

DATA STRUCTURES AND ALGORITHMS

LECTURE 13

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2023 - 2024

In Lecture 12...

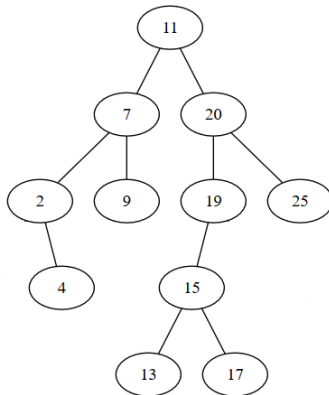
- Binary tree - representation and traversals
- Binary search trees

- AVL trees

Binary search trees - Recap

- A *Binary Search Tree* 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 the information from x
- Obviously, the relation used to order the nodes can be considered in an abstract way (instead of having " \leq " as in the definition).

Binary Search Tree Example



- If we do an inorder traversal of a binary search tree, we will get the elements in increasing order (according to the relation used).

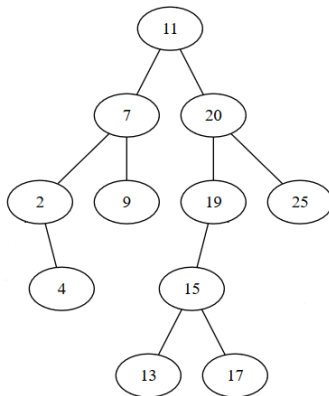
Balanced Binary Search Trees

- Specific operations for binary trees run in $O(h)$ time, which can be $\Theta(n)$ in worst case
- Best case is a balanced tree, where height of the tree is $\Theta(\log_2 n)$

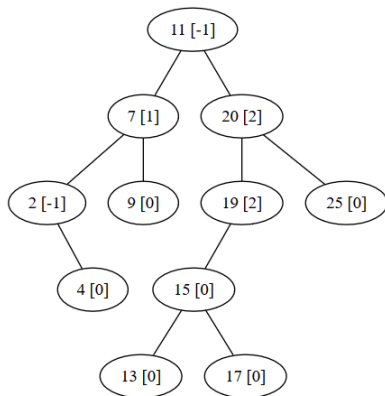
Balanced Binary Search Trees

- Specific operations for binary trees run in $O(h)$ time, which can be $\Theta(n)$ in worst case
- Best case is a balanced tree, where height of the tree is $\Theta(\log_2 n)$
- To reduce the complexity of algorithms, we want to keep the tree balanced. In order to do this, we want every node to be balanced.
- When a node loses its balance, we will perform some operations (called rotations) to make it balanced again.

- Definition: An AVL (Adelson-Velskii Landis) tree is a binary 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 (balancing information)
- Observations:
 - Height of an empty tree is -1
 - Height of a single node is 0

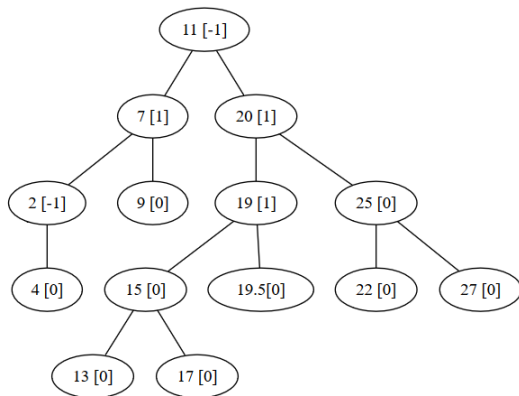


- Is this an AVL tree?



- Values in square brackets show the balancing information of a node. The tree is not an AVL tree, because the balancing information for nodes 19 and 20 is 2.

AVL Trees



- This is an AVL tree.

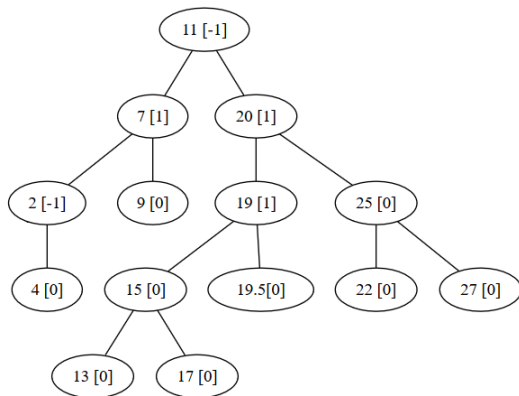
AVL Trees - rotations

- Adding or removing a node might result in a binary tree that violates the AVL tree property.
- In such cases, the property has to be restored and only after the property holds again is the operation (add or remove) considered finished.
- The AVL tree property can be restored with operations called **rotations**.

AVL Trees - rotations

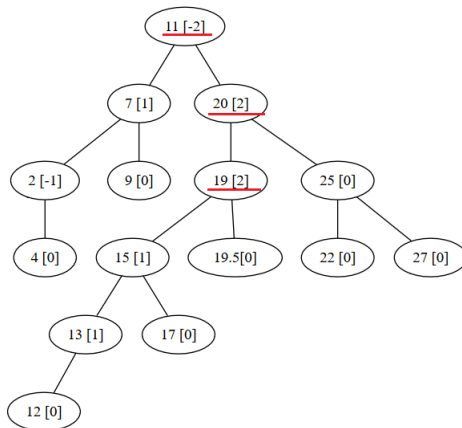
- Inserting in an AVL tree is similar to inserting in a BST: we need to find the position where the new element needs to be added and add the element there. However, this will change the balancing information of some nodes.
- After an insertion, only the nodes on the path to the modified node can change their height.
- We check the balancing information on the path from the modified node to the root (so from bottom to the top). 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 - rotations



- What if we insert element 12?

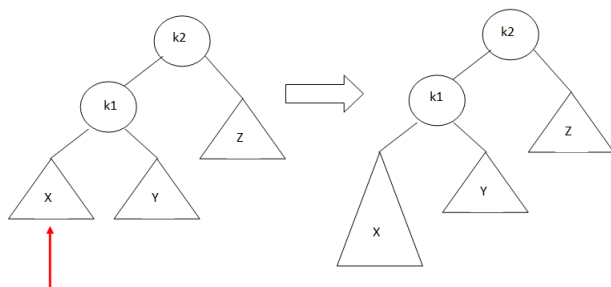
AVL Trees - rotations



- Red lines show the unbalanced nodes. We will rebalance node 19.

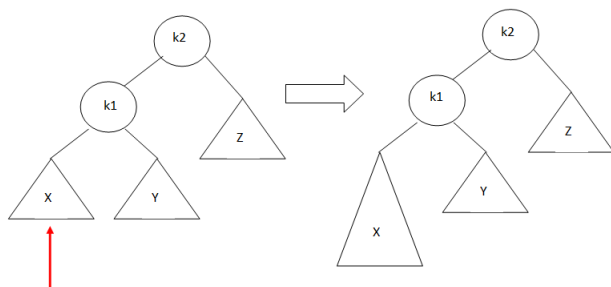
- Assume that at a given point α is the node that needs to be rebalanced.
- Since α was balanced before the insertion, and is not after the insertion, we know that a new node was inserted somewhere in a subtree of α and we can identify four cases in which a violation might occur:
 - Insertion into the left subtree of the left child of α
 - Insertion into the right subtree of the left child of α
 - Insertion into the left subtree of the right child of α
 - Insertion into the right subtree of the right child of α

AVL Trees - rotations - case 1



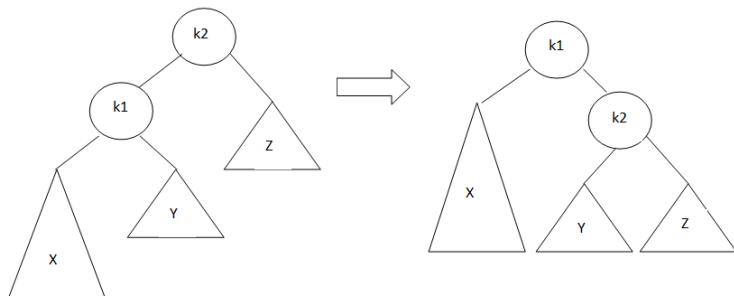
- Obs: X, Y and Z represent subtrees with the same height.

AVL Trees - rotations - case 1

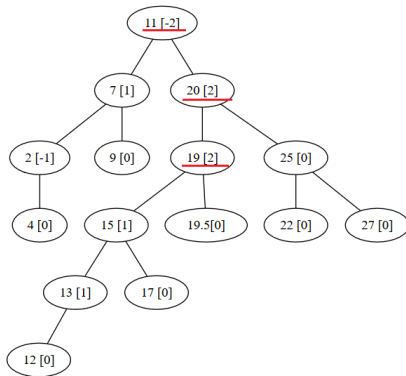


- Obs: X , Y and Z represent subtrees with the same height.
- Solution: single rotation to right

AVL Trees - rotation - Single Rotation to Right

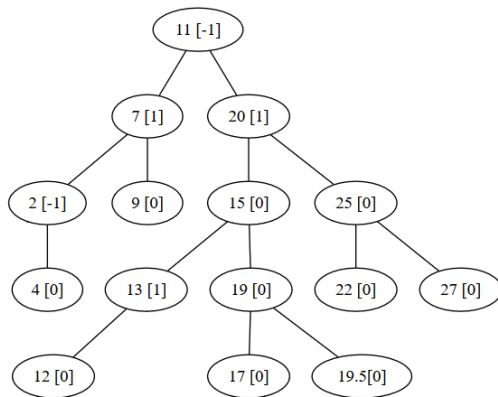


AVL Trees - rotations - case 1 example

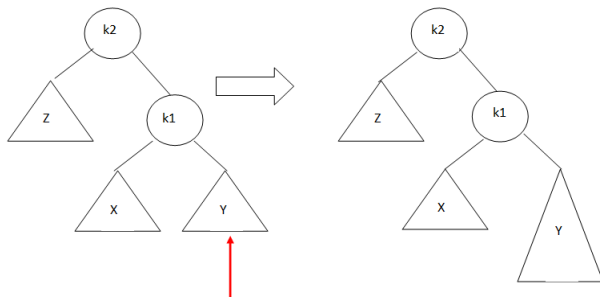


- Node 19 is imbalanced, because we inserted a new node (12) in the left subtree of the left child.
- Solution: **single rotation to right**

AVL Trees - rotation - case 1 example

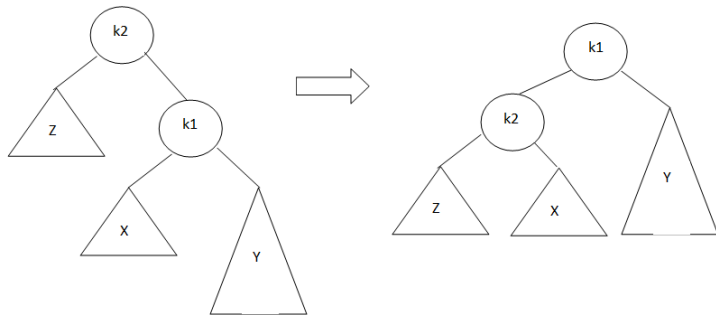


AVL Trees - rotations - case 4

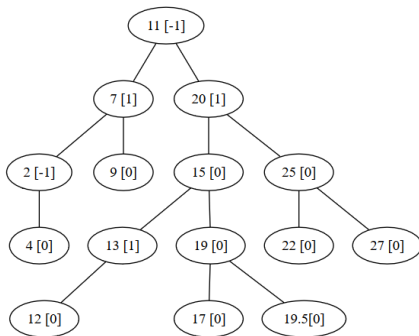


- Solution: **single rotation to left**

AVL Trees - rotation - Single Rotation to Left

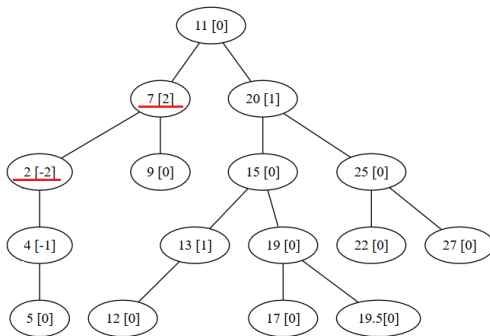


AVL Trees - rotations - case 4 example



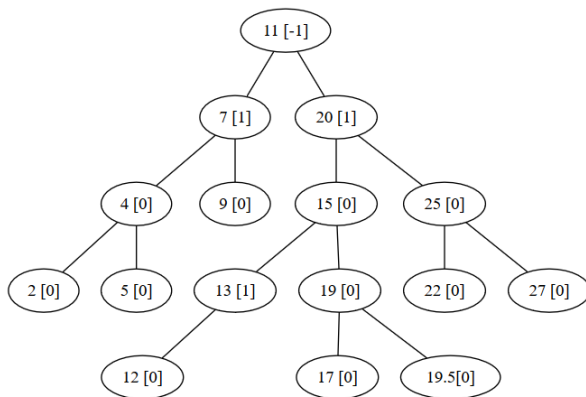
- Insert value 5

AVL Trees - rotations - case 4 example



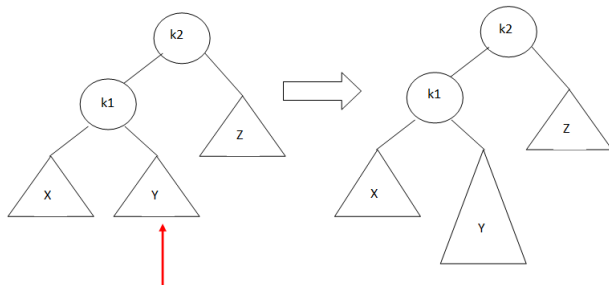
- Node 2 is imbalanced, because we inserted a new node (5) to the right subtree of the right child
- Solution: **single rotation to left**

AVL Trees - rotation - case 4 example



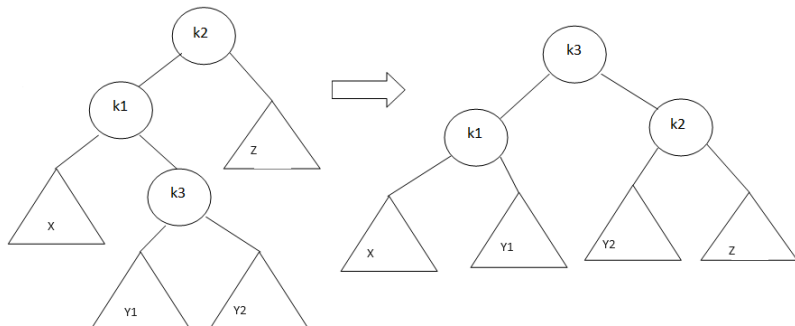
- After the rotation

AVL Trees - rotations - case 2

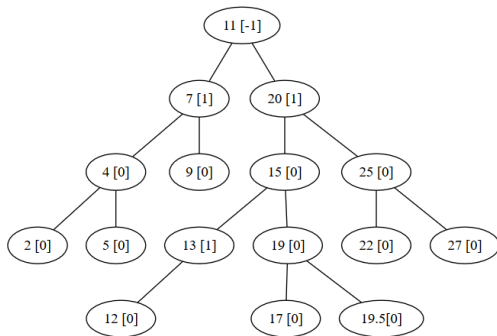


- Solution: **Double rotation to right**

AVL Trees - rotation - Double Rotation to Right

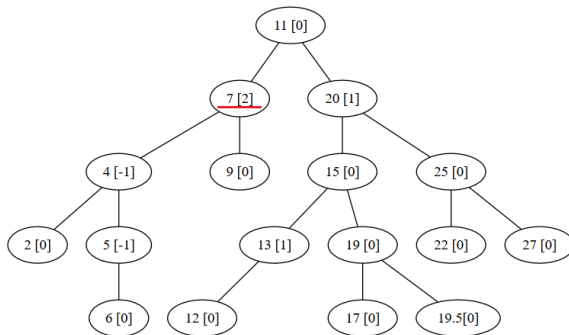


AVL Trees - rotations - case 2 example



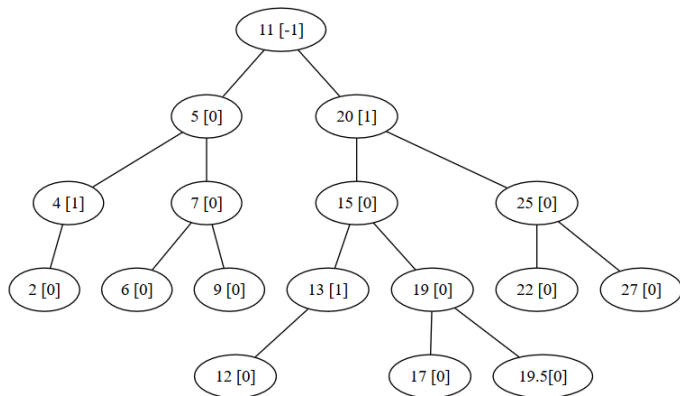
- Insert value 6

AVL Trees - rotations - case 2 example



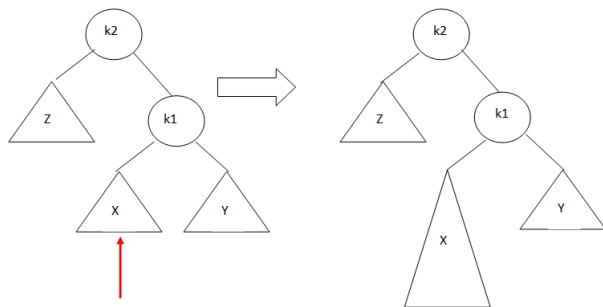
- Node 7 is imbalanced, because we inserted a new node (6) to the right subtree of the left child
- Solution: **double rotation to right**

AVL Trees - rotation - case 2 example



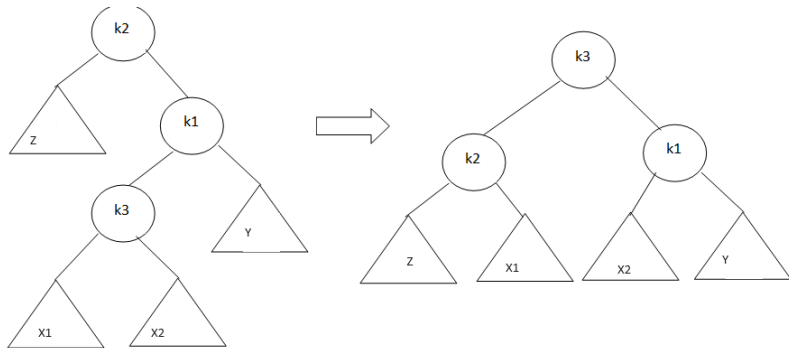
- After the rotation

AVL Trees - rotations - case 3

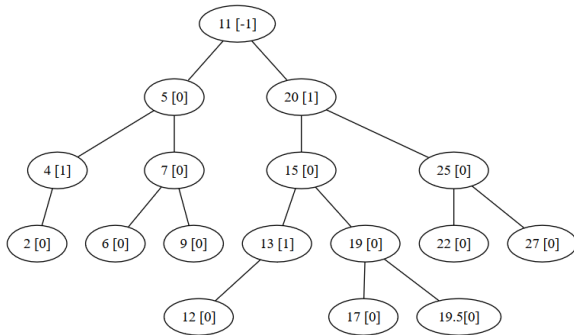


- Solution: **Double rotation to left**

AVL Trees - rotation - Double Rotation to Left

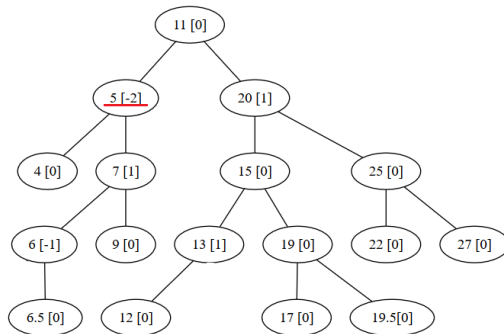


AVL Trees - rotations - case 3 example



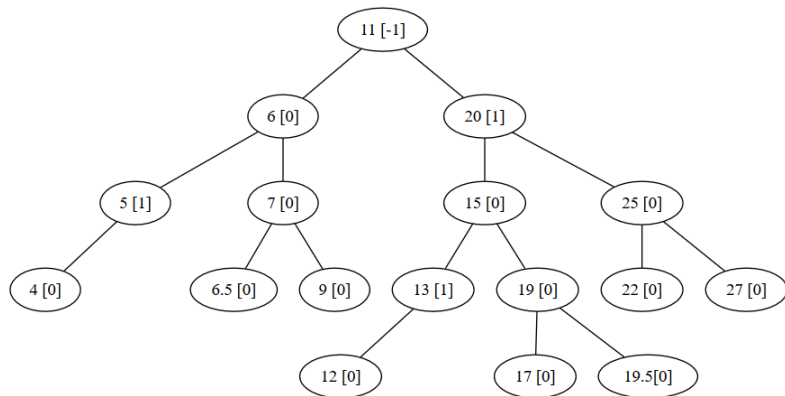
- Remove node with value 2 and insert value 6.5

AVL Trees - rotations - case 3 example



- Node 5 is imbalanced, because we inserted a new node (6.5) to the left subtree of the right child
- Solution: **double rotation to left**

AVL Trees - rotation - case 3 example



- After the rotation

Recognizing the rotations I

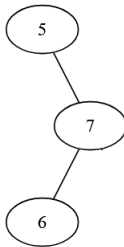
- One way of determining what kind of rotation we have is to consider the following terms:
 - *heavy-left* - a node is heavy-left if it has more nodes on the left side than on the right side. → a rotation to the right is needed to balance it.
 - *heavy-right* - a node is heavy-right if it has more nodes on the right side than on the left side. → a rotation to the left is needed to balance it.

Recognizing the rotations II

- When a node is imbalanced, we first check if it is *heavy-left* or *heavy-right*. If the node is *heavy-left* we look at its left child to see if it is *heavy-left* (a single notation will be enough) or *heavy-right* (we will need a double rotation).
- We do the same if the node is *heavy-right*, looking at its right child to see if it is *heavy-left* (a double rotation will be needed) or *heavy-right* (a single rotation is enough).

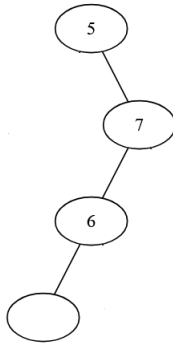
Double rotations I

- Some resources say that a double rotation can be achieved by doing two consecutive single rotations, but for the first one we do not have all the nodes.
- Let's see an example. Consider the following simple case:



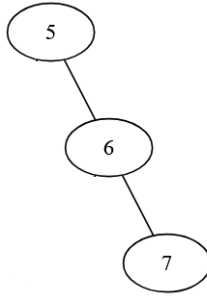
- This is the simplest, classic case of a double left rotation. However, we could also imagine that node 6 has a left child.

Double rotations II



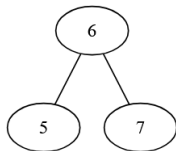
- In this case, we actually have an imbalance at node 7, where we need a single right rotation to balance it.

Double rotations III



- Now, we need a single left rotation to balance the tree.

Double rotations IV



- So, we have done a double left rotation by first doing a single right rotation (with incomplete nodes) and then a single left one.

AVL rotations example I

- Start with an empty AVL tree
- Insert 2

AVL rotations example II

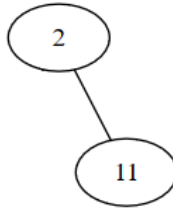


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example III

- No rotation is needed
- Insert 11

AVL rotations example IV

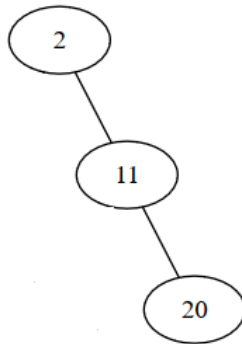


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example V

- No rotation is needed
- Insert 20

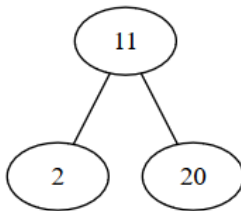
AVL rotations example VI



- Do we need a rotation?
- If yes, on which node and what type of rotation?

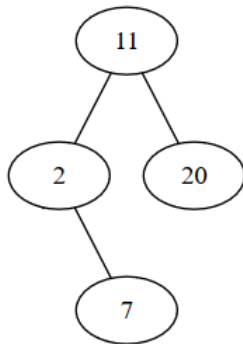
AVL rotations example VII

- Yes, we need a single left rotation on node 2
- After the rotation:



- Insert 7

AVL rotations example VIII

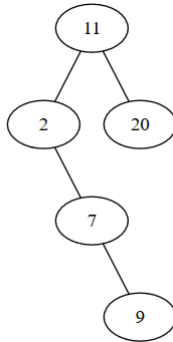


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example IX

- No rotation is needed
- Insert 9

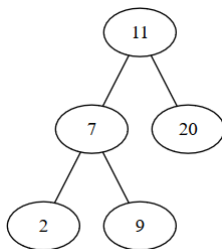
AVL rotations example X



- Do we need a rotation?
- If yes, on which node and what type of rotation?

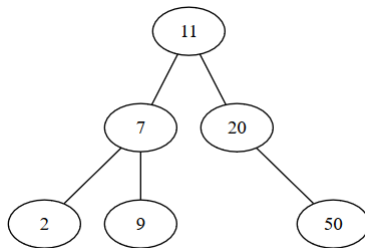
AVL rotations example XI

- Yes, we need a single left rotation on node 2
- After the rotation:



- Insert 50

AVL rotations example XII

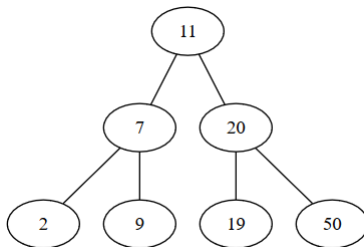


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XIII

- No rotation is needed
- Insert 19

AVL rotations example XIV

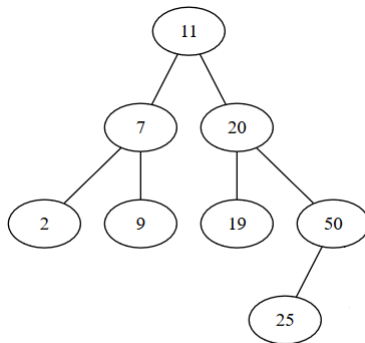


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XV

- No rotation is needed
- Insert 25

AVL rotations example XVI

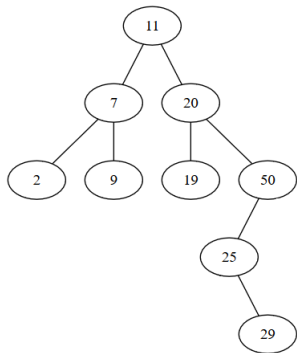


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XVII

- No rotation is needed
- Insert 29

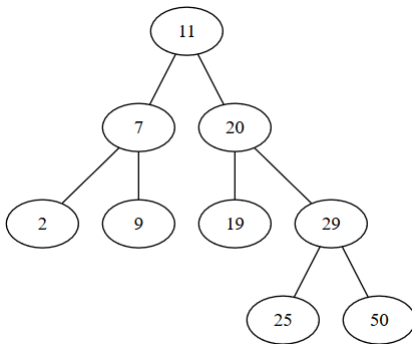
AVL rotations example XVIII



- Do we need a rotation?
- If yes, on which node and what type of rotation?

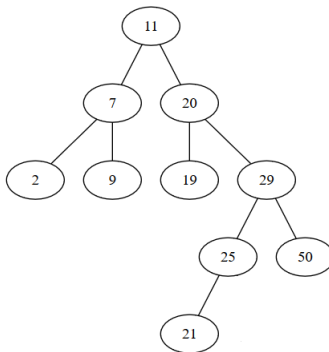
AVL rotations example XIX

- Yes, we need a double right rotation on node 50
- After the rotation



- Insert 21

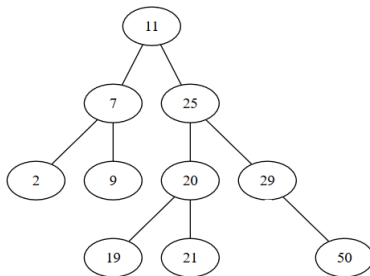
AVL rotations example XX



- Do we need a rotation?
- If yes, on which node and what type of rotation?

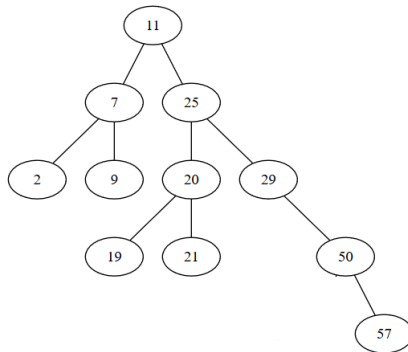
AVL rotations example XXI

- Yes, we need a double left rotation on node 20
- After the rotation



- Insert 57

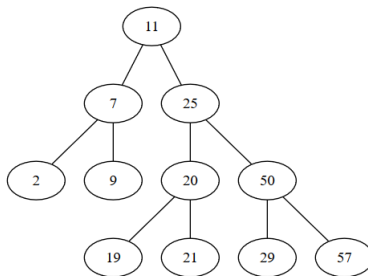
AVL rotations example XXII



- Do we need a rotation?
- If yes, on which node and what type of rotation?

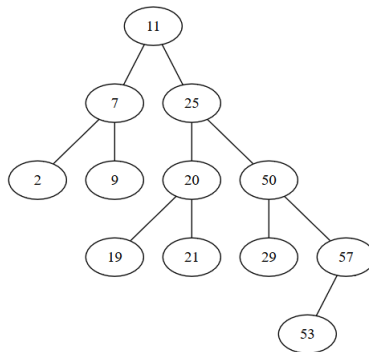
AVL rotations example XXIII

- Yes, we need a single left rotation on node 50
- After the rotation



- Insert 53

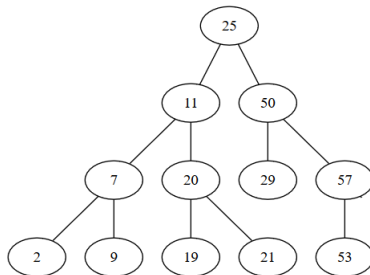
AVL rotations example XXIV



- Do we need a rotation?
- If yes, on which node and what type of rotation?

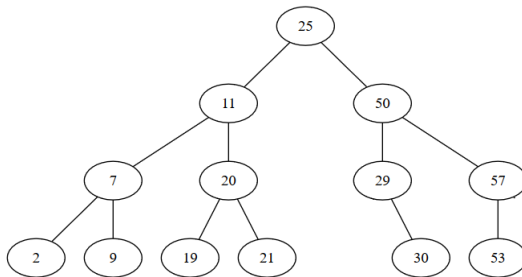
AVL rotations example XXV

- Yes, we need a single left rotation on node 11
- After the rotation



- Insert 30

AVL rotations example XXVI

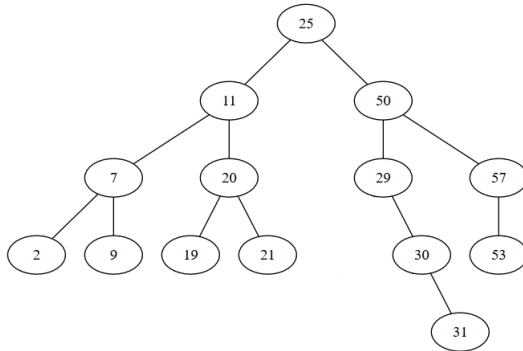


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XXVII

- No rotation is needed
- Insert 31

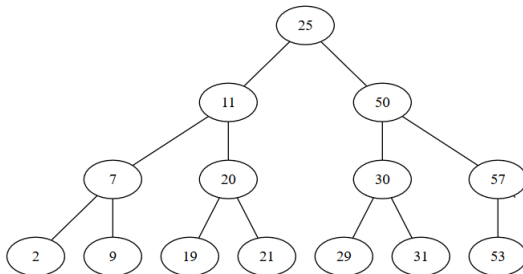
AVL rotations example XXVIII



- Do we need a rotation?
- If yes, on which node and what type of rotation?

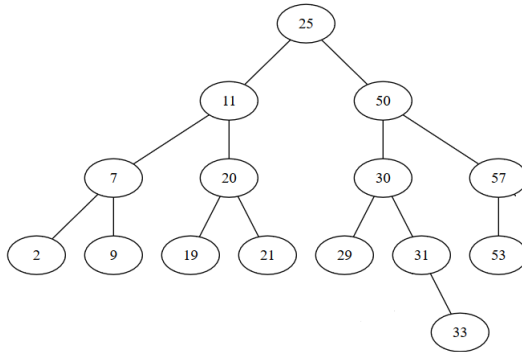
AVL rotations example XXIX

- Yes, we need a single left rotation on node 29
- After the rotation



- Insert 33

AVL rotations example XXX

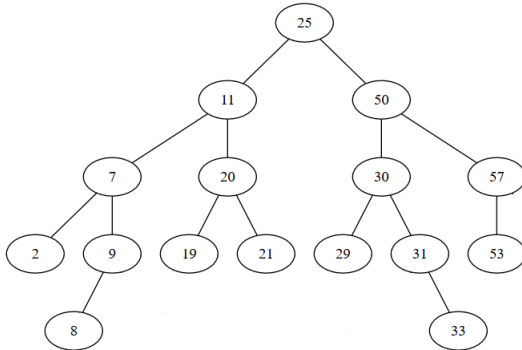


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XXXI

- No rotation is needed
- Insert 8

AVL rotations example XXXII

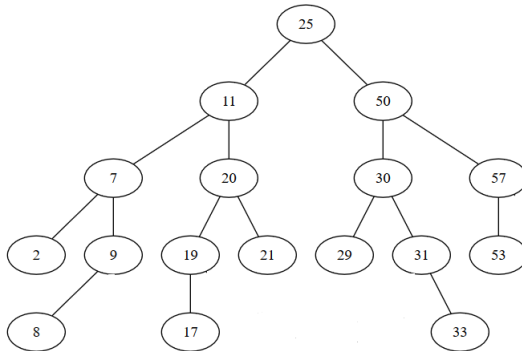


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XXXIII

- No rotation is needed
- Insert 17

AVL rotations example XXXIV

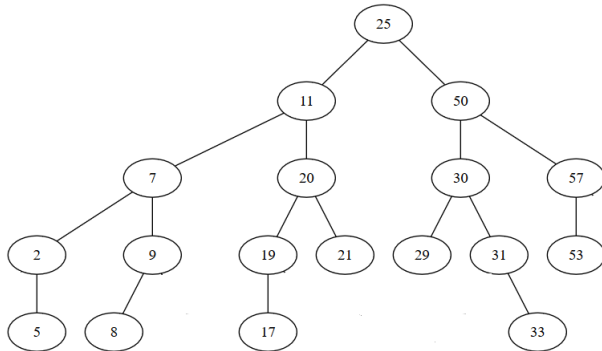


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XXXV

- No rotation is needed
- Insert 5

AVL rotations example XXXVI

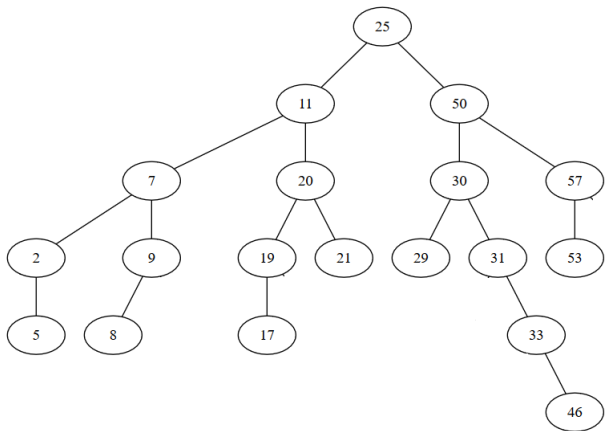


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XXXVII

- No rotation is needed
- Insert 46

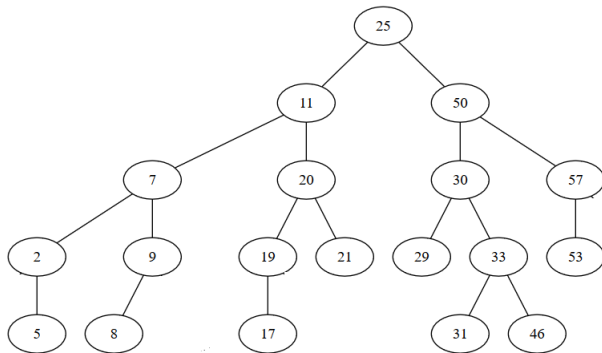
AVL rotations example XXXVIII



- Do we need a rotation?
- If yes, on which node and what type of rotation?

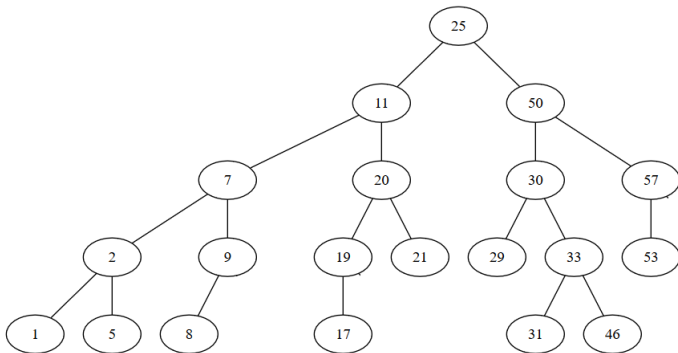
AVL rotations example XXXIX

- Yes, we need a single left rotation on node 31
- After the rotation



- Insert 1

AVL rotations example XL



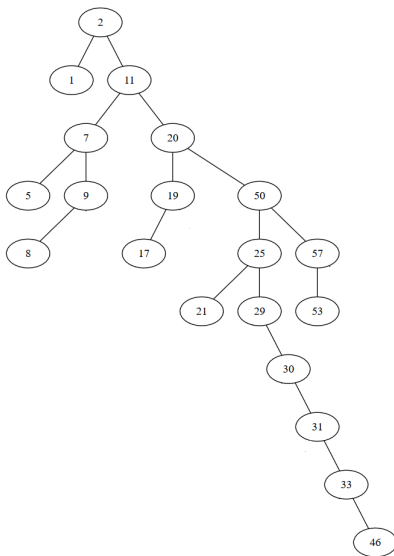
- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XLI

- No rotation is needed

Comparison to BST

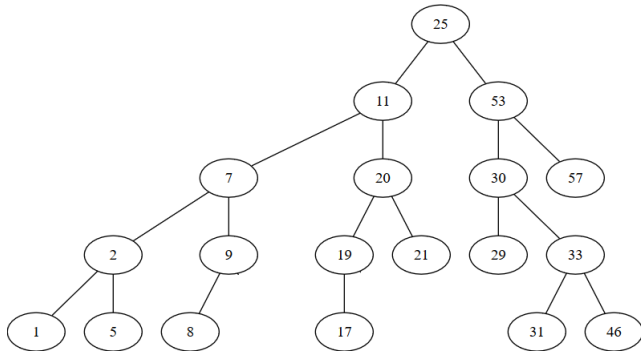
- If, instead of using an AVL tree, we used a binary search tree, after the insertions the tree would have been:



Example of remove I

- Remove 50

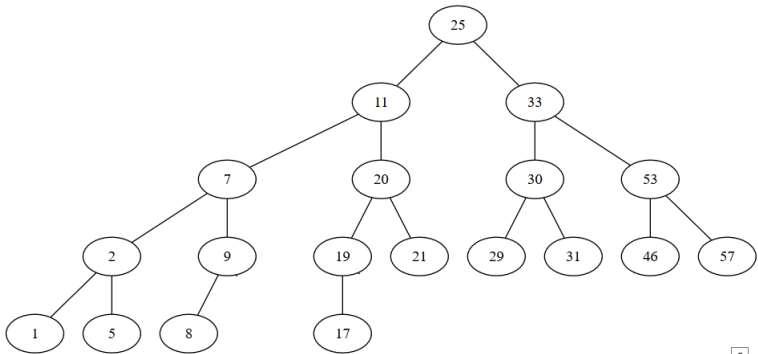
Example of remove II



- Do we need a rotation?
- If yes, on which node and what type of rotation?

Example of remove III

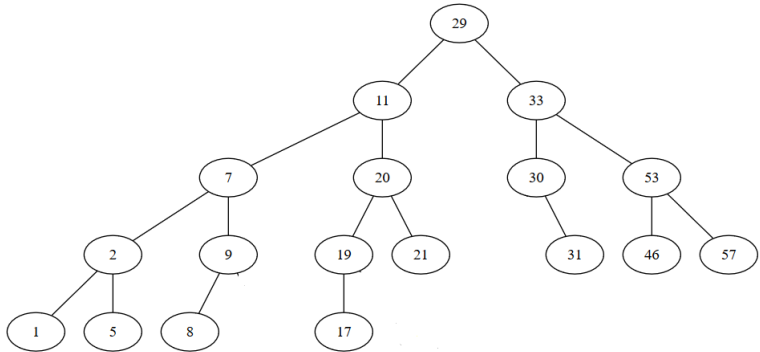
- Yes we need double right rotation on node 53
- After the rotation



G

- Remove 25

Example of remove IV

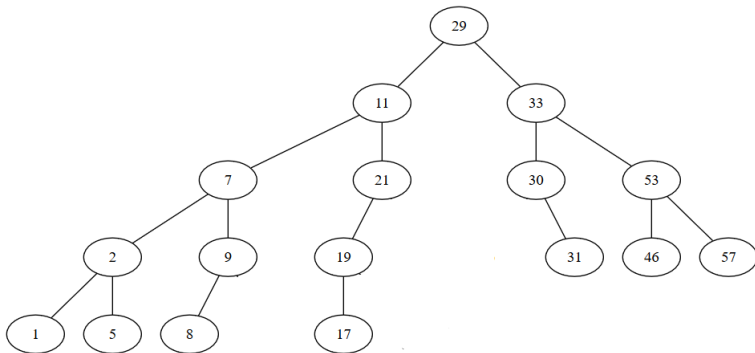


- Do we need a rotation?
- If yes, on which node and what type of rotation?

Example of remove V

- No rotation is needed
- Remove 20

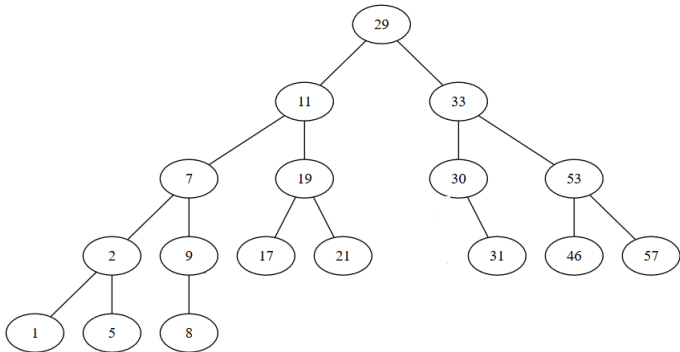
Example of remove VI



- Do we need a rotation?
- If yes, on which node and what type of rotation?

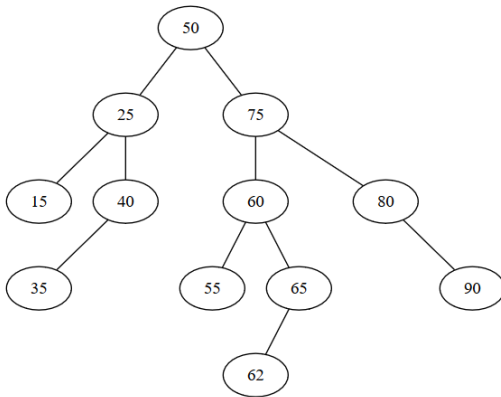
Example of remove VII

- Yes, we need a single right rotation on node 21
- After the rotation



Rotations for remove

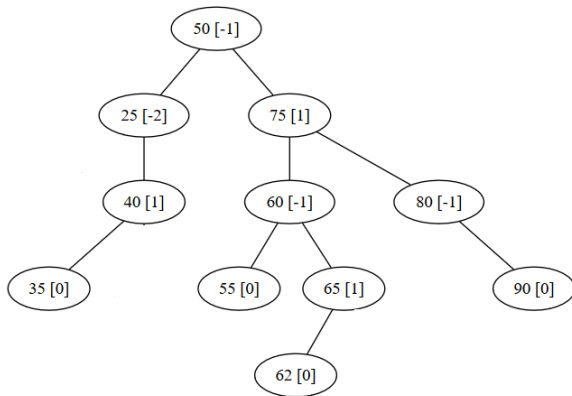
- When we remove a node, we might need more than 1 rotation:



- Remove value 15

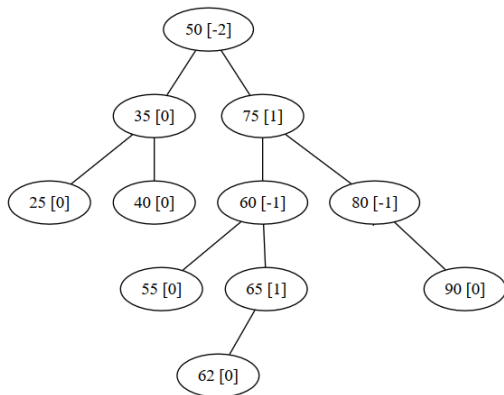
Rotations for remove

- After remove:



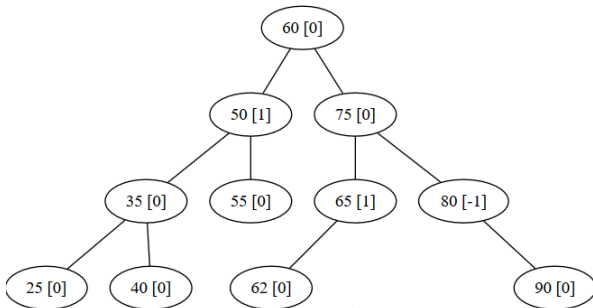
Rotations for remove

- After the rotation



Rotations for remove

- After the second rotation



AVL Trees - representation

- What structures do we need for an AVL Tree?

AVL Trees - representation

- What structures do we need for an AVL Tree?

AVLNode:

info: TComp *//information from the node*

left: \uparrow AVLNode *//address of left child*

right: \uparrow AVLNode *//address of right child*

h: Integer *//height of the node*

AVLTree:

root: \uparrow AVLNode *//root of the tree*

- **Obs:** you might need to keep a relation in the AVLTree structure as well, we will consider the \leq relation by default.

AVL Tree - implementation

- We will implement the *insert* operation for the AVL Tree.
- We need to implement some operations to make the implementation of *insert* simpler:

AVL Tree - implementation

- We will implement the *insert* operation for the AVL Tree.
- We need to implement some operations to make the implementation of *insert* simpler:
 - A subalgorithm that (re)computes the height of a node
 - A subalgorithm that computes the balance factor of a node
 - Four subalgorithms for the four rotation types (we will implement only one)
- And we will assume that we have a function, *createNode* that creates and returns a node containing a given information (left and right are NIL, height is 0).

AVL Tree - height of a node

subalgorithm `recomputeHeight(node)` **is:**

//pre: node is an \uparrow AVLNode. All descendants of node have their height (h) set

//to the correct value

//post: if node \neq NIL, h of node is set

AVL Tree - height of a node

subalgorithm `recomputeHeight(node)` **is:**

//pre: node is an \uparrow AVLNode. All descendants of node have their height (h) set

//to the correct value

//post: if node \neq NIL, h of node is set

if `node \neq NIL` **then**

if `[node].left = NIL and [node].right = NIL` **then**

`[node].h \leftarrow 0`

else if `[node].left = NIL` **then**

`[node].h \leftarrow [[node].right].h + 1`

else if `[node].right = NIL` **then**

`[node].h \leftarrow [[node].left].h + 1`

else

`[node].h \leftarrow max ([[node].left].h, [[node].right].h) + 1`

end-if

end-if

end-subalgorithm

- Complexity: $\Theta(1)$

AVL Tree - balance factor of a node

function balanceFactor(node) **is:**

//pre: node is an \uparrow AVLNode. All descendants of node have their height (h) set

//to the correct value

//post: returns the balance factor of the node

AVL Tree - balance factor of a node

function balanceFactor(node) **is:**

//pre: node is an \uparrow AVLNode. All descendants of node have their height (h) set

//to the correct value

//post: returns the balance factor of the node

if [node].left = NIL **and** [node].right = NIL **then**

 balanceFactor \leftarrow 0

else if [node].left = NIL **then**

 balanceFactor \leftarrow -1 - [[node].right].h *//height of empty tree is -1*

else if [node].right = NIL **then**

 balanceFactor \leftarrow [[node].left].h + 1

else

 balanceFactor \leftarrow [[node].left].h - [[node].right].h

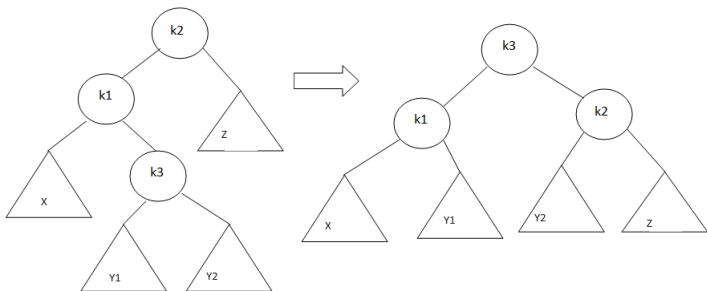
end-if

end-subalgorithm

- Complexity: $\Theta(1)$

AVL Tree - rotations

- Out of the four rotations, we will only implement one, double right rotation (DRR).
- The other three rotations can be implemented similarly (RLR, SRR, SLR).



function DRR(node) **is:** *//pre: node is an \uparrow AVLNode on which we perform the double right rotation*

//post: DRR returns the new root after the rotation

k2 \leftarrow node

k1 \leftarrow [node].left

k3 \leftarrow [k1].right

k3left \leftarrow [k3].left

k3right \leftarrow [k3].right

function DRR(node) **is:** *//pre: node is an \uparrow AVLNode on which we perform the double right rotation*

//post: DRR returns the new root after the rotation

k2 \leftarrow node

k1 \leftarrow [node].left

k3 \leftarrow [k1].right

k3left \leftarrow [k3].left

k3right \leftarrow [k3].right

//reset the links

newRoot \leftarrow k3

[newRoot].left \leftarrow k1

[newRoot].right \leftarrow k2

[k1].right \leftarrow k3left

[k2].left \leftarrow k3right

//continued on the next slide

```
//recompute the heights of the modified nodes  
recomputeHeight(k1)  
recomputeHeight(k2)  
recomputeHeight(newRoot)  
DRR  $\leftarrow$  newRoot  
end-function
```

- Complexity: $\Theta(1)$

AVL Tree - insert

function insertRec(node, elem) **is**

//pre: node is a \uparrow AVLNode, elem is the value we insert in the (sub)tree that

//has node as root

//post: insertRec returns the new root of the (sub)tree after the insertion

if node = NIL **then**

 insertRec \leftarrow createNode(elem)

else if elem \leq [node].info **then**

 [node].left \leftarrow insertRec([node].left, elem)

else

 [node].right \leftarrow insertRec([node].right, elem)

end-if

//continued on the next slide...

AVL Tree - insert

```
recomputeHeight(node)
balance  $\leftarrow$  getBalanceFactor(node)
if balance = -2 then
```

AVL Tree - insert

```
recomputeHeight(node)
balance  $\leftarrow$  getBalanceFactor(node)
if balance = -2 then
  //right subtree has larger height, we will need a rotation to the LEFT
  rightBalance  $\leftarrow$  getBalanceFactor([node].right)
  if rightBalance < 0 then
```

AVL Tree - insert

```
recomputeHeight(node)
balance ← getBalanceFactor(node)
if balance = -2 then
  //right subtree has larger height, we will need a rotation to the LEFT
  rightBalance ← getBalanceFactor([node].right)
  if rightBalance < 0 then
    //the right subtree of the right subtree has larger height, SRL
    node ← SRL(node)
  else
    node ← DRL(node)
  end-if
//continued on the next slide...
```


else if balance = 2 **then**

//left subtree has larger height, we will need a RIGHT rotation

leftBalance \leftarrow getBalanceFactor([node].left)

if leftBalance > 0 **then**

AVL Tree - insert

```
else if balance = 2 then  
  //left subtree has larger height, we will need a RIGHT rotation  
  leftBalance  $\leftarrow$  getBalanceFactor([node].left)  
  if leftBalance > 0 then  
    //the left subtree of the left subtree has larger height, SRR  
    node  $\leftarrow$  SRR(node)  
  else  
    node  $\leftarrow$  DRR(node)  
  end-if  
end-if  
insertRec  $\leftarrow$  node  
end-function
```

- Complexity of the *insertRec* algorithm: $O(\log_2 n)$
- Since *insertRec* receives as parameter a pointer to a node, we need a wrapper function to do the first call on the root

subalgorithm insert(tree, elem) **is**

//pre: tree is an AVL Tree, elem is the element to be inserted

//post: elem was inserted to tree

tree.root \leftarrow insertRec(tree.root, elem)

end-subalgorithm

- remove subalgorithm can be implemented similarly (start from the remove from BST and add the rotation part).

- Today we have talked about
 - AVL trees
- Extra reading - sample exam questions (solutions will be discussed in Lecture 14)