

DATA STRUCTURES AND ALGORITHMS

LECTURE 13

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2019 - 2020

In Lecture 12...

- Binary Trees
- Binary Search Trees

- AVL Trees
- Problems/Applications

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.

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?

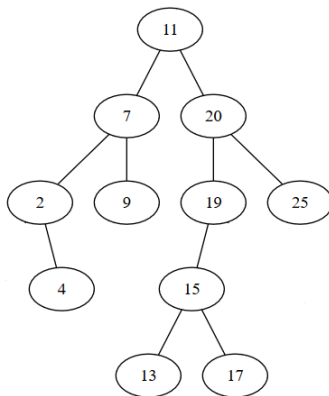
Balanced Binary Search Trees

- 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 $\Theta(\log_2 n)$

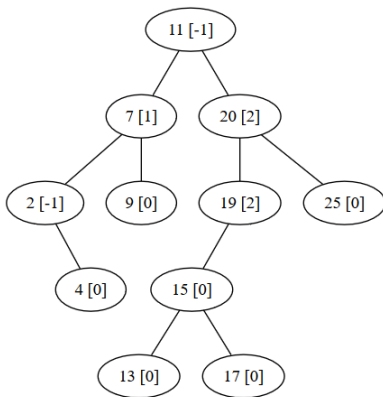
Balanced Binary Search Trees

- 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.

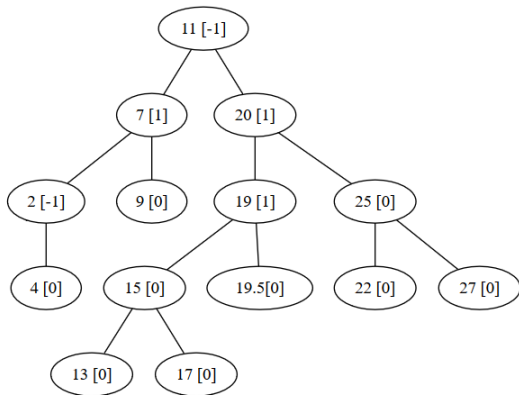
- Definition: An AVL (Adelson-Velskii Landis) tree is a binary tree which satisfies the following property (AVL tree property):
 - If x is a node of the AVL tree:
 - the difference between the height of the left and right subtree of x is 0, 1 or -1 (balancing information)
- Observations:
 - Height of 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.



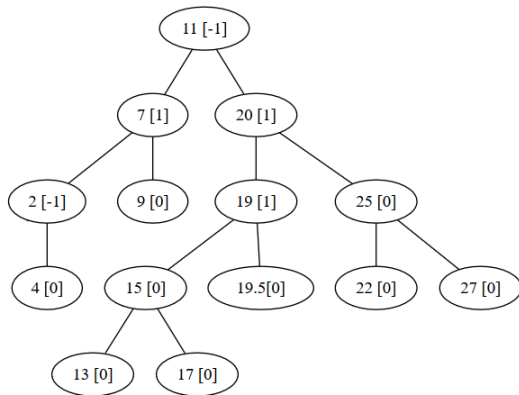
- This is an AVL tree.

- 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

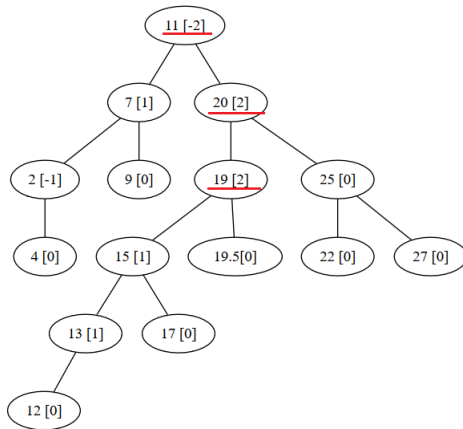
- After an insertion, only the nodes on the path to the modified node can change their height.
- We check the balancing information on the path from the modified node to the root. When we find a node that does not respect the AVL tree property, we perform a suitable *rotation* to rebalance the (sub)tree.

AVL Tress - rotations



- What if we insert element 12?

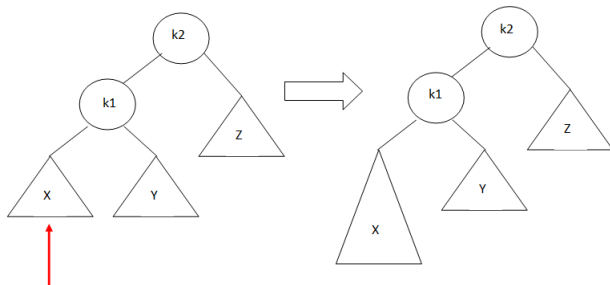
AVL Trees - rotations



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

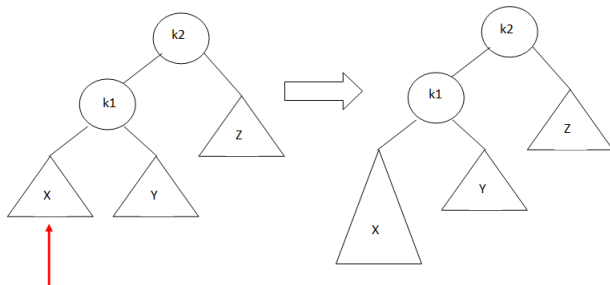
- Assume that at a given point α is the node that needs to be rebalanced.
- Since α was balanced before the insertion, and is not after the insertion, we can identify four cases in which a violation might occur:
 - Insertion into the left subtree of the left child of α
 - Insertion into the right subtree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



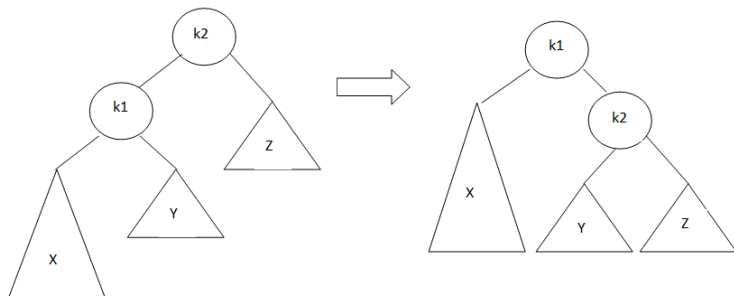
- Obs: X , Y and Z represent subtrees with the same height.

AVL Trees - rotations - case 1

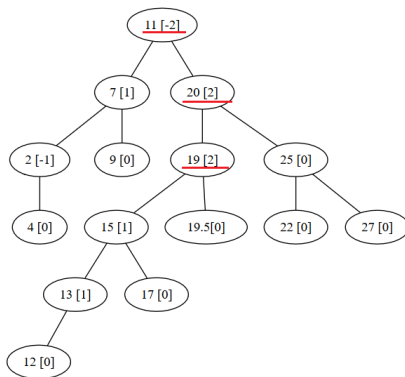


- Obs: X , Y and Z represent subtrees with the same height.
- Solution: single rotation to right

AVL Trees - rotation - Single Rotation to Right

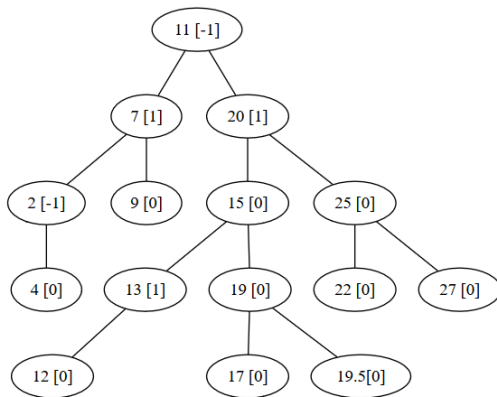


AVL Trees - rotations - case 1 example

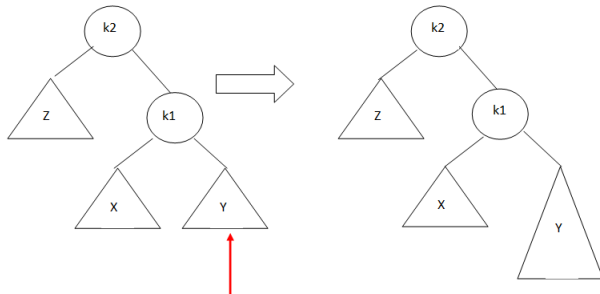


- Node 19 is imbalanced, because we inserted a new node (12) in the left subtree of the left child.
- Solution: **single rotation to right**

AVL Trees - rotation - case 1 example

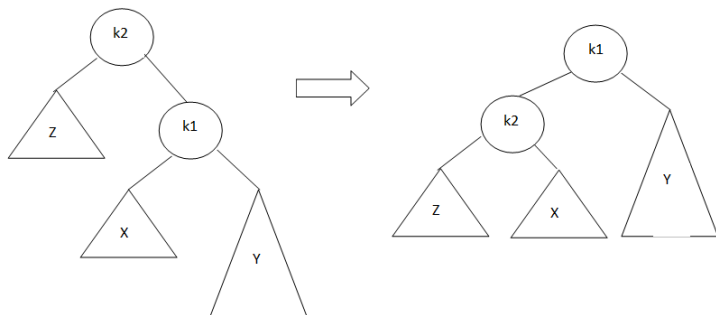


AVL Trees - rotations - case 4

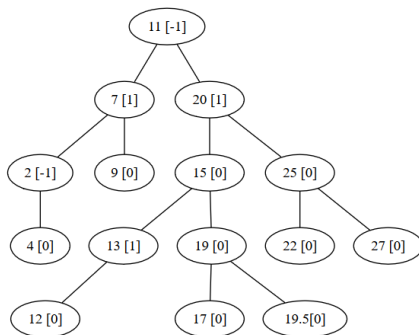


- Solution: **single rotation to left**

AVL Trees - rotation - Single Rotation to Left

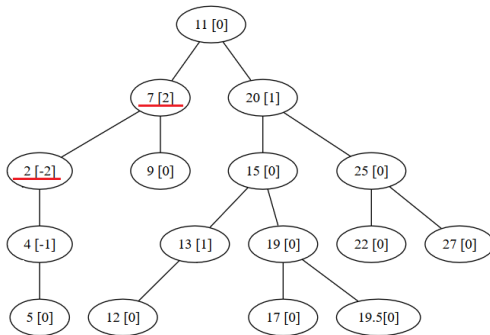


AVL Trees - rotations - case 4 example



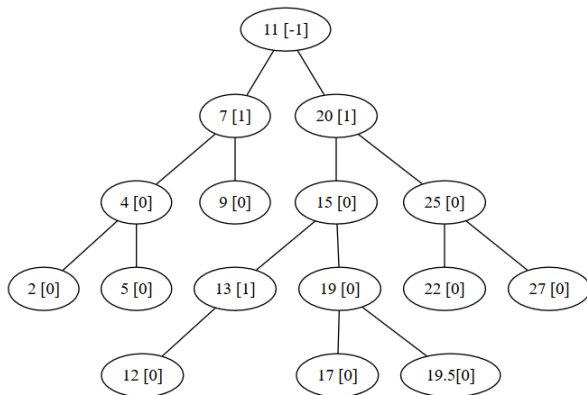
- Insert value 5

AVL Trees - rotations - case 4 example



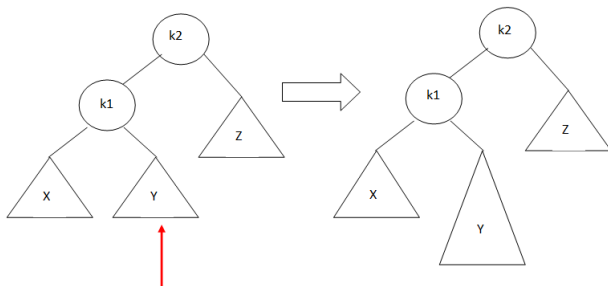
- Node 2 is imbalanced, because we inserted a new node (5) to the right subtree of the right child
- Solution: **single rotation to left**

AVL Trees - rotation - case 4 example



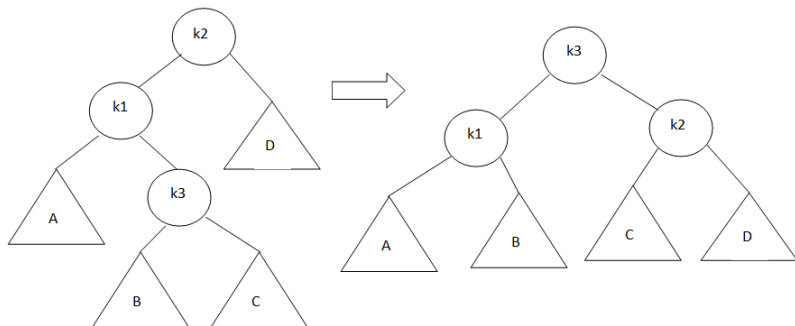
- After the rotation

AVL Trees - rotations - case 2

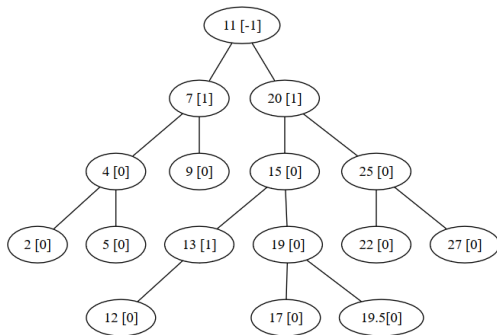


- Solution: **Double rotation to right**

AVL Trees - rotation - Double Rotation to Right

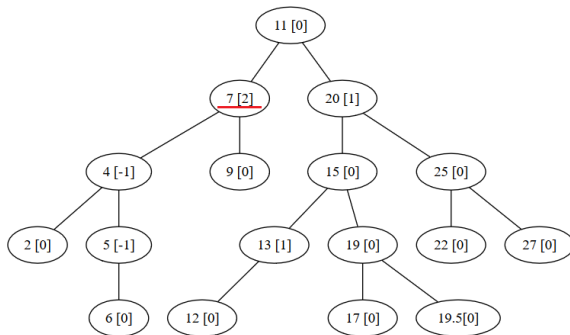


AVL Trees - rotations - case 2 example



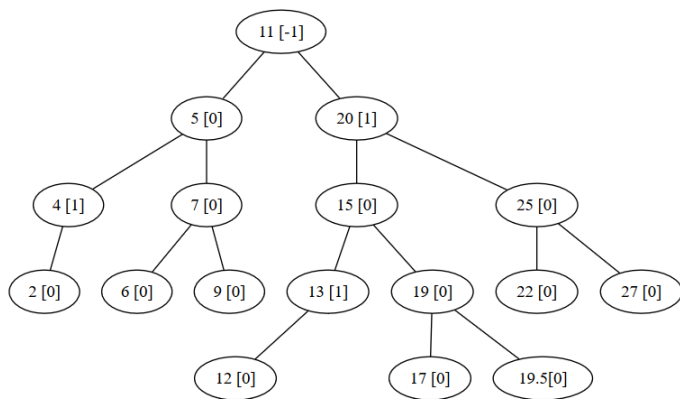
- Insert value 6

AVL Trees - rotations - case 2 example



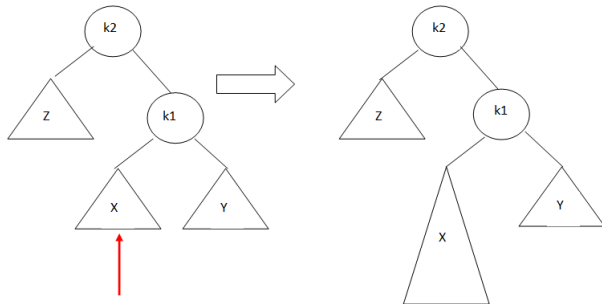
- Node 7 is imbalanced, because we inserted a new node (6) to the right subtree of the left child
- Solution: **double rotation to right**

AVL Trees - rotation - case 2 example



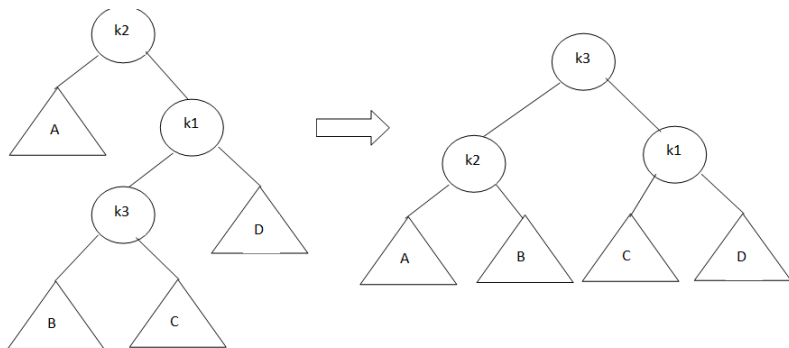
- After the rotation

AVL Trees - rotations - case 3

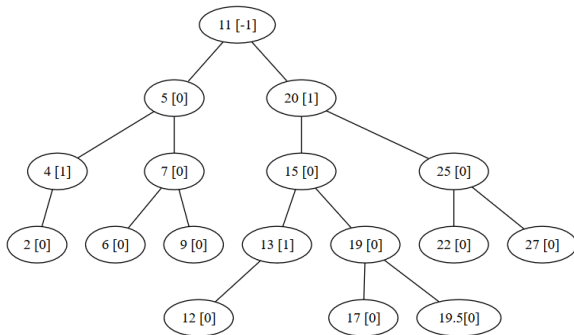


- Solution: **Double rotation to left**

AVL Trees - rotation - Double Rotation to Left

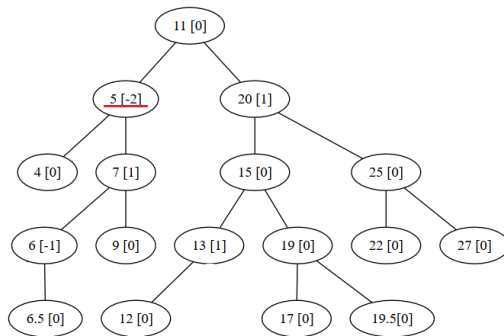


AVL Trees - rotations - case 3 example



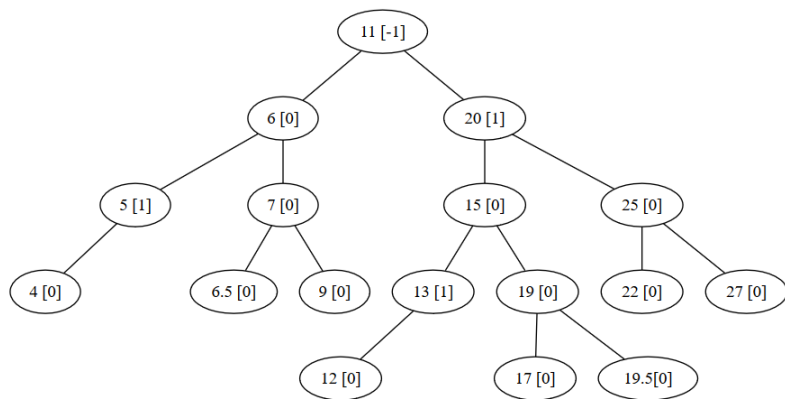
- Remove node with value 2 and insert value 6.5

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

AVL rotations example I

- Start with an empty AVL tree
- Insert 2

AVL rotations example II

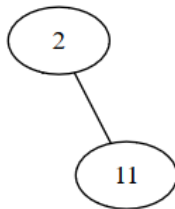


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example III

- No rotation is needed
- Insert 11

AVL rotations example IV

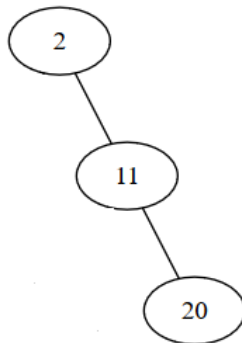


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example V

- No rotation is needed
- Insert 20

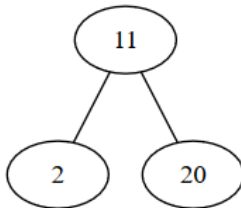
AVL rotations example VI



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

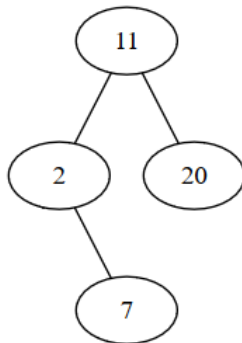
AVL rotations example VII

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



- Insert 7

AVL rotations example VIII

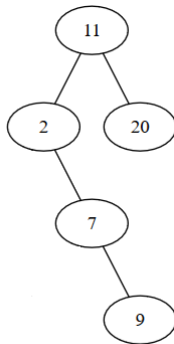


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example IX

- No rotation is needed
- Insert 9

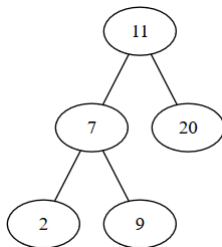
AVL rotations example X



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

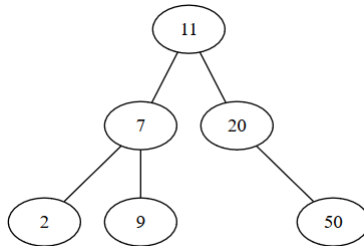
AVL rotations example XI

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



- Insert 50

AVL rotations example XII

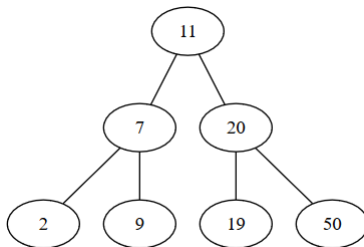


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XIII

- No rotation is needed
- Insert 19

AVL rotations example XIV

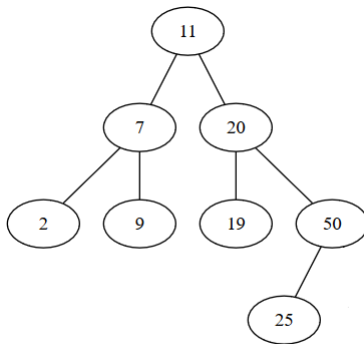


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XV

- No rotation is needed
- Insert 25

AVL rotations example XVI

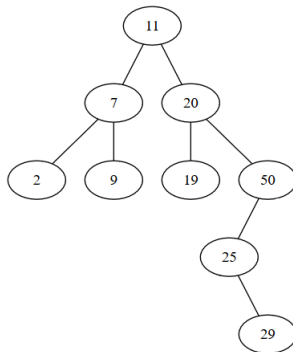


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XVII

- No rotation is needed
- Insert 29

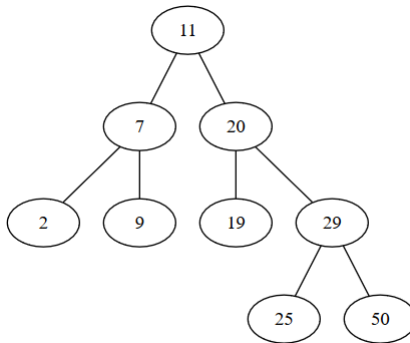
AVL rotations example XVIII



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

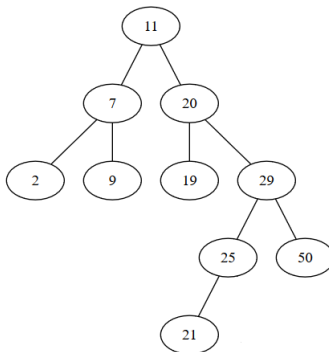
AVL rotations example XIX

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



- Insert 21

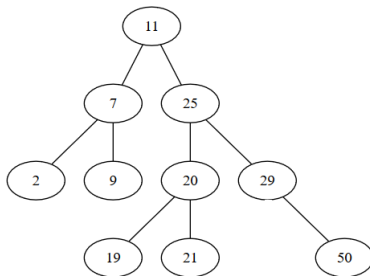
AVL rotations example XX



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

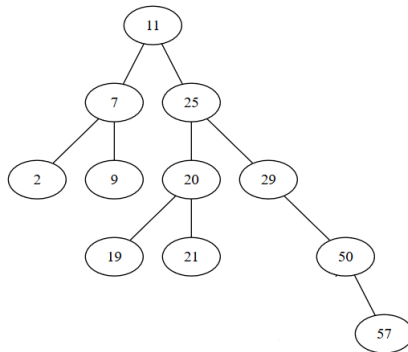
AVL rotations example XXI

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



- Insert 57

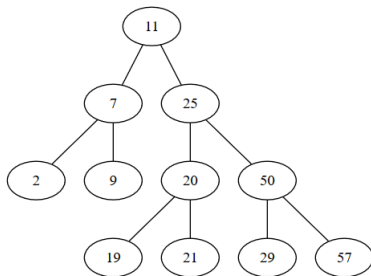
AVL rotations example XXII



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

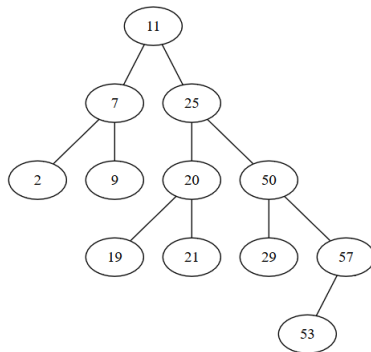
AVL rotations example XXIII

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



- Insert 53

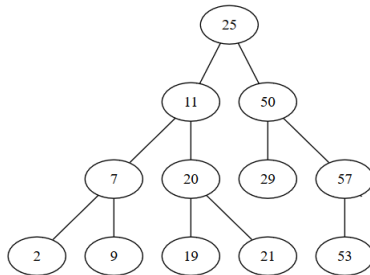
AVL rotations example XXIV



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

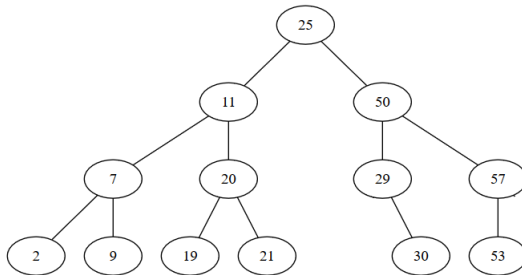
AVL rotations example XXV

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



- Insert 30

AVL rotations example XXVI

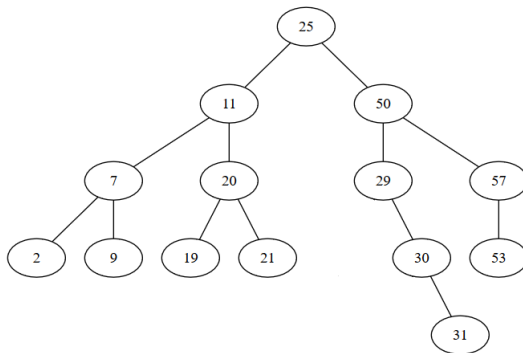


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XXVII

- No rotation is needed
- Insert 31

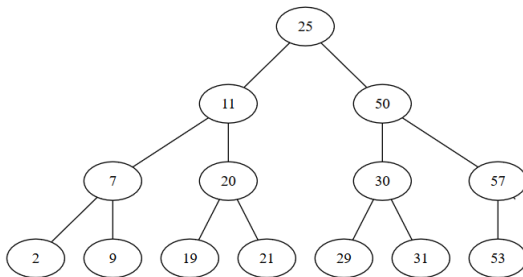
AVL rotations example XXVIII



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

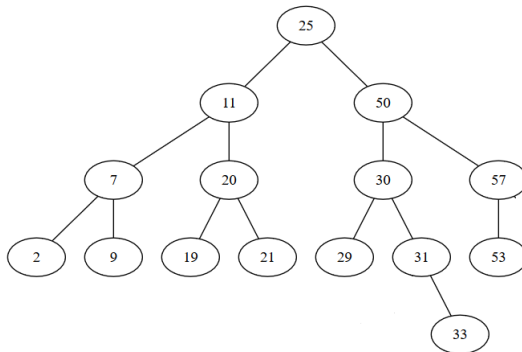
AVL rotations example XXIX

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



- Insert 33

AVL rotations example XXX

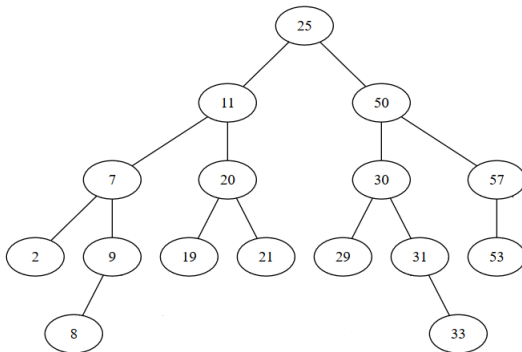


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XXXI

- No rotation is needed
- Insert 8

AVL rotations example XXXII

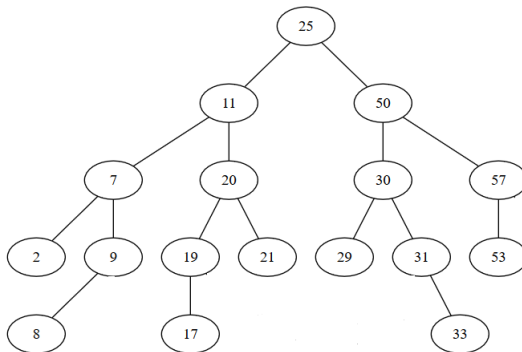


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XXXIII

- No rotation is needed
- Insert 17

AVL rotations example XXXIV

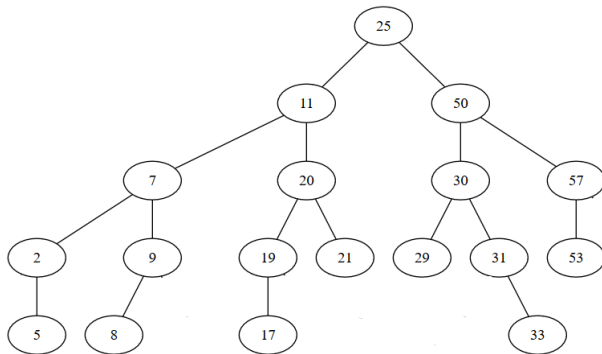


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XXXV

- No rotation is needed
- Insert 5

AVL rotations example XXXVI

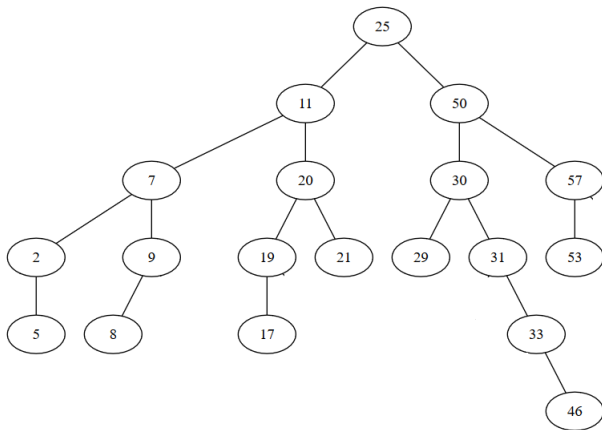


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XXXVII

- No rotation is needed
- Insert 46

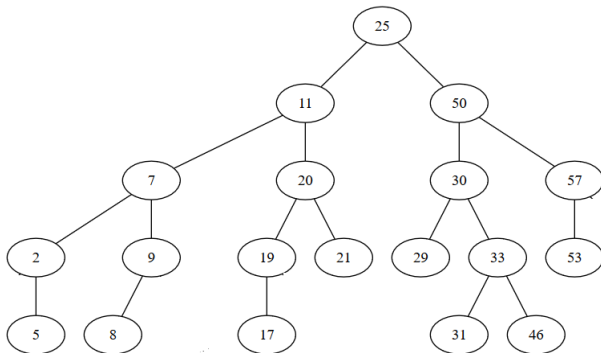
AVL rotations example XXXVIII



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

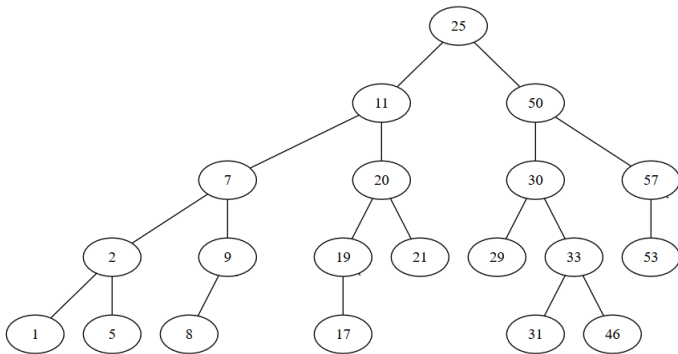
AVL rotations example XXXIX

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



- Insert 1

AVL rotations example XL

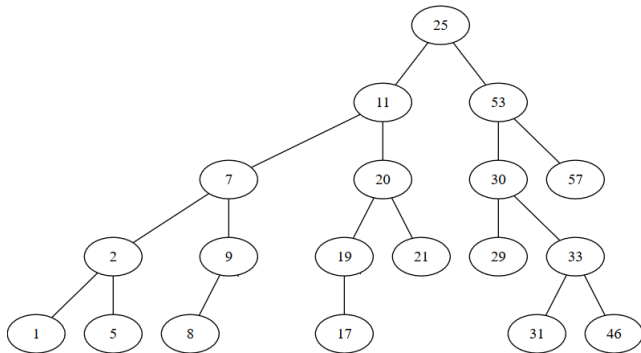


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XLI

- No rotation is needed
- Remove 50

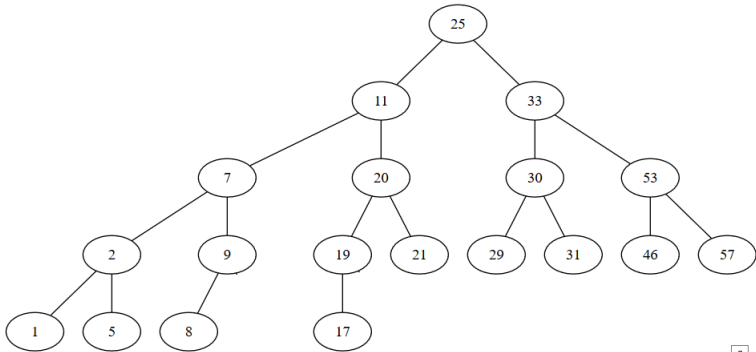
AVL rotations example XLII



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

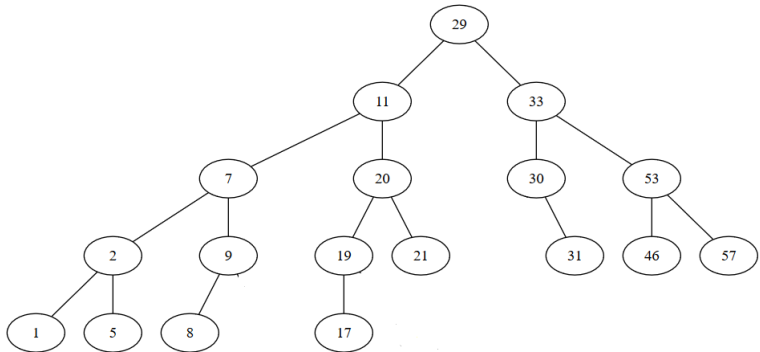
AVL rotations example XLIII

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



- Remove 25

AVL rotations example XLIV

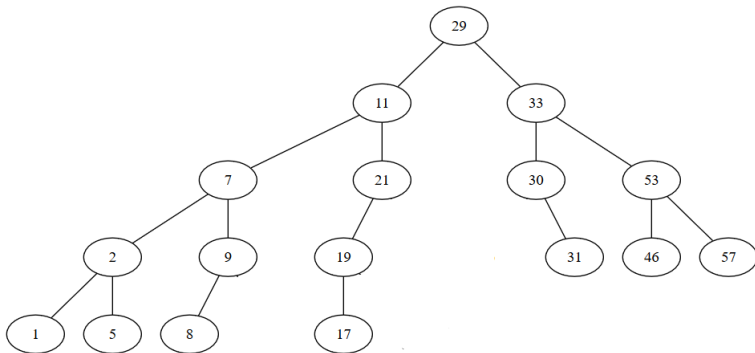


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XLV

- No rotation is needed
- Remove 20

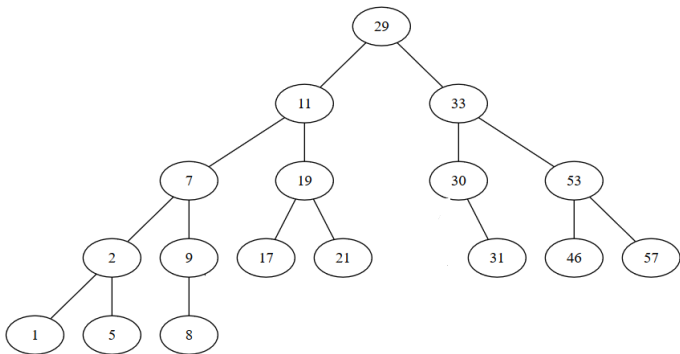
AVL rotations example XLVI



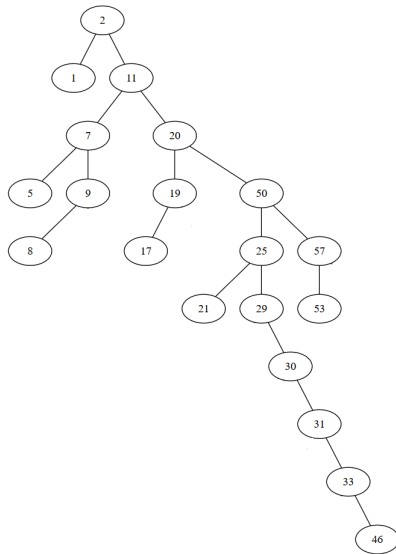
- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XLVII

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

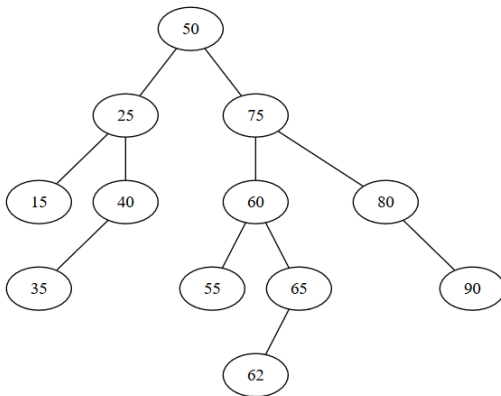


- If, instead of using an AVL tree, we used a binary search tree, after the insertions the tree would have been:



Rotations for remove

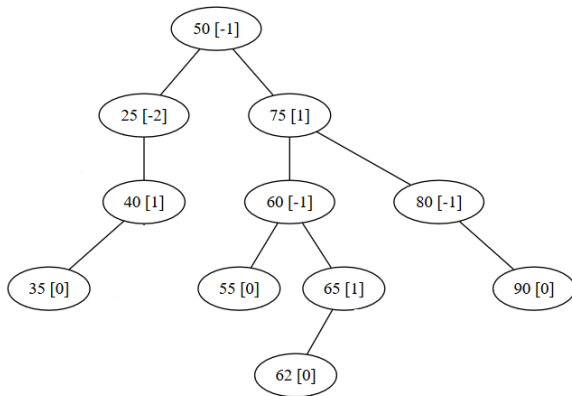
- When we remove a node, we might need more than 1 rotation:



- Remove value 15

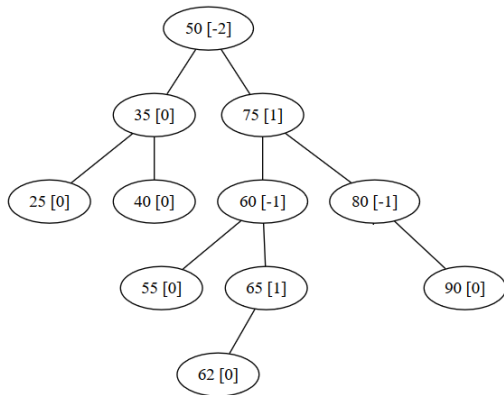
Rotations for remove

- After remove:



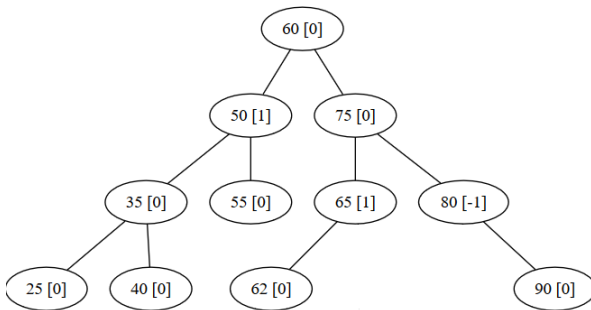
Rotations for remove

- 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*

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

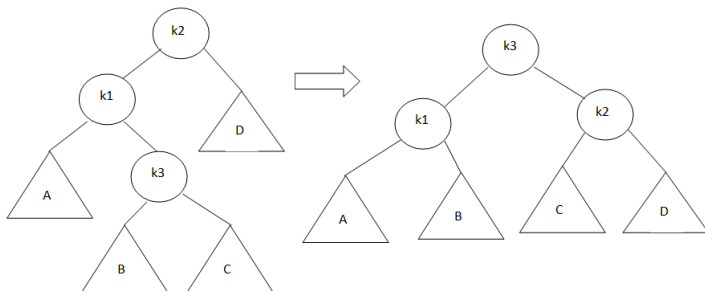
end-if

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).

Delimiter matching

- Given a sequence of round brackets (parentheses), (square) brackets and curly brackets, verify if the brackets are opened and closed correctly.
- For example:
 - The sequence $()([[][(())])$ - is correct
 - The sequence $[()()()]$ - is correct
 - The sequence $[()])$ - is not correct (one extra closed round bracket at the end)
 - The sequence $[()]$ - is not correct (brackets closed in wrong order)
 - The sequence $\{[[]] ()$ - is not correct (curly bracket is not closed)

Bracket matching - Solution Idea

- Stacks are suitable for this problem, because the bracket that was opened last should be the first to be closed. This matches the LIFO property of the stack.
- The main idea of the solution:
 - Start parsing the sequence, element-by-element
 - If we encounter an open bracket, we push it to a stack
 - If we encounter a closed bracket, we pop the last open bracket from the stack and check if they match
 - If they don't match, the sequence is not correct
 - If they match, we continue
 - If the stack is empty when we finished parsing the sequence, it was correct

Bracket matching - Extension

- How can we extend the previous idea so that in case of an error we will also signal the position where the problem occurs?
- Remember, we have 3 types of errors:
 - Open brackets that are never closed
 - Closed brackets that were not opened
 - Mismatch

Bracket matching - Extension

- How can we extend the previous idea so that in case of an error we will also signal the position where the problem occurs?
- Remember, we have 3 types of errors:
 - Open brackets that are never closed
 - Closed brackets that were not opened
 - Mismatch
- Keep count of the current position in the sequence, and push to the stack $\langle \text{delimiter}, \text{position} \rangle$ pairs.

Evaluating an arithmetic expression

- We want to write an algorithm that can compute the result of an arithmetic expression:
- For example:
 - $2+3*4 = 14$
 - $((2+4)*7)+3*(9-5) = 54$
 - $(((((3+1)*3)/((9-5)+2))-((3*(7-4)) + 6)) = -13$
- An arithmetic expression is composed of *operators* (+, -, * or /), parentheses and *operands* (the numbers we are working with). For simplicity we are going to use single digits as operands and we suppose that the expression is correct.

Infix and postfix notations

- The arithmetic expressions presented on the previous slide are in the so-called *infix* notation. This means that the *operators* are between the two operands that they refer to. Humans usually use this notation, but for a computer algorithm it is complicated to compute the result of an expression in an infix notation.
- Computers can work a lot easier with the *postfix* notation, where the operator comes after the operands.

Infix and postfix notations

- Examples of expressions in infix notation and the corresponding postfix notations:

Infix notation	Postfix notation
$1+2$	$12+$
$1+2-3$	$12+3-$
$4*3+6$	$43*6+$
$4*(3+6)$	$436+*$
$(5+6)*(4-1)$	$56+41-*$
$1+2*(3-4/(5+6))$	$123456+/-*+$

- The order of the operands is the same for both the infix and the postfix notations, only the order of the operators changes
- The operators have to be ordered taking into consideration operator precedence and the parentheses

Infix and postfix notations

- So, evaluating an arithmetic expression is divided into two subproblems:
 - Transform the infix notation into a postfix notation
 - Evaluate the postfix notation
- Both subproblems are solved using stacks and queues.

Infix to postfix transformation - The main idea

- Use an auxiliary stack for the operators and parentheses and a queue for the result.
- Start parsing the expression.
- If an operand is found, push it to the queue
- If an open parenthesis is found, it is pushed to the stack.
- If a closed parenthesis is found, pop elements from the stack and push them to the queue until an open parenthesis is found (but do not push parentheses to the queue).

Infix to postfix transformation - The main idea

- If an operator (`opCurrent`) is found:
 - If the stack is empty, push the operator to the stack
 - While the top of the stack contains an operator with a higher or equal precedence than the current operator, pop and push to the queue the operator from the stack. Push `opCurrent` to the stack when the stack becomes empty, its top is a parenthesis or an operator with lower precedence.
 - If the top of the stack is open parenthesis or operator with lower precedence, push `opCurrent` to the stack.
- When the expression is completely parsed, pop everything from the stack and push to the queue.

Infix to postfix transformation - Example

- Let's follow the transformation of $1+2*(3-4/(5+6))+7$

Infix to postfix transformation - Example

- Let's follow the transformation of $1+2*(3-4/(5+6))+7$

Input	Operation	Stack	Queue
1	Push to Queue		1
+	Push to stack	+	1
2	Push to Queue	+	12
*	Check (no Pop) and Push	+*	12
(Push to stack	+*(12
3	Push to Queue	+*(123
-	Check (no Pop) and Push	+*(-	123
4	Push to Queue	+*(-	1234
/	Check (no Pop) and Push	+*(-/	1234
(Push to stack	+*(-/(1234
5	Push to Queue	+*(-/(12345
+	Check (no Pop) and Push	+*(-/(+	12345
6	Push to Queue	+*(-/(+	123456
)	Pop and push to Queue till (+*(-/	123456+
)	Pop and push to Queue till (+*	123456+/-
+	Check, Pop twice and Push	+	123456+/-*+
7	Push to Queue	+	123456+/-*+7
over	Pop everything and push to Queue		123456+/-*+7+

Infix to postfix transformation - Implementation

```
function infixToPostfix(expr) is:  
    init(st)  
    init(q)  
    for elem in expr execute  
        if @elem is an operand then  
            push(q, elem)  
        else if @ elem is open parenthesis then  
            push(st, elem)  
        else if @elem is a closed parenthesis then  
            while @ top(st) is not an open parenthesis execute  
                op ← pop(st)  
                push(q, op)  
            end-while  
            pop(st) //get rid of open parenthesis  
        else //we have operand  
//continued on the next slide
```

Infix to postfix transformation - Implementation

```
    while not isEmpty(st) and @ top(st) not open parenthesis and @
top(st) has >= precedence than elem execute
        op ← pop(st)
        push(q, op)
    end-while
    push(st, elem)
end-if
end-for
while not isEmpty(st) execute
    op ← pop(st)
    push(q, op)
end-while
infixtoPostfix ← q
end-function
```

- Complexity: $\Theta(n)$ - where n is the length of the sequence

Evaluation of expression in postfix notation

- Once we have the postfix notation we can compute the value of the expression using a stack
- The main idea of the algorithm:
 - Use an auxiliary stack
 - Start parsing the expression
 - If an operand is found, it is pushed to the stack
 - If an operator is found, two values are popped from the stack, the operation is performed and the result is pushed to the stack
 - When the expression is parsed, the stack contains the result

Evaluation of postfix notation - Example

- Let's follow the evaluation of $123456+/-*+7+$

Evaluation of postfix notation - Example

- Let's follow the evaluation of $123456+/-*+7+$

Pop from the queue	Operation	Stack
1	Push	1
2	Push	1 2
3	Push	1 2 3
4	Push	1 2 3 4
5	Push	1 2 3 4 5
6	Push	1 2 3 4 5 6
+	Pop, add, Push	1 2 3 4 11
/	Pop, divide, Push	1 2 3 0
-	Pop, subtract, Push	1 2 3
*	Pop, multiply, Push	1 6
+	Pop, add, Push	7
7	Push	7 7
+	Pop, add, Push	14

Evaluation of postfix notation - Implementation

```
function evaluatePostfix(q) is:  
  init(st)  
  while not isEmpty(q) execute  
    elem  $\leftarrow$  pop(q)  
    if @ elem is an operand then  
      push(st, elem)  
    else  
      op1  $\leftarrow$  pop(st)  
      op2  $\leftarrow$  pop(st)  
      @ compute the result of op2 elem op1 in variable result  
      push(st, result)  
    end-if  
  end-while  
  result  $\leftarrow$  pop(st)  
  evaluatePostfix  $\leftarrow$  result  
end-function
```

- Complexity: $\Theta(n)$ - where n is the length of the expression

Evaluation of an arithmetic expression

- Combining the two functions we can compute the result of an arithmetic expression.
- How can we evaluate directly the expression in infix notation with one single function? *Hint: use two stacks.*
- How can we add exponentiation as a fifth operation?

Think about it

- Assume you have a binary tree, but you do not know how it looks like, but you have the *preorder* and *inorder* traversal of the tree. Give an algorithm for building the tree based on these two traversals.
- For example:
 - Preorder: A B F G H E L M
 - Inorder: B G F H A L E M