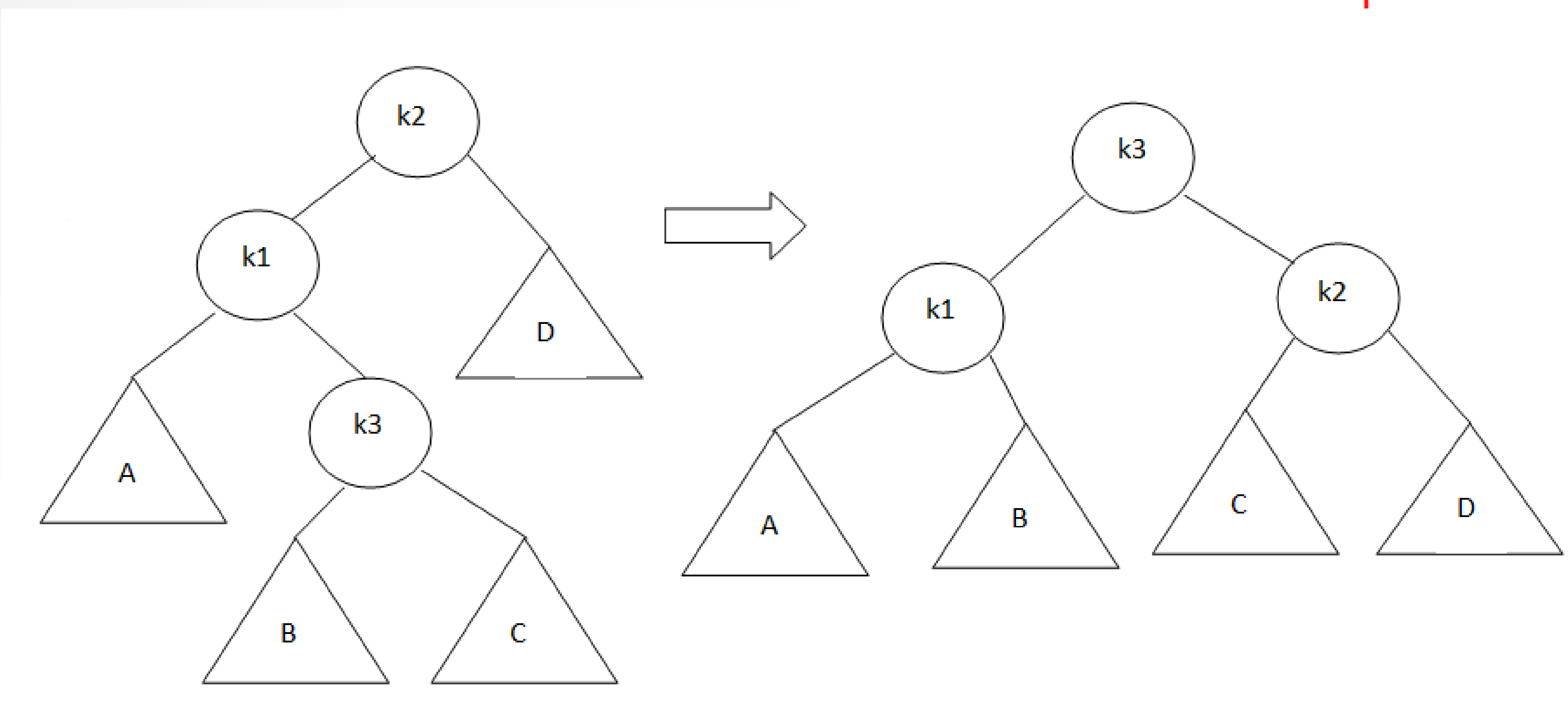


- X, Y and Z represent subtrees with the same height.
- Solution: single rotation to right

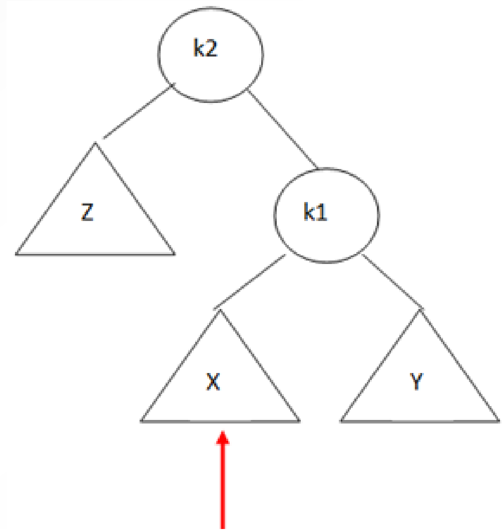
AVL Trees – insert: case 2



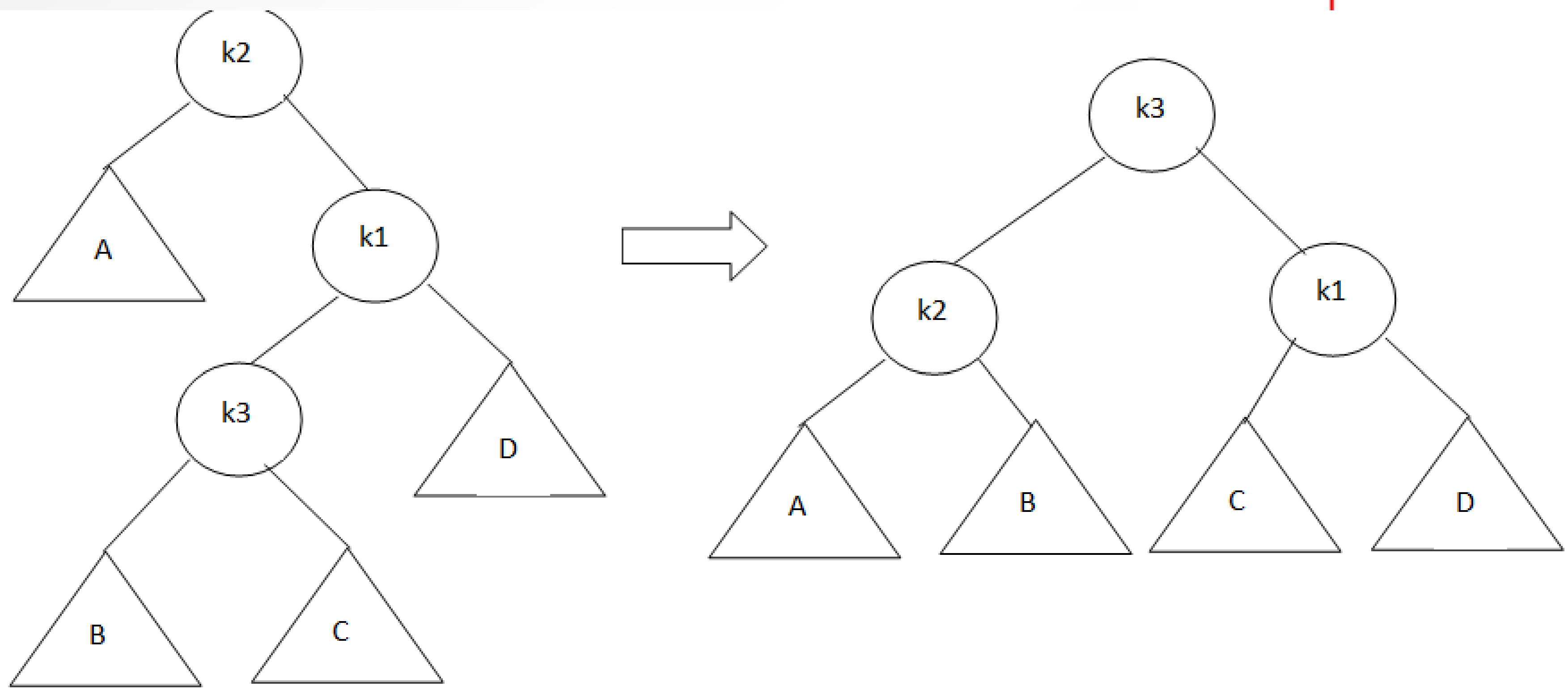
Double rotation to right



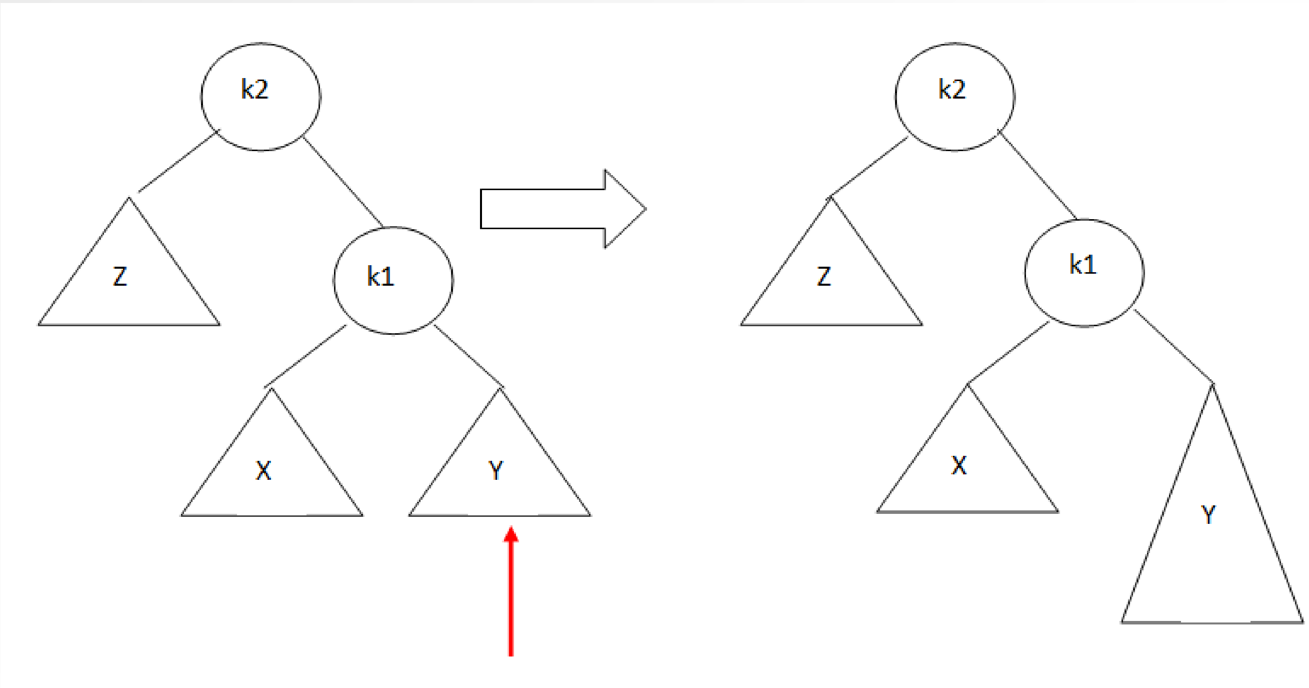
AVL Trees – insert: case 3

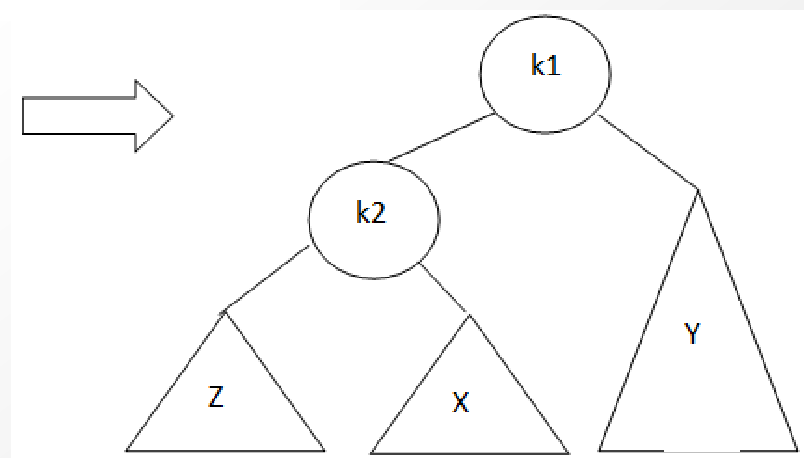


Double rotation to left



AVL Trees – insert: case 4

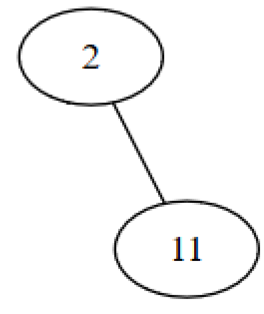


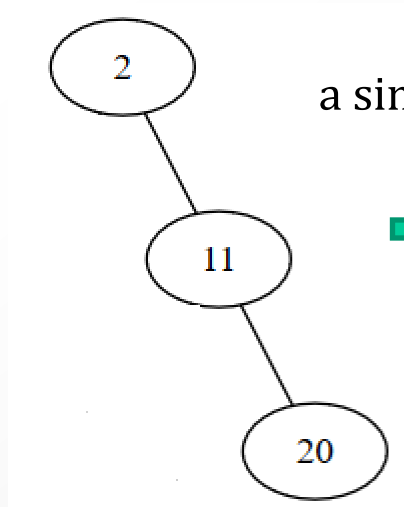


Single rotation
to left

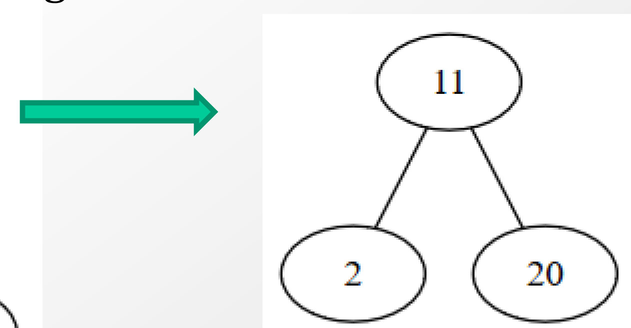
AVL insert: example

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



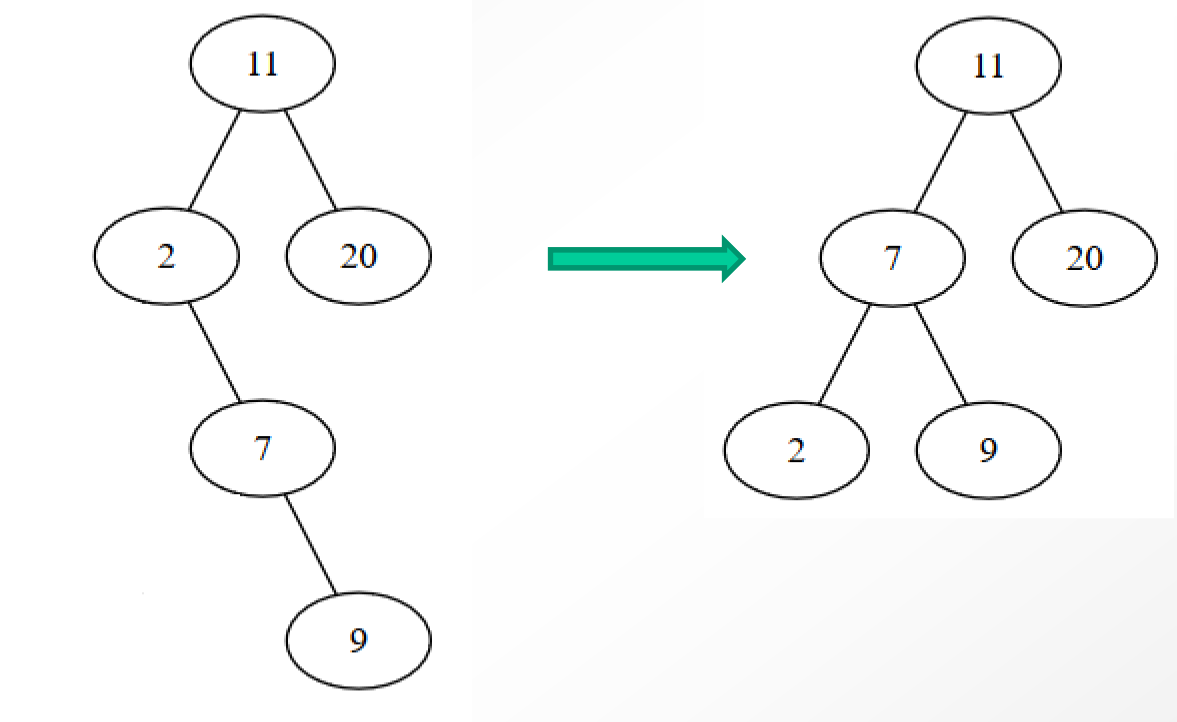


a single left rotation on node 2



AVL insert: example

- Operation:
  Insert 9

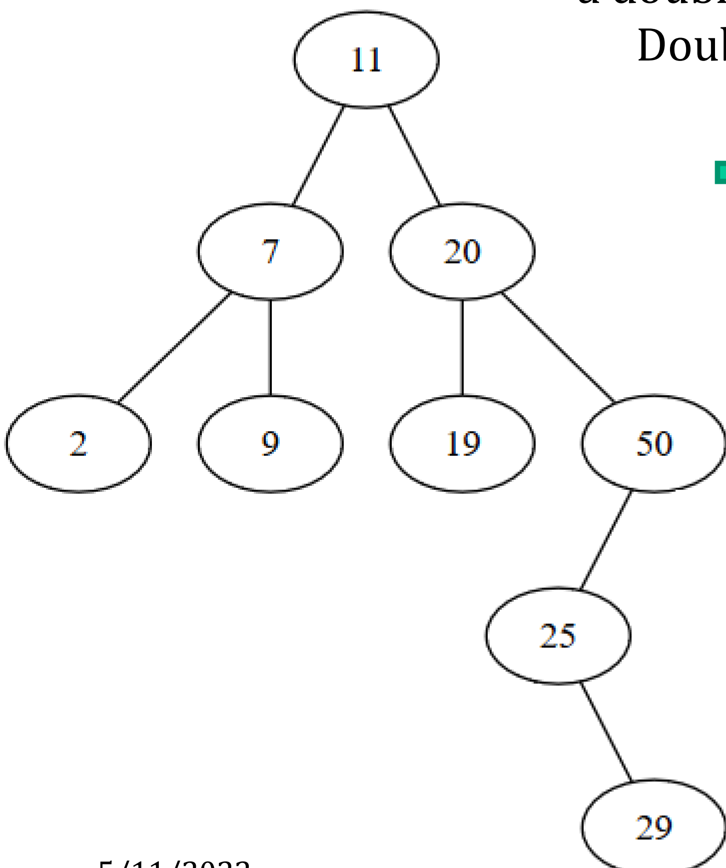


a single left rotation on node 2

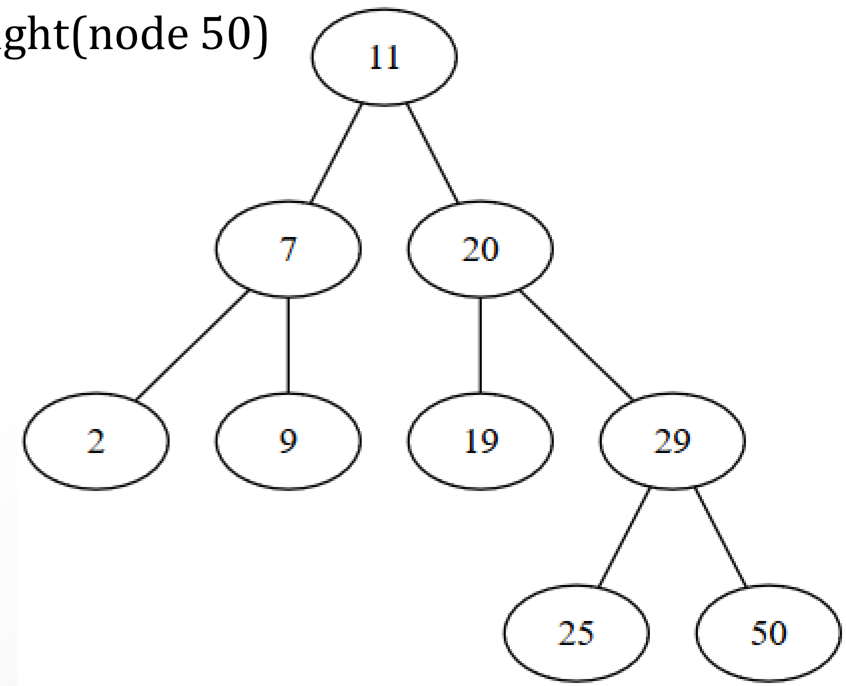
- Insert 50
- Insert 19
- Insert 25
- Insert 29

AVL insert: example

- Operation: insert 29



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



AVL insert: example

- Operation: add 21 to the previous tree

a double left rotation on node 20