DATA STRUCTURES AND ALGORITHMS

LECTURE 13

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2024 - 2025

In Lecture 12...

- Binary search trees

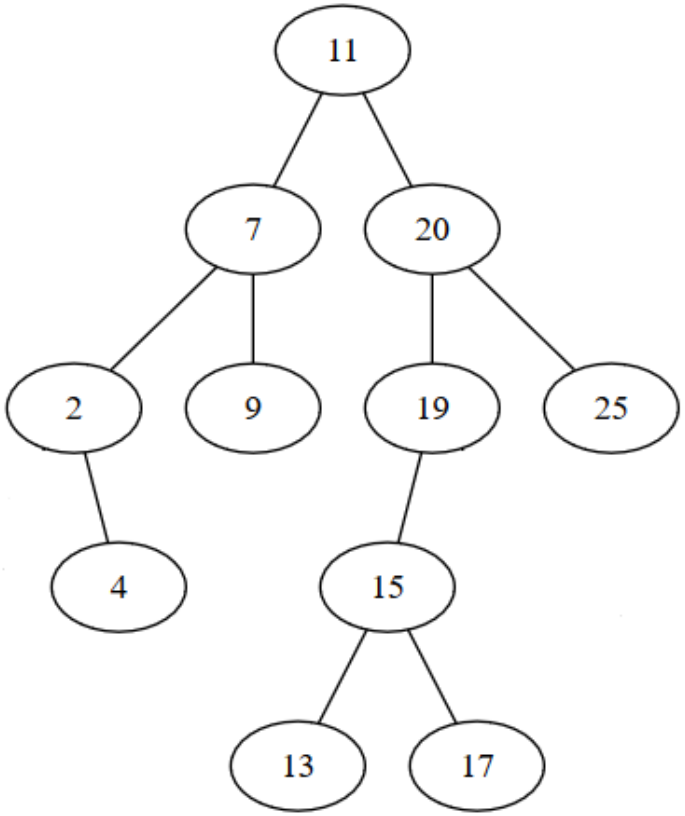
Today

- AVL trees
- Cuckoo hashing
- Perfect hashing

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 $\Theta(log_2 n)$

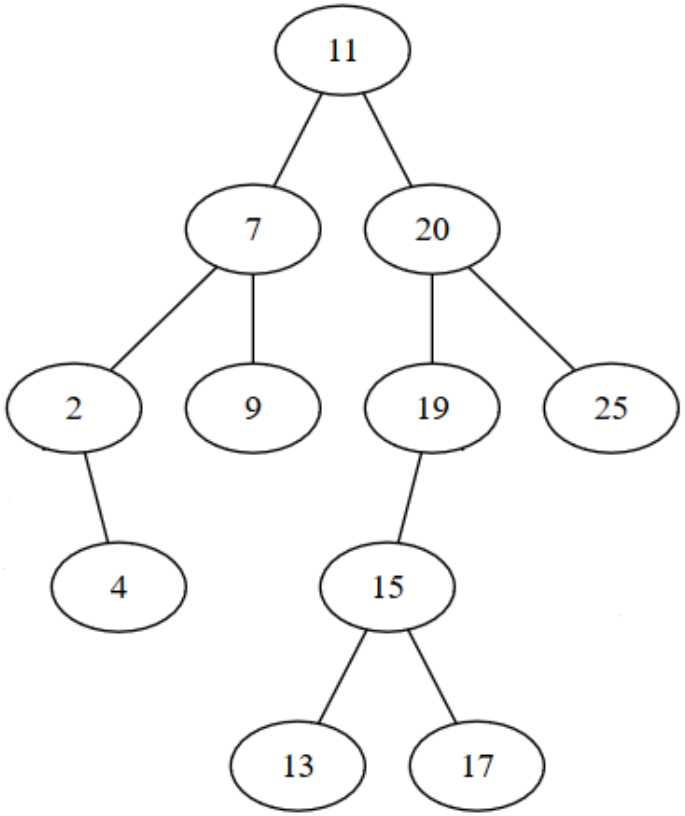
Balanced Binary Search Trees

- 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.

AVL Trees

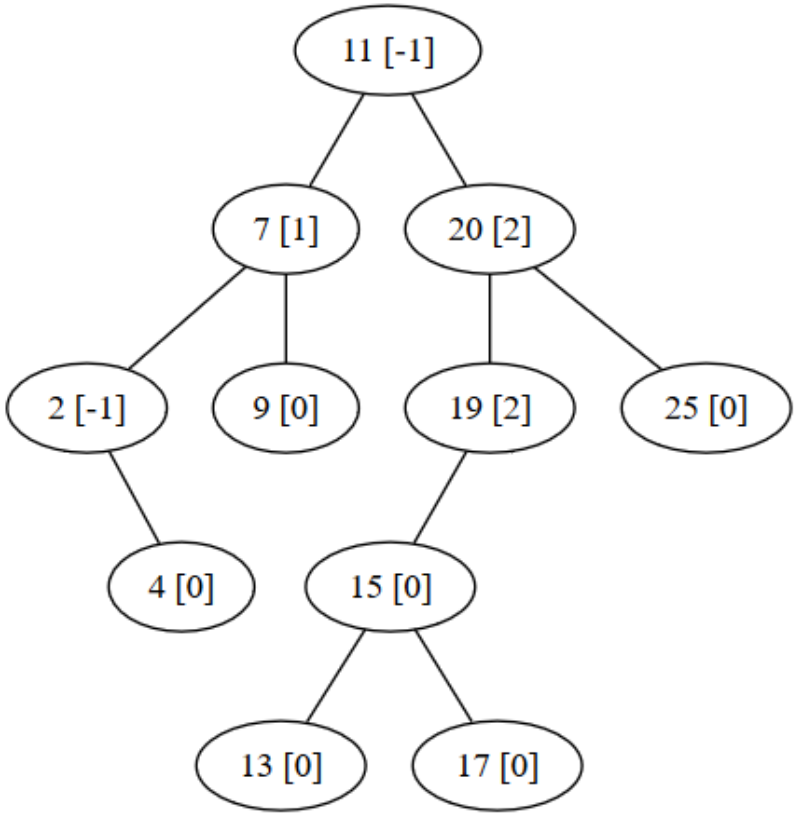
- Definition: An AVL (Adelson-Velskii Landis) tree is a binary tree which satisfies the following property (AVL tree property):
  - If x is a node of the AVL tree:
    - the difference between the height of the left and right subtree of x is 0, 1 or -1 (balancing information)
- Observations:
- Height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

AVL Trees



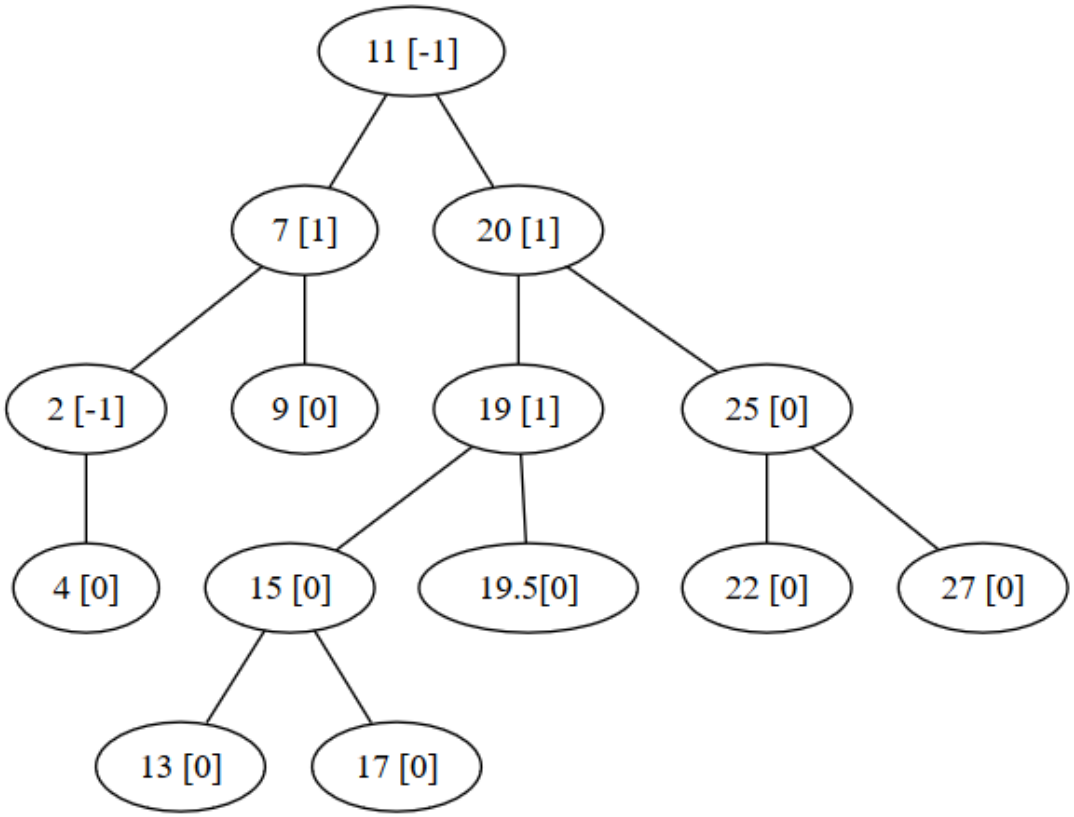
- Is this an AVL tree?

AVL Trees



- 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.

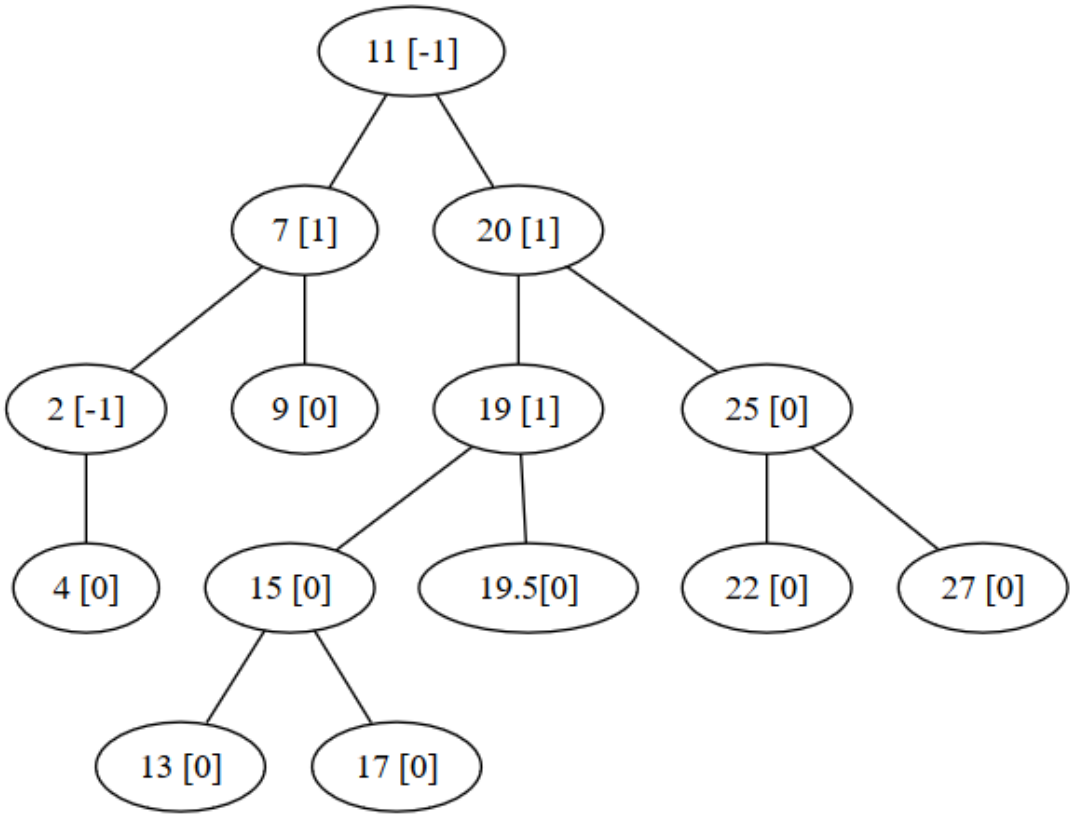
AVL Trees - rotations

- 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

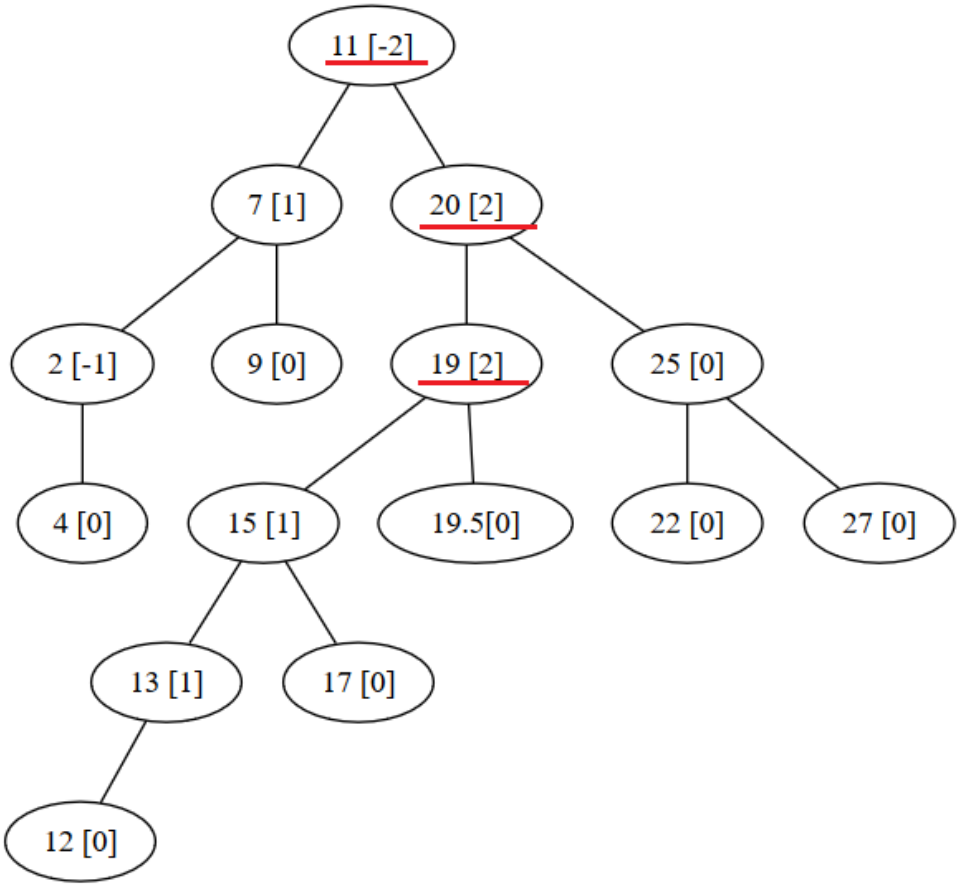
- Inserting in an AVL tree is similar to inserting in a BST: we need to find the position where the new element needs to be added and add the element there. However, this will change the balancing information of some nodes.
- After an insertion, only the nodes on the path to the modified node can change 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

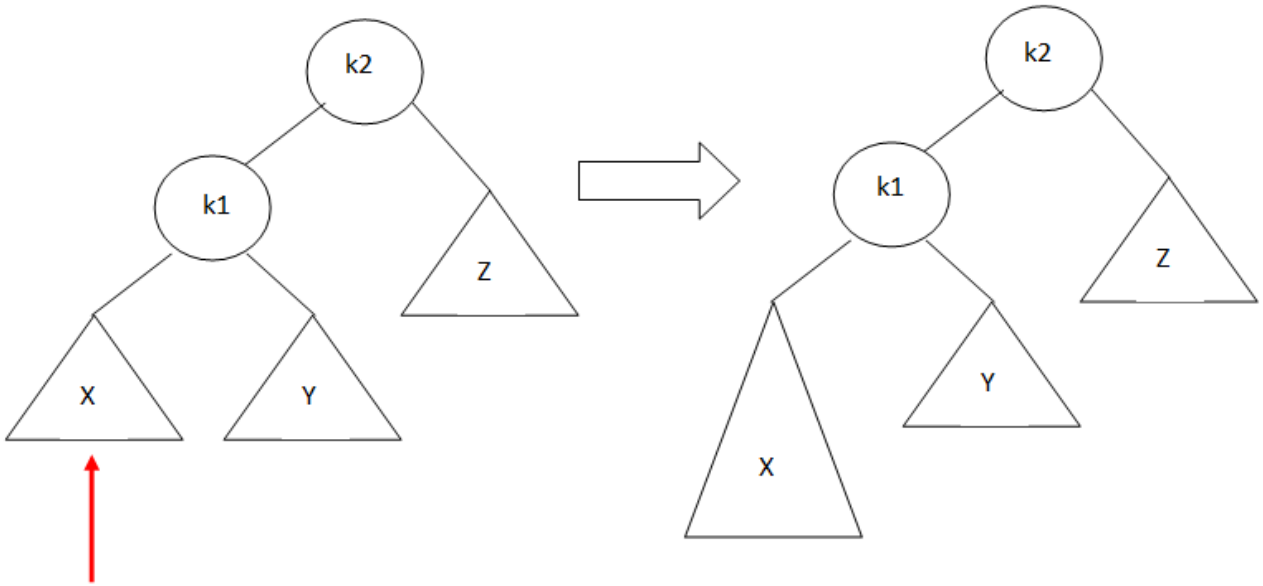


- Red lines show the unbalanced nodes. We will rebalance node 19.

AVL Trees - rotations

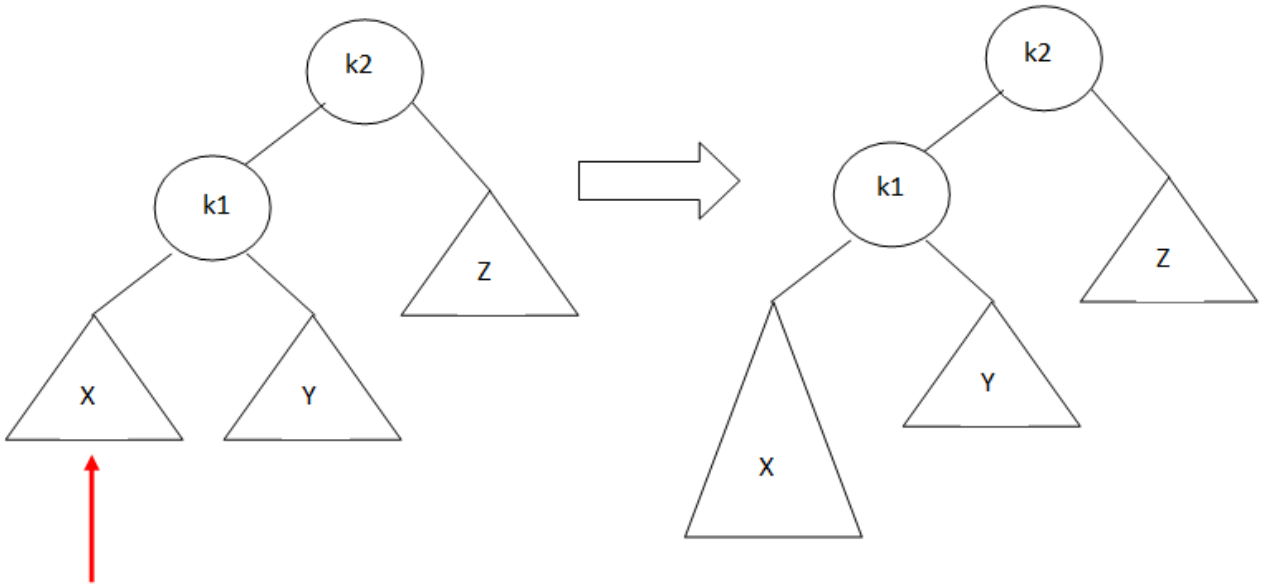
- Assume that at a given point α is the node that needs to be rebalanced. Obviously, α is not a leaf node, it has either a left or a right child, or both.
- Since α was balanced before the insertion, and is not after the insertion, we know that a new node was inserted somewhere in a subtree of α and 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



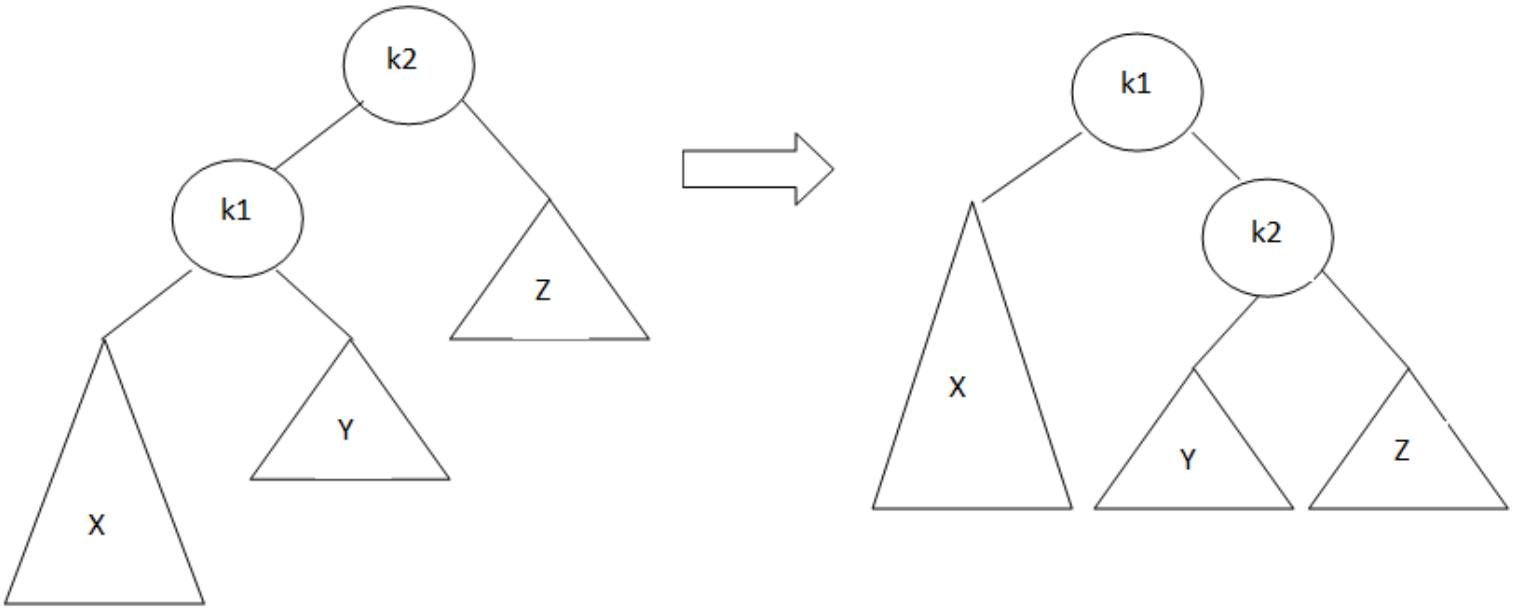
- Obs: X, Y and Z represent subtrees with the same height.

AVL Trees - rotations - case 1

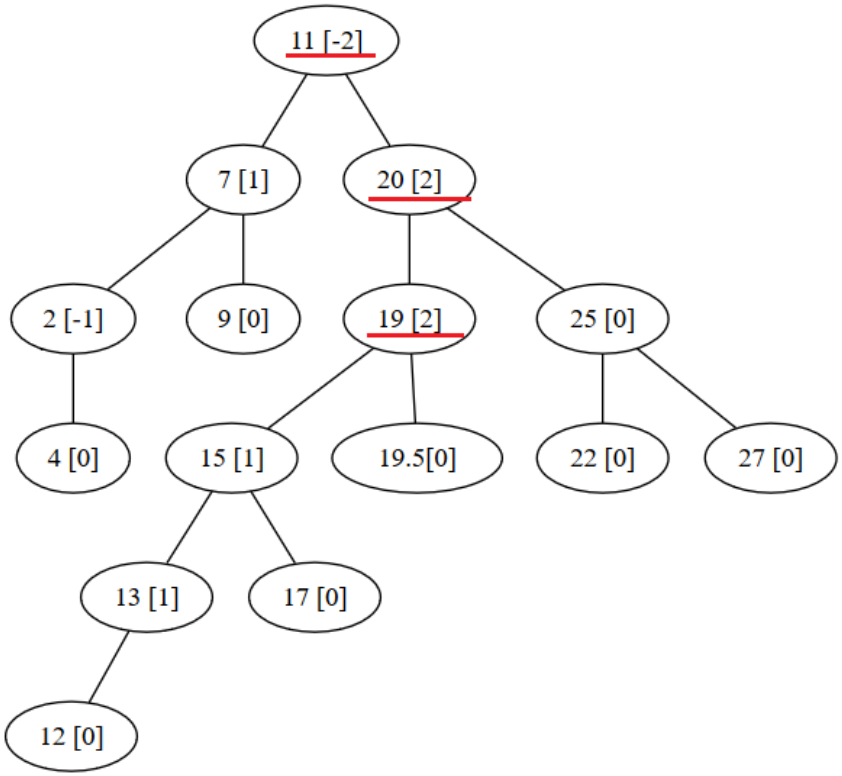


- Obs: X, Y and Z represent subtrees with the same height.
- Solution: single rotation to right

AVL Trees - rotation - Single Rotation to Right

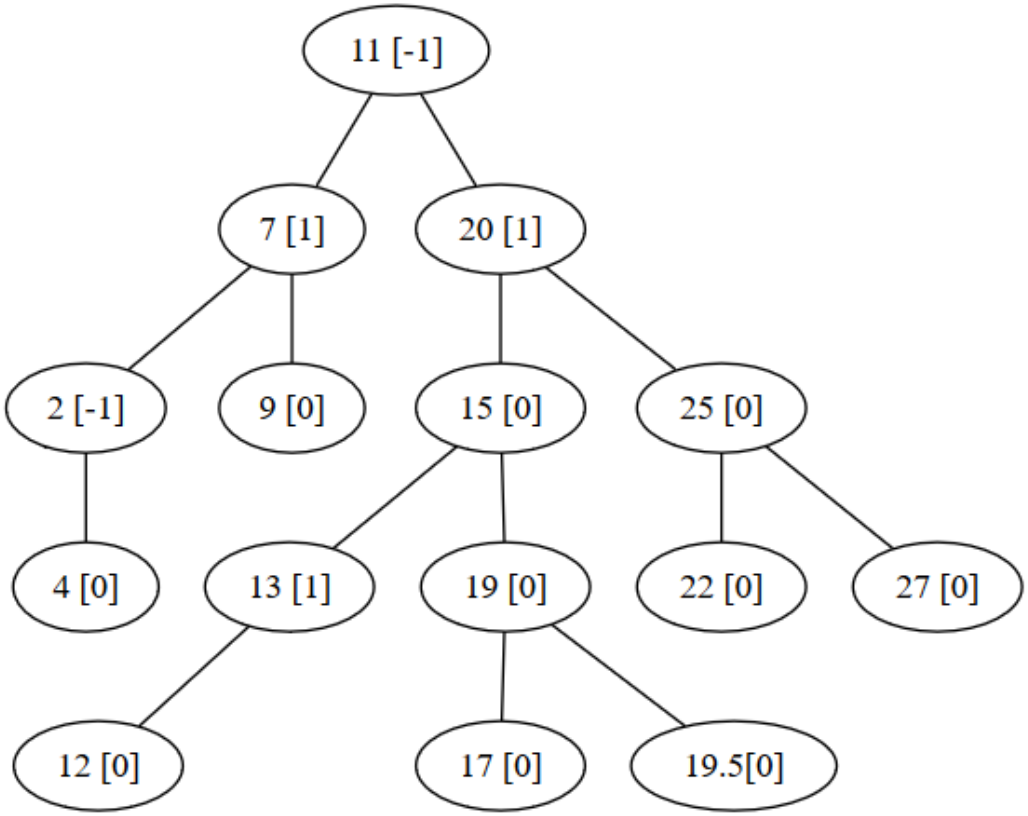


AVL Trees - rotations - case 1 example

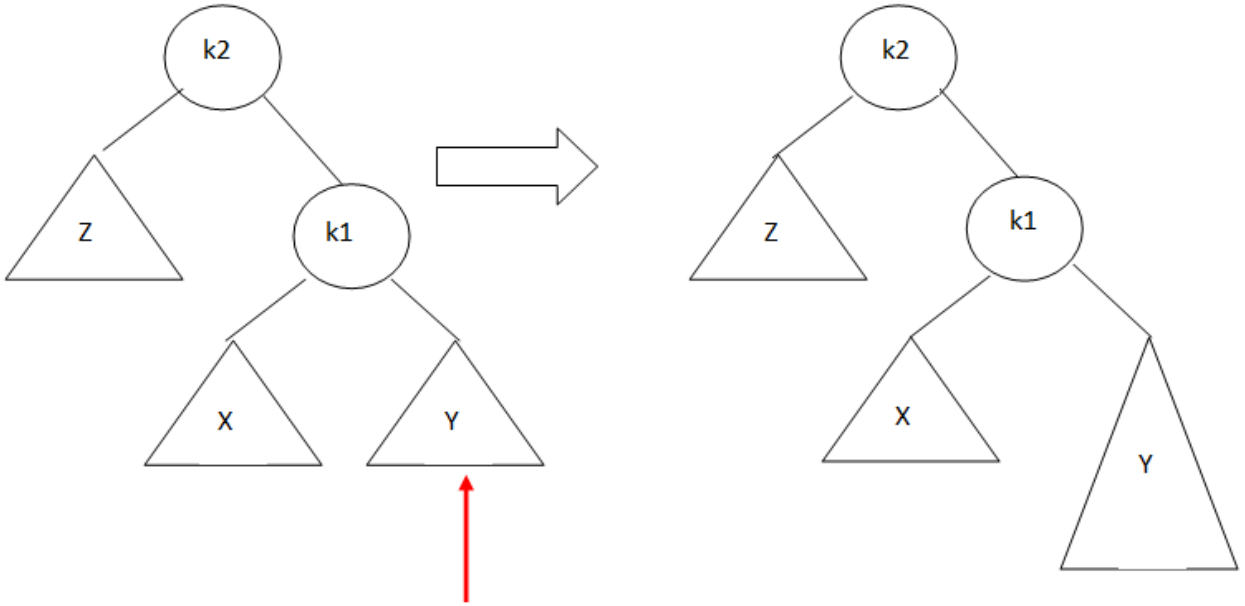


- Node 19 is imbalanced, because we inserted a new node (12) in the left subtree of the left child.
- Solution: **single rotation to right**

AVL Trees - rotation - case 1 example

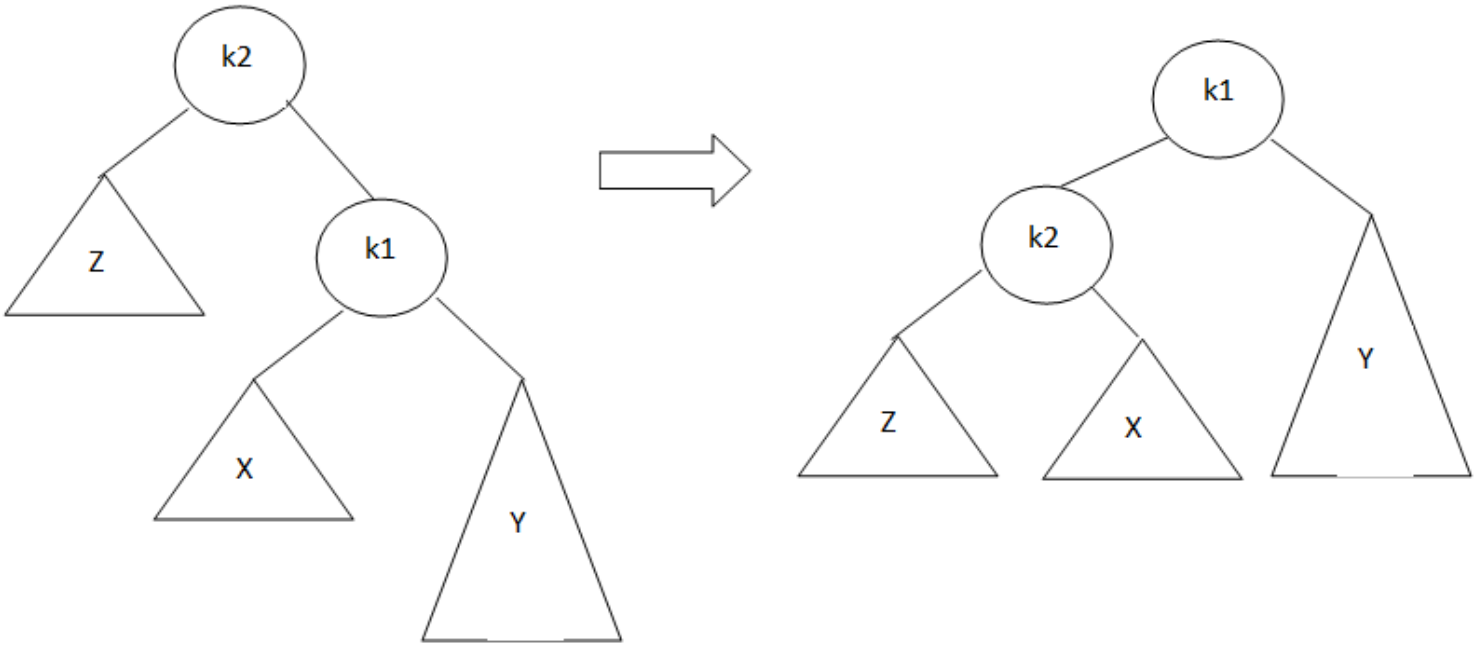


AVL Trees - rotations - case 4

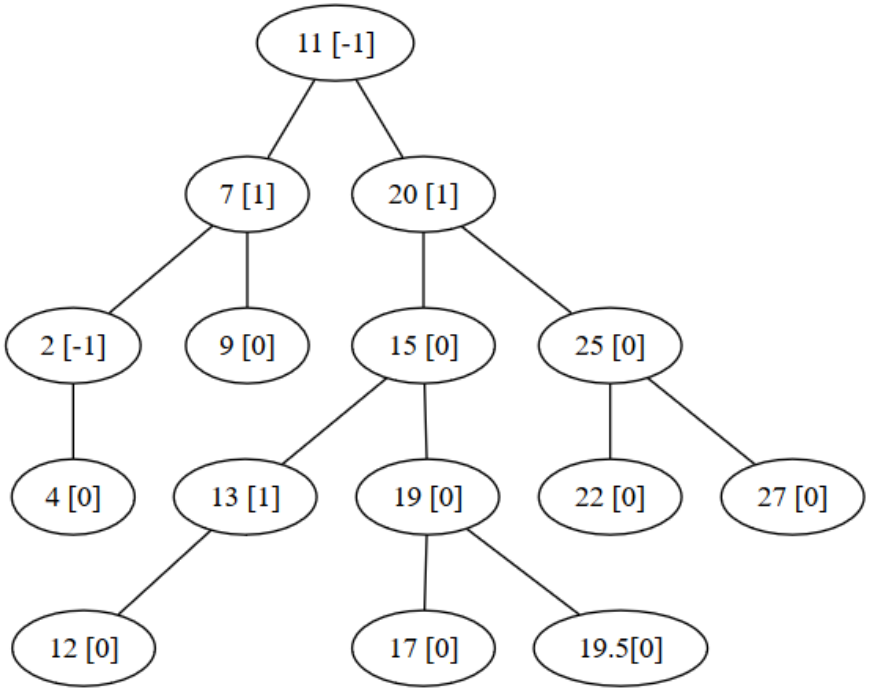


- Solution: **single rotation to left**

AVL Trees - rotation - Single Rotation to Left

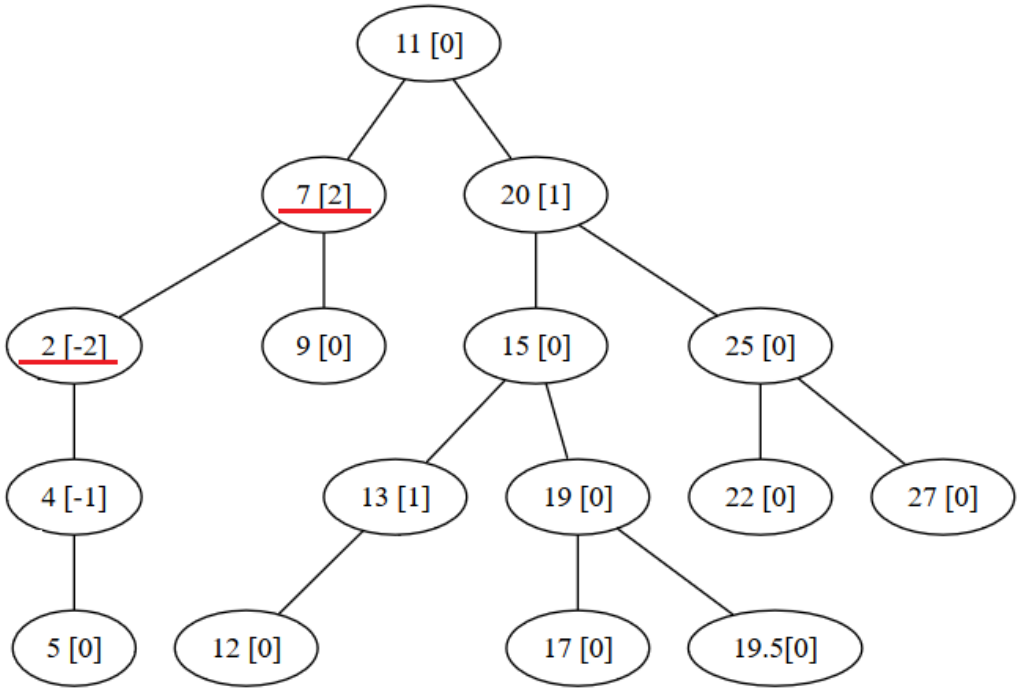


AVL Trees - rotations - case 4 example



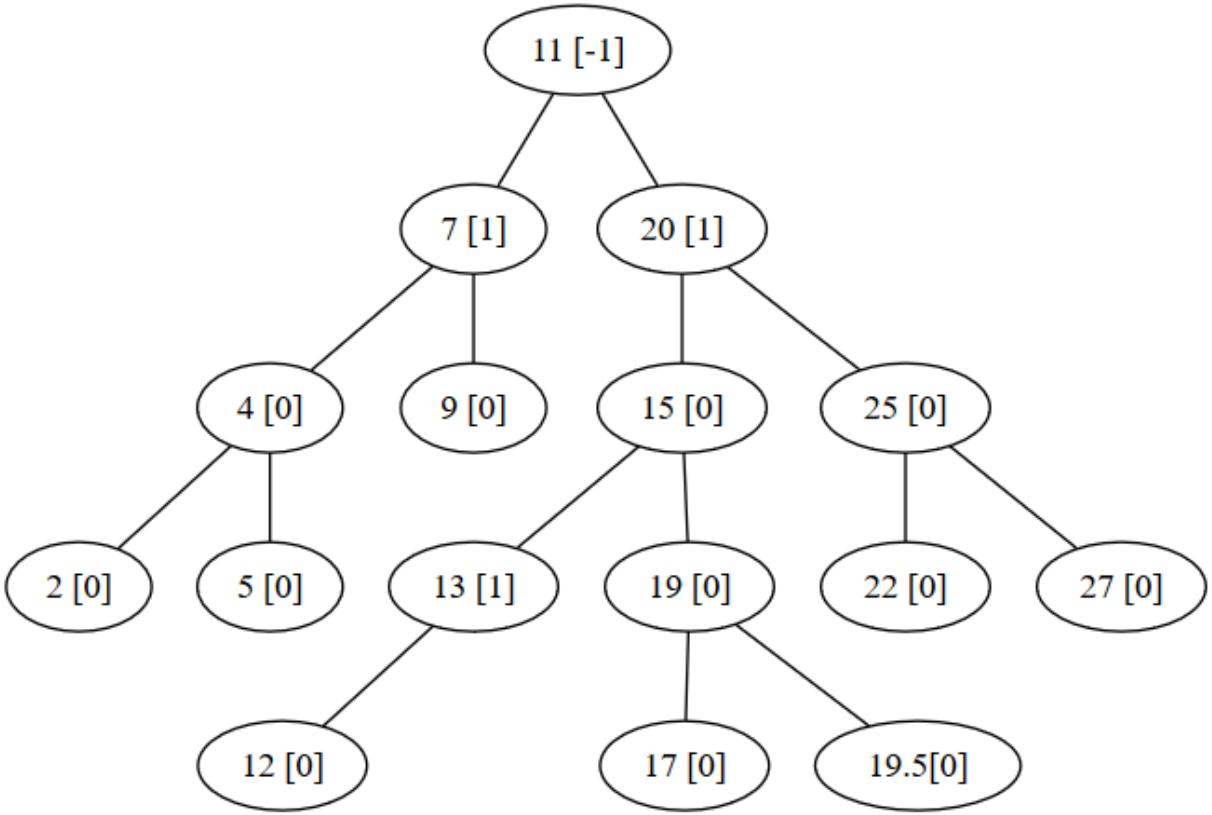
- Insert value 5

AVL Trees - rotations - case 4 example



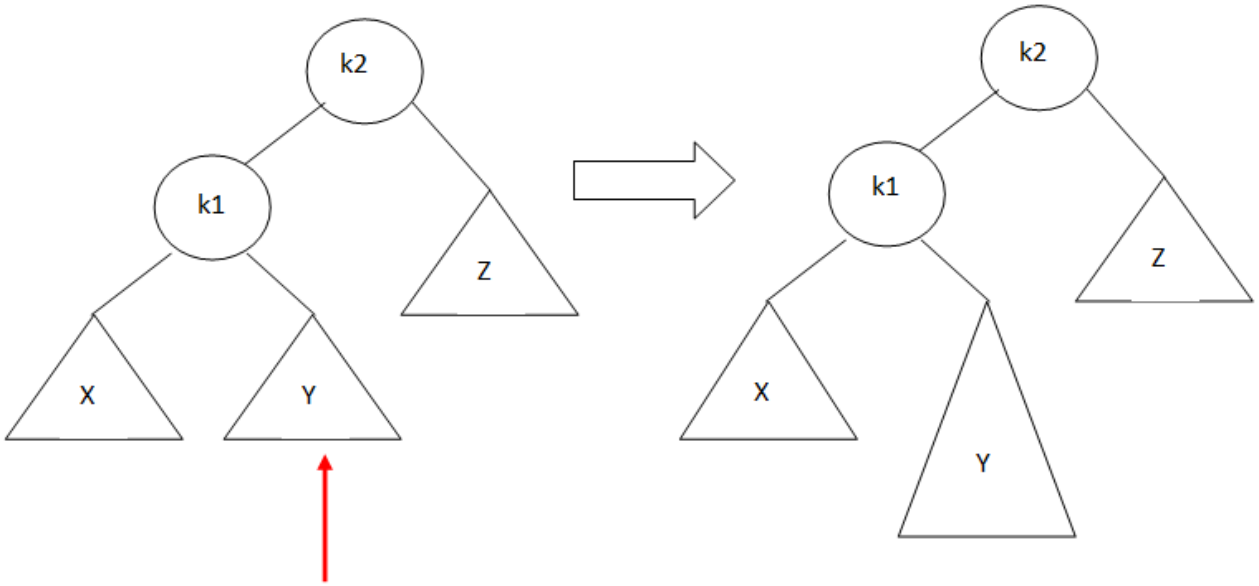
- Node 2 is imbalanced, because we inserted a new node (5) to the right subtree of the right child
- Solution: **single rotation to left**

AVL Trees - rotation - case 4 example



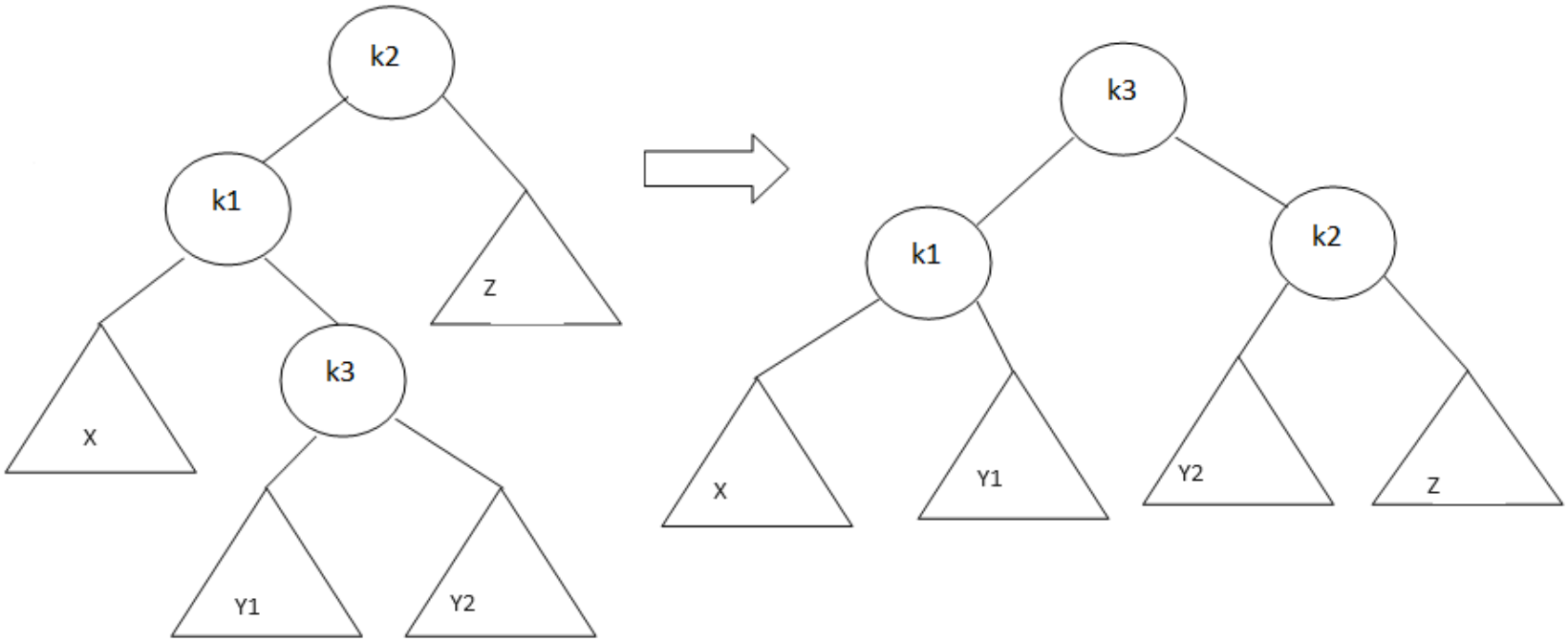
- After the rotation

AVL Trees - rotations - case 2

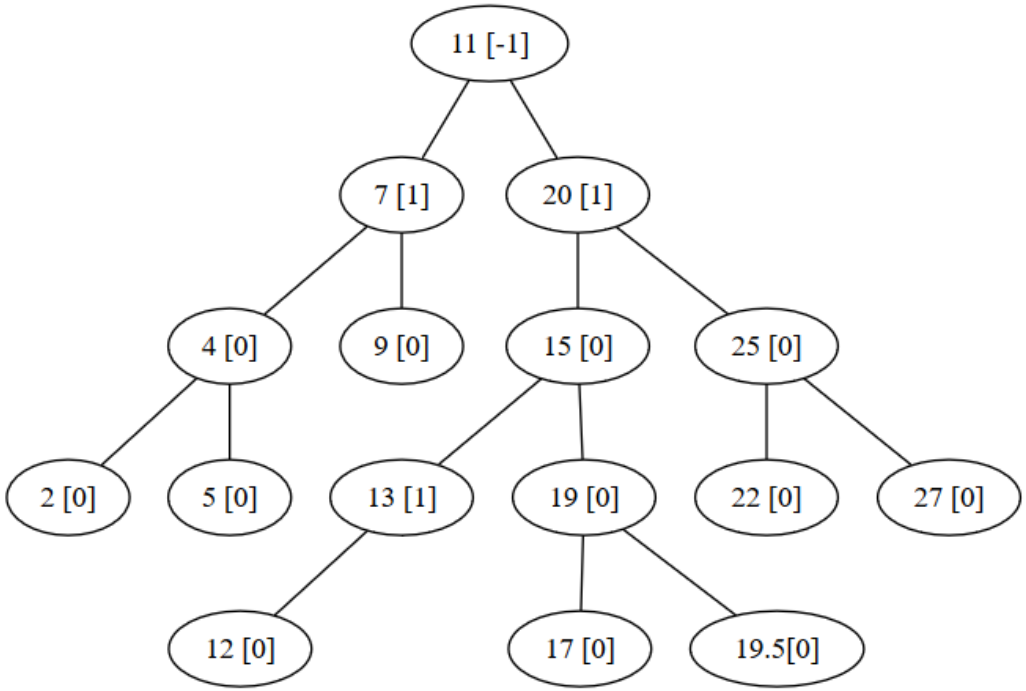


- Solution: **Double rotation to right**

AVL Trees - rotation - Double Rotation to Right

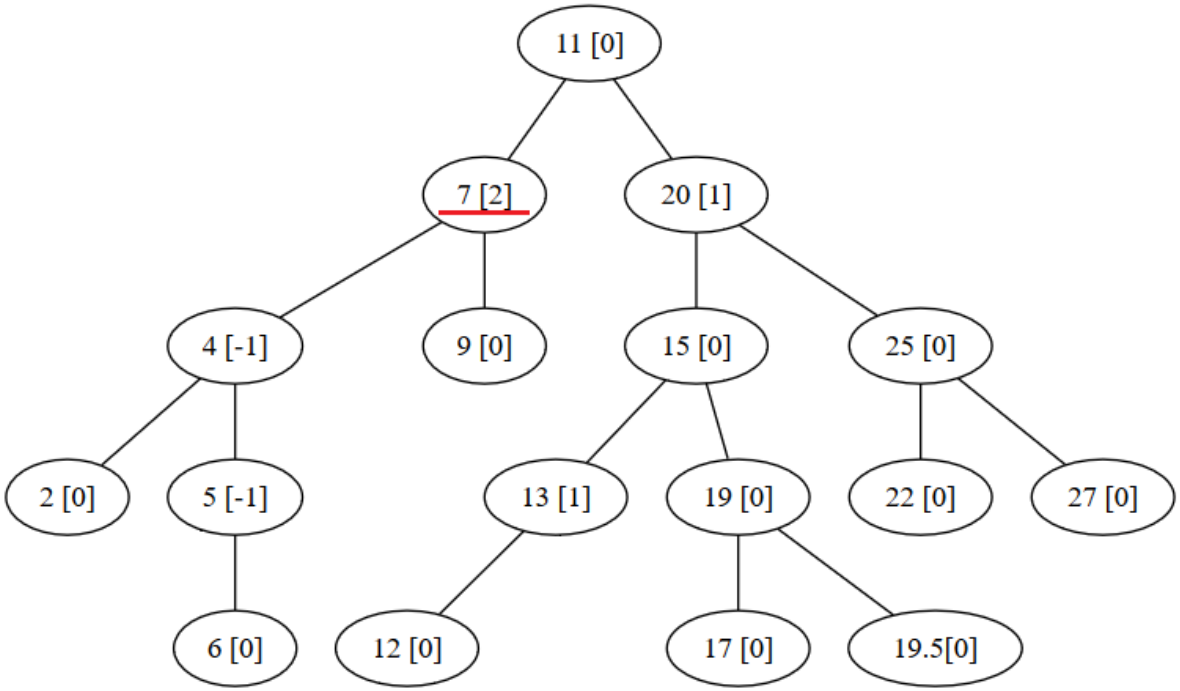


AVL Trees - rotations - case 2 example



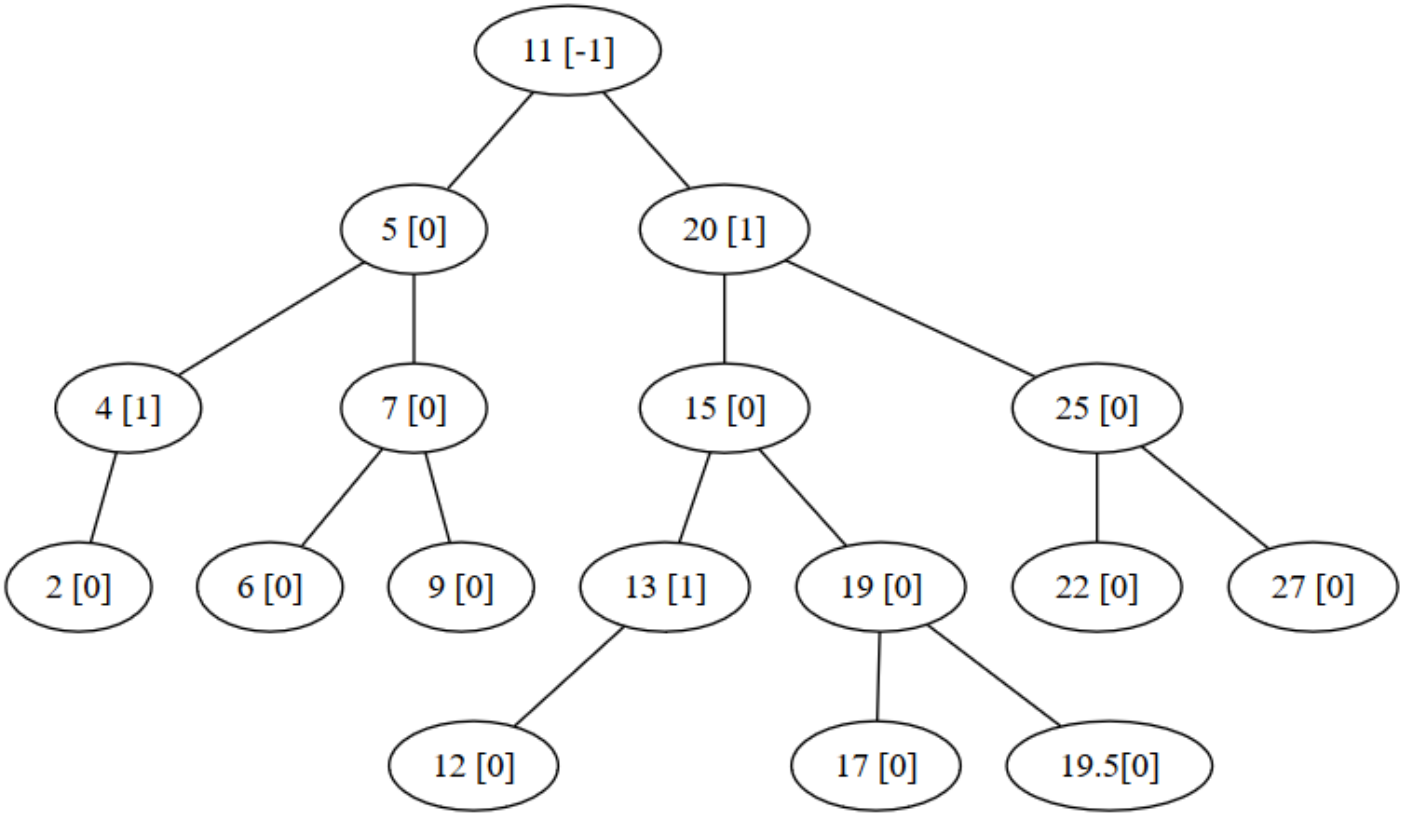
- Insert value 6

AVL Trees - rotations - case 2 example



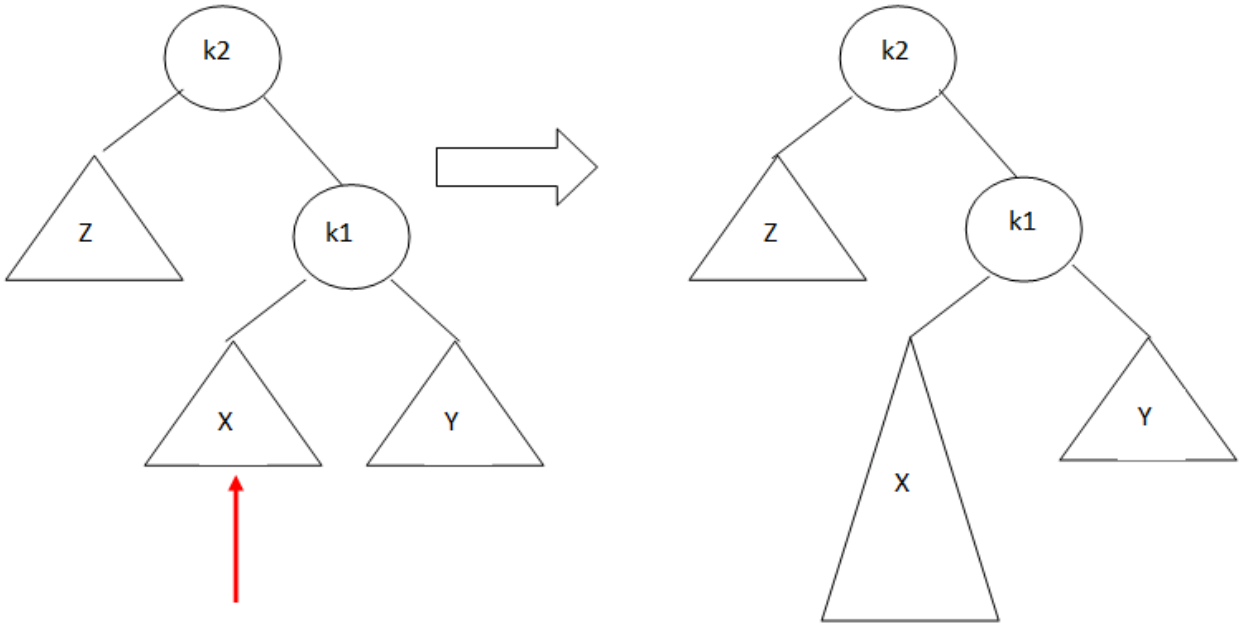
- Node 7 is imbalanced, because we inserted a new node (6) to the right subtree of the left child
- Solution: **double rotation to right**

AVL Trees - rotation - case 2 example



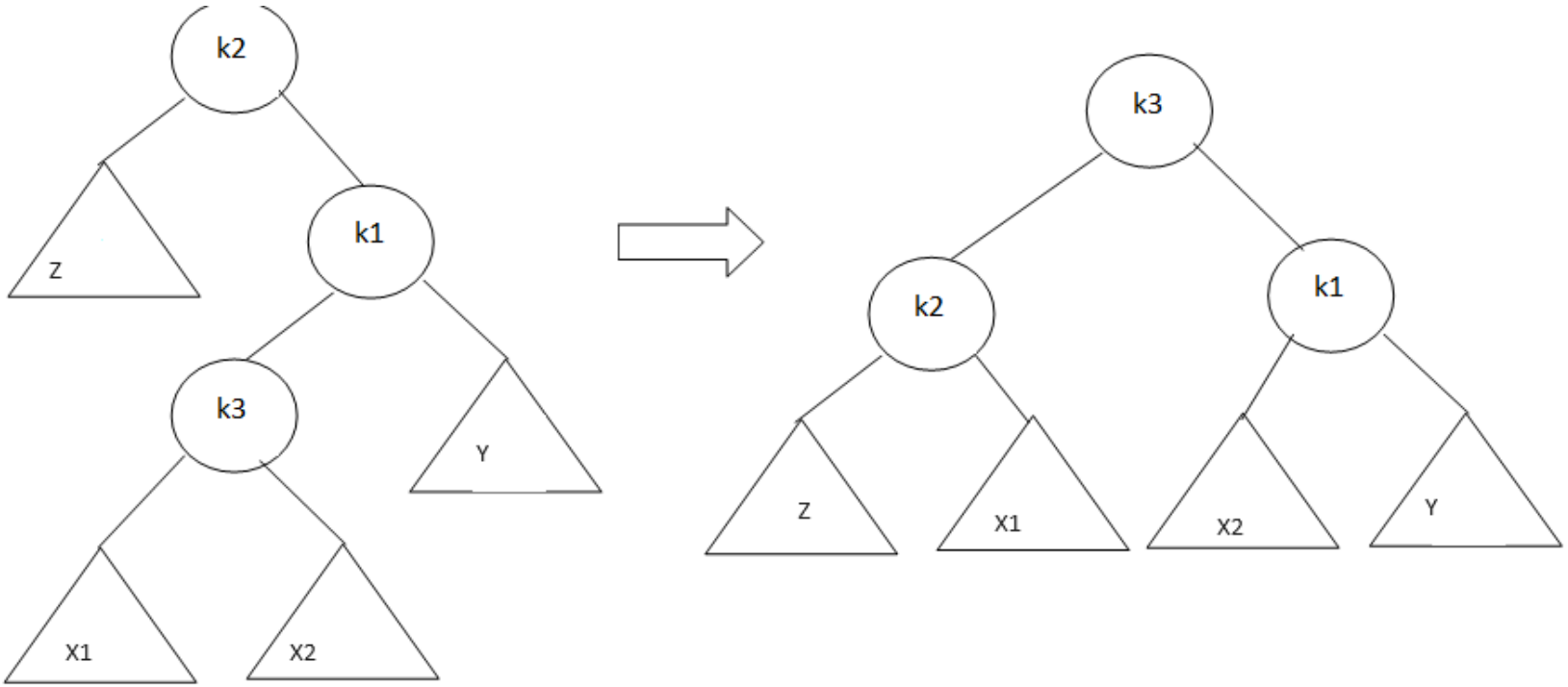
- After the rotation

AVL Trees - rotations - case 3

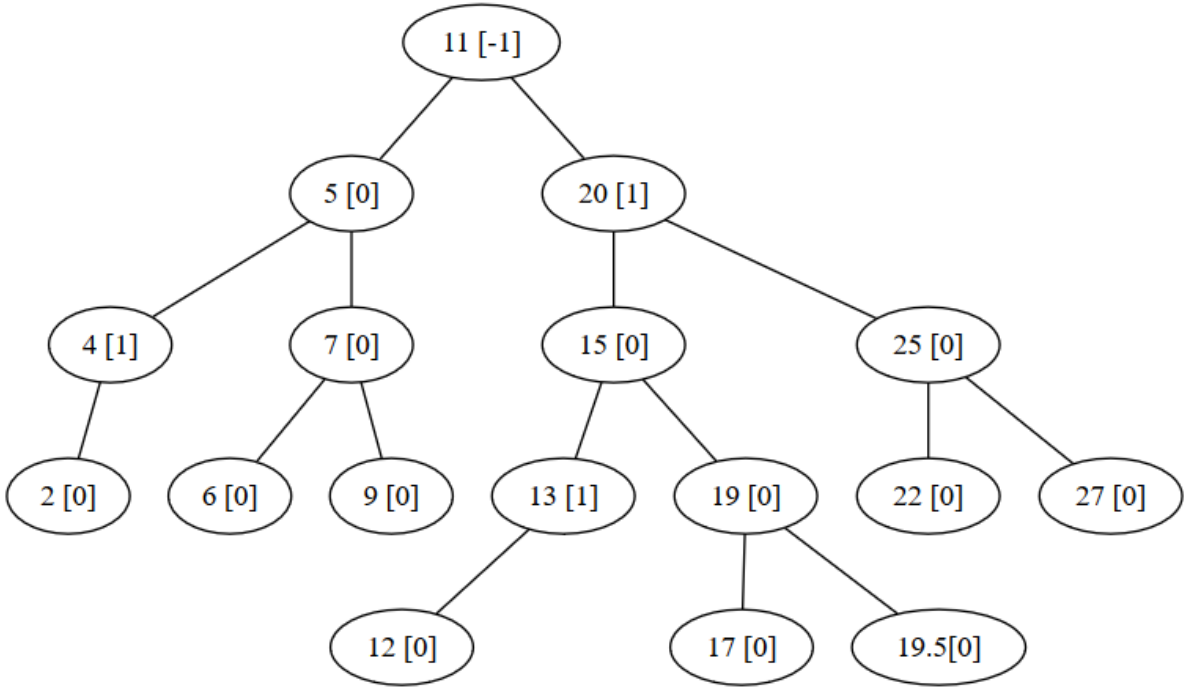


- Solution: **Double rotation to left**

AVL Trees - rotation - Double Rotation to Left

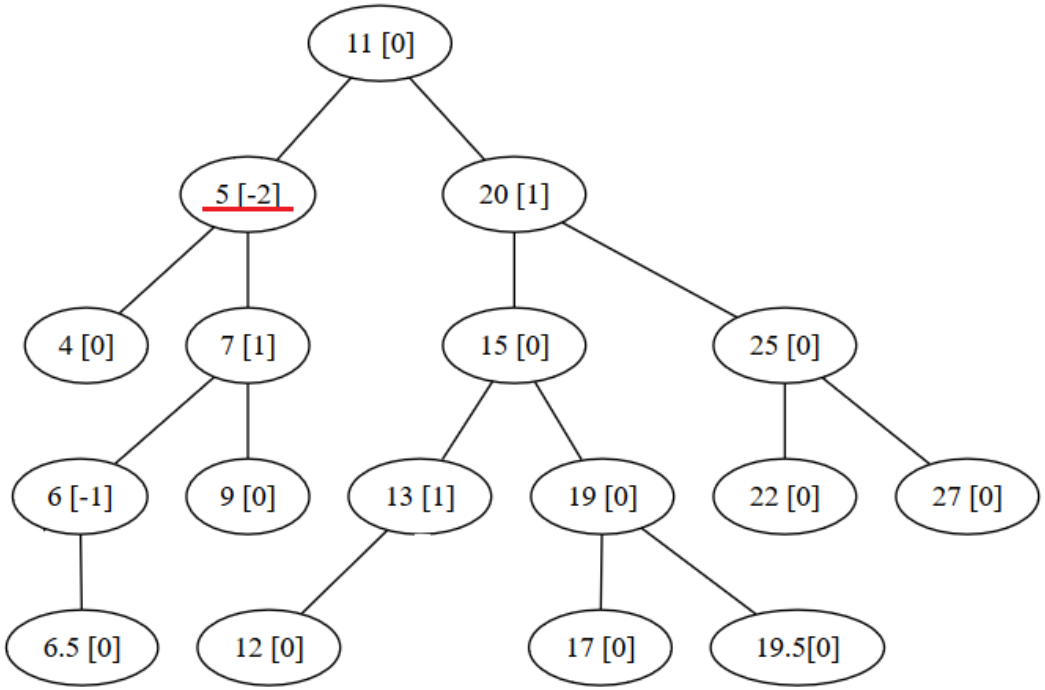


AVL Trees - rotations - case 3 example



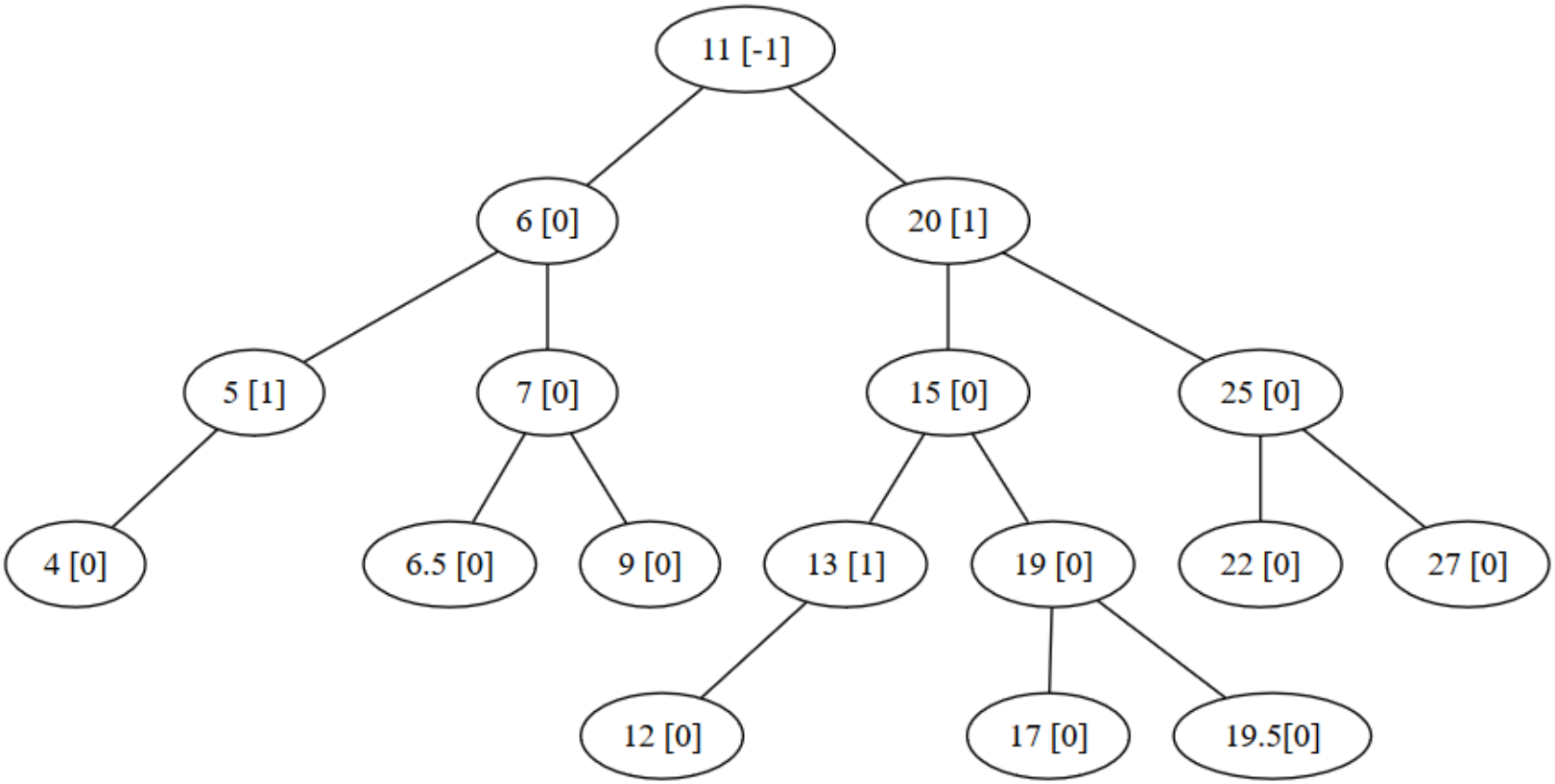
- Remove node with value 2 and insert value 6.5

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. $\rightarrow$ 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. $\rightarrow$ a rotation to the left is needed to balance it.

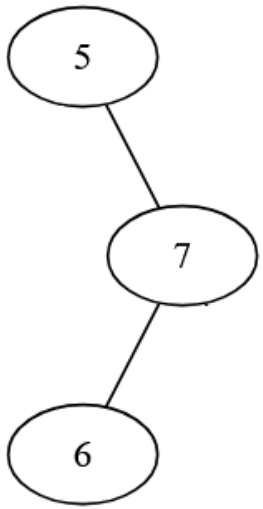
Recognizing the rotations II

- 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).

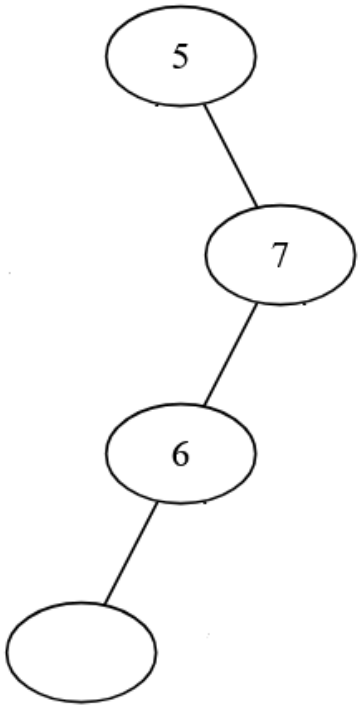
Double rotations I

- 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:



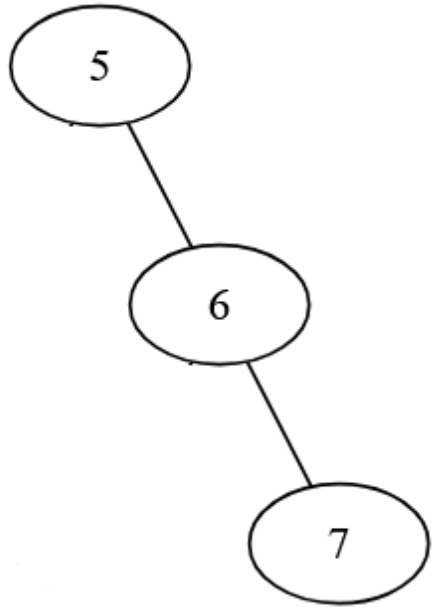
- This is the simplest, classic case of a double left rotation. However, we could also imagine that node 6 has a left child.

Double rotations II



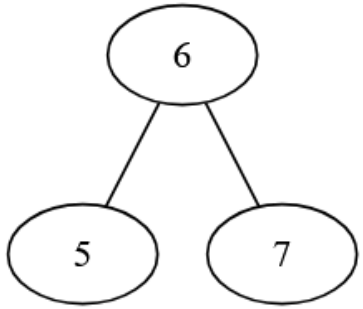
- In this case, we actually have an imbalance at node 7, where we need a single right notation to balance it.

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

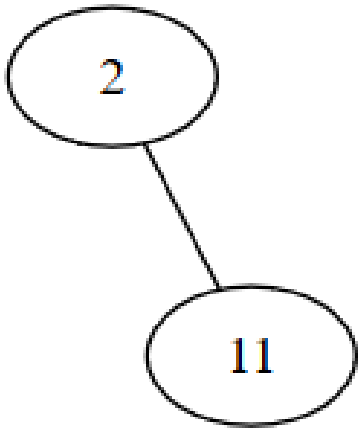
AVL rotations example II

- 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

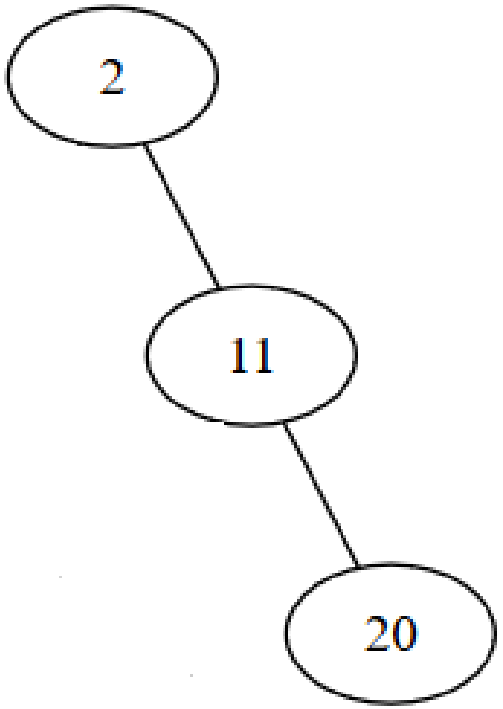


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example V

- No rotation is needed
- Insert 20

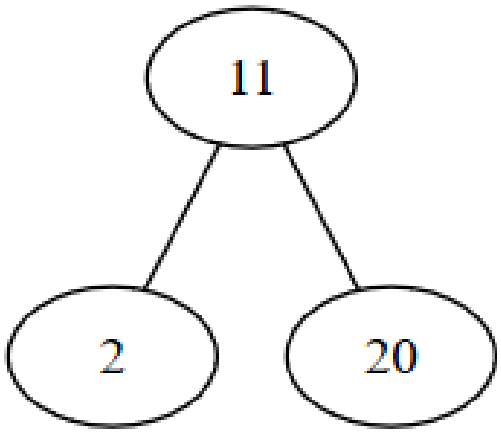
AVL rotations example VI



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

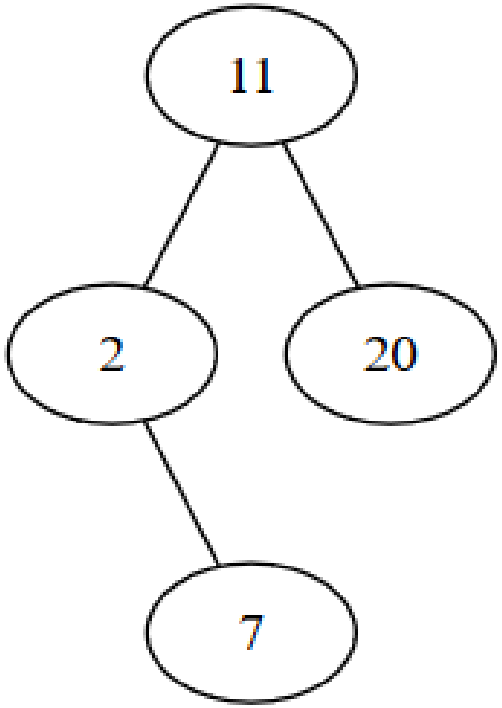
AVL rotations example VII

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



- Insert 7

AVL rotations example VIII

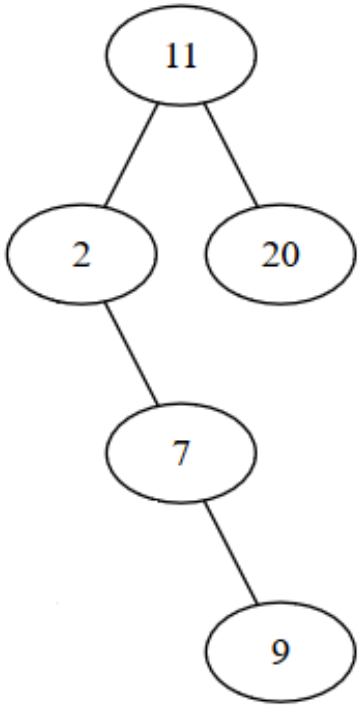


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example IX

- No rotation is needed
- Insert 9

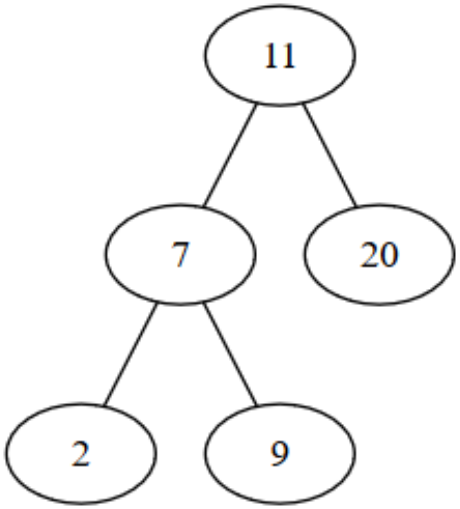
AVL rotations example X



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

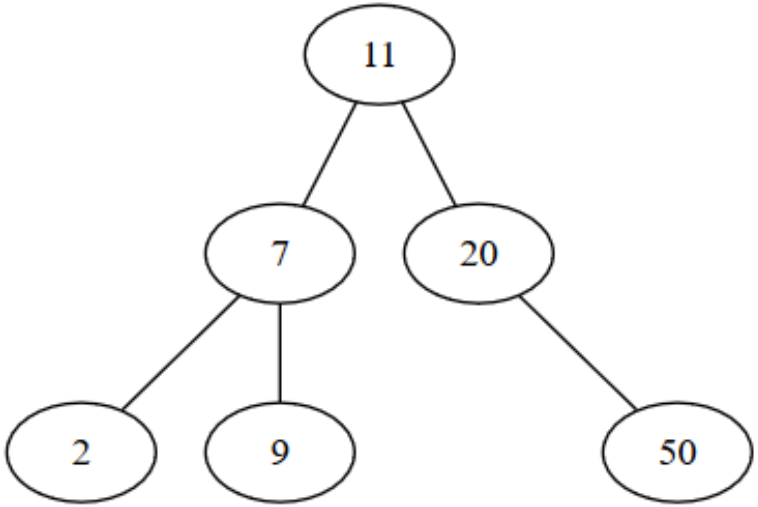
AVL rotations example XI

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



- Insert 50

AVL rotations example XII

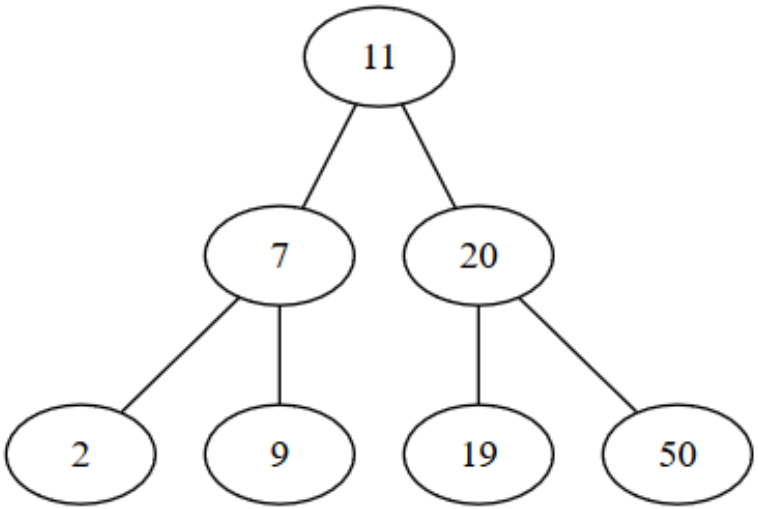


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XIII

- No rotation is needed
- Insert 19

AVL rotations example XIV

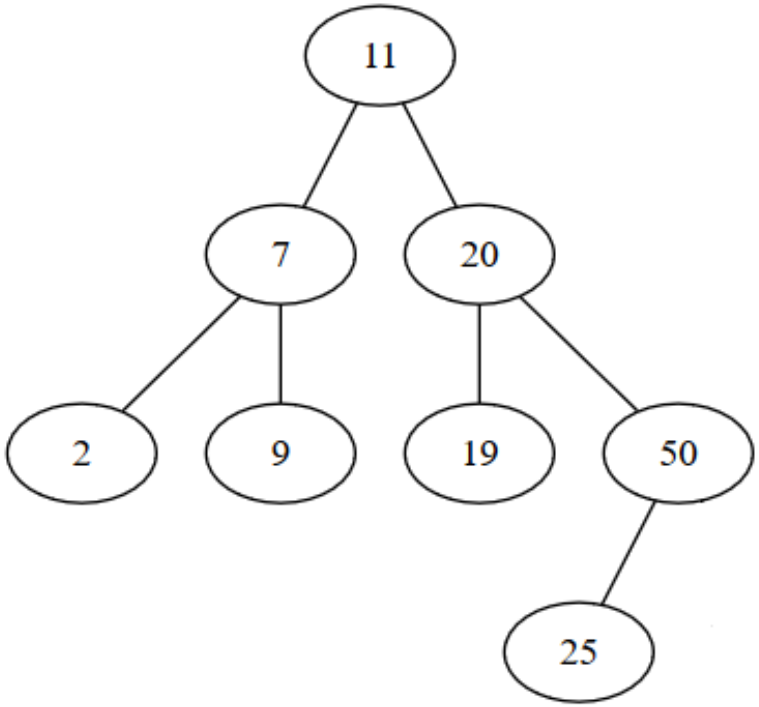


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XV

- No rotation is needed
- Insert 25

AVL rotations example XVI

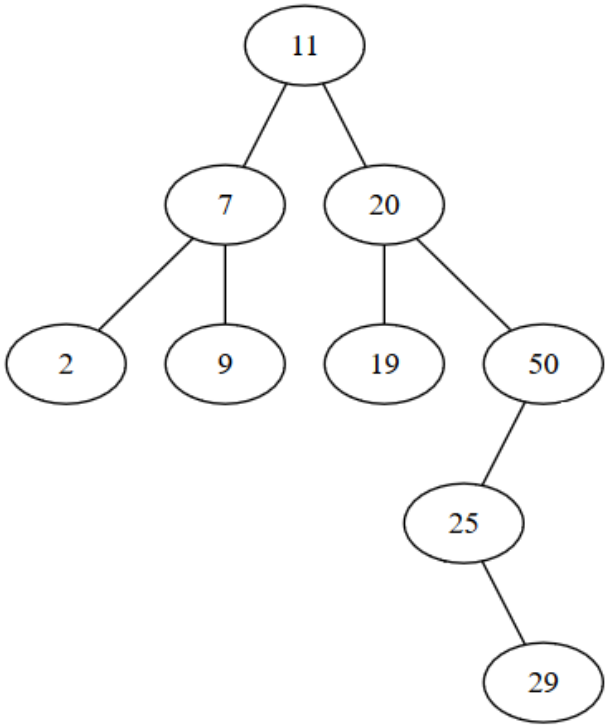


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XVII

- No rotation is needed
- Insert 29

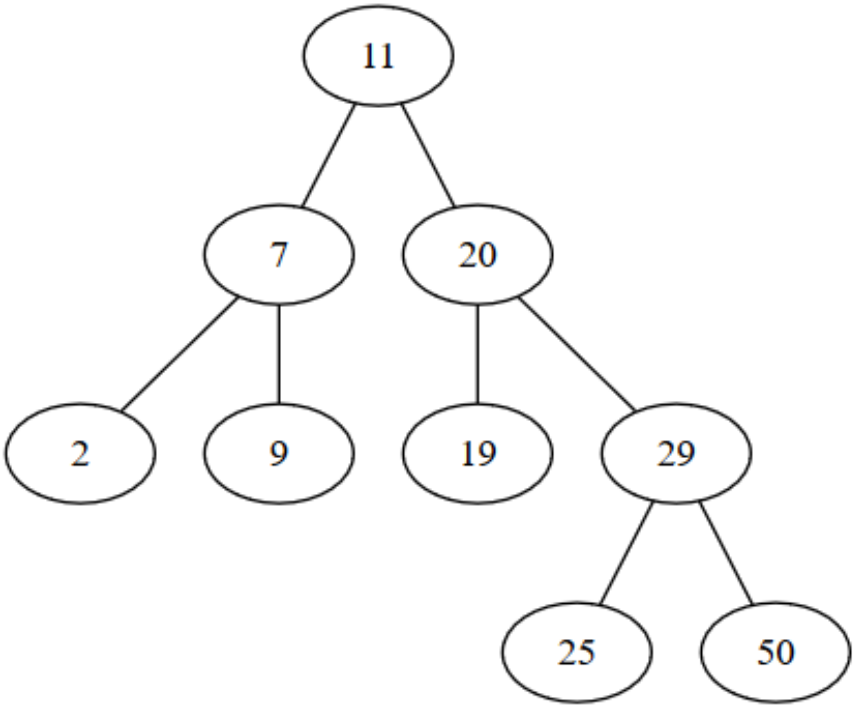
AVL rotations example XVIII



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

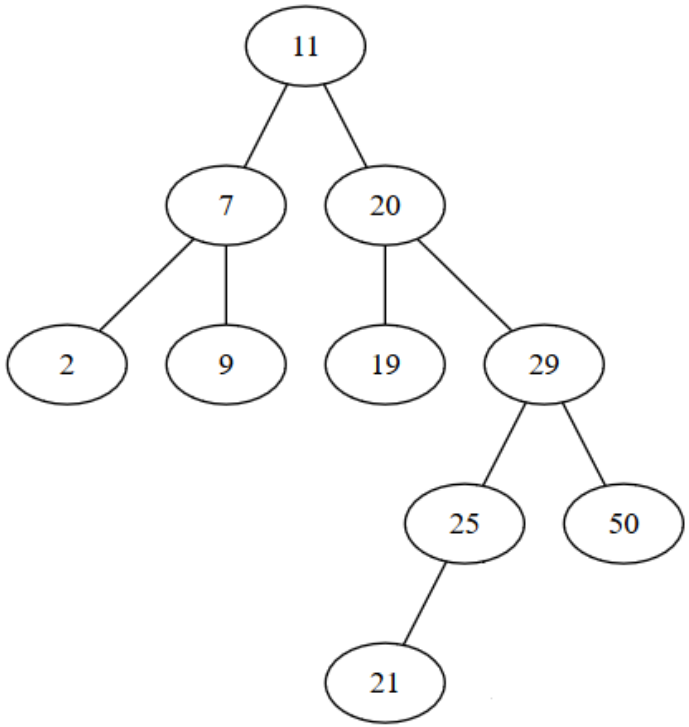
AVL rotations example XIX

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



- Insert 21

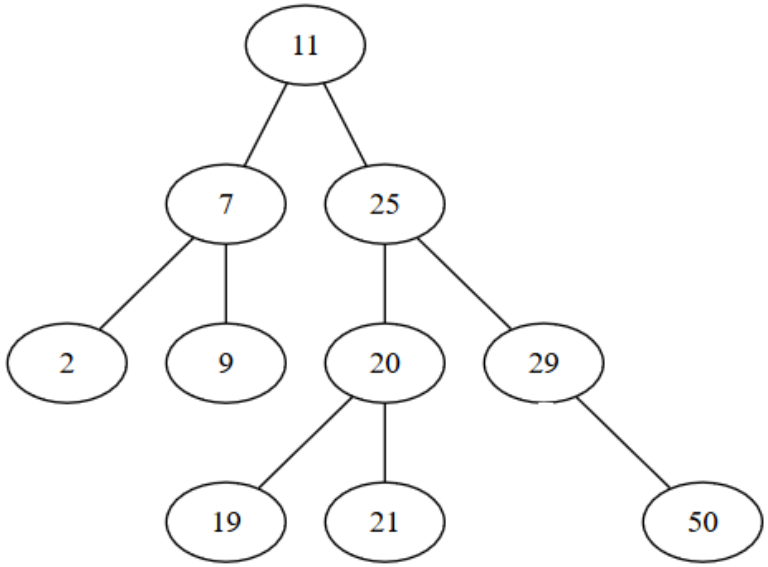
AVL rotations example XX



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

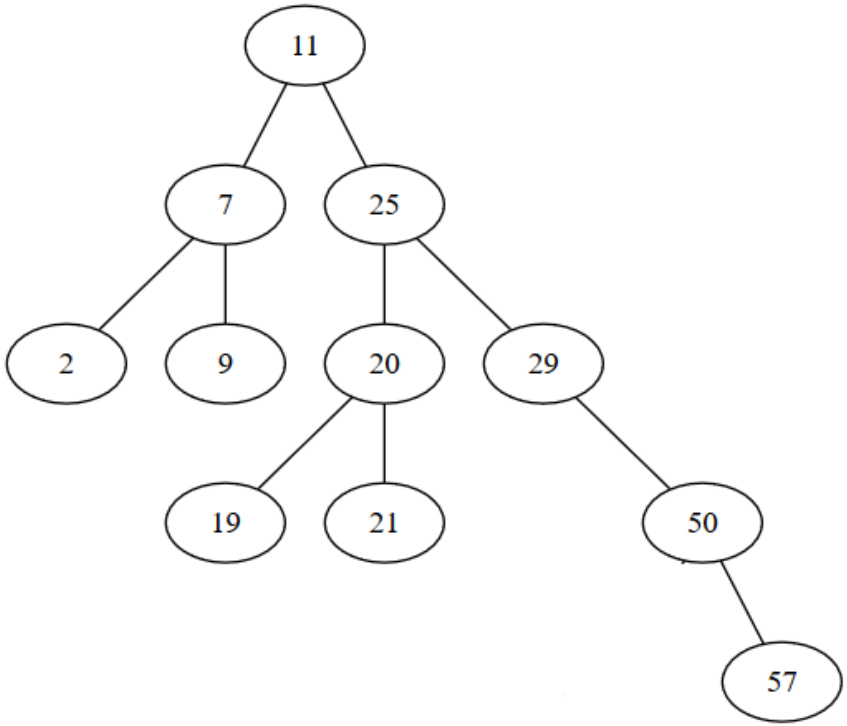
AVL rotations example XXI

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



- Insert 57

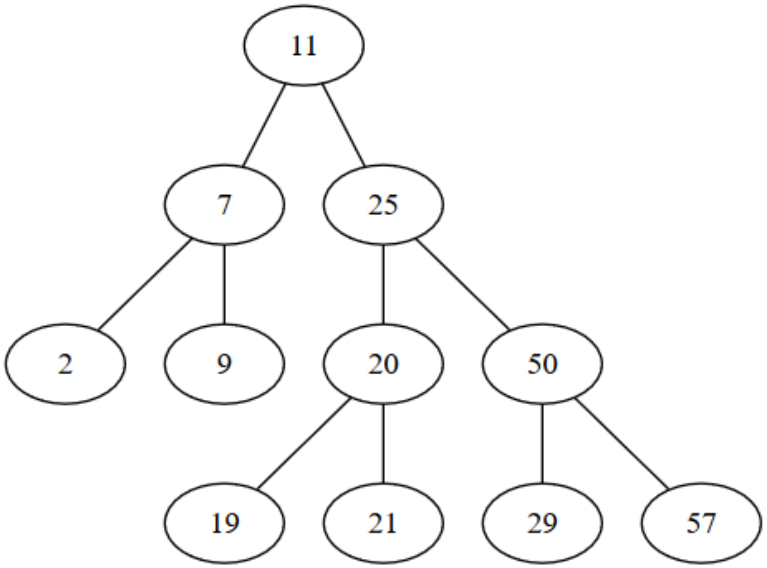
AVL rotations example XXII



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

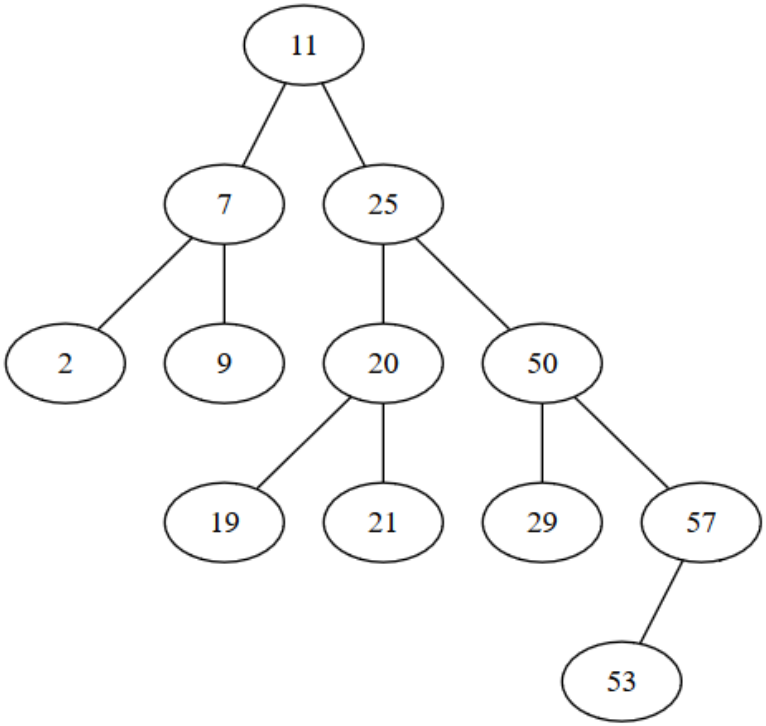
AVL rotations example XXIII

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



- Insert 53

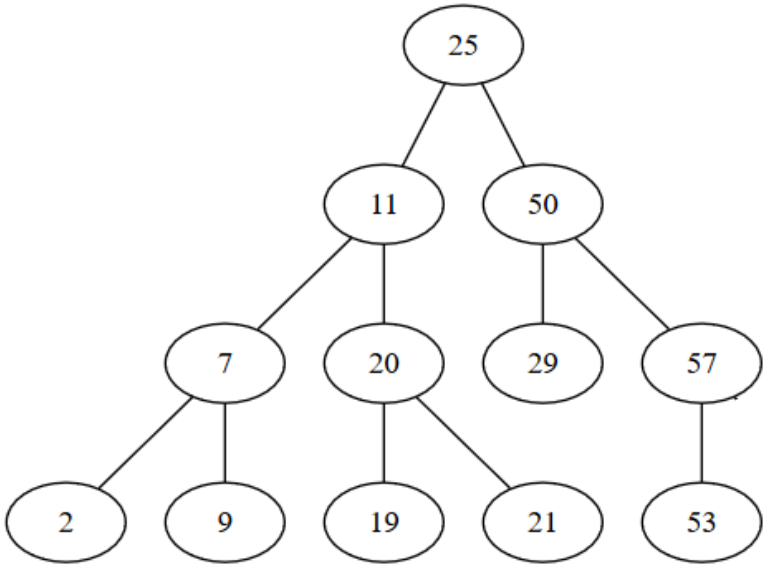
AVL rotations example XXIV



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

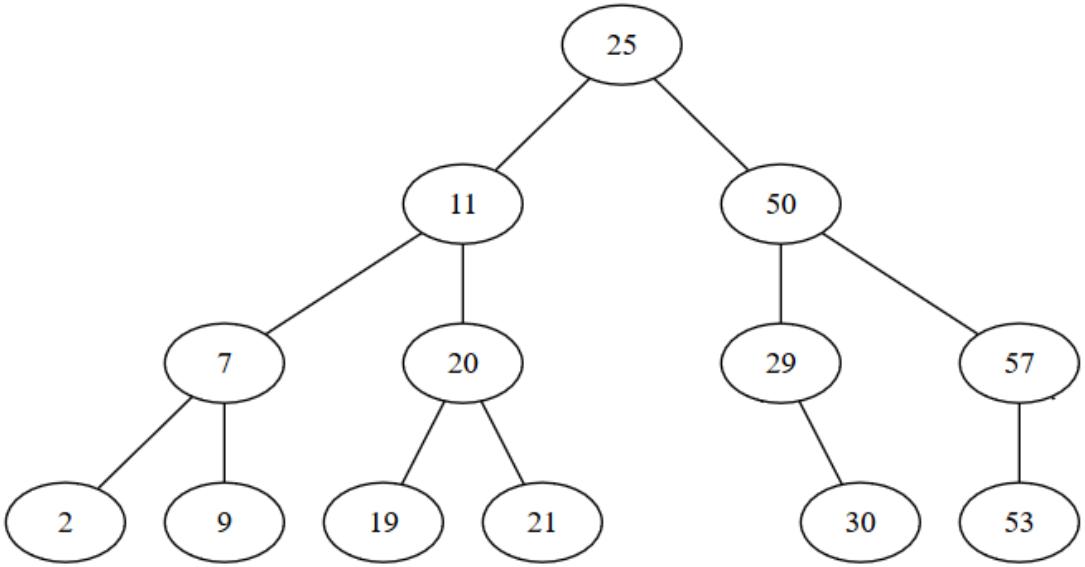
AVL rotations example XXV

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



- Insert 30

AVL rotations example XXVI

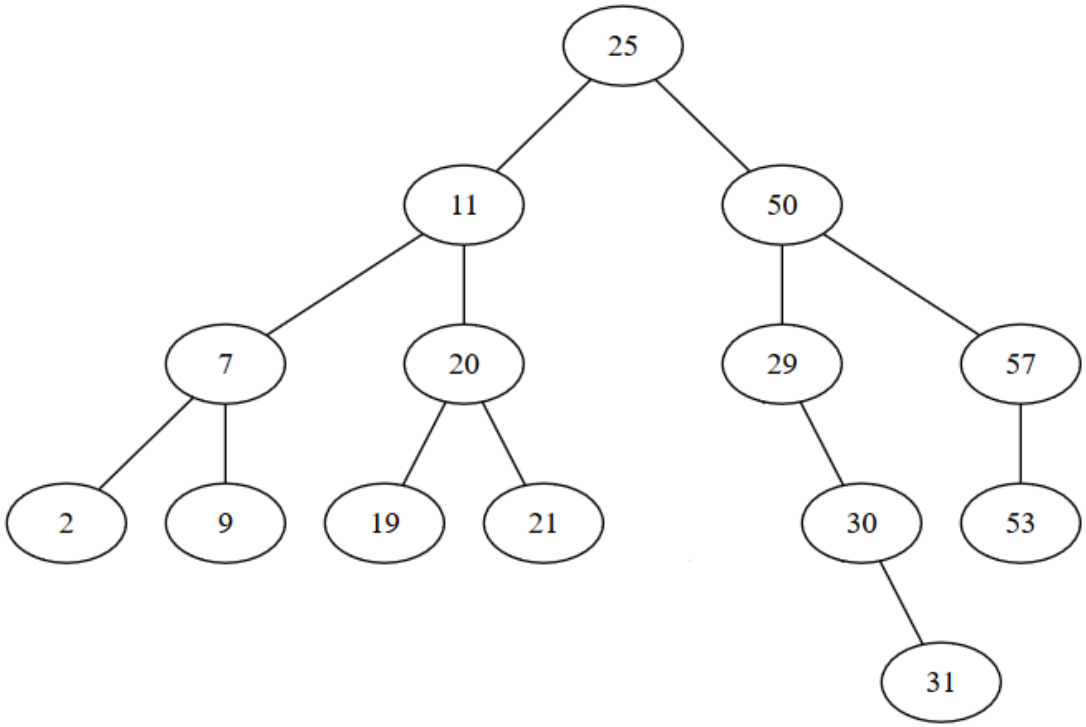


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XXVII

- No rotation is needed
- Insert 31

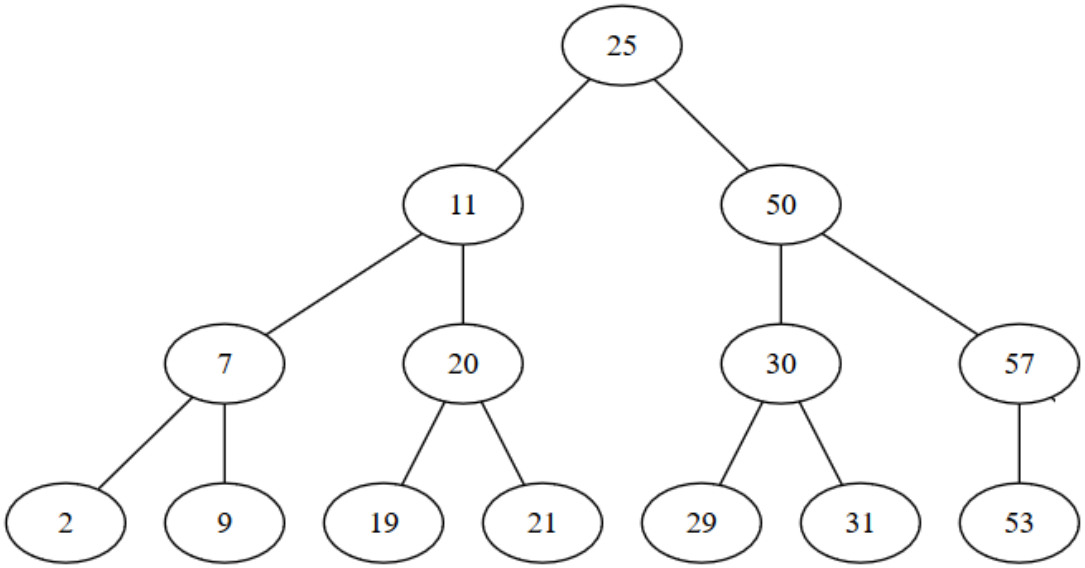
AVL rotations example XXVIII



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

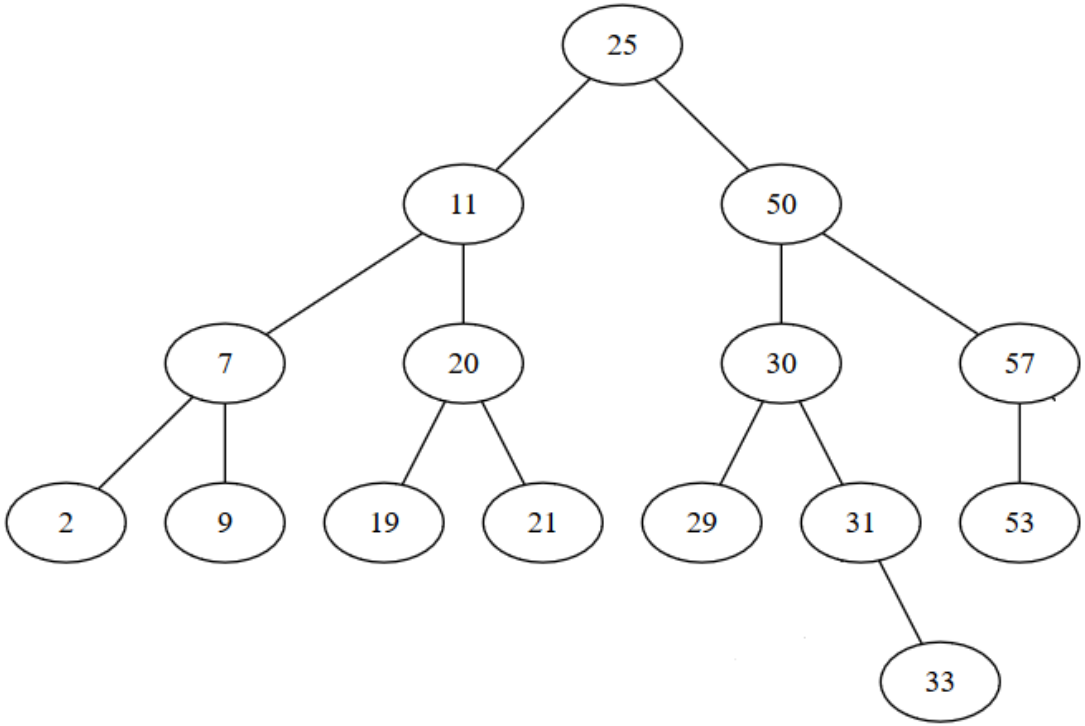
AVL rotations example XXIX

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



- Insert 33

AVL rotations example XXX

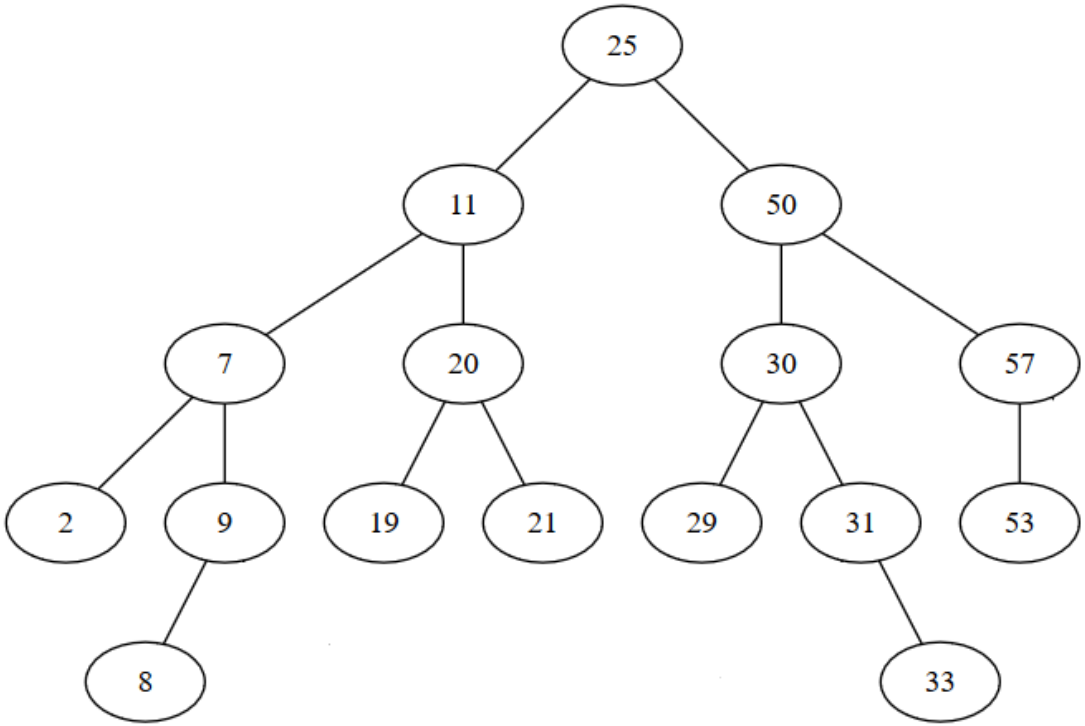


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XXXI

- No rotation is needed
- Insert 8

AVL rotations example XXXII

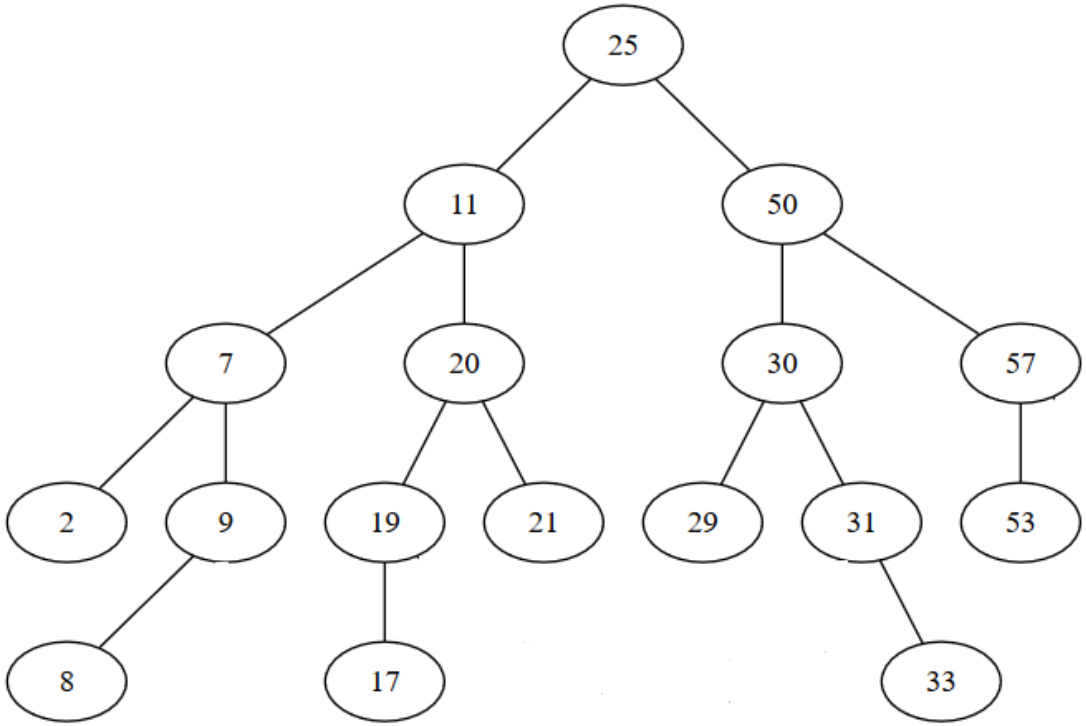


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XXXIII

- No rotation is needed
- Insert 17

AVL rotations example XXXIV

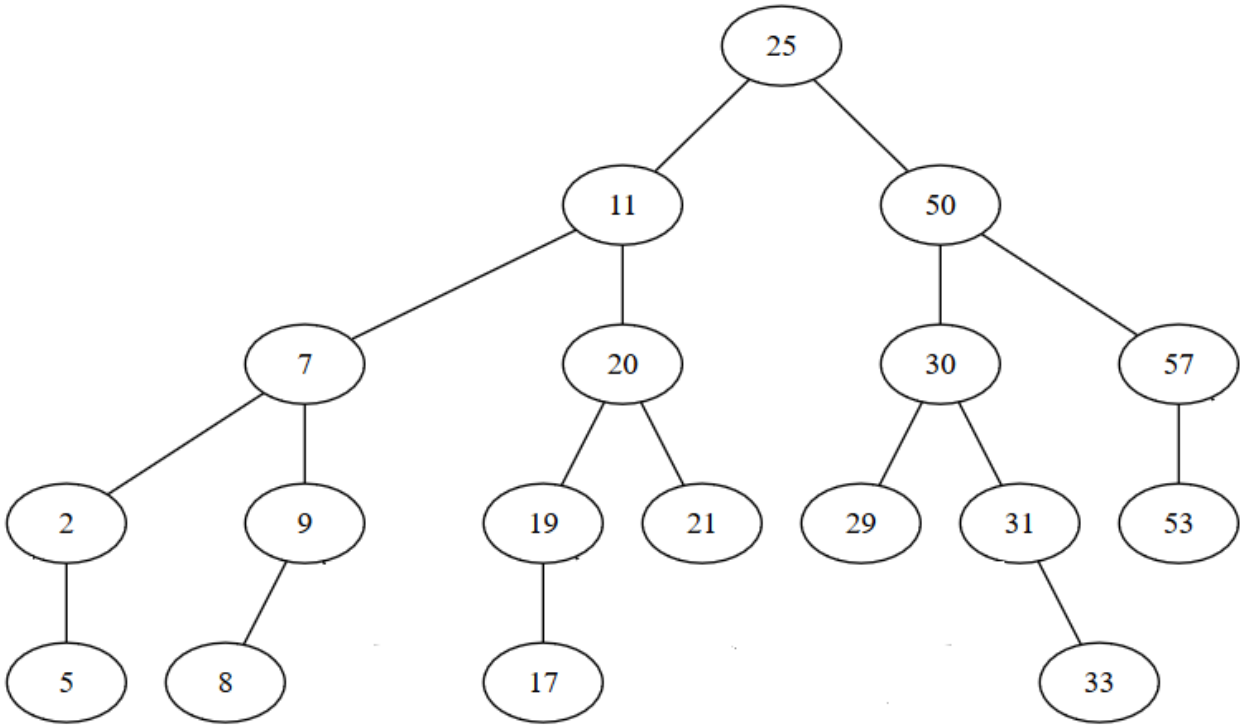


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XXXV

- No rotation is needed
- Insert 5

AVL rotations example XXXVI

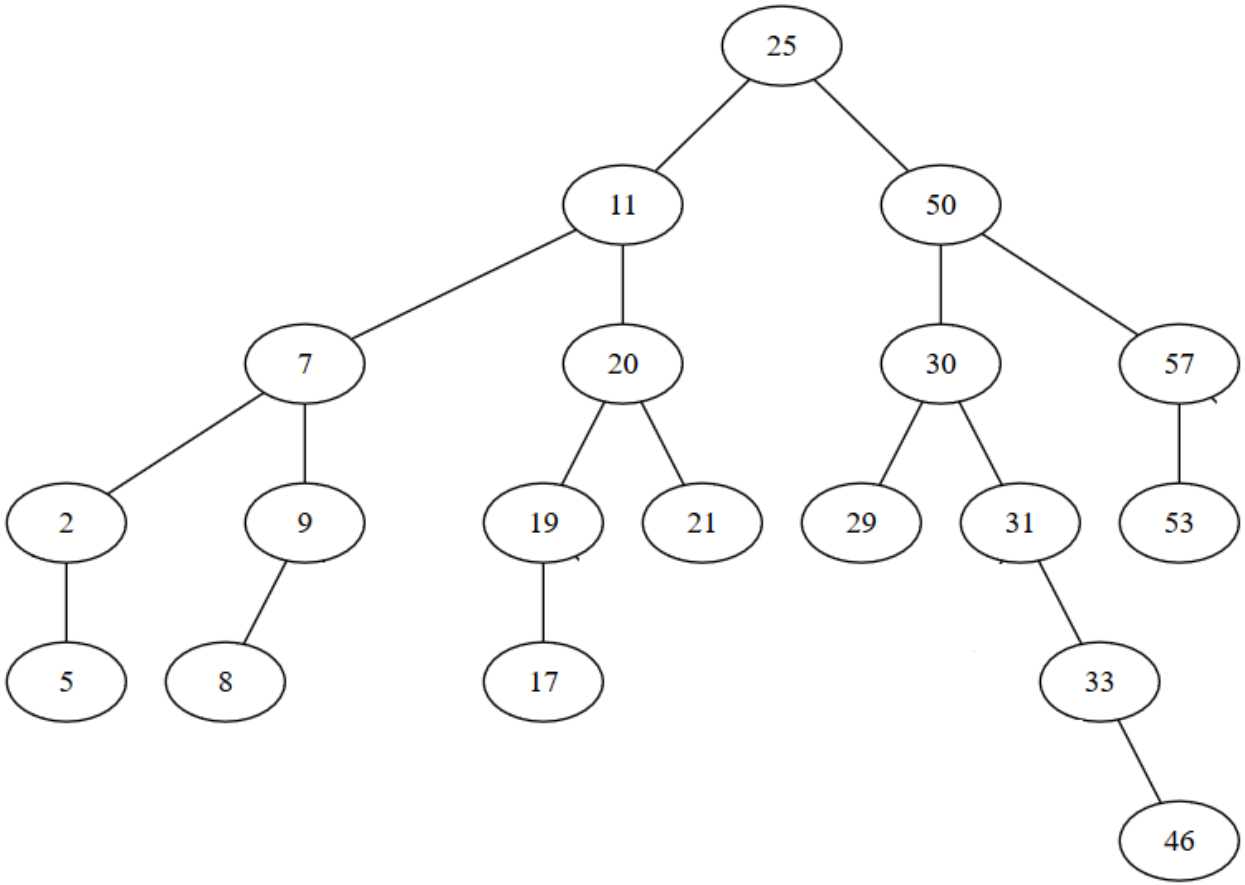


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XXXVII

- No rotation is needed
- Insert 46

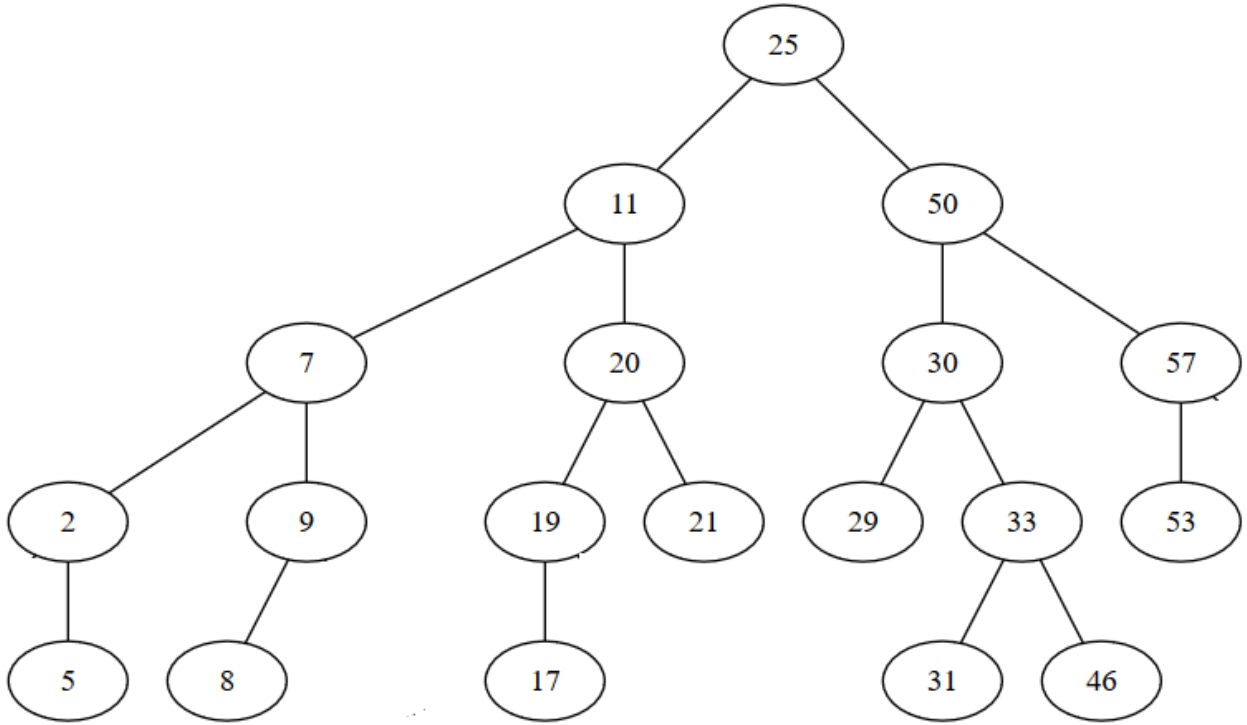
AVL rotations example XXXVIII



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

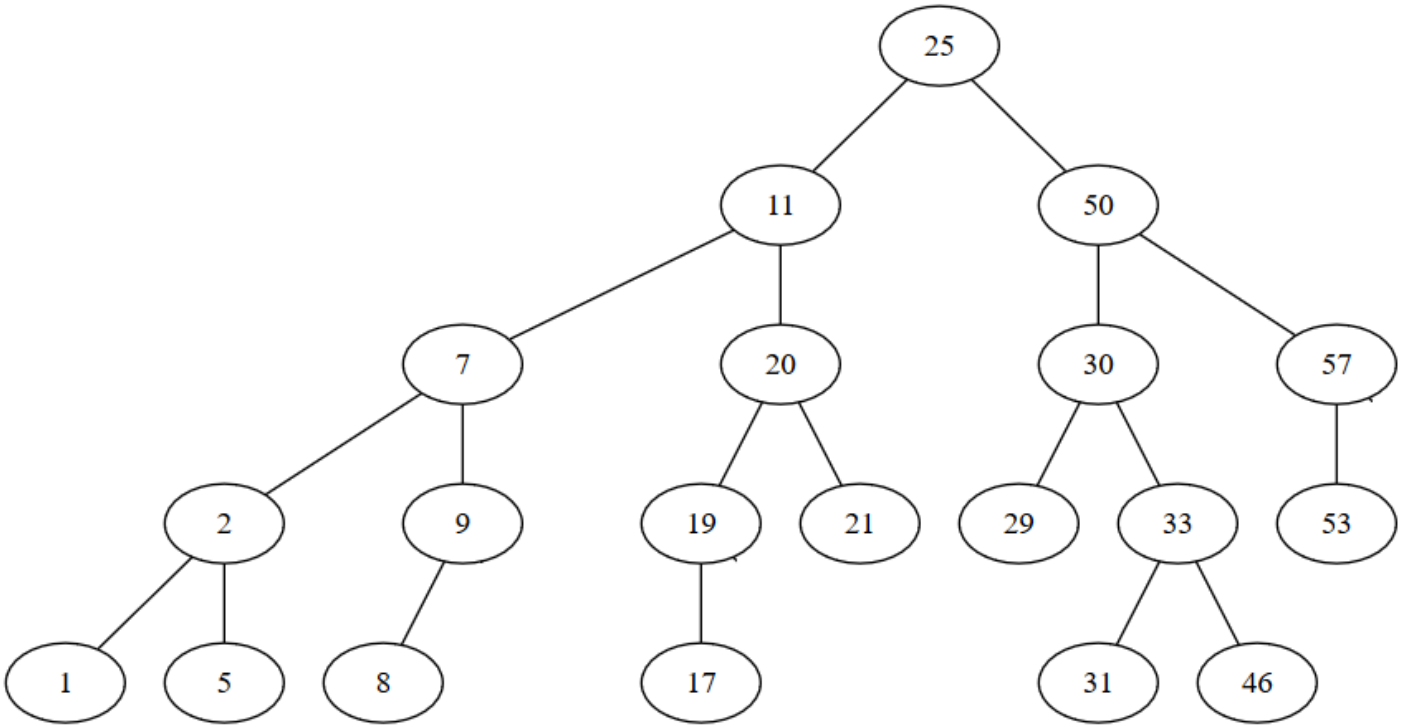
AVL rotations example XXXIX

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



- Insert 1

AVL rotations example XL



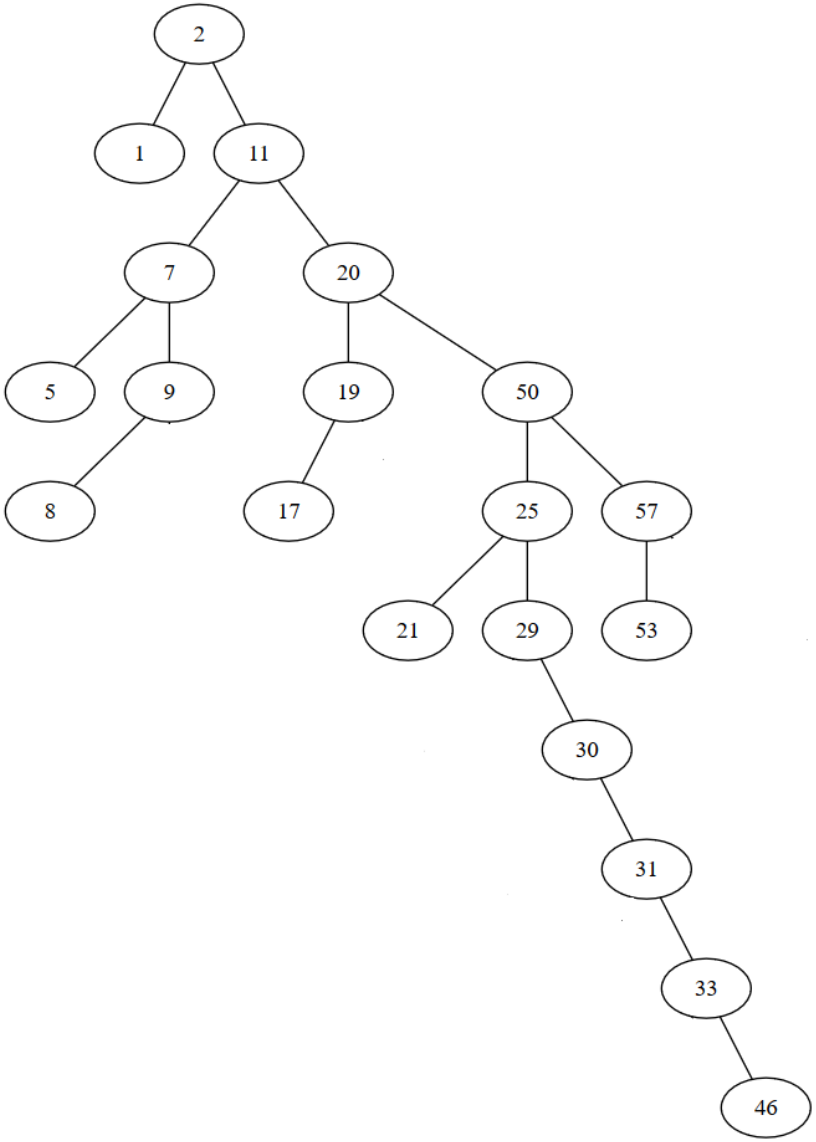
- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XLI

- No rotation is needed

Comparison to BST

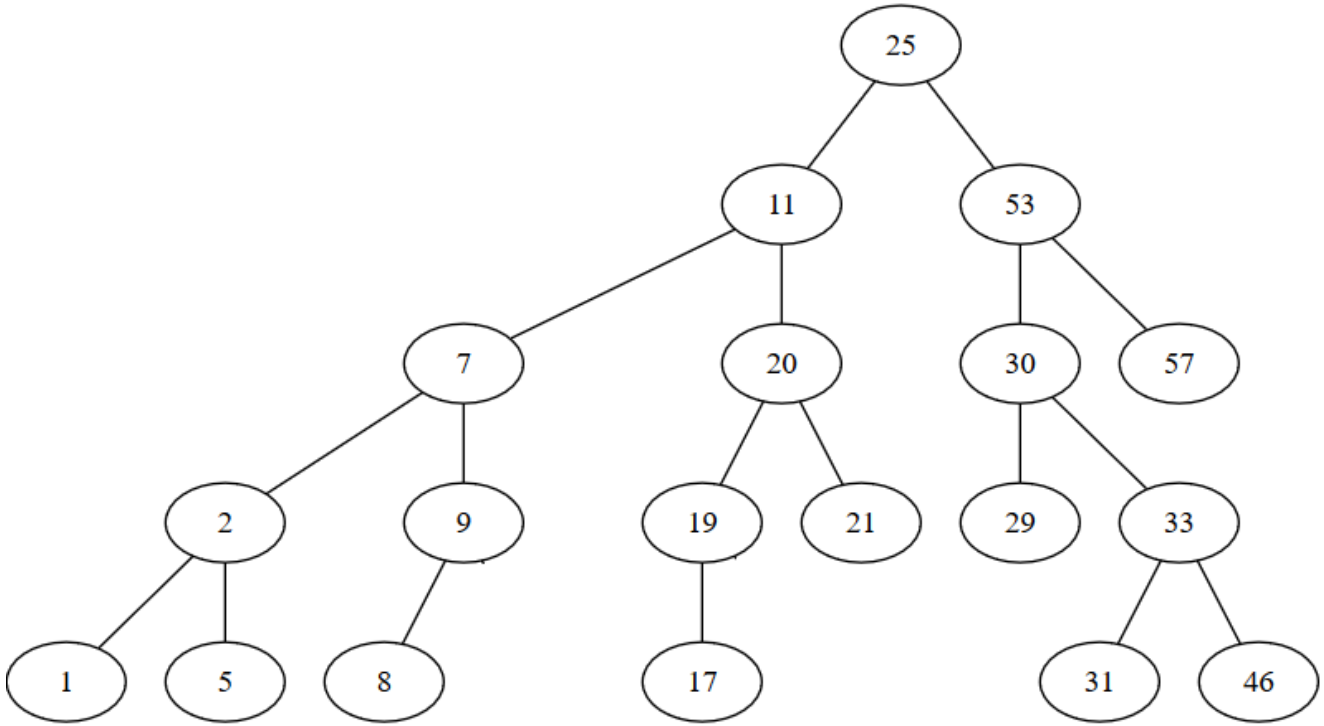
- If, instead of using an AVL tree, we used a binary search tree, after the insertions the tree would have been:


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

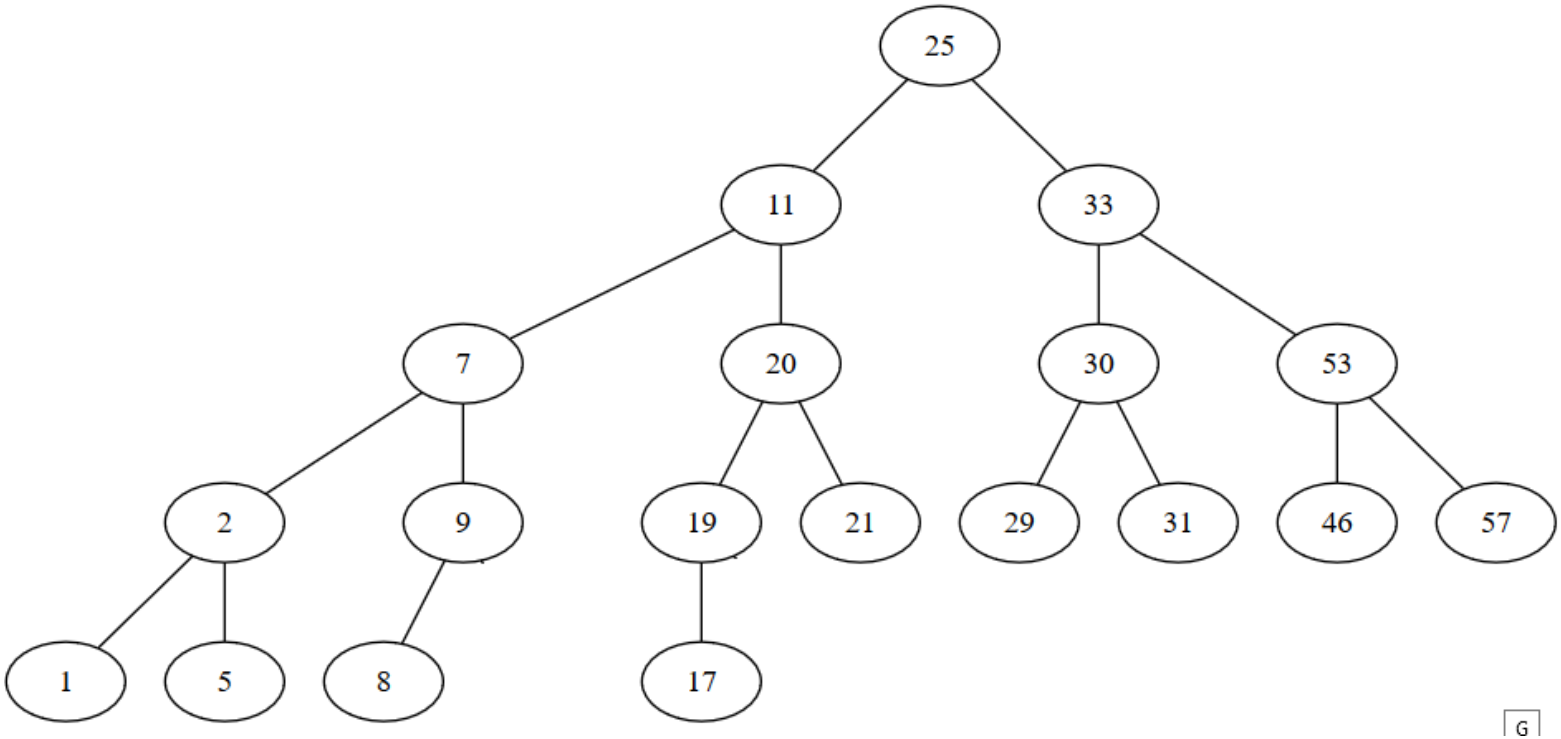
Example of remove II



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

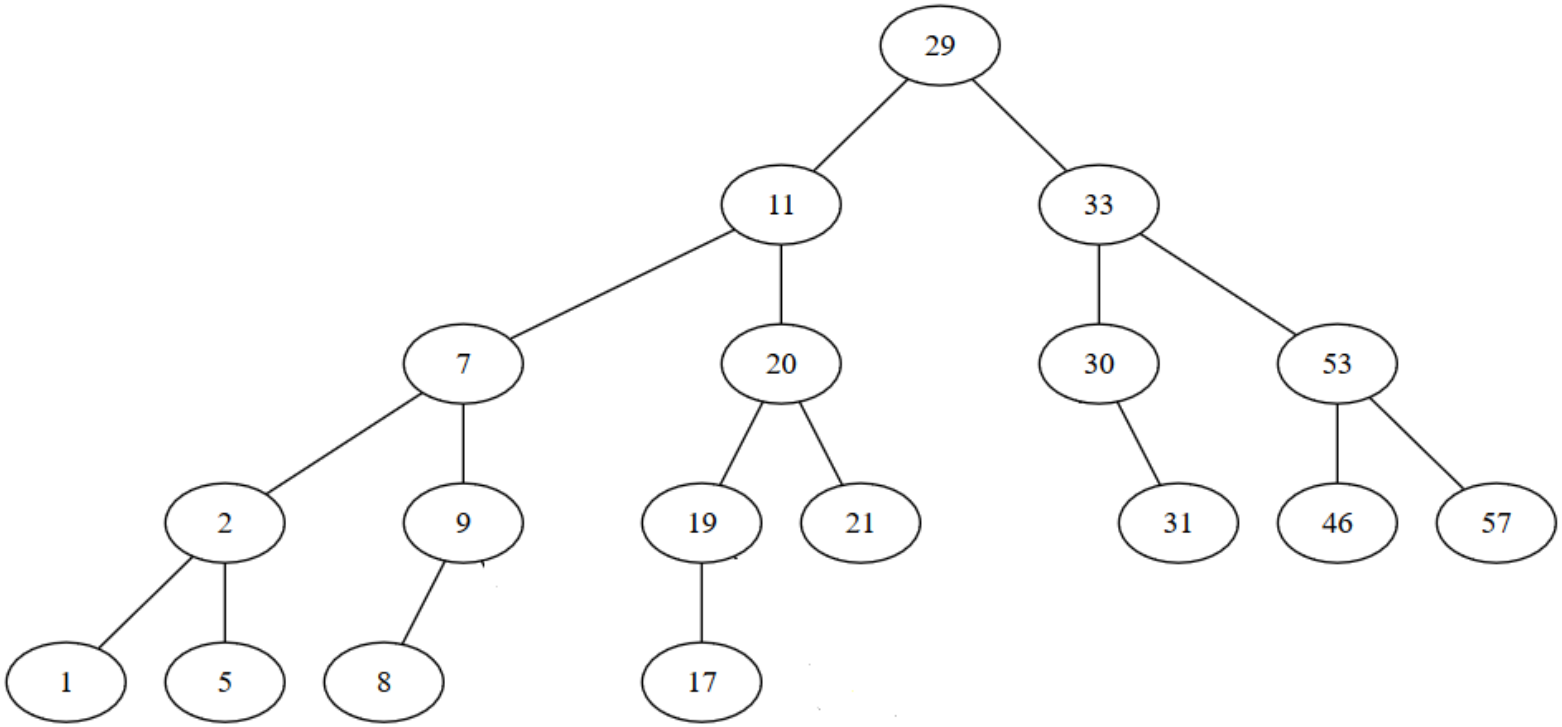
Example of remove III

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



- Remove 25

Example of remove IV

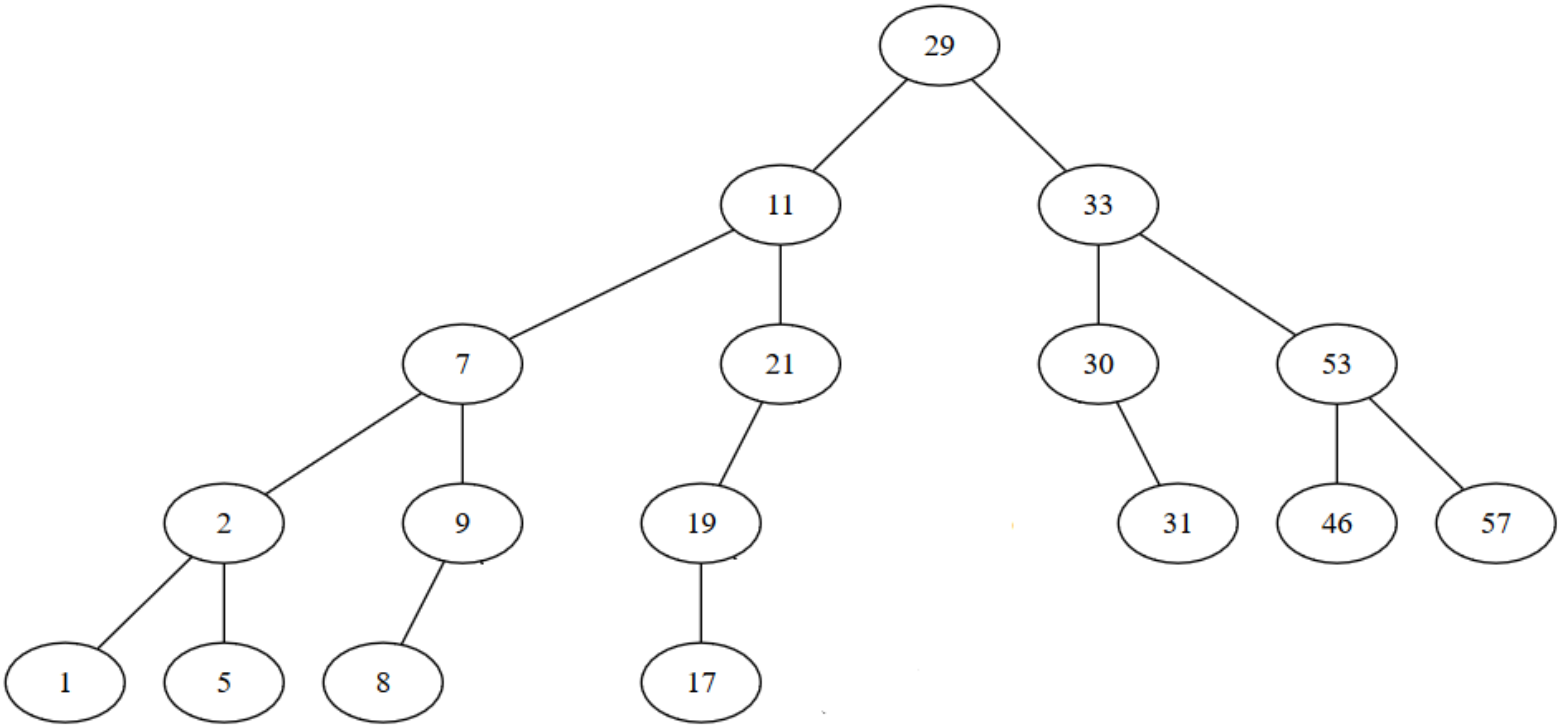


- Do we need a rotation?
- If yes, on which node and what type of rotation?

Example of remove V

- No rotation is needed
- Remove 20

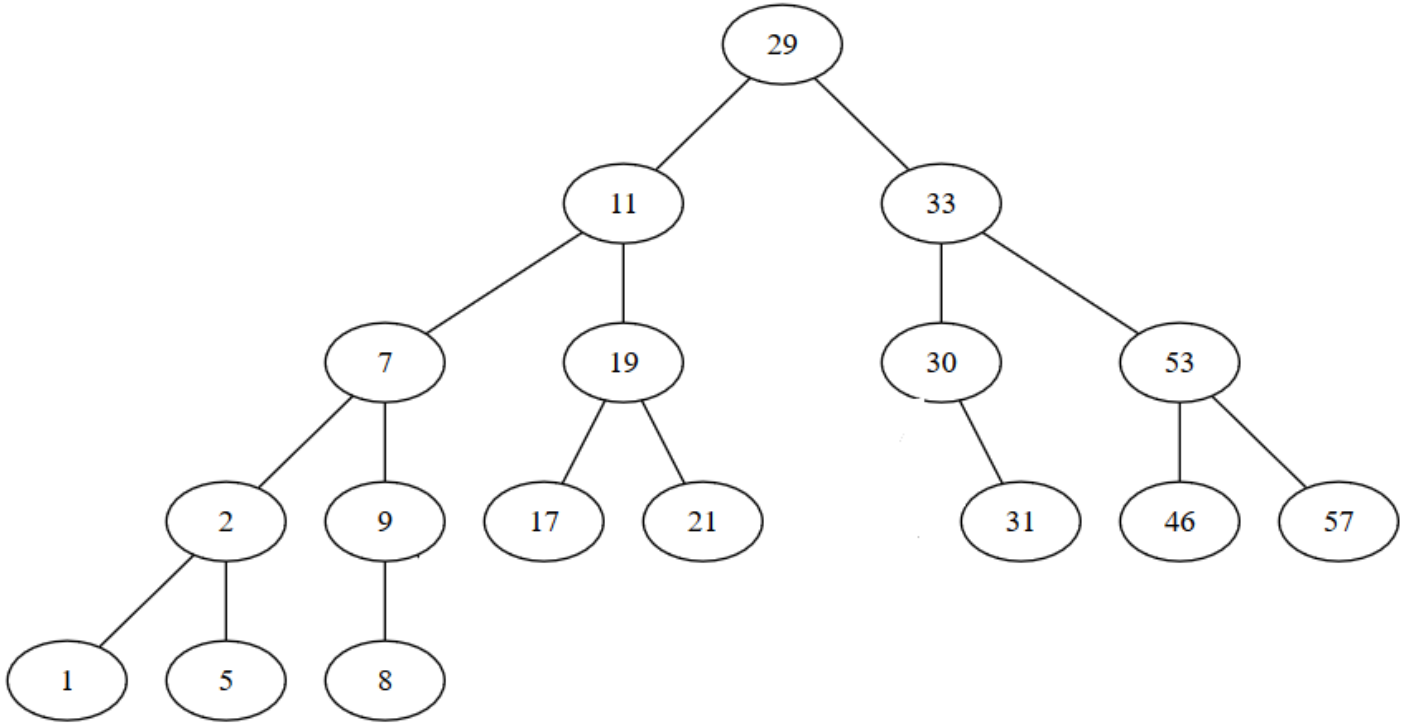
Example of remove VI



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

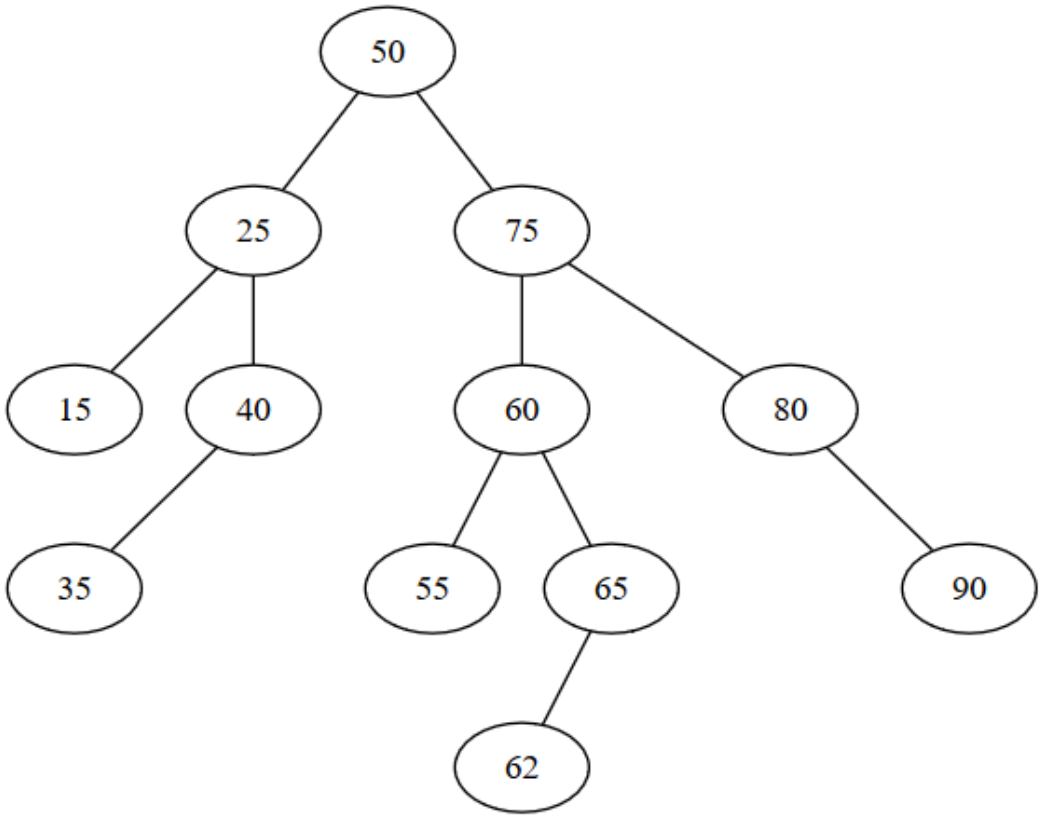
Example of remove VII

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



Rotations for remove

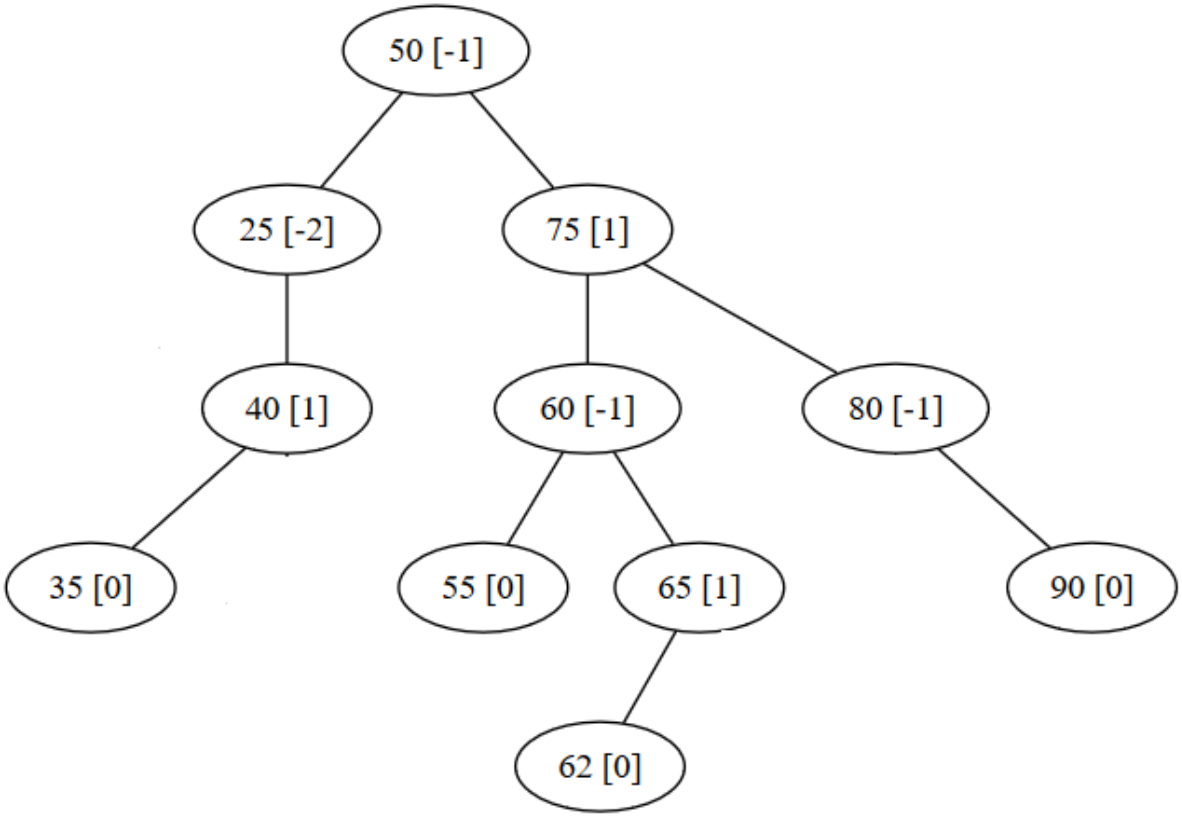
- When we remove a node, we might need more than 1 rotation:



- Remove value 15

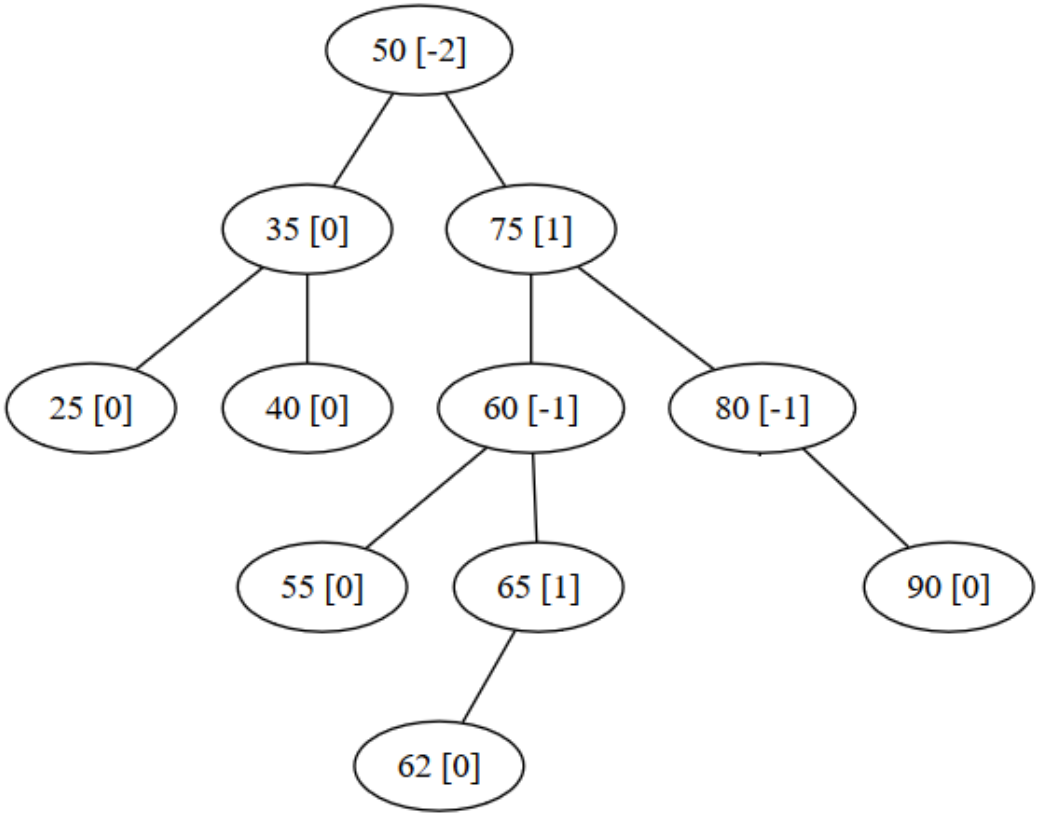
Rotations for remove

- After remove:



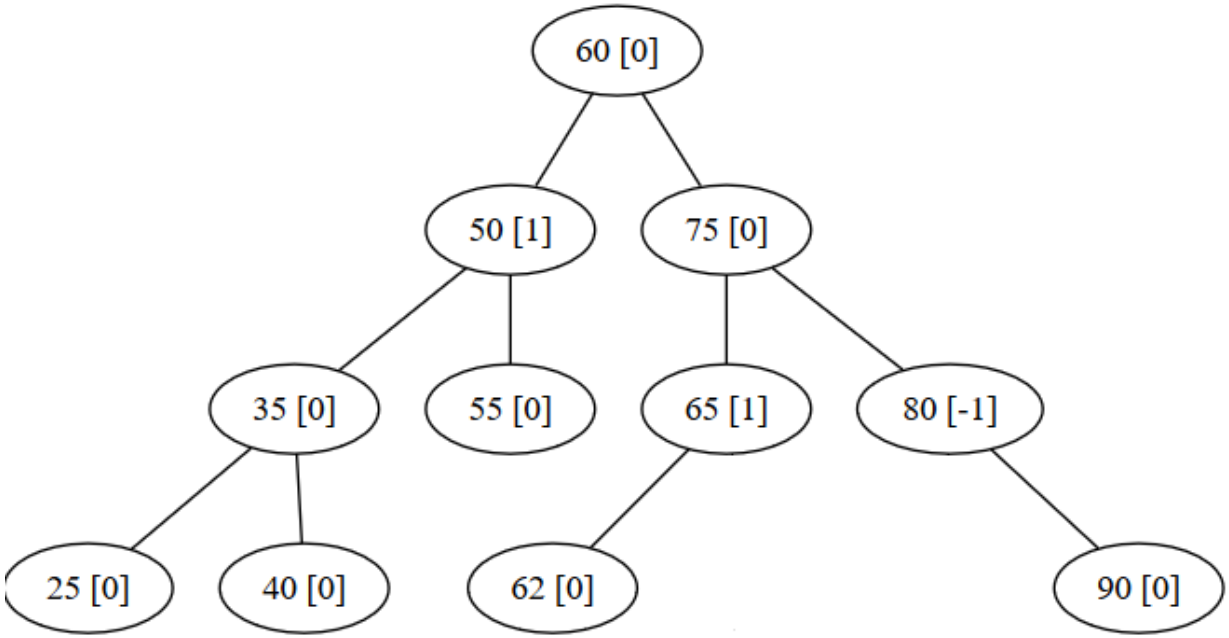
Rotations for remove

- 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: ↑ AVLNode *//address of left child*
right: ↑ AVLNode *//address of right child*
h: Integer *//height of the node*

AVLTree:
root: ↑ 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 ↑ AVLNode. All descendants of node have their height (h) set
//to the correct value
//post: if node ≠ 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 ↑ AVLNode. All descendants of node have their height (h) set
//to the correct value
//post: returns the balance factor of the node

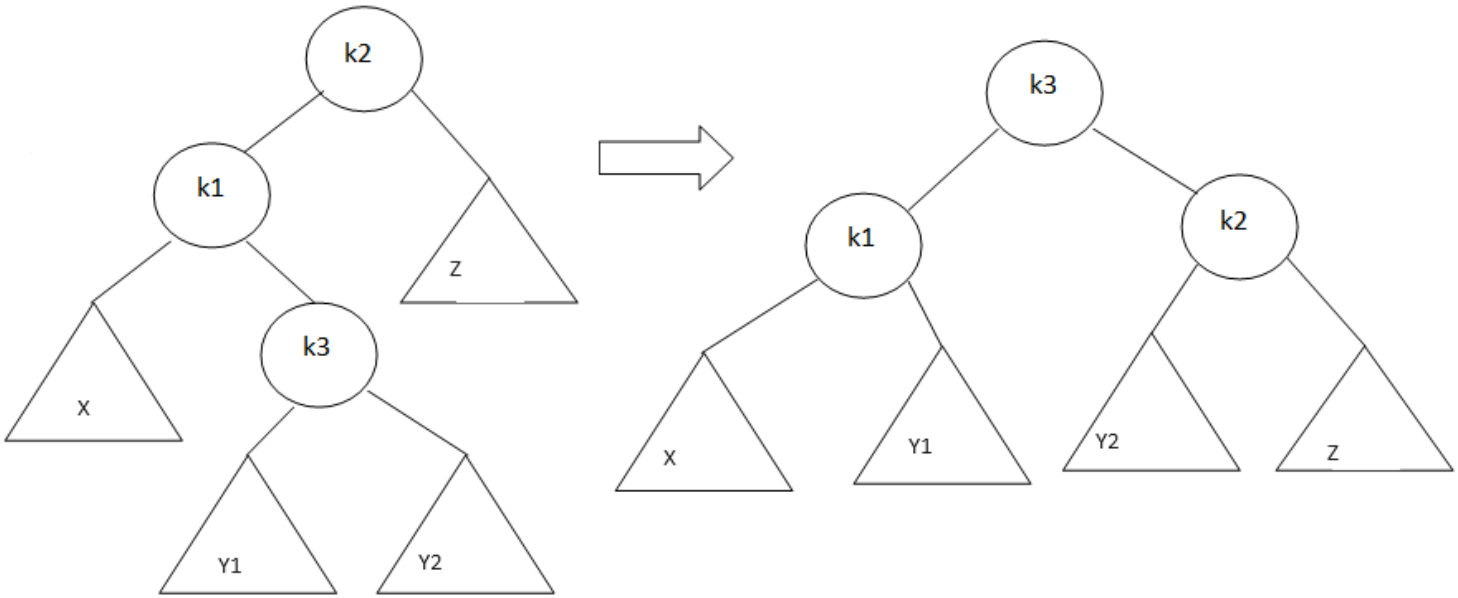
AVL Tree - balance factor of a node

```
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 ↑ 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

AVL Tree - DRR

//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 ↑ 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**

AVL Tree - insert

```
    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).

Cuckoo hashing

Cuckoo hashing

- In cuckoo hashing we have two hash tables of the same size, each of them more than half empty and each hash table has its hash function (so we have two different hash functions).
- For each element to be added we can compute two positions: one from the first hash table and one from the second. In case of cuckoo hashing, it is guaranteed that an element will be on one of these positions.
- Search is simple, because we only have to look at these two positions.
- Delete is simple, because we only have to look at these two positions and set to empty the one where we find the element.

- When we want to insert a new element we will compute its position in the first hash table. If the position is empty, we will place the element there.
- If the position in the first hash table is not empty, we will kick out the element that is currently there, and place the new element into the first hash table.
- The element that was kicked off, will be placed at its position in the second hash table. If that position is occupied, we will kick off the element from there and place it into its position in the first hash table.
- We repeat the above process until we will get an empty position for an element.
- If we get back to the same location with the same key we have a cycle and we cannot add this element $\Rightarrow$ resize, rehash

Cuckoo hashing - example

- Assume that we have two hash tables, with $m = 11$ positions and the following hash functions:
  - h1(k) = k % 11
  - h2(k) = (k div 11) % 11

Position	0	1	2	3	4	5	6	7	8	9	10
T											

Position	0	1	2	3	4	5	6	7	8	9	10
T											

Cuckoo hashing - example

- Insert key 20

Cuckoo hashing - example

- Insert key 20
  - h1(20) = 9 - empty position, element added in the first table
- Insert key 50

Cuckoo hashing - example

- Insert key 20
  - h1(20) = 9 - empty position, element added in the first table
- Insert key 50
  - h1(50) = 6 - empty position, element added in the first table

Position	0	1	2	3	4	5	6	7	8	9	10
T							50			20	

Position	0	1	2	3	4	5	6	7	8	9	10
T											

Cuckoo hashing - example

- Insert key 53

Cuckoo hashing - example

- Insert key 53
  - h1(53) = 9 - occupied
  - 53 goes in the first hash table, and it sends 20 in the second to position h2(20) = 1

Position	0	1	2	3	4	5	6	7	8	9	10
T							50			53	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20									

Cuckoo hashing - example

- Insert key 75

Cuckoo hashing - example

- Insert key 75
  - h1(75) = 9 - occupied
  - 75 goes in the first hash table, and it sends 53 in the second to position h2(53) = 4

Position	0	1	2	3	4	5	6	7	8	9	10
T							50			75	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20			53						

Cuckoo hashing example

- Insert key 100

Cuckoo hashing example

- Insert key 100
  - h1(100) = 1 - empty position
- Insert key 67

Cuckoo hashing example

- Insert key 100
  - h1(100) = 1 - empty position
- Insert key 67
  - h1(67) = 1 - occupied
  - 67 goes in the first hash table, and it sends 100 in the second to position h2(100) = 9

Position	0	1	2	3	4	5	6	7	8	9	10
T		67					50			75	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20			53					100	

Cuckoo hashing example

- Insert key 105

Cuckoo hashing example

- Insert key 105
  - h1(105) = 6 - occupied
  - 105 goes in the first hash table, and it sends 50 in the second to position h2(50) = 4
  - 50 goes in the second hash table, and it sends 53 to the first one, to position h1(53) = 9
  - 53 goes in the first hash table, and it sends 75 to the second one, to position h2(75) = 6

Position	0	1	2	3	4	5	6	7	8	9	10
T		67					105			53	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20			50		75			100	

Cuckoo hashing example

- Insert key 3

Cuckoo hashing example

- Insert key 3
  - h1(3) = 3 - empty position
- Insert key 36

Cuckoo hashing example

- Insert key 3
  - h1(3) = 3 - empty position
- Insert key 36
  - h1(36) = 3 - occupied
  - 36 goes in the first hash table, and it sends 3 in the second to position h2(3) = 0

Position	0	1	2	3	4	5	6	7	8	9	10
T		67		36			105			53	

Position	0	1	2	3	4	5	6	7	8	9	10
T	3	20			50		75			100	

Cuckoo hashing example

- Insert key 39

Cuckoo hashing example

- Insert key 39
  - h1(39) = 6 - occupied
  - 39 goes in the first hash table and it sends 105 in the second to position h2(105) = 9
  - 105 goes to the second hash table and it sends 100 in the first to position h1(100) = 1
  - 100 goes in the first hash table and it sends 67 in the second to position h2(67) = 6
  - 67 goes in the second hash table and it sends 75 in the first to position h1(75) = 9
  - 75 goes in the first hash table and it sends 53 in the second to position h2(53) = 4
  - 53 goes in the second hash table and it sends 50 in the first to position h1(50) = 6
  - 50 goes in the first hash table and it sends 39 in the second to position h2(39) = 3

Cuckoo hashing example

Position	0	1	2	3	4	5	6	7	8	9	10
T		100		36			50			75	

Position	0	1	2	3	4	5	6	7	8	9	10
T	3	20		39	53		67			105	

Cuckoo hashing

- It can happen that we cannot insert a key because we get in a cycle. In these situation we have to increase the size of the tables and rehash the elements.
- While in some situation insert moves a lot of elements, it can be shown that if the load factor of the tables is below 0.5, the probability of a cycles is low and it is very unlikely that more than $O(log_2 n)$ elements will be moved.

Cuckoo hashing

- If we use two tables and each position from a table holds one element at most, the tables have to have load factor below 0.5 to work well.
- If we use three tables, the tables can have load factor of 0.91 and for 4 tables we have 0.97

Perfect hashing

Perfect hashing

- Assume that we know all the keys in advance and we use *separate chaining* for collision resolution $\Rightarrow$ the more lists we make, the shorter the lists will be (reduced number of collisions) $\Rightarrow$ if we could make a large number of list, each would have one element only (no collision).
- How large should we make the hash table to make sure that there are no collisions?
- If $M = N^2$, it can be shown that the table is collision free with probability at least $1/2$.
- Start building the hash table. If you detect a collision, just choose a new hash function and start over (expected number of trials is at most 2).

Perfect hashing

- Having a table of size N^2 is impractical.
- Solution instead:
  - Use a hash table of size N (*primary* hash table).
  - Instead of using linked list for collision resolution (as in separate chaining) each element of the hash table is another hash table (*secondary hash table*)
  - Make the secondary hash table of size ${n_j}^2$, where n_j is the number of elements from this hash table.
  - Each secondary hash table will be constructed with a different hash function, and will be reconstructed until it is collision free.
- This is called **perfect hashing**.
- It can be shown that the total space needed for the secondary hash tables is at most $2N$.

Perfect hashing

- Perfect hashing requires multiple hash functions, this is why we use *universal hashing*.

Perfect hashing

- Perfect hashing requires multiple hash functions, this is why we use *universal hashing*.
- Let p be a prime number, larger than the largest possible key.
- The universal hash function family $\mathcal{H}$ can be defined as:

$$\mathcal{H} = \{H_{a,b}(x) = ((a * x + b) \ \% \ p) \ \% \ m)$$

$$\text{where } 1 \leq a \leq p-1, 0 \leq b \leq p-1$$

- a and b are chosen randomly when the hash function is initialized.

Perfect hashing - example

- Insert into a hash table with perfect hashing the letters from "PERFECT HASHING EXAMPLE". Since we want no collisions at all, we are going to consider only the unique letters: "PERFCTHASINGXML"
- Since we are inserting $N = 15$ elements, we will take $m = 15$.
- For each letter, the *hashCode* is the index of the letter in the alphabet.

Letter	P	E	R	F	C	T	H	A	S	I	N	G	X	M	L
HashCode	16	5	18	6	3	20	8	1	19	9	14	7	24	13	12

Perfect hashing - example

- p has to be a prime number larger than the maximum key $\Rightarrow$ 29
- The hash function will be:

$$H_{a,b}(x) = ((a * x + b) \ \% \ p) \ \% \ m$$

- where a will be 3 and b will be 2 (chosen randomly).

Letter	P	E	R	F	C	T	H	A	S	I	N	G	X	M	L
HashCode	16	5	18	6	3	20	8	1	19	9	14	7	24	13	12

Perfect hashing - example

- p has to be a prime number larger than the maximum key $\Rightarrow$ 29
- The hash function will be:

$$H_{a,b}(x) = ((a * x + b) \ \% \ p) \ \% \ m$$

- where a will be 3 and b will be 2 (chosen randomly).

Letter	P	E	R	F	C	T	H	A	S	I	N	G	X	M	L
HashCode	16	5	18	6	3	20	8	1	19	9	14	7	24	13	12
H(HashCode)	6	2	12	5	11	4	11	5	1	0	0	8	1	12	9

Perfect hashing - example

- Collisions:
  - position 0 - I, N
  - position 1 - S, X
  - position 2 - E
  - position 4 - T
  - position 5 - F, A
  - position 6 - P
  - position 8 - G
  - position 9 - L
  - position 11 - C, H
  - position 12 - R, M

Perfect hashing - example

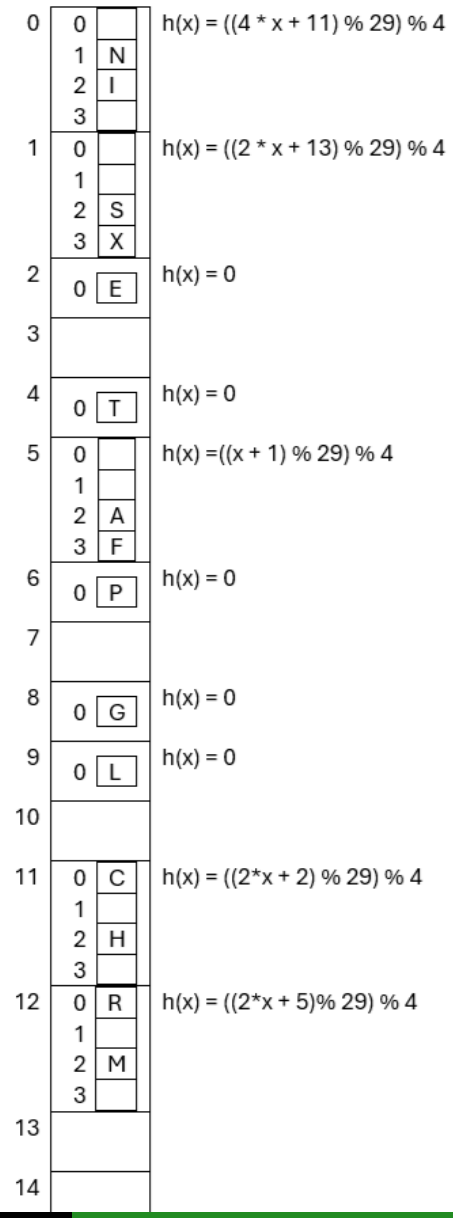
- For the positions where we have no collision (only one element hashed to it) we will have a secondary hash table with only one element, and hash function $h(x) = 0$
- For the positions where we have two elements, we will have a secondary hash table with 4 positions and different hash functions, taken from the same universe, with different random values for a and b.
- For example for position 0, we can define $a = 4$ and $b = 11$ and we will have:
  $h(I) = h(9) = 2$
  $h(N) = h(14) = 1$

Perfect hashing - example

- Assume that for the secondary hash table from position 1 we will choose $a = 5$ and $b = 2$.
- Positions for the elements will be:
  $h(S) = h(19) = ((5 * 19 + 2)\%29)\%4 = 2$
  $h(X) = h(24) = ((5 * 24 + 2)\%29)\%4 = 2$
- In perfect hashing we should not have collisions, so we will simply chose another hash function: another random values for a and b. Choosing for example $a = 2$ and $b = 13$, we will have h(S) = 2 and h(X) = 3.

- The fully built table:

Hash function for primary table: h(x) = ((3*x + 2) % 29 % 15



Perfect hashing

- When perfect hashing is used and we search for an element we will have to check at most 2 positions (position in the primary and in the secondary table).
- This means that worst case performance of the table is $\Theta(1)$.
- But in order to use perfect hashing, we need to have static keys: once the table is built, no new elements can be added.

Summary

- Today we have talked about
  - AVL trees
  - Cucko hashing
  - Perfect hashing