DATA STRUCTURES AND ALGORITHMS

LECTURE 13

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2020 - 2021

In Lecture 12...

- Binary Trees
- Binary Search Trees

Today

- AVL Trees
- Misc

- Starting from an initially empty Binary Search Tree and the relation $\leq$, insert into it, in the given order, the following values: 10, 20, 5, 7, 15, 5, 30, 3, 5, 5, 1, 9, 29, 2.

Binary Search Tree with duplicate values

- Starting from an initially empty Binary Search Tree and the relation $\leq$, 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?

Binary Search Tree with duplicate values

- Starting from an initially empty Binary Search Tree and the relation $\leq$, 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?
- Remove 3 (show both options)

Binary Search Tree with duplicate values

- Starting from an initially empty Binary Search Tree and the relation $\leq$, 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?
- Remove 3 (show both options)
- How would you count now how many times the value 5 is in the tree now?

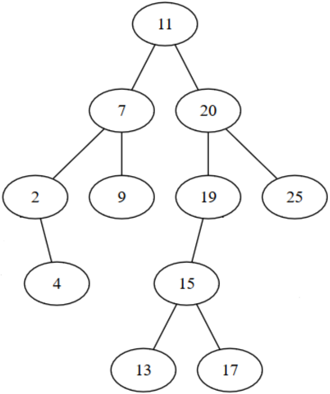
Balanced Binary Search Trees

- 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 $\Theta(log_2 n)$

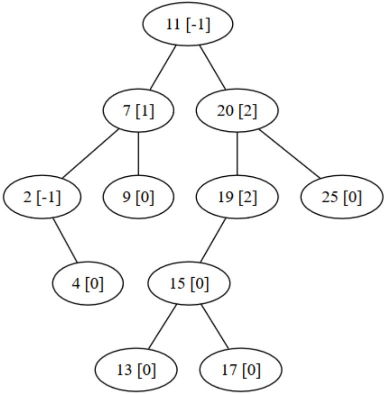
- Specific operations for binary trees run in $O(h)$ time, which can be $\theta(n)$ in worst case
- Best case is a balanced tree, where height of the tree is $\Theta(log_2 n)$
- To reduce the complexity of algorithms, we want to keep the tree balanced. In order to do this, we want every node to be balanced.
- 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

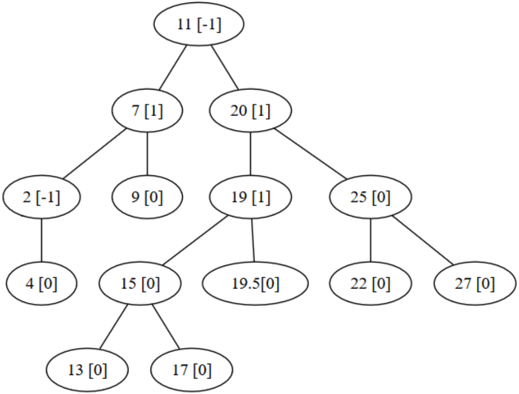
AVL Trees



- 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.

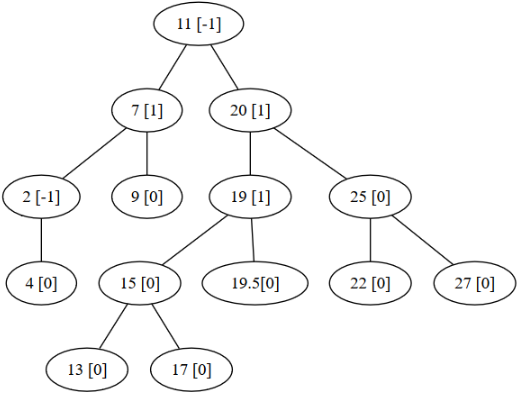


- This is an AVL tree.

AVL Trees - rotations

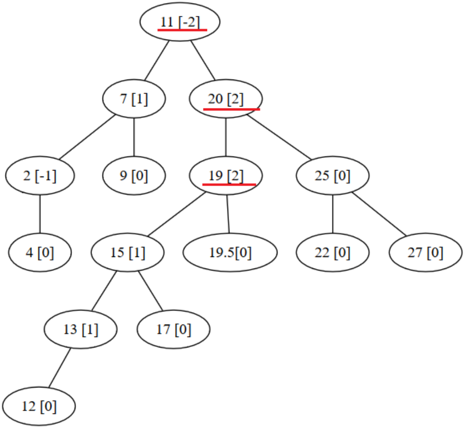
- Adding or removing a node might result in a binary tree that violates the AVL tree property.
- In such cases, the property has to be restored and only after the property holds again is the operation (add or remove) considered finished.
- The AVL tree property can be restored with operations called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

- 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. When we find a node that does not respect the AVL tree property, we perform a suitable *rotation* to rebalance the (sub)tree.



- What if we insert element 12?

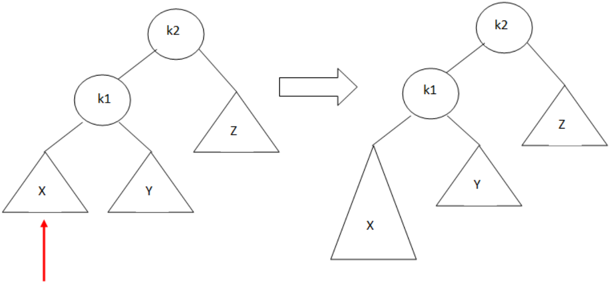
AVL Trees - rotations



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

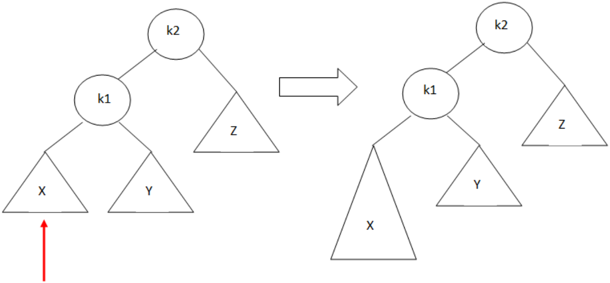
- Assume that at a given point α is the node that needs to be rebalanced.
- Since α was balanced before the insertion, and is not after the insertion, we can identify four cases in which a violation might occur:
  - Insertion into the left subtree of the left child of α
  - Insertion into the right subtree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



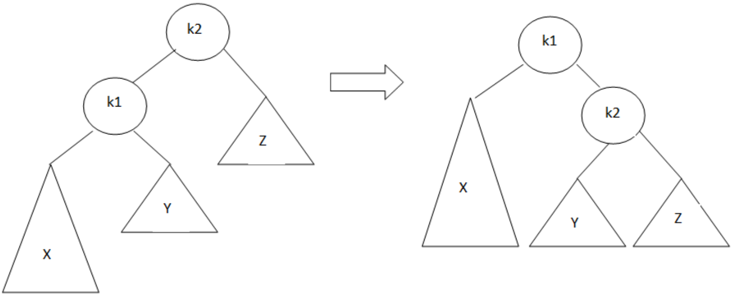
- Obs: X, Y and Z represent subtrees with the same height.

AVL Trees - rotations - case 1

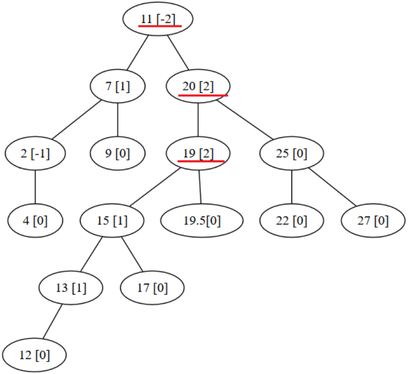


- Obs: X, Y and Z represent subtrees with the same height.
- Solution: single rotation to right

AVL Trees - rotation - Single Rotation to Right

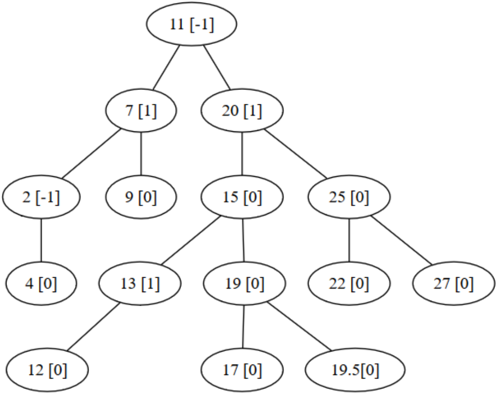


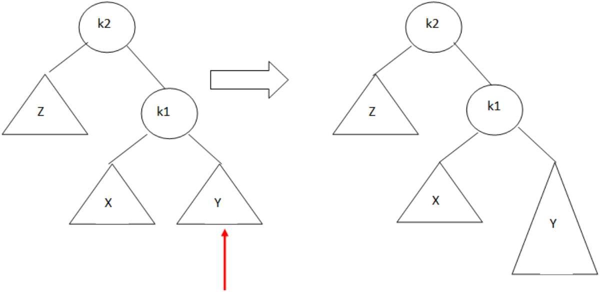
AVL Trees - rotations - case 1 example



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

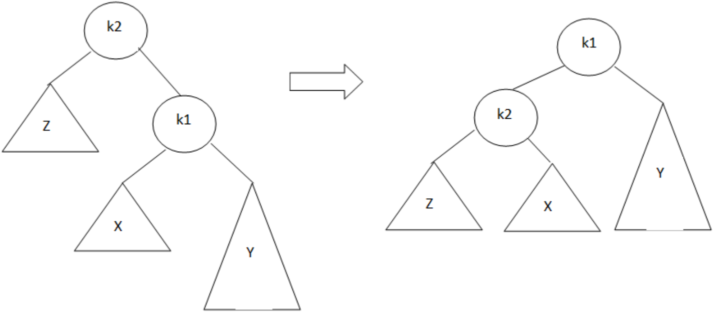
AVL Trees - rotation - case 1 example

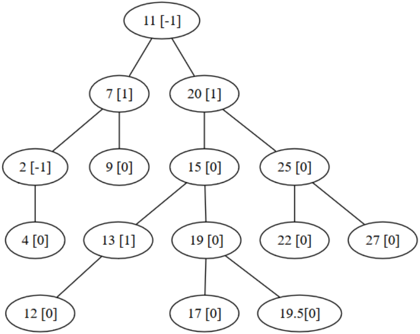




- Solution: **single rotation to left**

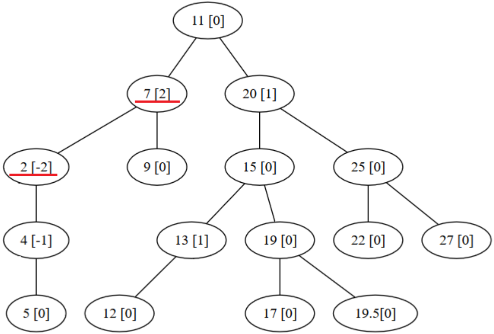
AVL Trees - rotation - Single Rotation to Left





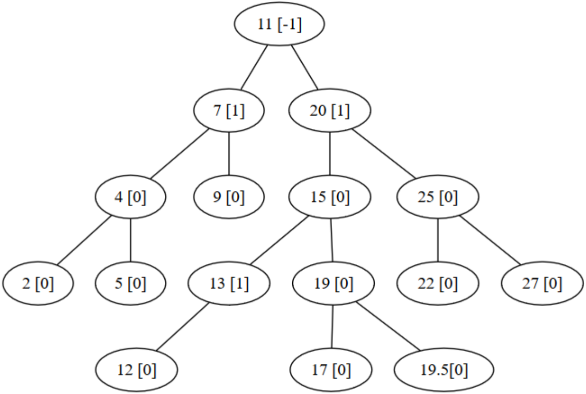
- Insert value 5

AVL Trees - rotations - case 4 example



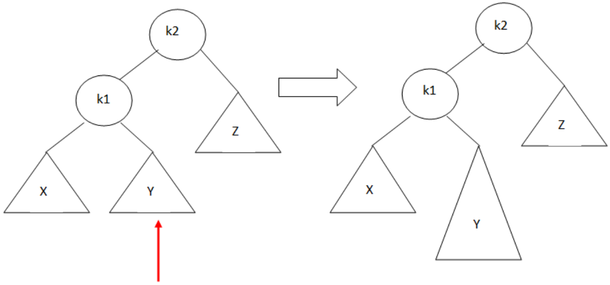
- Node 2 is imbalanced, because we inserted a new node (5) to the right subtree of the right child
- Solution: **single rotation to left**

AVL Trees - rotation - case 4 example



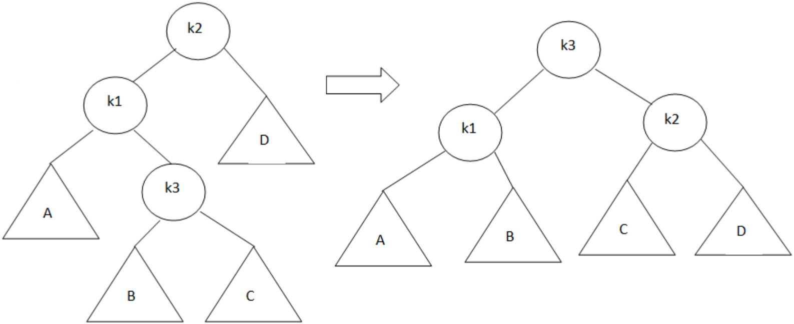
- After the rotation

AVL Trees - rotations - case 2

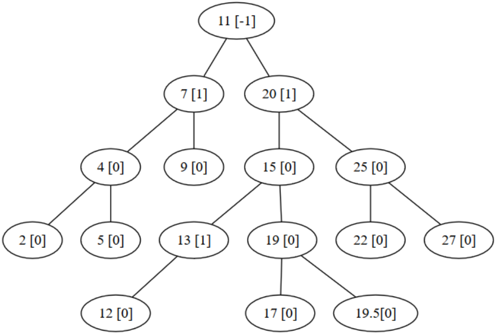


- Solution: **Double rotation to right**

AVL Trees - rotation - Double Rotation to Right

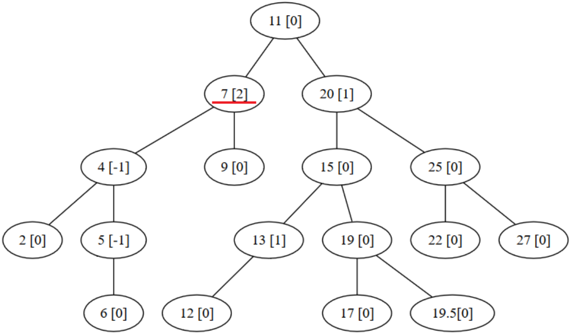


AVL Trees - rotations - case 2 example



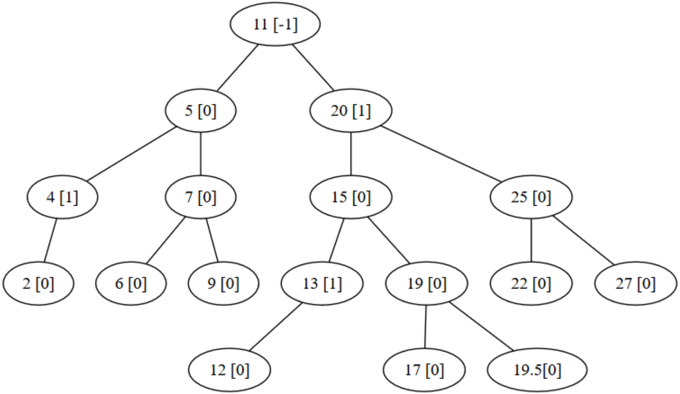
- Insert value 6

AVL Trees - rotations - case 2 example



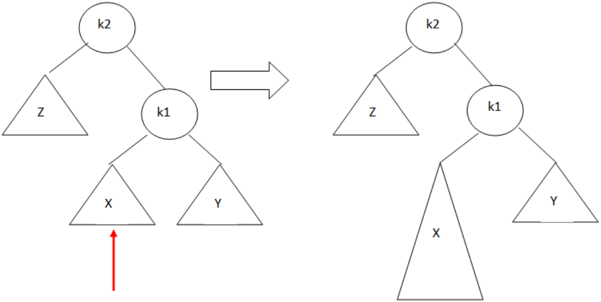
- Node 7 is imbalanced, because we inserted a new node (6) to the right subtree of the left child
- Solution: **double rotation to right**

AVL Trees - rotation - case 2 example

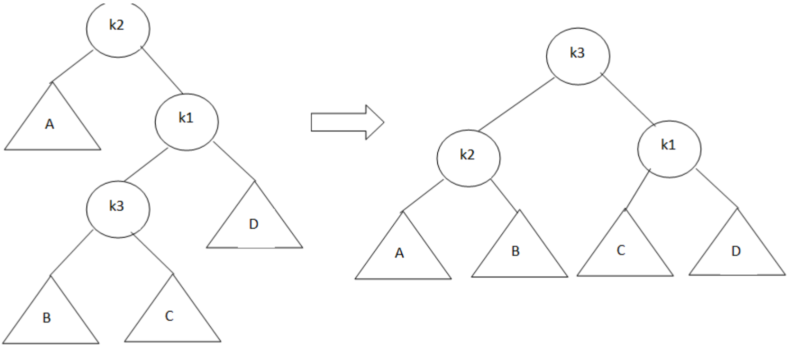


- After the rotation

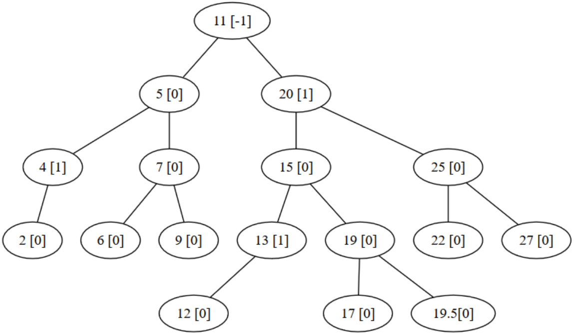
AVL Trees - rotations - case 3



- Solution: **Double rotation to left**

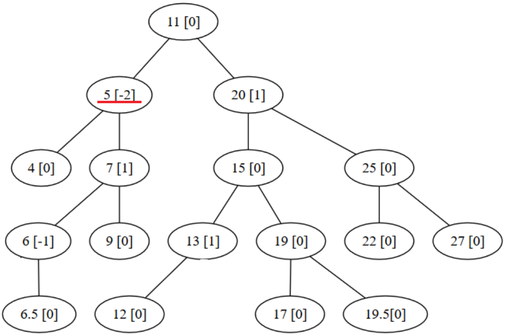

k2
A
k1
k3
D
B
C
k3
k2
k1
A
B
C
D

AVL Trees - rotations - case 3 example



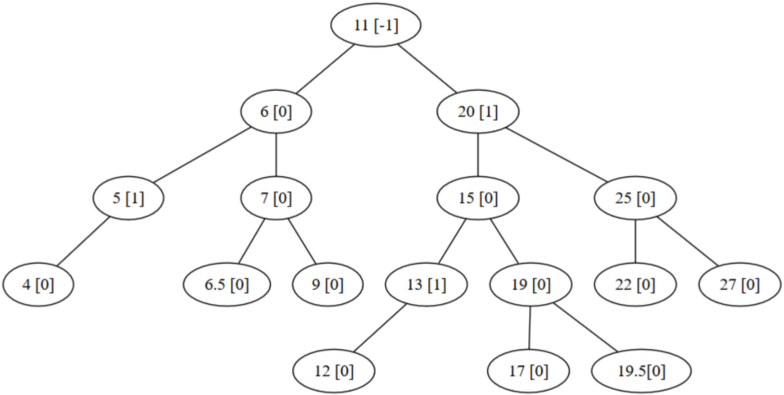
- Remove node with value 2 and insert value 6.5

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

AVL rotations example I

- Start with an empty AVL tree
- Insert 2

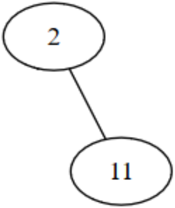
AVL rotations example II

- 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

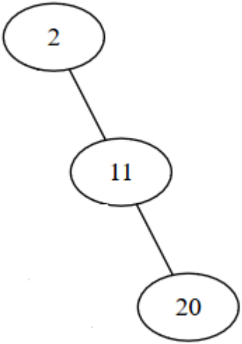


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example V

- No rotation is needed
- Insert 20

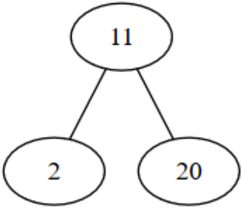
AVL rotations example VI



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

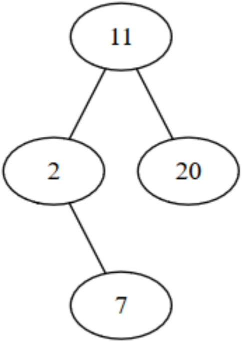
AVL rotations example VII

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



- Insert 7

AVL rotations example VIII

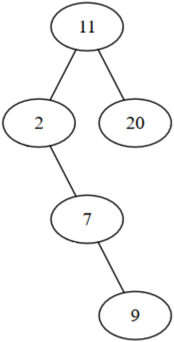


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example IX

- No rotation is needed
- Insert 9

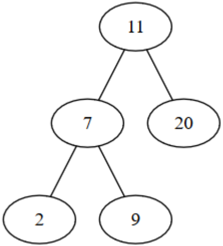
AVL rotations example X



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

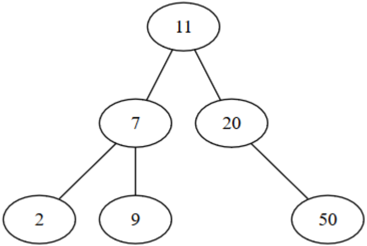
AVL rotations example XI

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



- Insert 50

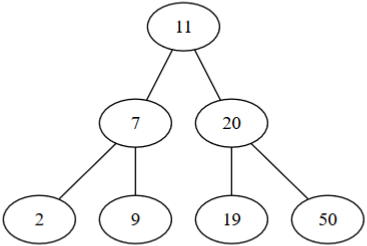
AVL rotations example XII



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

- No rotation is needed
- Insert 19

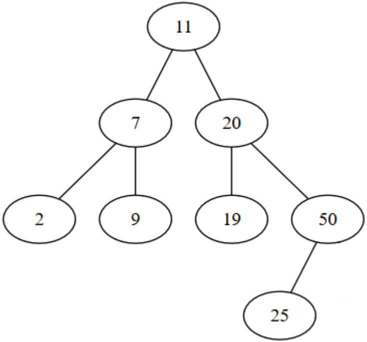
AVL rotations example XIV



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

- No rotation is needed
- Insert 25

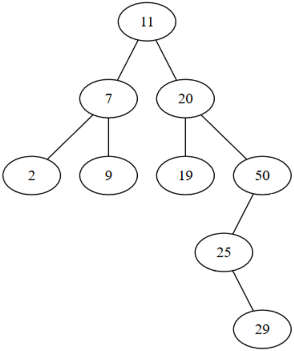
AVL rotations example XVI



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

- No rotation is needed
- Insert 29

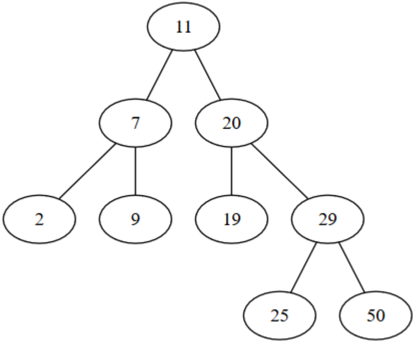
AVL rotations example XVIII



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

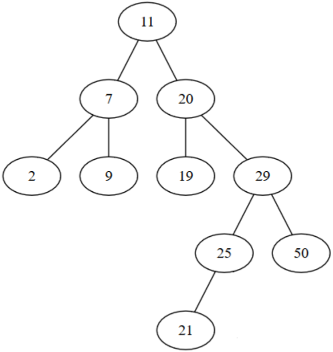
AVL rotations example XIX

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



- Insert 21

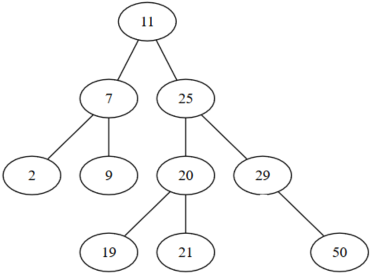
AVL rotations example XX



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

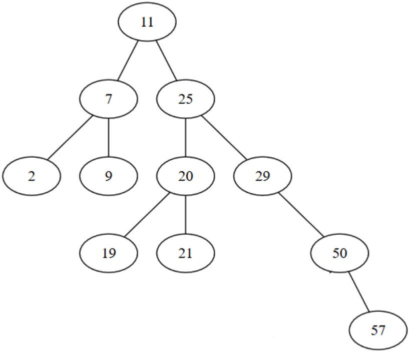
AVL rotations example XXI

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



- Insert 57

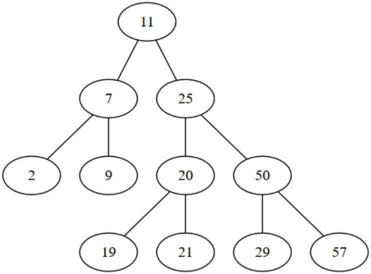
AVL rotations example XXII



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

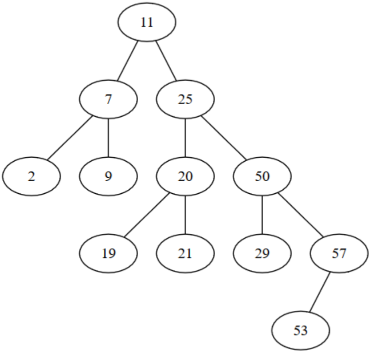
AVL rotations example XXIII

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



- Insert 53

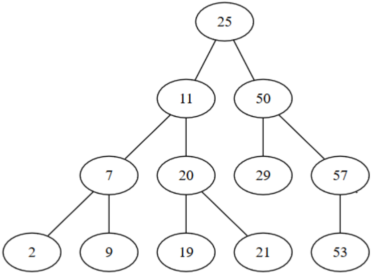
AVL rotations example XXIV



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

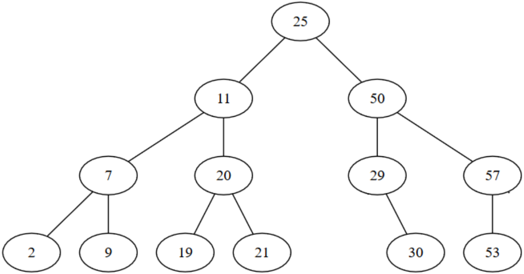
AVL rotations example XXV

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



- Insert 30

AVL rotations example XXVI

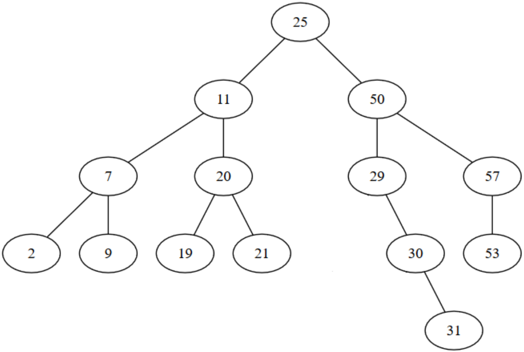


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XXVII

- No rotation is needed
- Insert 31

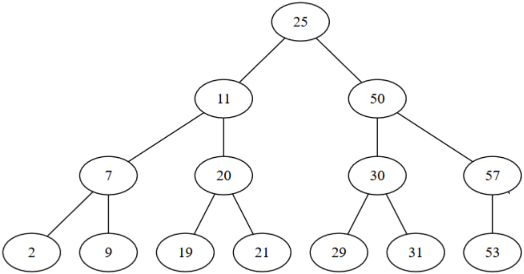
AVL rotations example XXVIII



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

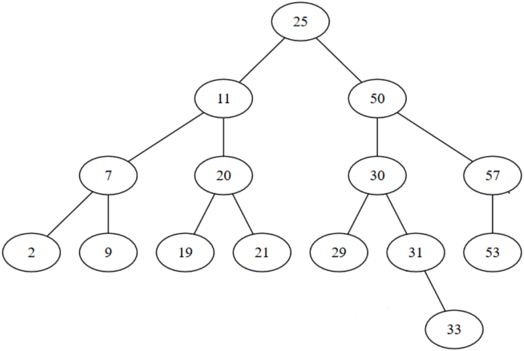
AVL rotations example XXIX

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



- Insert 33

AVL rotations example XXX

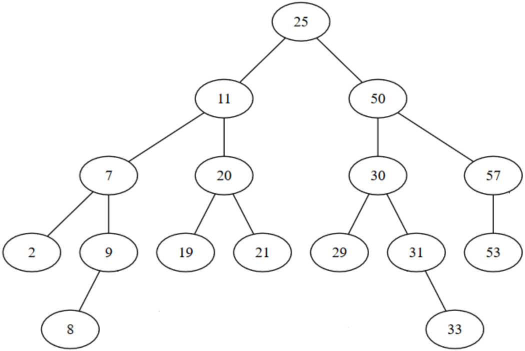


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XXXI

- No rotation is needed
- Insert 8

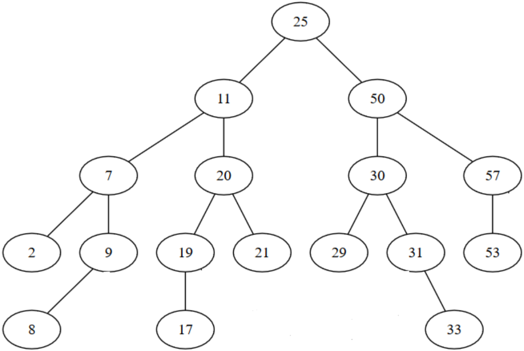
AVL rotations example XXXII



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

- No rotation is needed
- Insert 17

AVL rotations example XXXIV

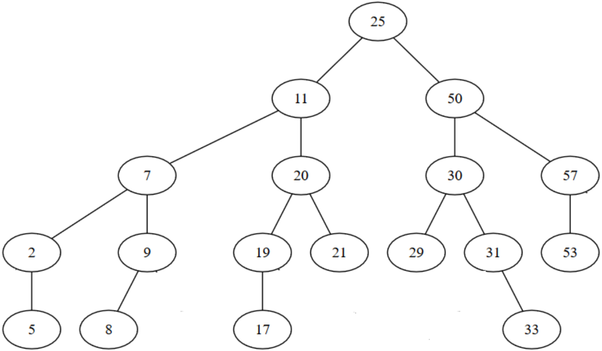


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XXXV

- No rotation is needed
- Insert 5

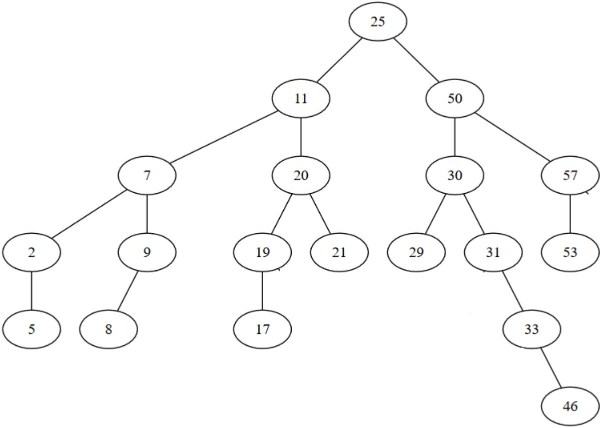
AVL rotations example XXXVI



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

- No rotation is needed
- Insert 46

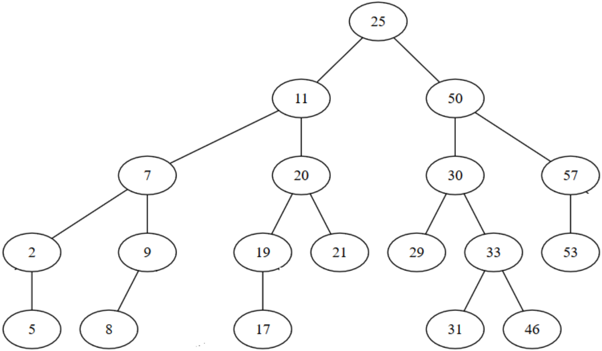
AVL rotations example XXXVIII



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

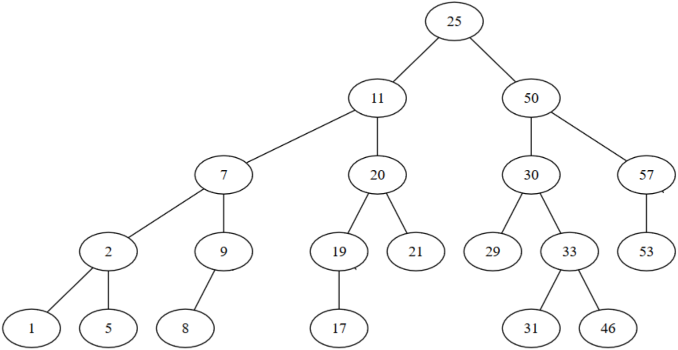
AVL rotations example XXXIX

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



- Insert 1

AVL rotations example XL



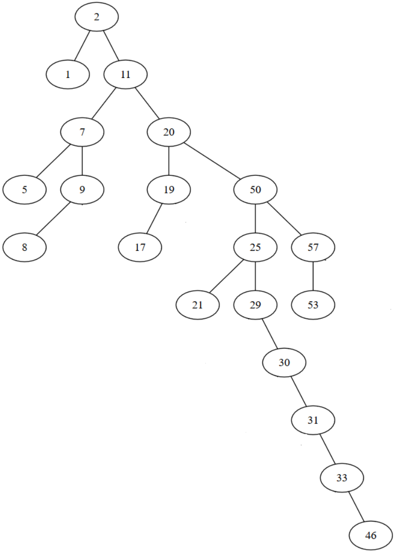
- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XLI

- No rotation is needed

Comparison to BST

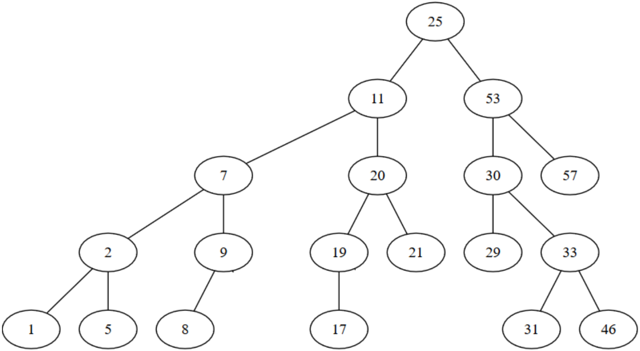
- If, instead of using an AVL tree, we used a binary search tree, after the insertions the tree would have been:


2
1
11
7
20
5
9
19
50
8
17
25
57
21
29
53
30
31
33
46

Example of remove I

- Remove 50

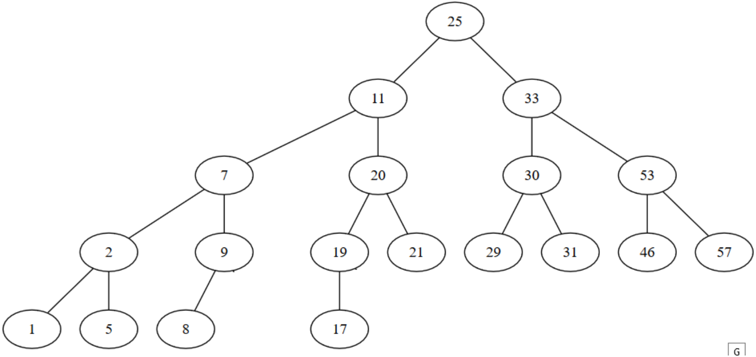
Example of remove II



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

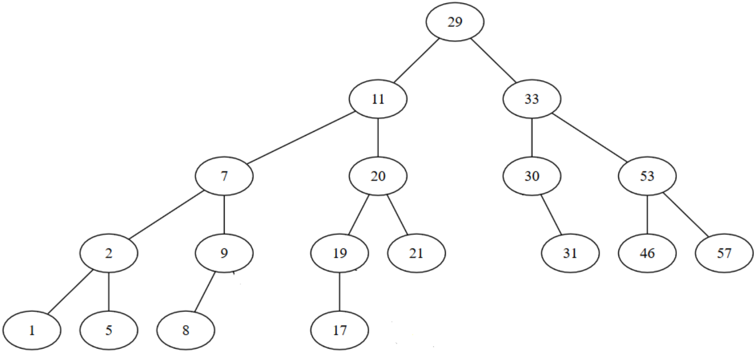
Example of remove III

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



- Remove 25

Example of remove IV

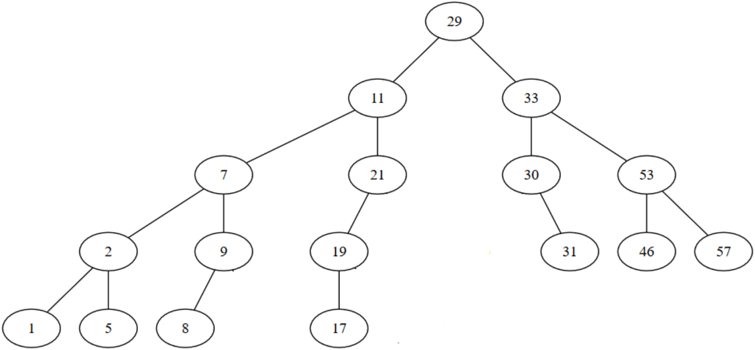


- Do we need a rotation?
- If yes, on which node and what type of rotation?

Example of remove V

- No rotation is needed
- Remove 20

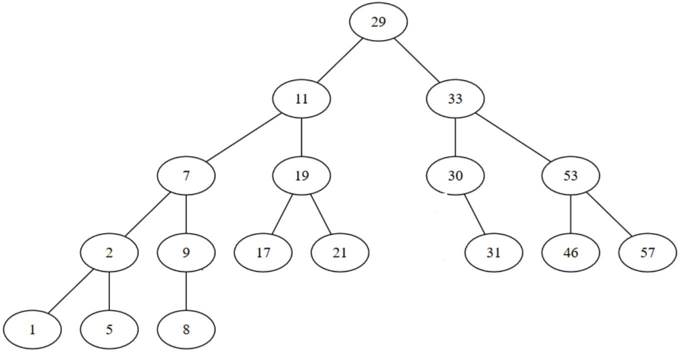
Example of remove VI



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

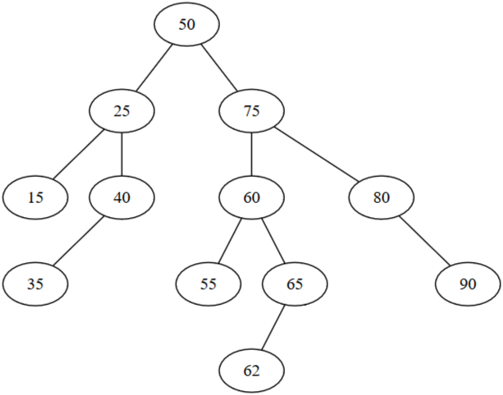
Example of remove VII

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



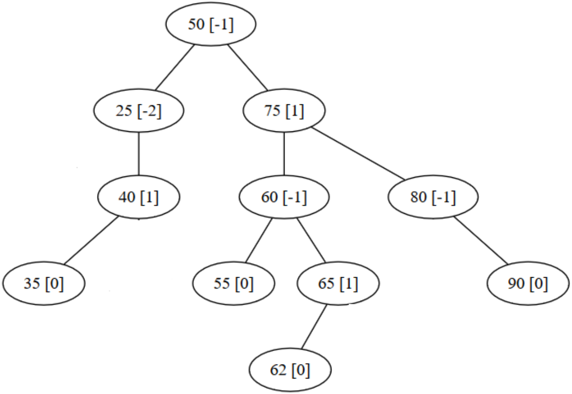
Rotations for remove

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

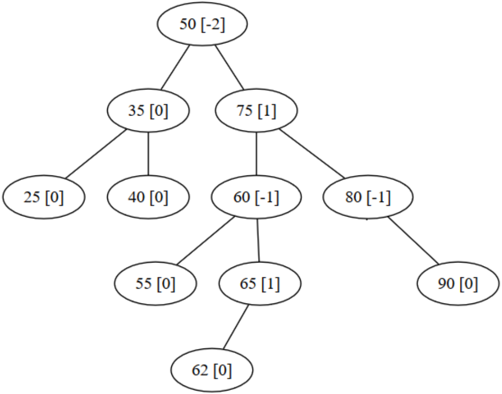


- Remove value 15

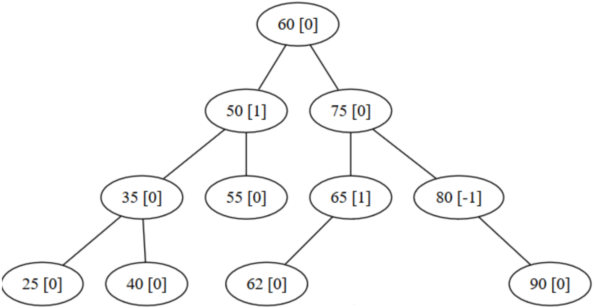
- After remove:



- After the rotation



- After the second rotation



- What structures do we need for an AVL Tree?

- What structures do we need for an AVL Tree?

AVLNode:
  info: TComp *//information from the node*
  left: ↑ AVLNode *//address of left child*
  right: ↑ AVLNode *//address of right child*
  h: Integer *//height of the node*

AVLTree:
  root: ↑ AVLNode *//root of the tree*

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 ↑ AVLNode. All descendants of node have their height (h) set
//to the correct value
//post: if node ≠ NIL, h of node is set
  if node ≠ NIL then
    if [node].left = NIL and [node].right = NIL then
      [node].h ← 0
    else if [node].left = NIL then
      [node].h ← [[node].right].h + 1
    else if [node].right = NIL then
      [node].h ← [[node].left].h + 1
    else
      [node].h ← 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

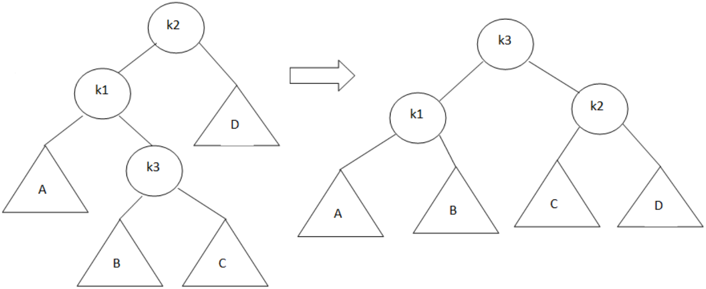
AVL Tree - balance factor of a node

function balanceFactor(node) **is:**
//pre: node is an ↑ 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 ← 0
  else if [node].left = NIL **then**
    balanceFactor ← -1 - [[node].right].h *//height of empty tree is -1*
  else if [node].right = NIL **then**
    balanceFactor ← [[node].left].h + 1
  else
    balanceFactor ← [[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).



AVL Tree - DRR

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 ← node
k1 ← [node].left
k3 ← [k1].right
k3left ← [k3].left
k3right ← [k3].right
```

AVL Tree - DRR

function DRR(node) **is:** *//pre: node is an ↑ AVLNode on which we perform the double right rotation*
//post: DRR returns the new root after the rotation

```
    k2 ← node
    k1 ← [node].left
    k3 ← [k1].right
    k3left ← [k3].left
    k3right ← [k3].right
    //reset the links
    newRoot ← k3
    [newRoot].left ← k1
    [newRoot].right ← k2
    [k1].right ← k3left
    [k2].left ← k3right
```

//continued on the next slide

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

- Complexity: $\Theta(1)$

AVL Tree - insert

```
function insertRec(node, elem) is
//pre: node is a ↑ 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 ← createNode(elem)
  else ifelem ≤ [node].info then
     [node].left ← insertRec([node].left, elem)
  else
    [node].right ← 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 ← 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
```

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...
```

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**

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

AVL Tree - insert

- 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).

Huffman coding

- The *Huffman coding* can be used to encode characters (from an alphabet) using variable length codes.
- In order to reduce the total number of bits needed to encode a message, characters that appear more frequently have shorter codes.
- Since we use variable length code for each character, *no code can be the prefix of any other code* (if we encode letter E with 01 and letter X with 010011, during decoding, when we find a 01, we will not know whether it is E or the beginning of X).

Huffman coding

- When building the Huffman encoding for a message, we first have to compute the frequency of every character from the message, because we are going to define the codes based on the frequencies.

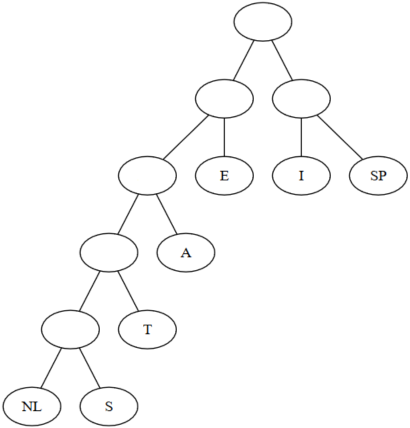
- Assume that we have a message with the following letters and frequencies

Character	a	e	i	s	t	space	newline
Frequency	10	15	12	3	4	13	1

Huffman coding

- For defining the Huffman code a binary tree is build in the following way:
  - Start with trees containing only a root node, one for every character. Each tree has a weight, which is frequency of the character.
  - Get the two trees with the least weight (if there is a tie, choose randomly), combine them into one tree which has as weight the sum of the two weights.
  - Repeat until we have only one tree.

Huffman coding



Huffman coding

- Code for each character can be read from the tree in the following way: start from the root and go towards the corresponding leaf node. Every time we go left add the bit 0 to encoding and when we go right add bit 1.
- Code for the characters:
  - NL - 00000
  - S - 00001
  - T - 0001
  - A - 001
  - E - 01
  - I - 10
  - SP - 11
- In order to encode a message, just replace each character with the corresponding code

Huffman coding

- Assume we have the following code and we want to decode it: 0110110001000100111001000000
- We do not know where the code of each character ends, but we can use the previously built tree to decode it.
- Start parsing the code and iterate through the tree in the following way:
  - Start from the root
  - If the current bit from the code is 0 go to the left child, otherwise go to the right child
  - If we are at a leaf node we have decoded a character and have to start over from the root
- The decoded message: E I SP T T A SP I E NL