Datenstrukturen und Algorithmen

Vorlesung 12

Überblick

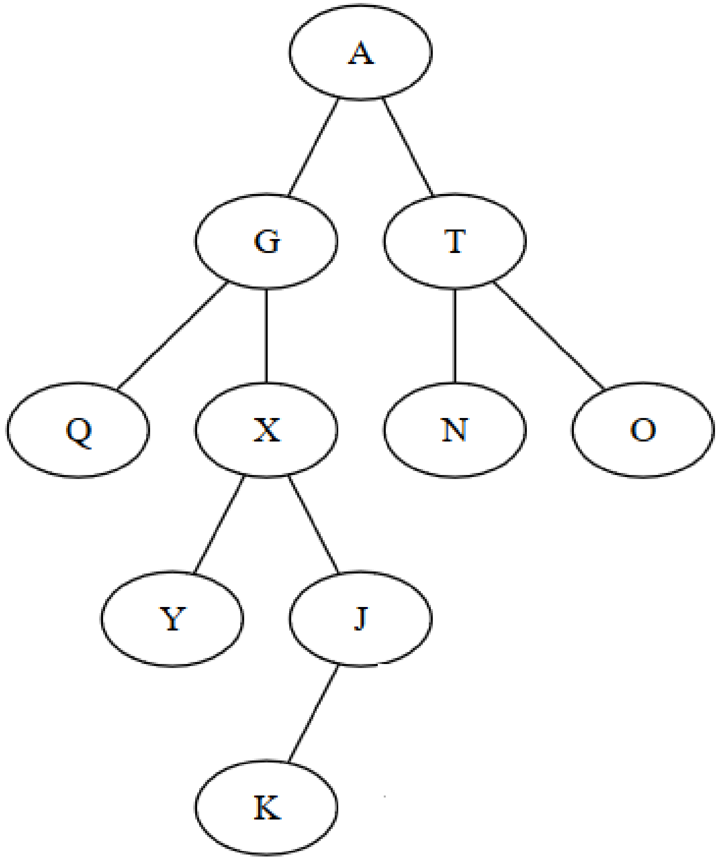
- Vorige Woche:
  - Hashtabellen: Hopscotch hashing
  - Bäume - allgemeines
  - Binärbäume
    - Einführung
    - ADT Binary Tree
    - Traversierungen in Präordnung und Inordnung
- Heute betrachten wir:
  - Binärbäume: Traversierungen in Postordnung und Level-Ordnung
  - Binärsuchbäume
  - AVL-Bäume

Traversieren in Postordnung (postorder)

- Postordnung Reihenfolge:
  - Durchlaufe linken Teilbaum
  - Durchlaufe rechten Teilbaum
  - Besuche Wurzel

- Bei der Traversierung der Teilbäume gilt dieselbe Reihenfolge

Traversieren in Postordnung - Beispiel



- Traversieren in Postordnung: Q, Y, K, J, X, G, N, O, T, A

Traversieren in Postordnung– rekursive Implementierung

```
subalgorithm postorder recursive(node) is:
//pre: node ist ein ↑ BTNode
    if node ≠ NIL then
        postorder_recursive([node].left)
        postorder_recursive([node].right)
        @visit [node].info
    end-if
end-subalgorithm
```

- Man braucht wieder einen Wrapper-Algorithmus
- Die Traversierung braucht $\Theta(n)$ Zeit für einen Binärbaum mit n Knoten

Traversieren in Postordnung – nicht-rekursive Implementierung

- Man kann den Traversierungs-Algorithmus in Postorder auch iterativ implementieren, aber es ist ein bisschen komplizierter als Traversieren in Präordnung und Inordnung

- Es gibt eine Implementierung **mit zwei Stacks** oder eine Implementierung **mit einem Stack**

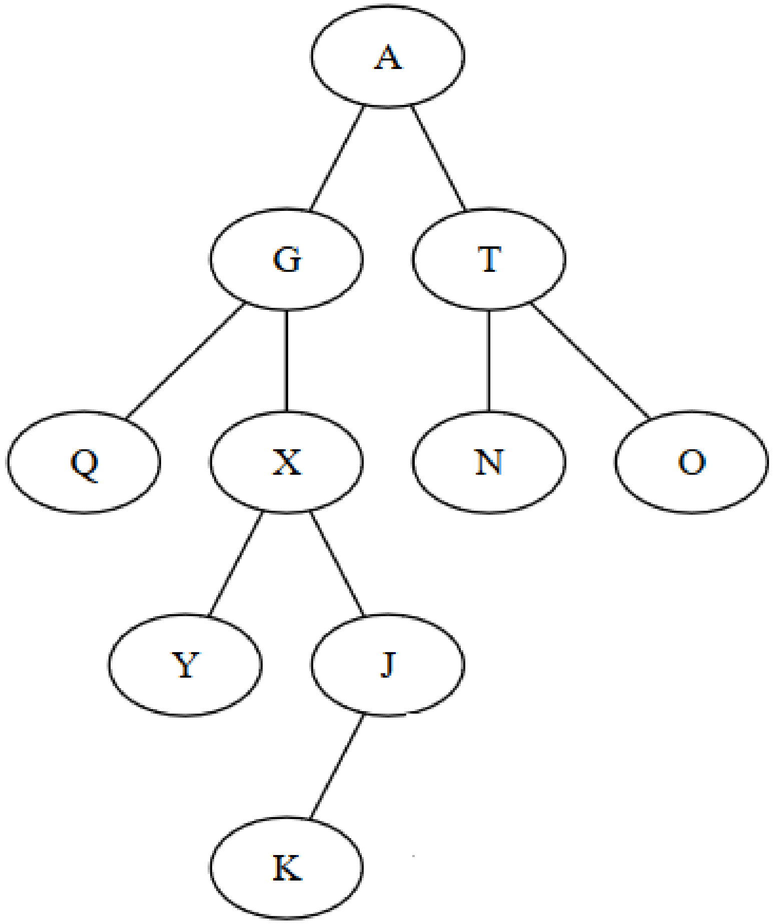
Traversieren in Postordnung mit **zwei** Stacks

- Die Hauptidee ist den Gegenteil der Traversierung in Postordnung in einem Stack aufzubauen und nachher der Reihe nach *pop* zu benutzen um die Knoten in Postorder zu traversieren
- Das Gegenteil der Traversierung in Postordnung ist ähnlich zu der Traversierung in Präordnung, aber das rechte Teilbaum muss als erster besucht werden anstatt das linke Teilbaum.
- Der Algorithmus ähnelt der Traversierung in Präordnung mit zwei Änderungen:
  - Wenn ein Knoten aus dem Stack entfernt wird, dann wird dieser in den zweiten Stack eingefügt (anstatt besucht zu werden)
  - Für einen Knoten, der aus dem Stack entfernt wird, wird zuerst das linke Kind und dann das rechte Kind in dem Stack eingefügt

Traversieren in Postordnung mit **einem** Stack

- Man fängt mit einem leeren Stack an und einem *currentNode,* der auf die Wurzel gesetzt wird
- Solange *currentNode* nicht NIL ist, füge das rechte Kind und *currentNode* in den Stack ein und setze *currentNode* auf das linke Kind
- Solange der Stack nicht leer ist:
  - Pop einen Knoten aus dem Stack (*currentNode*)
  - Falls *currentNode* ein rechtes Kind hat, der Stack nicht leer ist und er enthält das rechte Kind in dem Top, dann pop das rechte Kind, push *currentNode* und setze *currentNode* auf das rechte Kind
  - Ansonsten besuche *currentNode* und setze ihn auf NIL
  - Solange *currentNode* nicht NIL ist, push in dem Stack das rechte Kind von *currentNode* und *currentNode*, und setze *currentNode* auf das linke Kind

Traversieren in Postordnung mit **einem** Stack - Beispiel



- CurrentNode: A, Stack:
- CurrentNode: NIL, Stack: T A X G Q
- Besuche Q, currentNode: NIL, Stack: T A X G
- CurrentNode: X, Stack: T A G
- CurrentNode: NIL, Stack: T A G J X Y
- Besuche Y, CurrentNode: NIL, Stack: T A G J X
- CurrentNode: J, Stack: T A G X
- CurrentNode: NIL, Stack: T A G X J K
- Besuche K, CurrentNode: NIL, Stack: T A G X J
- Besuche J, CurrentNode: NIL, Stack: T A G X
- Besuche X, CurrentNode: NIL, Stack: T A G
- Besuche G, CurrentNode: NIL, Stack: T A
- CurrentNode: T, Stack: A
- CurrentNode: Nill, Stack: A O T N
- ...

Solange currentNode nicht NIL ist, füge das rechte Kind und currentNode in den Stack ein und setze currentNode auf das linke Kind

Solange der Stack nicht leer ist:

- Pop einen Knoten aus dem Stack (*currentNode*)
- Falls *currentNode* ein rechtes Kind hat, der Stack nicht leer ist und er enthält das rechte Kind in dem Top, dann pop das rechte Kind, push *currentNode* und setze *currentNode* auf das rechte Kind
- Ansonsten besuche *currentNode* und setze ihn auf NIL
- Solange *currentNode* nicht NIL ist, push in dem Stack das rechte Kind von *currentNode* und *currentNode*, und setze *currentNode* auf das linke Kind

Traversieren in Postordnung mit einem Stack

```
subalgorithm postorder(tree) is:
//pre: tree ist ein Binärbaum
    s: Stack //s ist ein Hilfs-Stack
    node ← tree.root
    while node ≠ NIL execute
        if [node].right ≠ NIL then
            push(s, [node].right)
        end-if
        push(s, node)
        node ← [node].left
    end-while
    while not isEmpty(s) execute
        node ← pop(s)
        if [node].right ≠ NIL and (not isEmpty(s)) and [node].right = top(s) then
            pop(s)
            push(s, node)
            node ← [node].right
//Fortsetzung auf der nächsten Folie
```

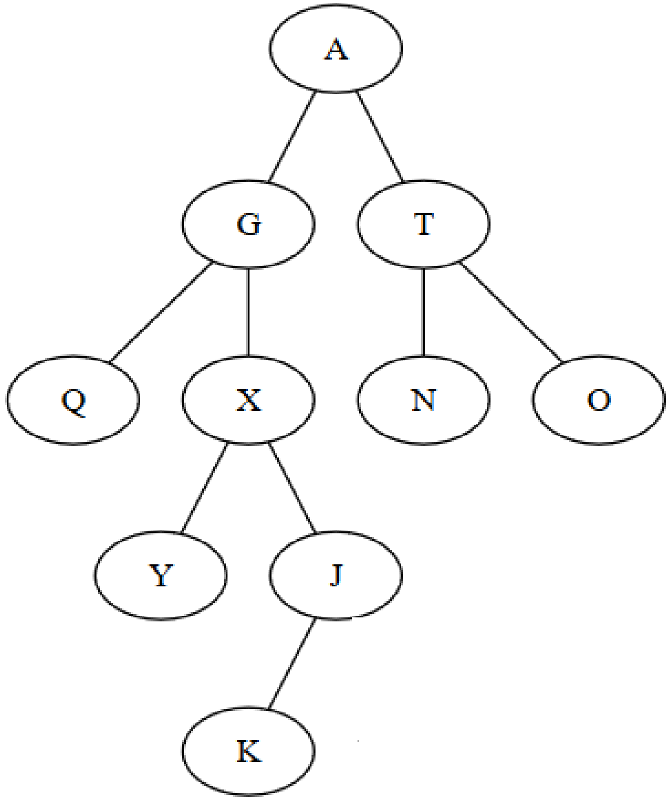
Traversieren in Postordnung mit einem Stack

```
            else
                @visit node
                node ← NIL
            end-if
            while node ≠ NIL execute
                if [node].right ≠ NIL then
                    push(s, [node].right)
                end-if
                push(s, node)
                node ← [node].left
            end-while
        end-while
    end-subalgorithm
```

- Zeitkomplexität: $\Theta(n)$, Speicherplatzkomplexität: $O(n)$

Traversieren in Level-Ordnung

- Im Falle des Traversierens in Level-Ordnung wird am Anfang die Wurzel besucht, dann alle Kinder der Wurzel, dann die Kinder der Kinder, usw.

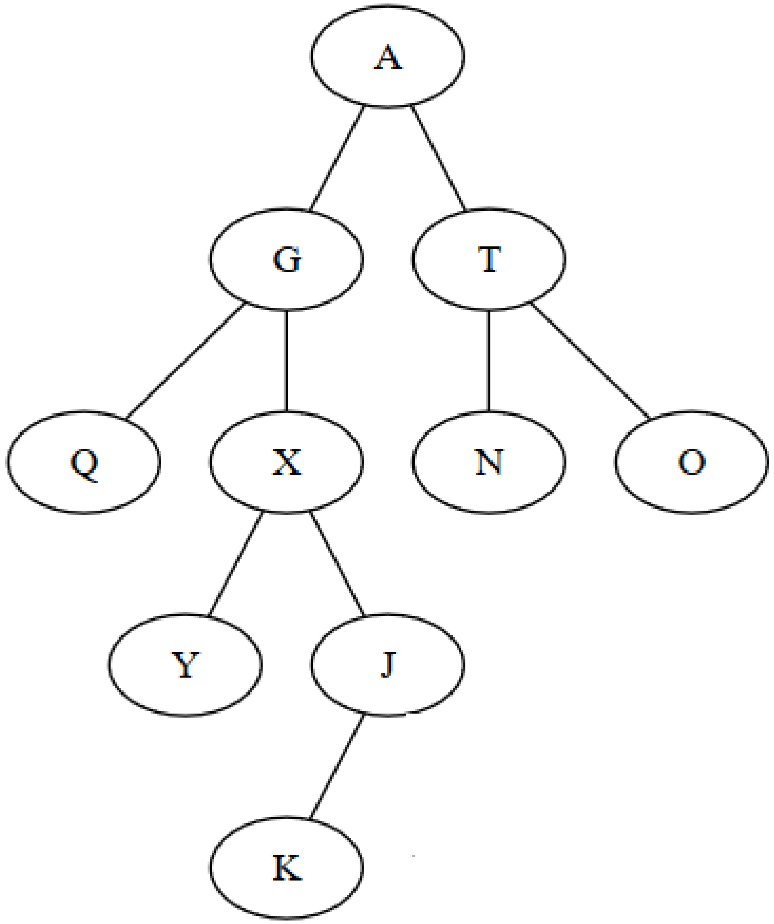


- Traversieren in Level-Ordnung (breadth-first): A, G, T, Q, X, N, O, Y, J, K

Traversieren in Level-Ordnung – Implementierung

- Bei dem Traversieren in Level-Ordnung gibt es nur eine nicht-rekursive Implementierung
- Der Algorithmus ähnelt dem Traversieren in Präordnung, aber anstatt eines Stacks benutzt man eine Schlange
  - Man fängt mit einer leeren Schlange an
  - Füge die Wurzel des Baumes in die Schlange ein
  - Solange die Schlange nicht leer ist:
    - Pop einen Knoten und besuche ihn
    - Füge das linke Kind des Knotens in die Schlange ein
    - Füge das rechte Kind des Knotens in die Schlange ein

Traversieren in Level-Ordnung - Beispiel



- Queue: A
- CurrentNode: A, Queue: G, T
- CurrentNode: G, Queue: T, Q, X
- CurrentNode: T, Queue: Q, X, N, O
- CurrentNode: Q, Queue: X, N, O
- CurrentNode: X, Queue: N, O, Y, J
- CurrentNode: N, Queue: O, Y, J
- CurrentNode: O, Queue: Y, J
- CurrentNode: Y, Queue: J
- CurrentNode: J, Queue: K
- CurrentNode K, Queue: NIL

Traversieren in Level-Ordnung – nicht-rekursive Implementierung

```
subalgorithm levelorder(tree) is:
//pre: tree ist ein Binärbaum
    q: Queue //q ist eine Hilfs-Schlange
    if tree.root ≠ NIL then
        push(q, tree.root)
    end-if
    while not isEmpty(q) execute
        currentNode ← pop(q)
        @visit currentNode
        if [currentNode].left ≠ NIL then
            push(q, [currentNode].left)
        end-if
        if [currentNode].right ≠ NIL then
            push(q, [currentNode].right)
        end-if
    end-while
end-subalgorithm
```

Traversieren in Level-Ordnung – nicht-rekursive Implementierung

- Zeitkomplexität: $\Theta(n)$
- Speicherplatzkomplexität: *O(n)* für die Schlange

Iterator für Binärbaum

- Die Schnittstelle des Binärbaumes enthält die Operation *iterator*, die einen Iterator zurückgibt
- Die Operation hat die Traversierungsmethode als Eingabeparameter: Präordnung, Inordnung, Postordnung, Level-Ordnung
- Die besprochenen Traversierungs-Algorithmen laufen alle Elemente des Binärbaumes auf einmal durch, aber ein Iterator muss Element für Element den Baum durchlaufen
- Der Iterator hat die Standardoperationen: *init, getCurrent, next, valid*

Inordnung Binärbaum Iterator

- Nehmen wir an, dass die Implementierung kein Vaterknoten enthält
- Welche Felder braucht der Iterator?

<u>InorderIterator:</u>
    bt: BinaryTree
    s: Stack
    currentNode: ↑ BTNode

- Was muss man in der *init* Operation tun?

Inordnung Binärbaum Iterator – init

```
subalgorithm init (it, bt) is:
//pre: it - ist ein InorderIterator, bt ist ein Binärbaum
    it.bt ← bt
    init(it.s)
    node ← bt.root
    while node ≠ NIL execute
        push(it.s, node)
        node ← [node].left
    end-while
    if not isEmpty(it.s) then
        it.currentNode ← top(it.s)
    else
        it.currentNode ← NIL
    end-if
end-subalgorithm
```

Inordnung Binärbaum Iterator – getCurrent

```
function getCurrent(it) is:
    getCurrent ← [it.currentNode].info
end-function
```

Inordnung Binärbaum Iterator – valid

```
function valid(it) is:
    if it.currentNode = NIL then
        valid ← false
    else
        valid ← true
    end-if
end-function
```

Inordnung Binärbaum Iterator – next

```
subalgorithm next(it) is:
    node ← pop(it.s)
    if [node].right ≠ NIL then
        node ← [node].right
        while node ≠ NIL execute
            push(it.s, node)
            node ← [node].left
        end-while
    end-if
    if not isEmpty(it.s) then
        it.currentNode ← top(it.s)
    else
        it.currentNode ← NIL
    end-if
end-subalgorithm
```

Binärsuchbaum

- Ein **Binärsuchbaum** ist ein Binärbaum mit folgenden Eigenschaften:
  - Falls x ein Knoten aus den Binärsuchbaum ist, dann:
    - Ist die Information aus x größer gleich jeder Information aus einem Knoten y des linken Teilbaumes
    - Ist die Information aus x kleiner gleich jeder Information aus einem Knoten y des rechten Teilbaumes
- In einem Binärsuchbaum ist die Information aus dem Knoten vom Typ *TComp*
- Die Ordnungsrelation kann abstrakt sein (nicht unbedingt „$\leq$“)

Binärsuchbaum - Repräsentierung I

- Man benutzt eine verkettete Repräsentierung mit dynamischer Allokation (ähnlich wie bei den Binärbäumen)
- Wir benutzen die Ordnungsrelation „≤“

<u>BSTNode:</u>
    info: TComp
    left: ↑ BSTNode
    right: ↑ BSTNode

<u>BinarySearchTree:</u>
    root: ↑ BSTNode

Binärsuchbaum - Repräsentierung II

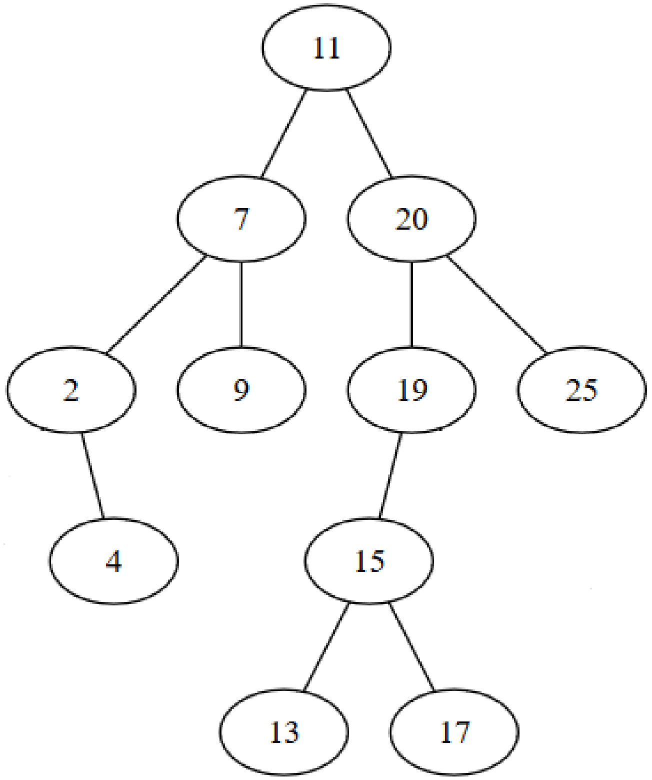
- Wie repräsentieren wir einen Binärsuchbaum verkettet auf Arrays?

```
BinarySearchTree:
    info: TComp[]
    left: Integer[]
    right: Integer[]
    root: Integer
    firstEmpty: Integer
    cap: Integer
```

- Wie repräsentieren wir einen Binärsuchbaum verkettet auf Arrays mit einer Struktur BSTNode?
- Für die nächsten Implementierungen benutzen wir die Repräsentierung mit dynamischer Allokation

Binärsuchbaum - Suchoperation

- Wie kann man ein Element in einem Binärsuchbaum finden?



Binärsuchbaum – Suchoperation – rekursive Implementierung

```
function search_rec (node, elem) is:
//pre: node ist ein BSTNode und elem ist das TComp, das gesucht wird
    if node = NIL then
        search_rec ← false
    else
        if [node].info = elem then
            search_rec ← true
        else if [node].info < elem then
            search_rec ← search_rec([node].right, elem)
        else
            search_rec ← search_rec([node].left, elem)
        end-if
    end-if
end-function
```

Binärsuchbaum – Suchoperation – rekursive Implementierung

- Komplexität der Suchoperation:
  - O(h) (welches O(n) ist)
- Da die Suchoperation einen Knoten als Eingabeparameter hat, braucht man eine Wrapper-Funktion:

function search (tree, e) **is**:

//pre: tree ist ein BinarySearchTree, e ist das gesuchte Element

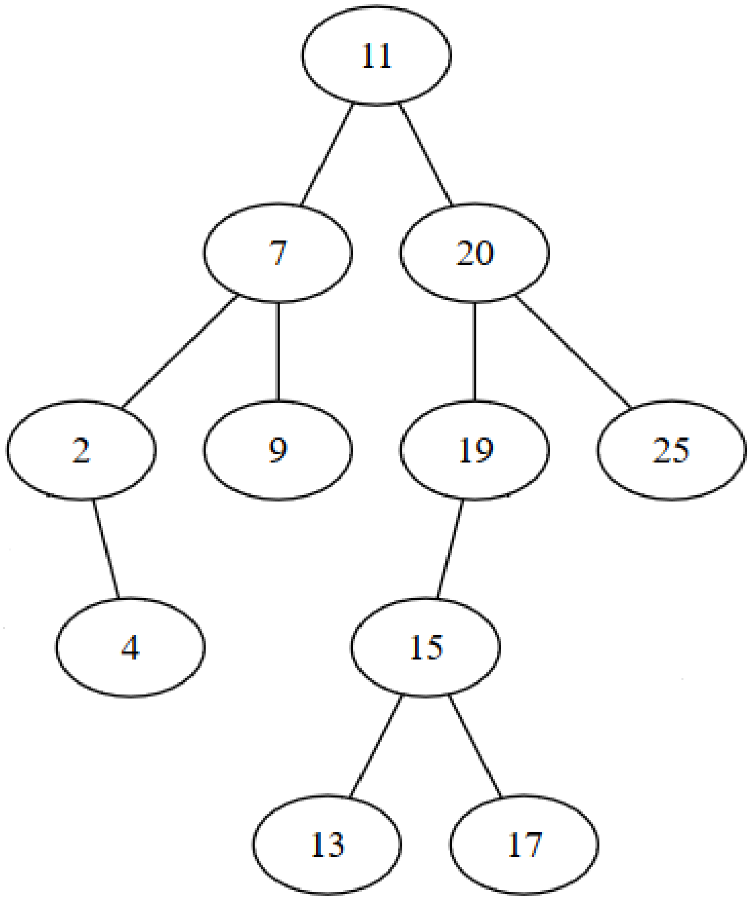
    search ← search_rec(tree.root, e)

end-function

Binärsuchbaum – Suchoperation – iterative Implementierung

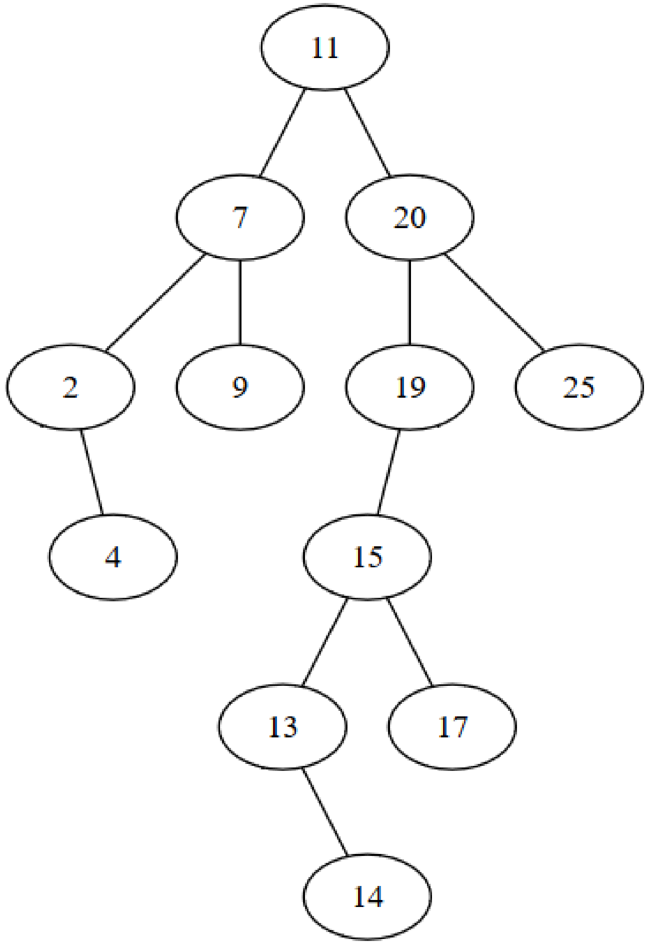
```
function search (tree, elem) is:
//pre: tree ist ein BinarySearchTree und elem ist das TComp, das gesucht wird
    currentNode ← tree.root
    found ← false
    while currentNode ≠ NIL and not found execute
        if [currentNode].info = elem then
            found ← true
        else if [currentNode].info < elem then
            currentNode ← [currentNode].right
        else
            currentNode ← [currentNode].left
        end-if
    end-while
    search ← found
end-function
```

Binärsuchbaum – Einfügeoperation



- Wie/Wo kann man den Wert 14 einfügen?

Binärsuchbaum – Einfügeoperation



BST – insert – rekursive Implementierung

- Wie kann man die Einfügeoperation implementieren?
- Man kann damit anfangen, einen neuen Knoten mit dem neuen Wert zu erstellen

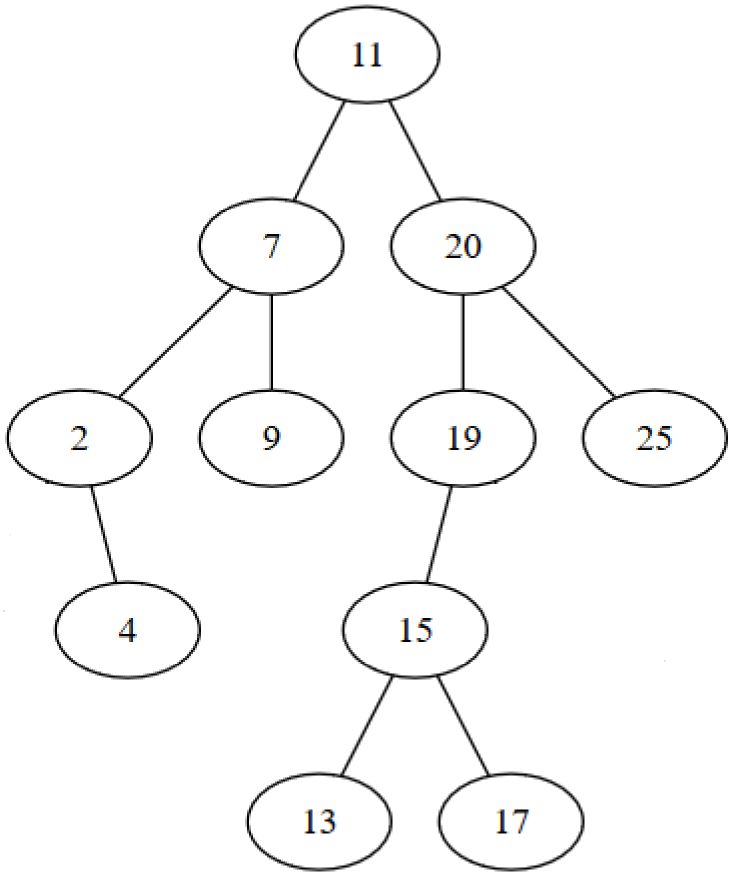
```
function initNode(e) is:
//pre: e ist ein TComp
//post: initNode ← ein Knoten mit e als Information
    allocate(node)
    [node].info ← e
    [node].left ← NIL
    [node].right ← NIL
    initNode ← node
end-function
```

BST – insert – rekursive Implementierung

```
function insert_rec(node, e) is:
//pre: node ist ein BSTNode, e ist TComp
//post: ein Knoten, der e enthält, wurde in dem Baum anfangend von node eingefügt
    if node = NIL then
        node ← initNode(e)
    else if [node].info ≥ e then
        [node].left ← insert_rec([node].left, e)
    else
        [node].right ← insert_rec([node].right, e)
    end-if
    Insert_rec ← node
end-function
```

- Komplexität: O(n)
- Genauso wie bei der Suchoperation, braucht man eine Wrapper-Funktion, welche diese Funktion mit der Wurzel aufruft

BST – finde das Minimum



- Wie kann man das minimale Element finden?

BST – finde das Minimum

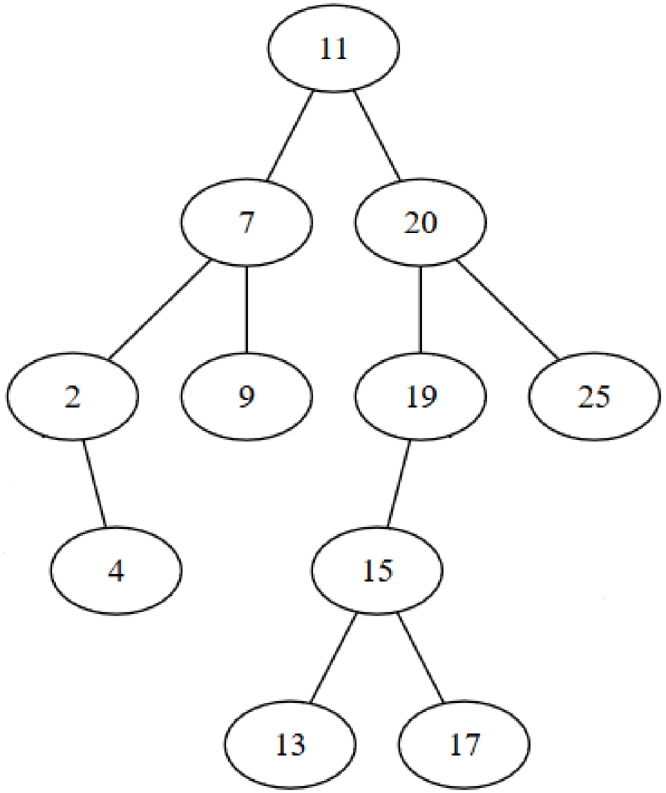
```
function minimum(tree) is:
//pre: tree ist ein BinarySearchTree
//post: minimum ← der minimale Wert aus dem Baum
    currentNode ← tree.root
    if currentNode = NIL then
        @empty tree, no minimum
    else
        while [currentNode].left ≠ NIL execute
            currentNode ← [currentNode].left
        end-while
        minimum ← [currentNode].info
    end-if
end-function
```

BST – finde das Minimum

- Komplexität: O(n)

- Es gibt unterschiedliche Minimum-Methoden:
  - Man kann den minimalen Wert für einen Teilbaum berechnen, in diesem Fall braucht man die Wurzel des Teilbaumes als Parameter
  - Man kann anstatt den Wert, den Knoten mit dem minimalen Wert zurückgeben
- Der maximale Wert kann ähnlich gefunden werden

Finde den Vaterknoten eines Knotens



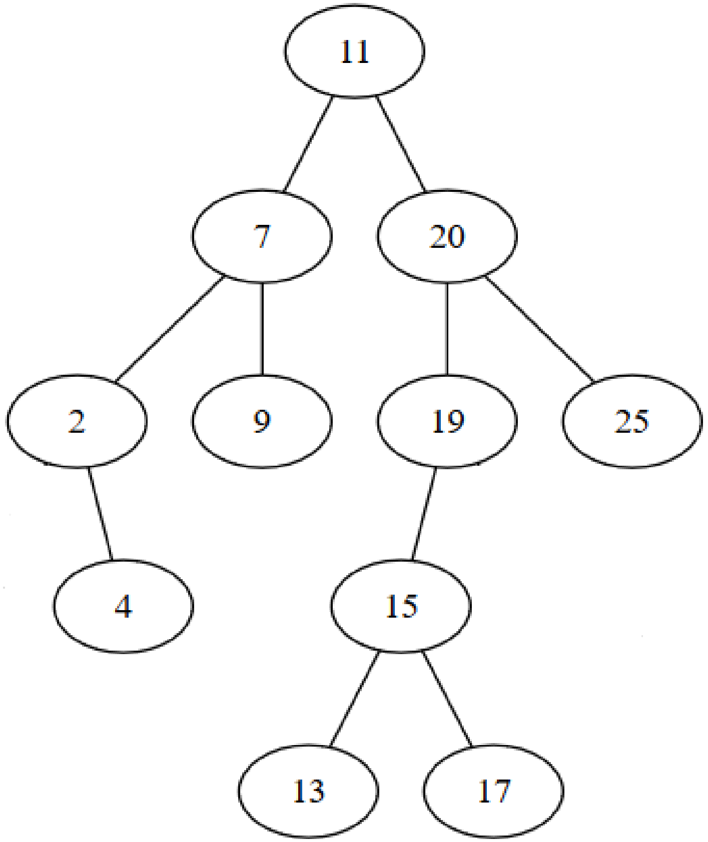
- Wie kann man den Vaterknoten eines gegebenen Knotens finden?
- Man nimmt an, dass der Vaterknoten in der Repräsentierung eines Knotens nicht gespeichert wird

Finde den Vaterknoten eines Knotens

```
function parent(tree, node) is:
//pre: tree ist ein BinarySearchTree, node ist ein Pointer zu einem BSTNode, node ≠ NIL
//post: gibt den Vaterknoten eines Knoten zurück, oder NIL falls der Knoten Wurzel ist
    c ← tree.root
    if c = node then //node ist die Wurzel
        parent ← NIL
    else
        while c ≠ NIL and [c].left ≠ node and [c].right ≠ node execute
            if [c].info ≥ [node].info then
                c ← [c].left
            else
                c ← [c].right
            end-if
        end-while
        parent ← c
    end-if
end-function
```

- Komplexität: O(n)

BST – finde den Nachfolger eines Knotens



- Wie kann man den Knoten mit dem nachfolgenden Wert für einen gegebenen Knoten finden? Z.B. Nachfolger von 11, oder 17, oder 13?

BST – finde den Nachfolger eines Knotens

```
function successor(tree, node) is:
//pre: tree ist ein BinarySearchTree, node ist ein Pointer zu einem BSTNode, node ≠ NIL
//post: gibt den Knoten mit dem nachfolgenden Wert für den gegebenen Knoten aus
//oder NIL falls der Knoten schon den maximalen Wert enthält
    if [node].right ≠ NIL then
        c ← [node].right
        while [c].left ≠ NIL execute
            c ← [c].left
        end-while
        successor ← c
    else
        c ← node
        p ← parent(tree, c)
        while p ≠ NIL and [p].left ≠ c execute //solange der Knoten c das rechte Kind ist muss
            c ← p                              //man höher gehen
            p ← parent(tree, p)
        end-while
        successor ← p
    end-if
end-function
```

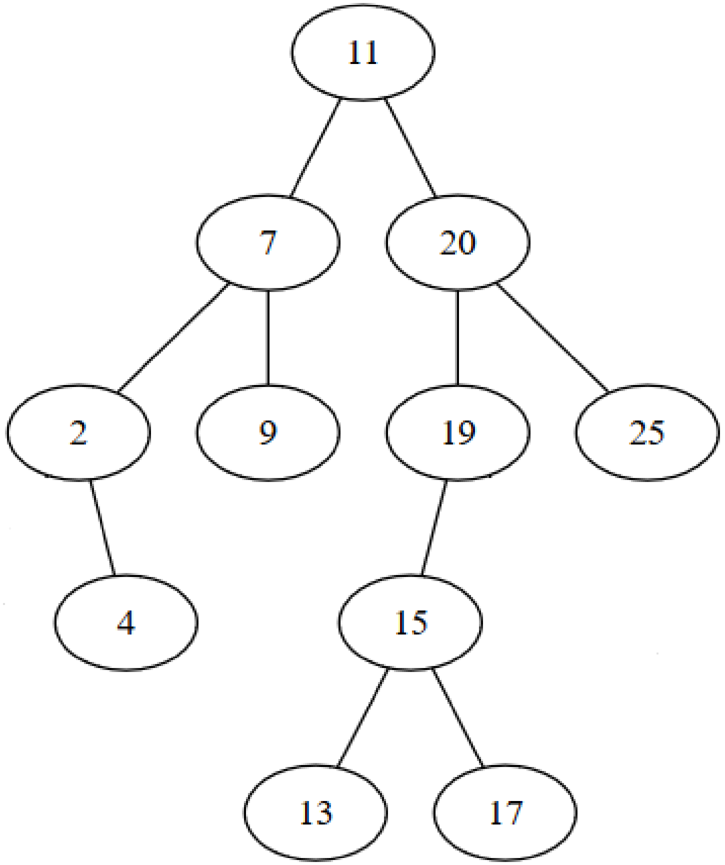
BST – finde den Nachfolger eines Knotens

- Komplexität:
  - Hängt von der Funktion *parent* ab:
    - Falls *parent* $\Theta(1)$ ist, dann ist die Komplexität des Nachfolgers O(n)
    - Falls *parent* $O(n)$ ist, dann ist die Komplexität des Nachfolgers $O(n^2)$

- Was ändert sich wenn die Nachfolger Funktion anstatt einen Knoten nur den Wert als Eingabeparameter hat?

- Ähnlich zu der Nachfolger Funktion kann man auch die Vorgänger Funktion definieren

BST – lösche einen Knoten



- Wie kann man den Wert 25 löschen? Oder den Wert 2 und 11?

BST – lösche einen Knoten

- Wenn man einen Knoten aus einem Binärbaum löschen will, dann gibt es drei Fälle:
  - Der Knoten, der gelöscht werden muss, hat kein Kind
    - Setze das entsprechende Kind des Vaterknotens auf NIL
  - Der Knoten, der gelöscht werden muss, hat ein Kind:
    - Ersetze den gelöschten Knoten mit seinem Kind (setze das Kind des Vaterknotens auf seinem Kind)
  - Der Knoten, der gelöscht werden muss, hat zwei Kinder:
    - Finde den maximalen Wert aus dem linken Teilbaum, ersetze den gelöschten Wert mit diesem maximalen Wert und lösche den maximalen Wert

    ODER

    - Finde den minimalen Wert aus dem rechten Teilbaum, ersetze den gelöschten Wert mit diesem minimalen Wert und lösche den minimalen Wert

BST – lösche einen Knoten

```
function removeRec(node, elem) is
//pre: node ist ein Pointer zu einem BSTreeNode und elem ist der Wert, der
//      gelöscht werden muss
//post: der Knoten mit dem Wert elem wurde aus dem (Teil)baum der mit
//          node anfängt gelöscht
    if node = NIL then
        removeRec ← NIL
    else if [node].info > elem then
        [node].left ← removeRec([node].left, elem)
        removeRec ← node
    else if [node].info < elem then
        [node].right ← removeRec([node].right, elem)
        removeRec ← node
    else //[node].info = elem, wir wollen node löschen
//Fortsetzung auf der nächsten Folie
```

BST – lösche einen Knoten

```
            if [node].left = NIL and [node].right = NIL then
                removeRec ← NIL
            else if [node].left = NIL then
                removeRec ← [node].right
            else if [node].right = NIL then
                removeRec ← [node].left
            else
                min ← minimum([node].right)
                [node].info ← [min].info
                [node].right ← removeRec([node].right, [min].info)
                removeRec ← node
            end-if
        end-if
    end-function
```

BST – lösche einen Knoten

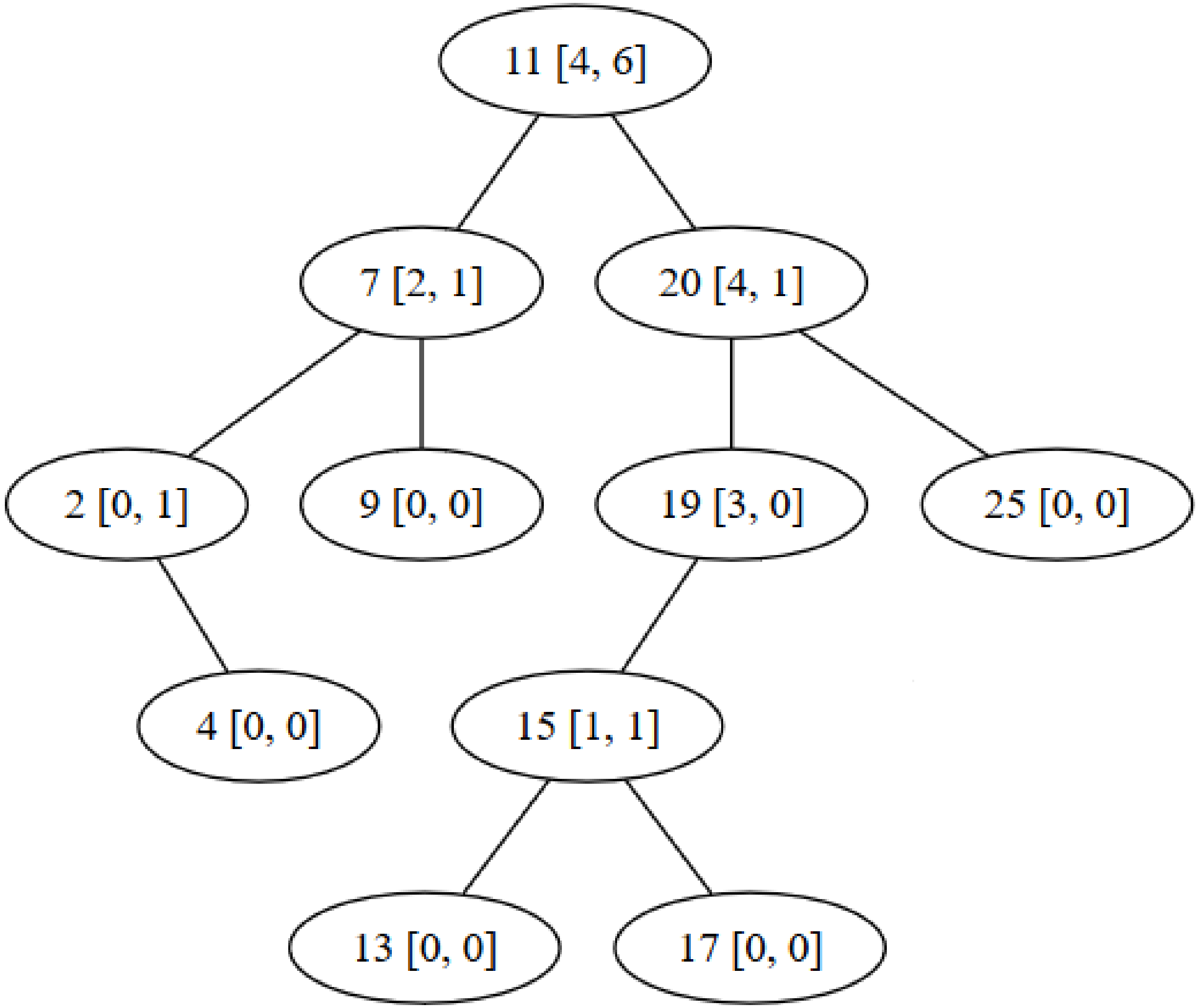
- In dem Löschalgorithmus nehmen wir an, dass die *minimum* Funktion den <u>Knoten</u> mit dem minimalen Wert zurückgibt und nicht nur den Wert
- Komplexität: O(n)

BST – denke nach

- Kann man eine Sortierte Liste auf einem Binärsuchbaum speichern?

- Listen haben Positionen für die Elemente und Operationen basierend auf diese Positionen. In einem SortedList kann man ein Element an einer gegebenen Position zurückgeben oder löschen. Wie kann man Positionen in dem Binärsuchbaum speichern?
- Tipp: speichere in jedem Knoten die Anzahl der Knoten in dem linken Teilbaum und/oder die Anzahl der Knoten in dem rechten Teilbaum. Das gibt uns automatisch die Position der Wurzel.
- Diese Werte müssen dann bei jeder Einfüge- und Löschoperation aktualisiert werden

BST



Binärsuchbaum mit Duplikatwerte

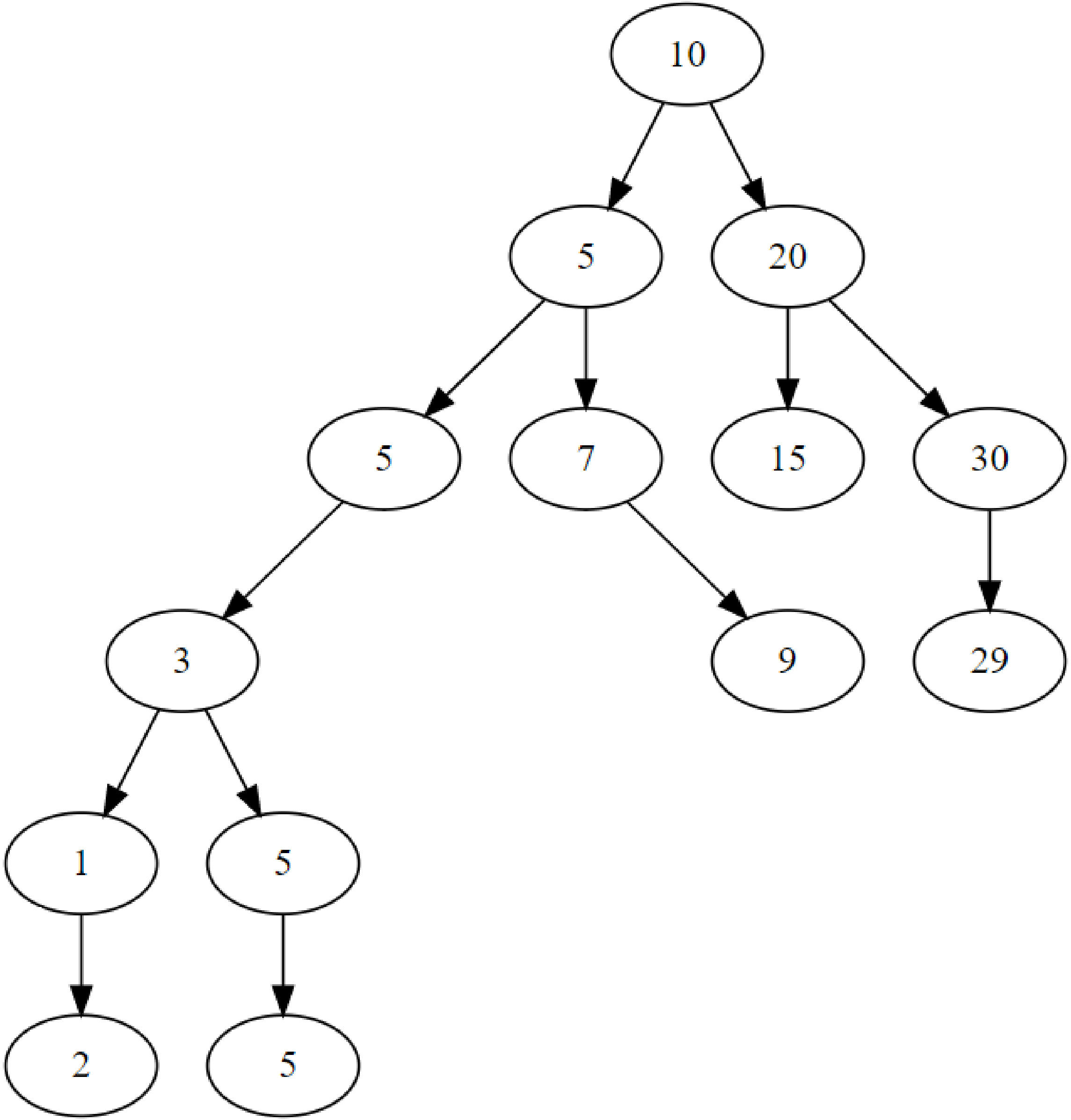
- Anfangend von einem leeren Binärsuchbaum mit der Relation $\leq$ füge, in der angegebenen Reihenfolge, folgende Werte ein:

  10, 20, 5, 7, 15, 5, 30, 3, 5, 5, 1, 9, 29, 2

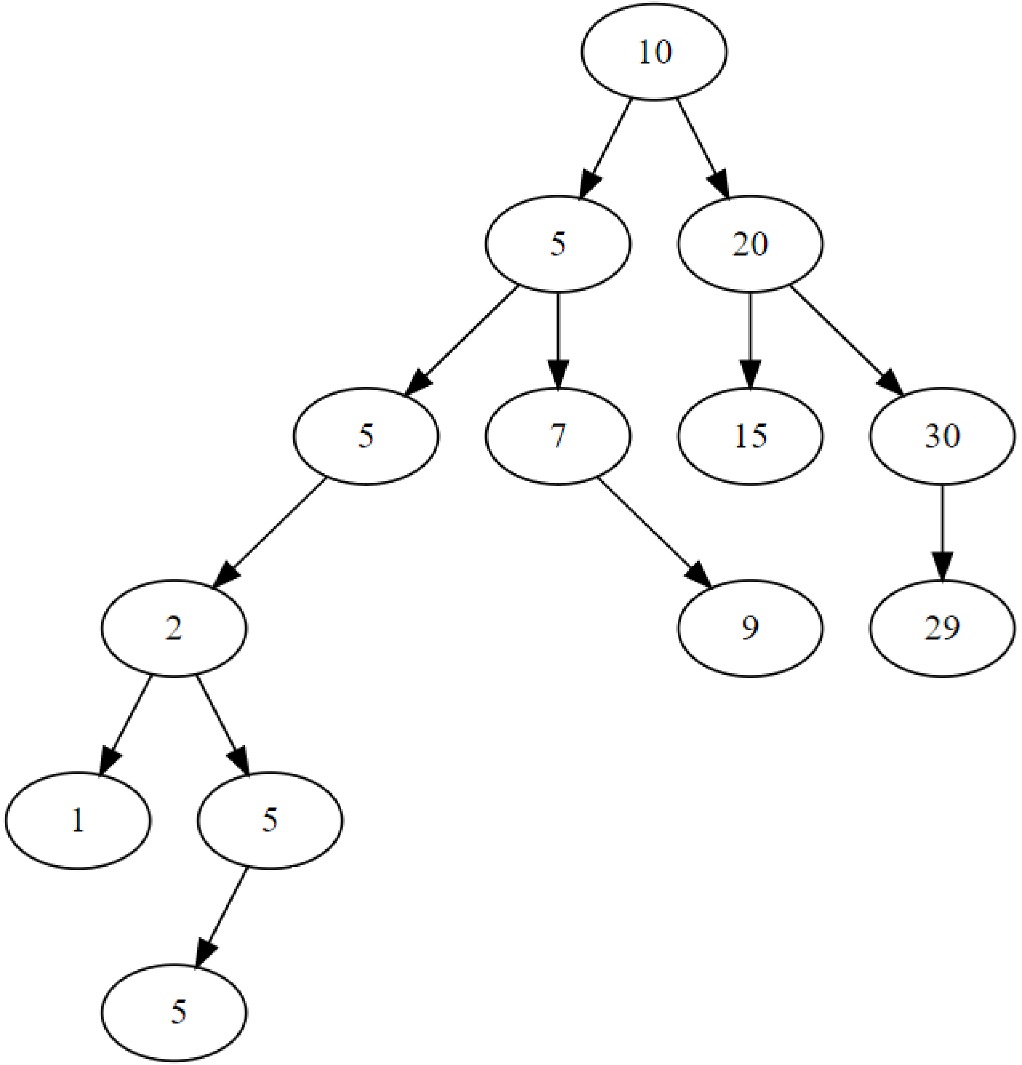
- Wo sollte man Werte einfügen die gleich sind mit dem Vaterknoten, in dem linken oder in dem rechten Kind?

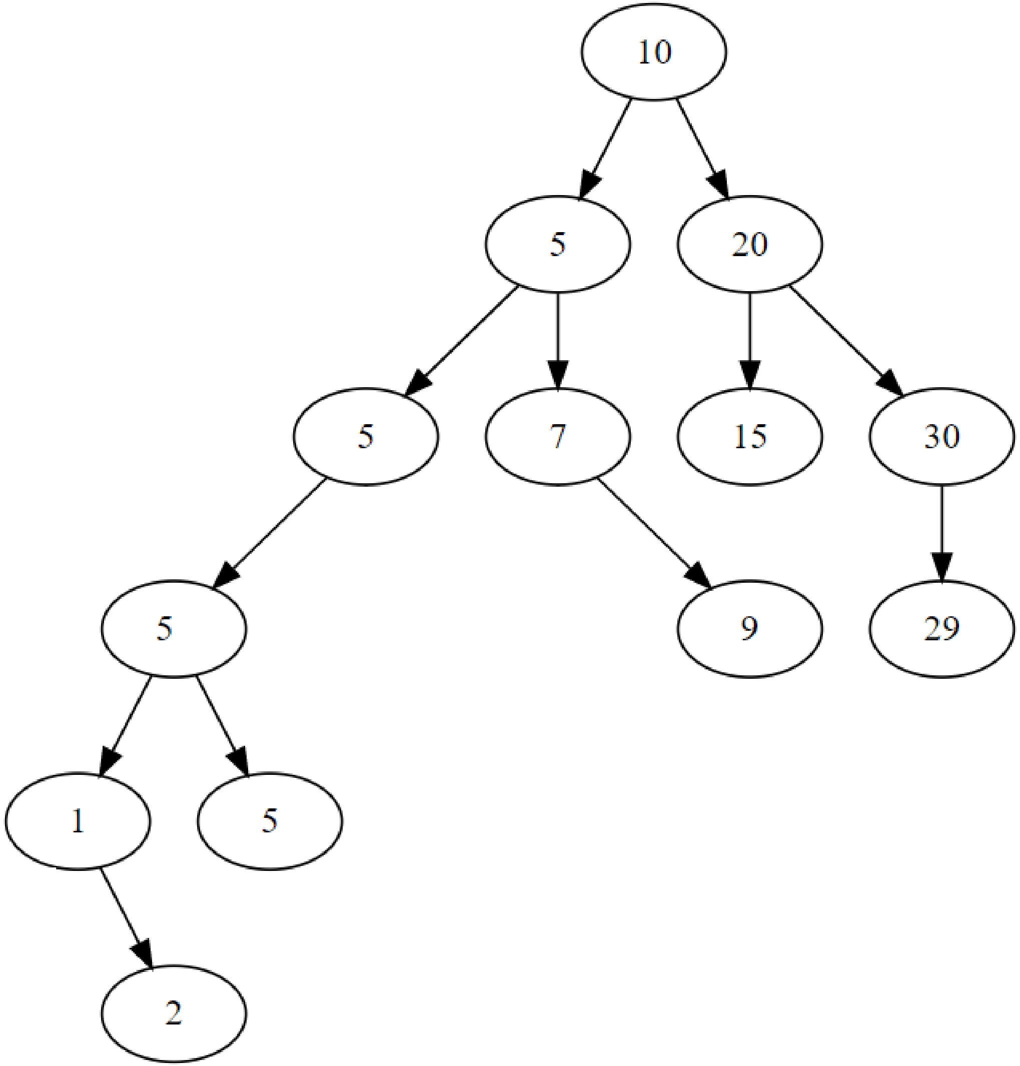
- Wie würdet ihr zählen wie viele Male ein Wert in dem Binärsuchbaum vorkommt?

- 10, 20, 5, 7, 15, 5, 30, 3, 5, 5, 1, 9, 29, 2
- Lösche 3



- Es gibt zwei Möglichkeiten:





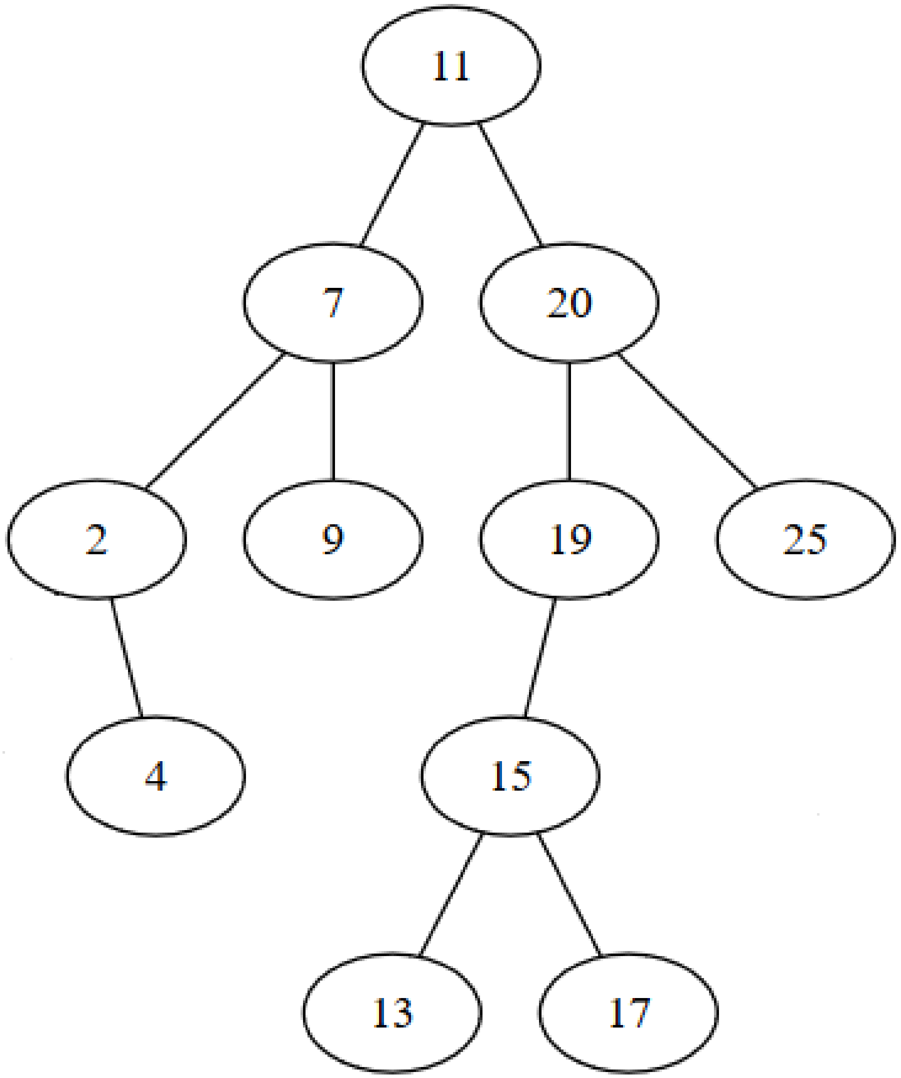
Balancierte Binärsuchbäume

- Spezifische Operationen für Binärsuchbäume haben die Zeitkomplexität *O(h),* das im schlimmsten Fall $\Theta(n)$ ist
- Bester Fall erreicht man, wenn der Baum balanciert ist, und die Höhe des Baumes $\Theta(log_2 n)$ ist
- Um die Komplexität des Algorithmus zu reduzieren, will man den Baum balanciert behalten.
- Wenn ein Knoten nicht mehr balanciert ist, dann muss man Knoten verschieben (Rotationen) bis der Baum wieder balanciert ist

AVL Bäume

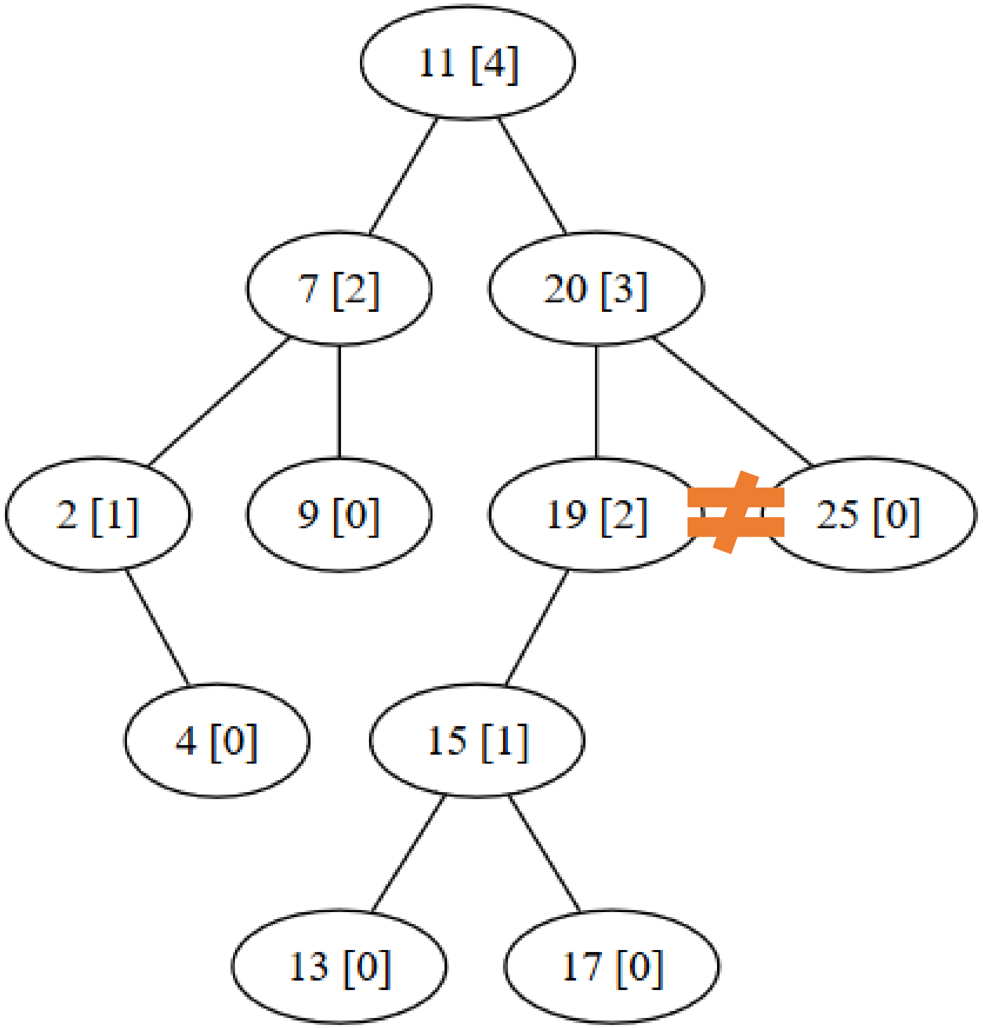
- Definition: Ein AVL (Adelson-Velskii, Landis) Baum ist ein Binärsuchbaum mit folgender Eigenschaft (AVL Baum-Eigenschaft)
  - Falls **x ein Knoten** des AVL Baumes ist, dann ist **der Unterschied der Höhen zwischen dem linken und dem rechten Teilbaum** von x **0, 1 oder -1** (Balancierungsinformation)
- Bemerkung:
  - Ein leerer AVL Baum hat die Höhe -1
  - Die Höhe eines Knotens ist 0

AVL Bäume



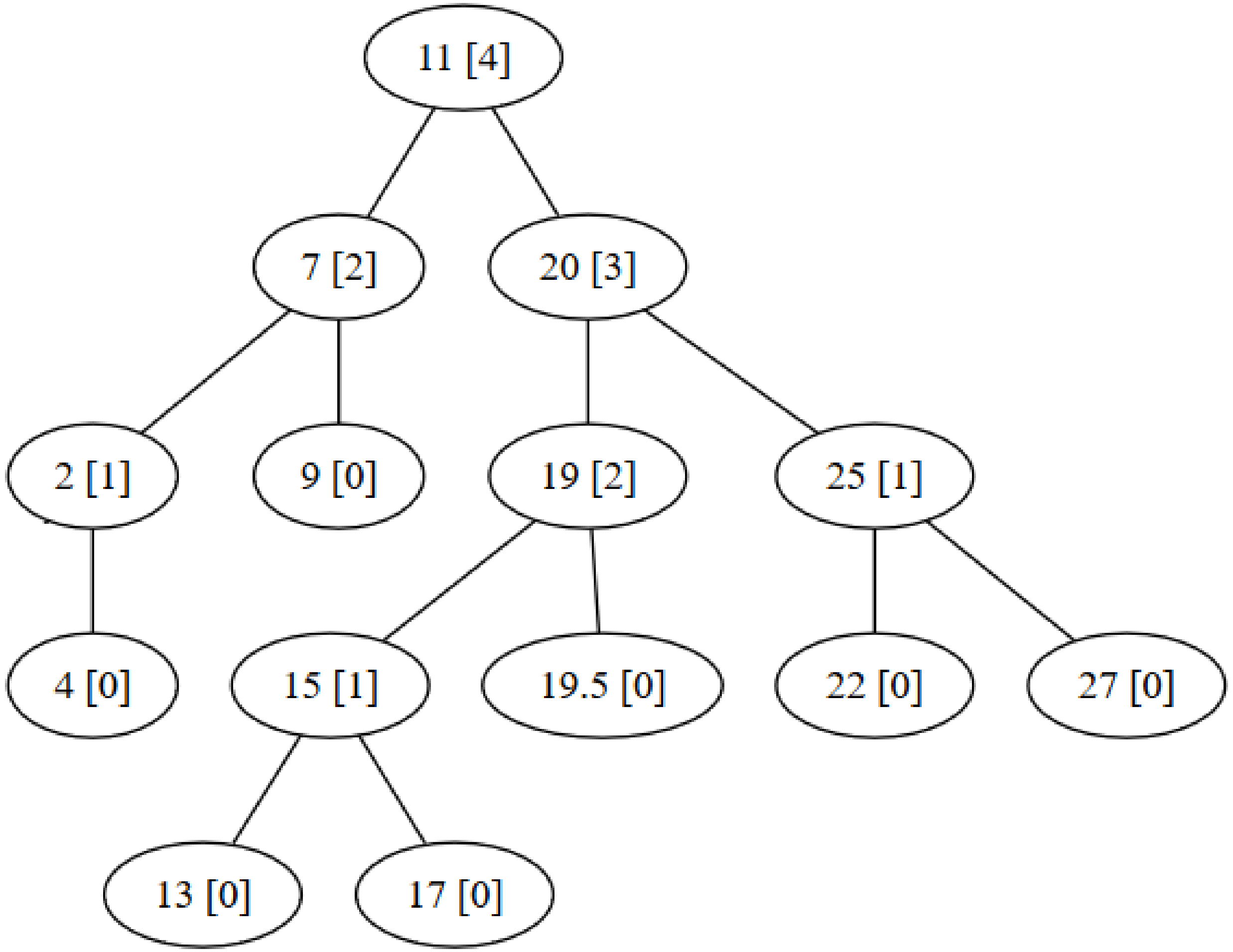
- Ist dieser ein AVL Baum?

AVL Bäume



- Die Werte in den Klammern zeigen die Höhen der Knoten.
- Der Baum ist kein AVL Baum, da der Unterschied der Höhen der Teilbäume von 19 und 20 größer als 1 ist (die Differenz ist 2)

AVL Bäume



- Dieser ist ein AVL Baum

AVL Bäume – Rotationen

- Es kann sein, dass die Einfüge- oder Löschoperation die AVL Eigenschaft verletzt

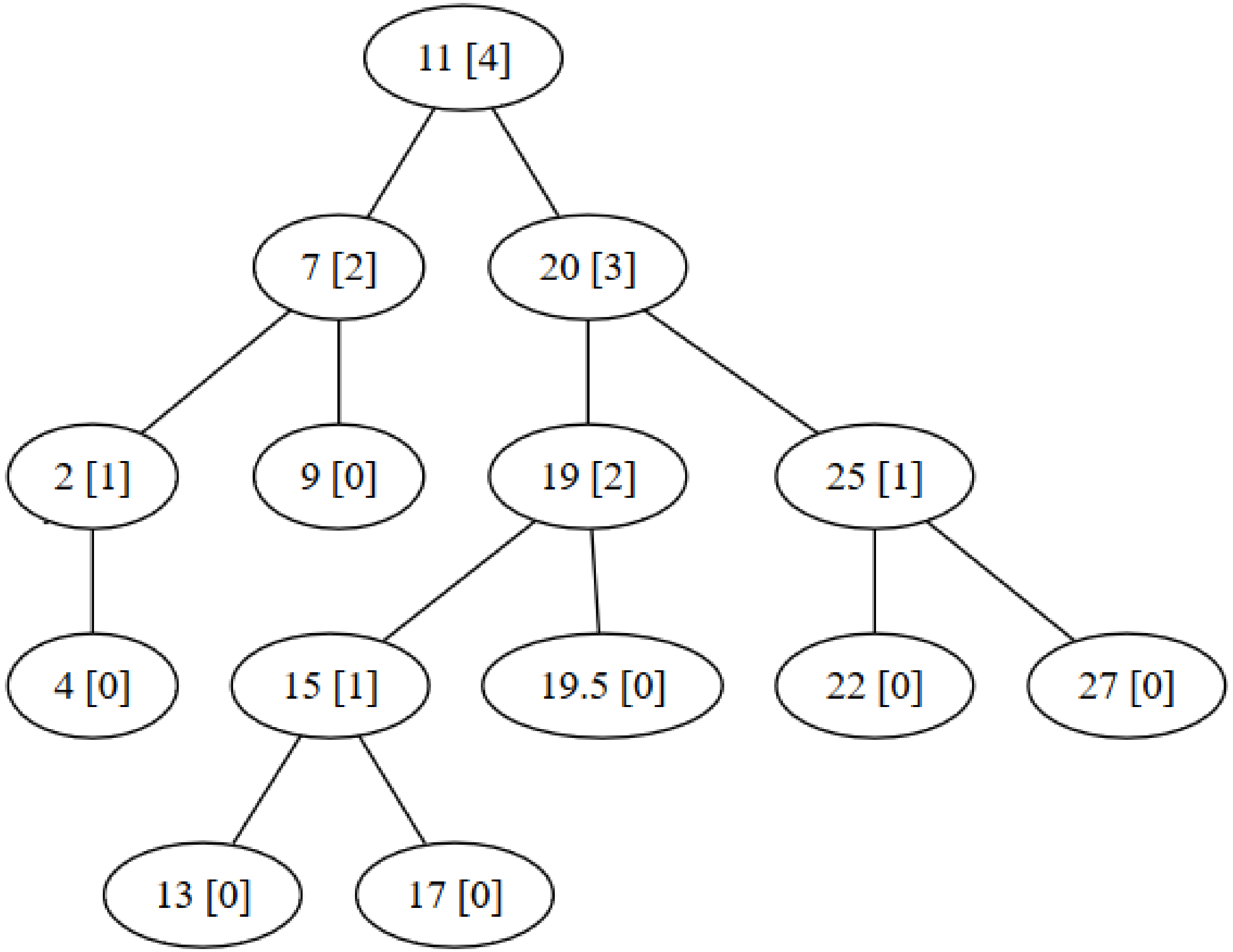
- In diesem Fall, muss die AVL Eigenschaft mithilfe von **Rotationen** aufbewahrt werden

AVL Bäume – Rotationen

- Nach einer Einfügeoperation, können nur die Höhen der Knoten auf dem Pfad zu dem eingefügten Knoten geändert werden

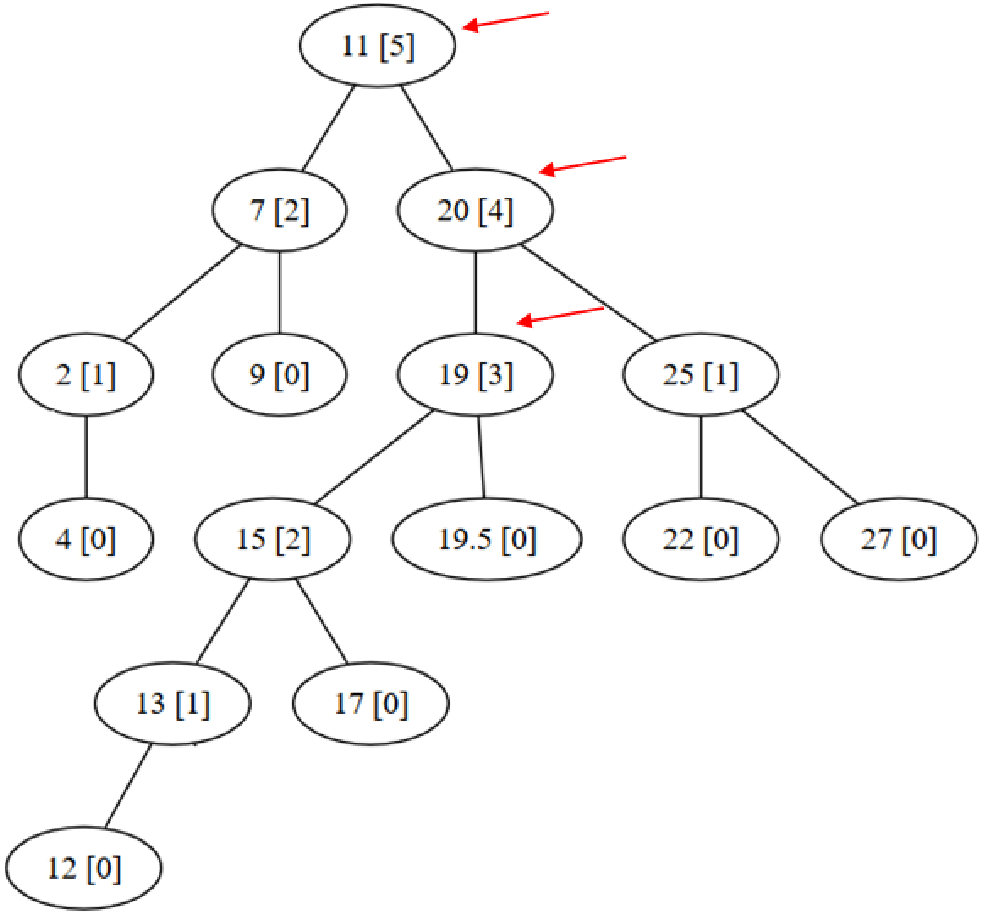
- Man überprüft die Balancierungsinformation und wenn man einen Knoten findet, der die AVL Eigenschaft verletzt, dann führt man eine Rotation aus

AVL Bäume – Rotationen



- Was passiert wenn man 12 einfügt?

AVL Bäume – Rotationen

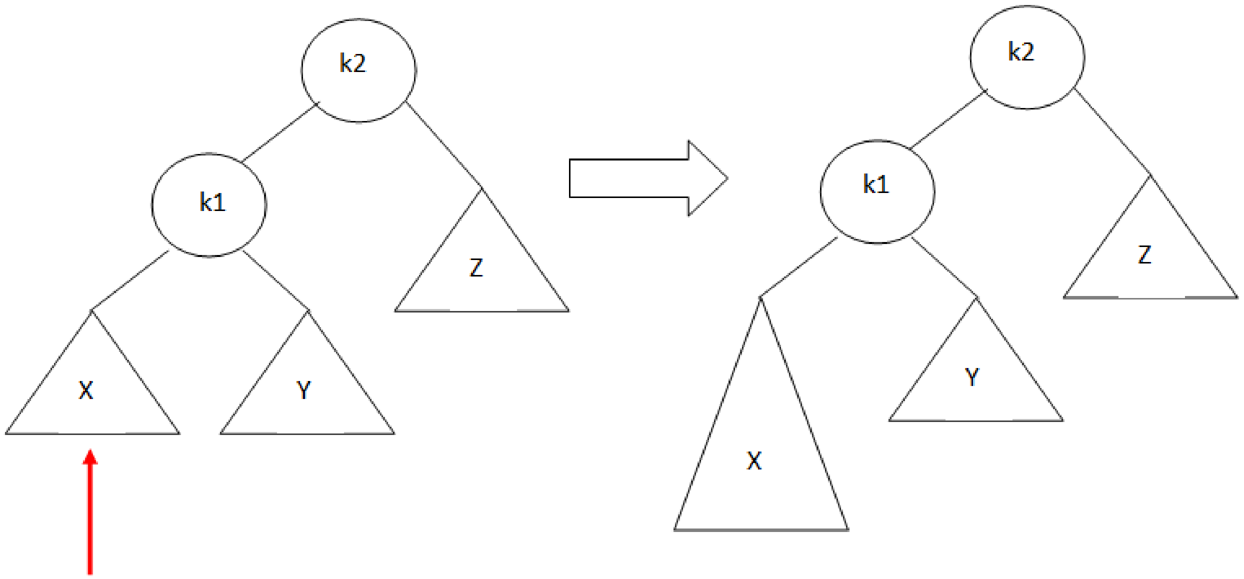


- Die roten Pfeile zeigen die unbalancierte Knoten. Wir müssen den Knoten 19 balancieren.

AVL Bäume – Rotationen

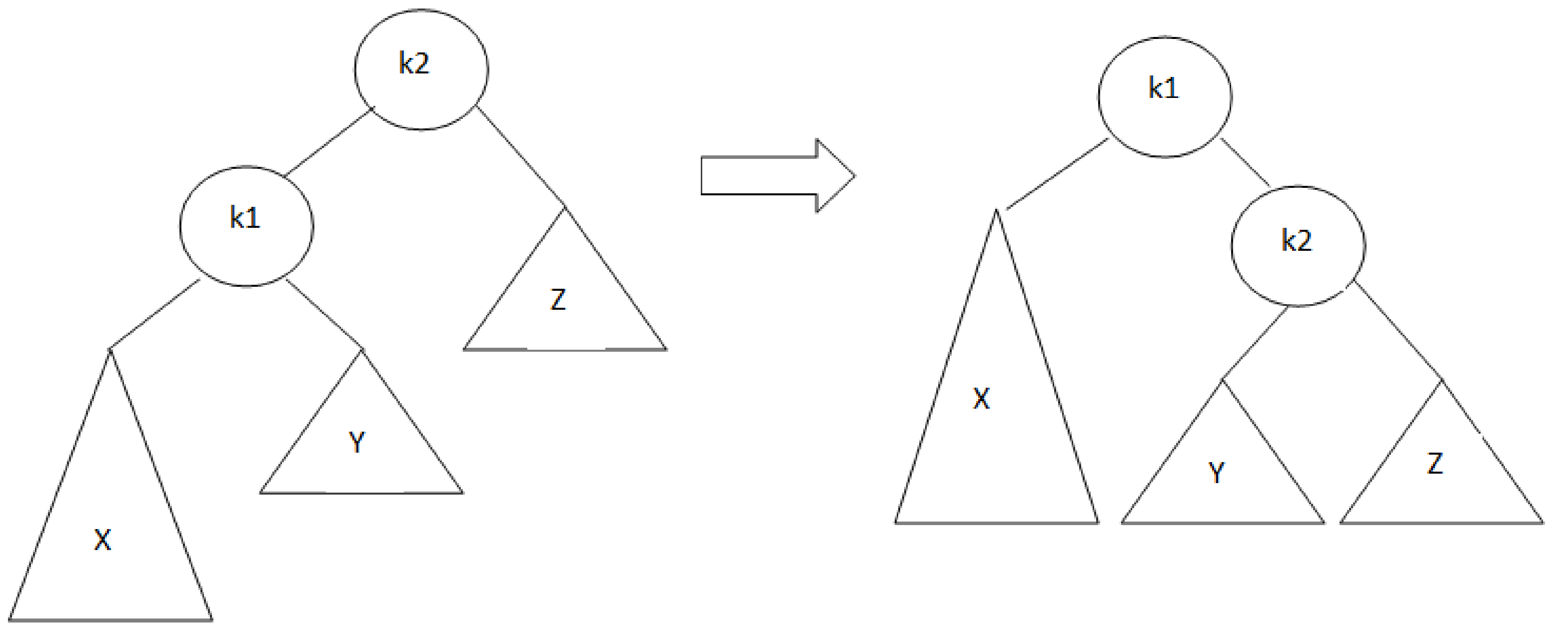
- Nehmen wir an, dass zu einem bestimmten Zeitpunkt der Knoten α balanciert werden muss
- Da α vor dem Einfügen balanciert war und nach dem Einfügen nicht mehr, kann man vier Fälle unterscheiden:
  - Einfügen in dem linken Teilbaum des linken Kindes von α
  - Einfügen in dem rechten Teilbaum des linken Kindes von α
  - Einfügen in dem linken Teilbaum des rechten Kindes von α
  - Einfügen in dem rechten Teilbaum des rechten Kindes von α

AVL Bäume – Rotationen – Fall 1

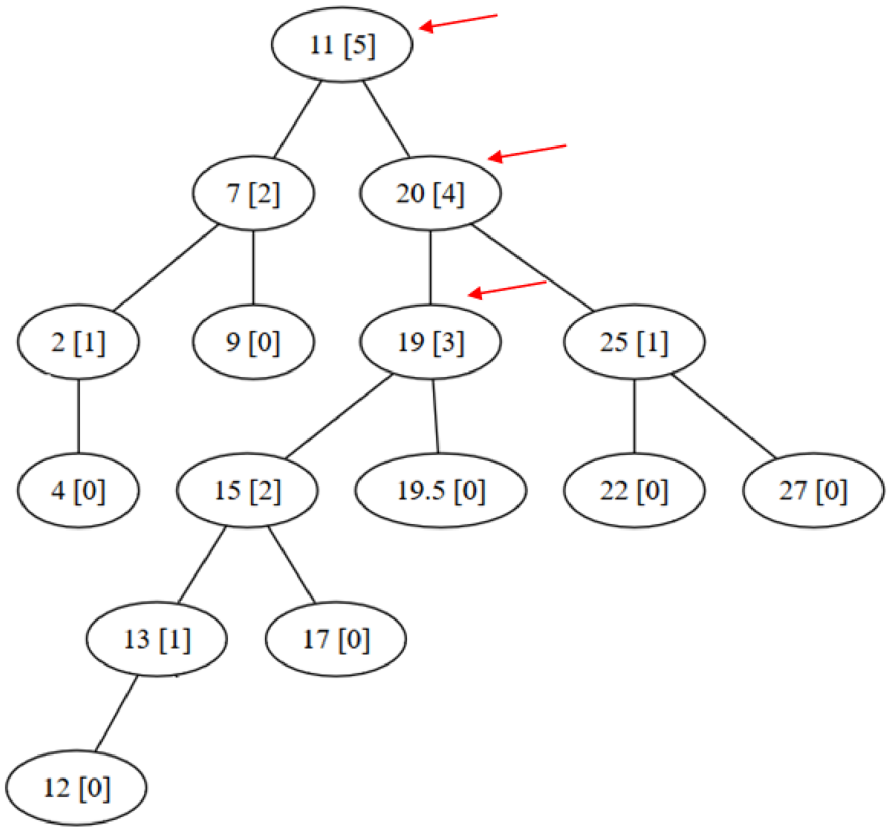


- Bem. X, Y und Z sind Teilbäume mit derselben Höhe vor dem Einfügen.
- Lösung: **einfache Rotation nach rechts**

AVL Bäume – Rotationen – Fall 1

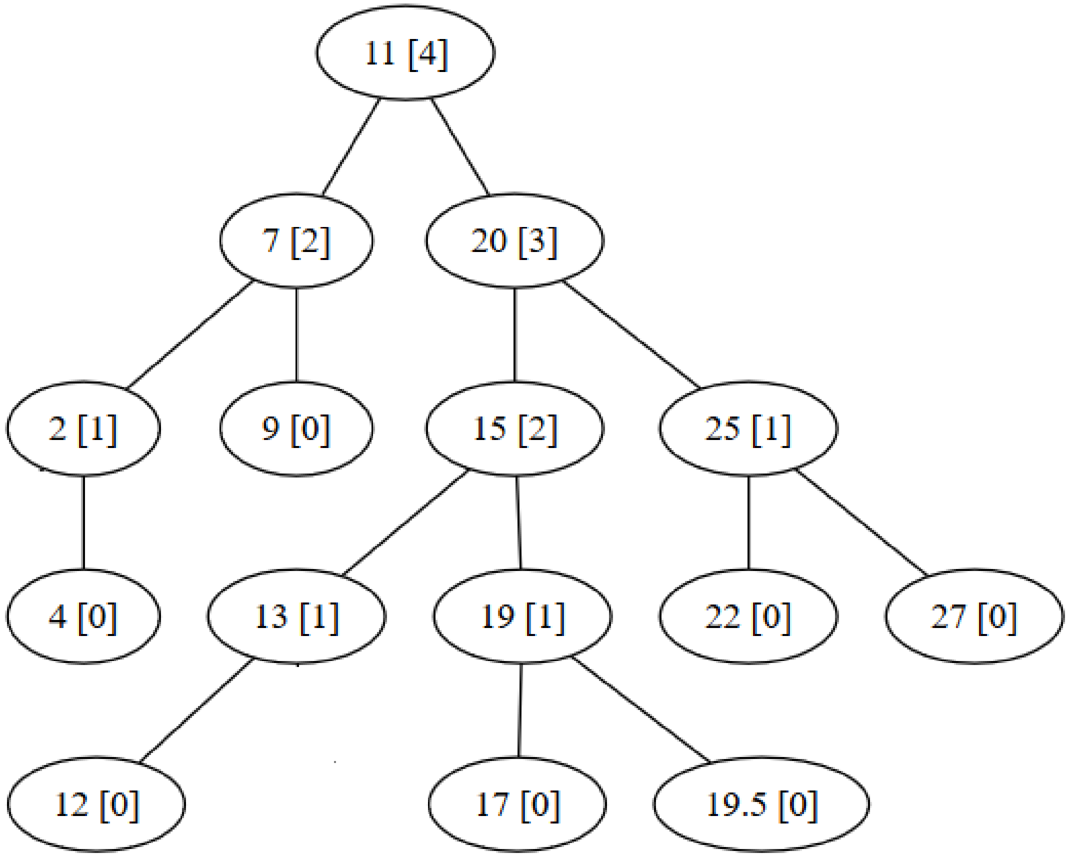


AVL Bäume – Rotationen – Fall 1 - Beispiel



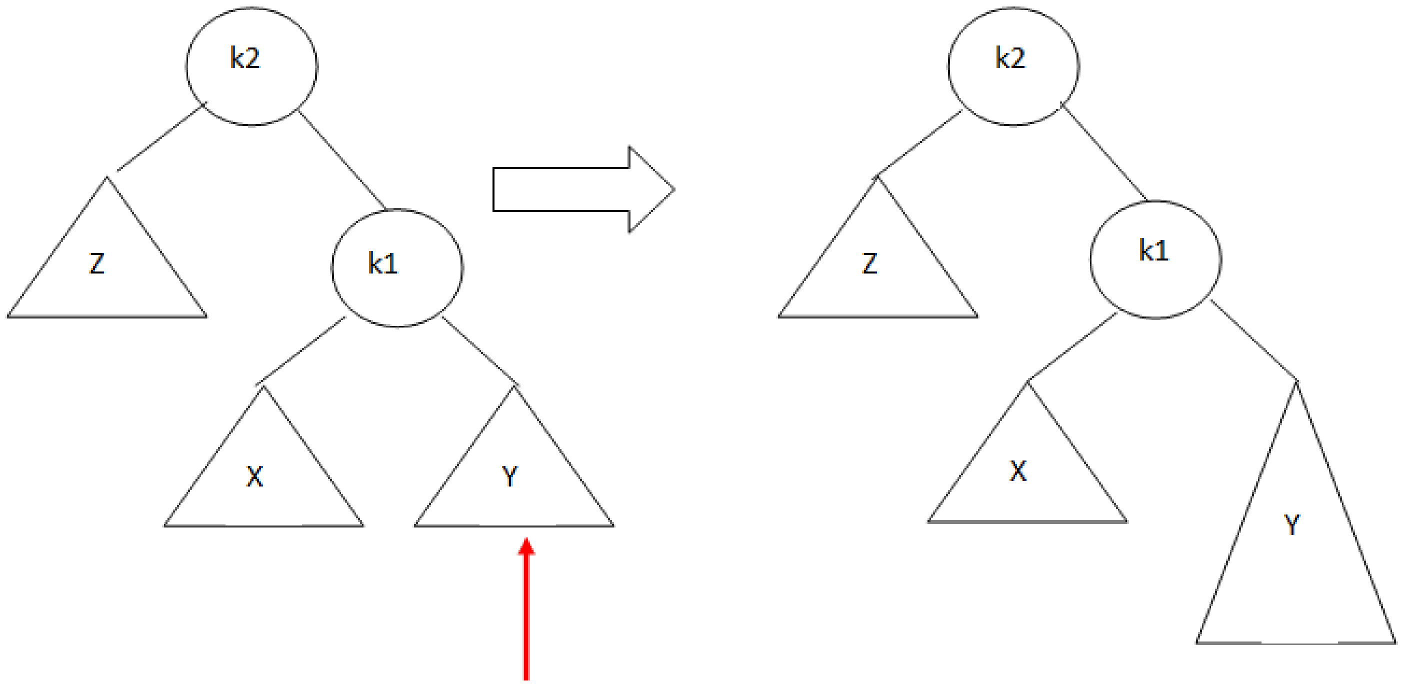
- Knoten 19 ist unbalanciert, da wir 12 in dem linken Teilbaum des linken Kindes eingefügt haben

AVL Bäume – Rotationen – Fall 1 - Beispiel



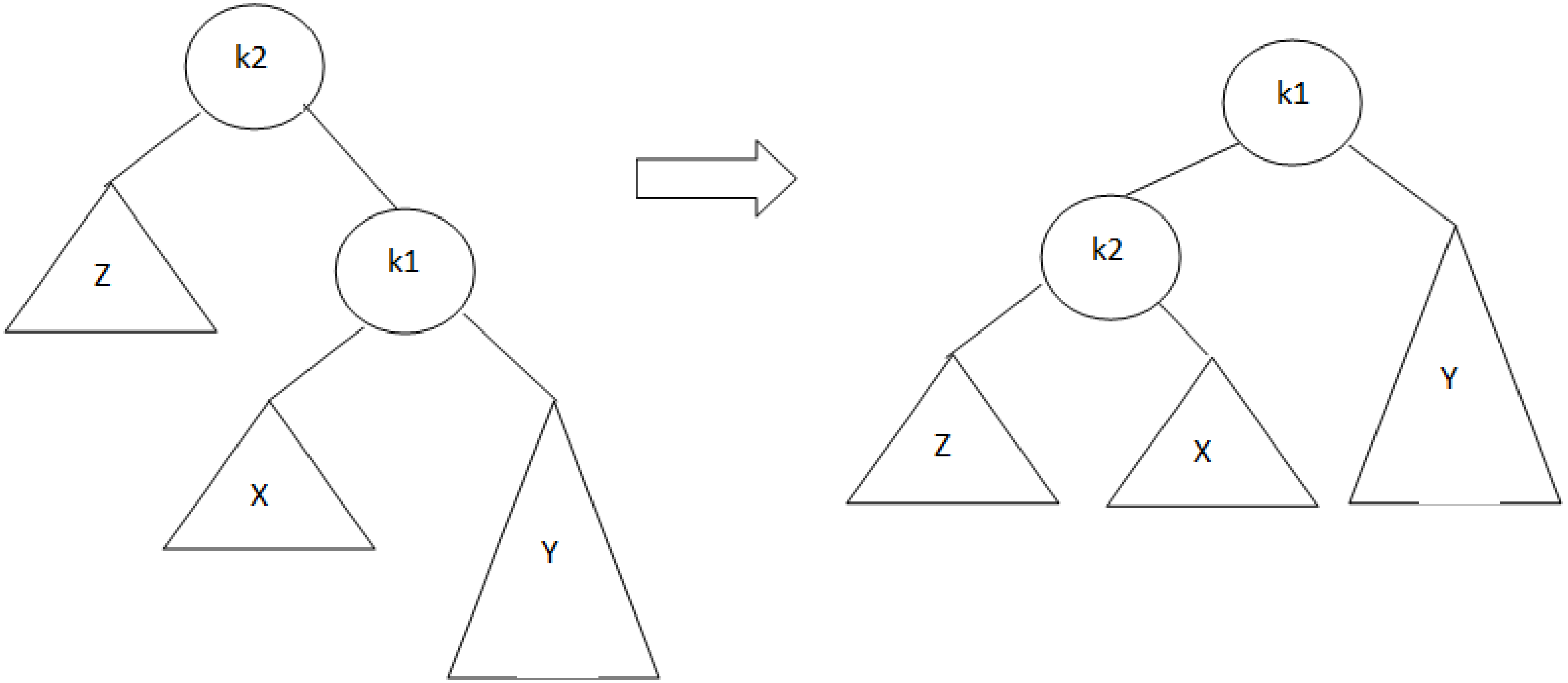
- Lösung: einfache Rotation nach rechts

AVL Bäume – Rotationen – Fall 4

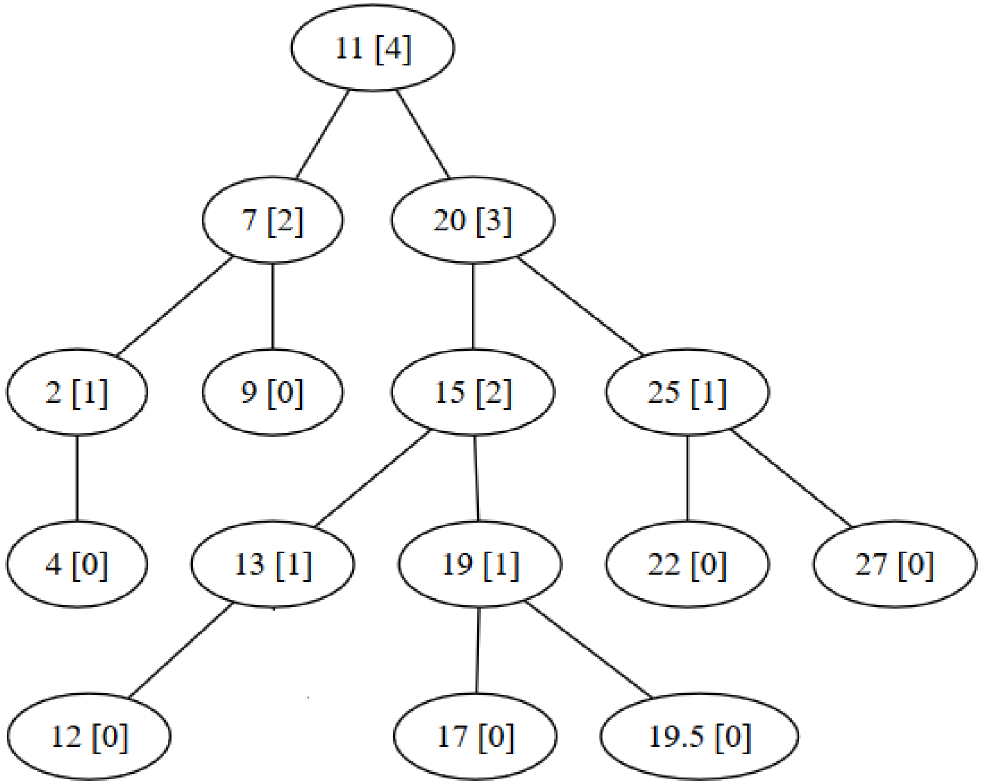


- Lösung: **einfache Rotation nach links**

AVL Bäume – Rotationen – Fall 4

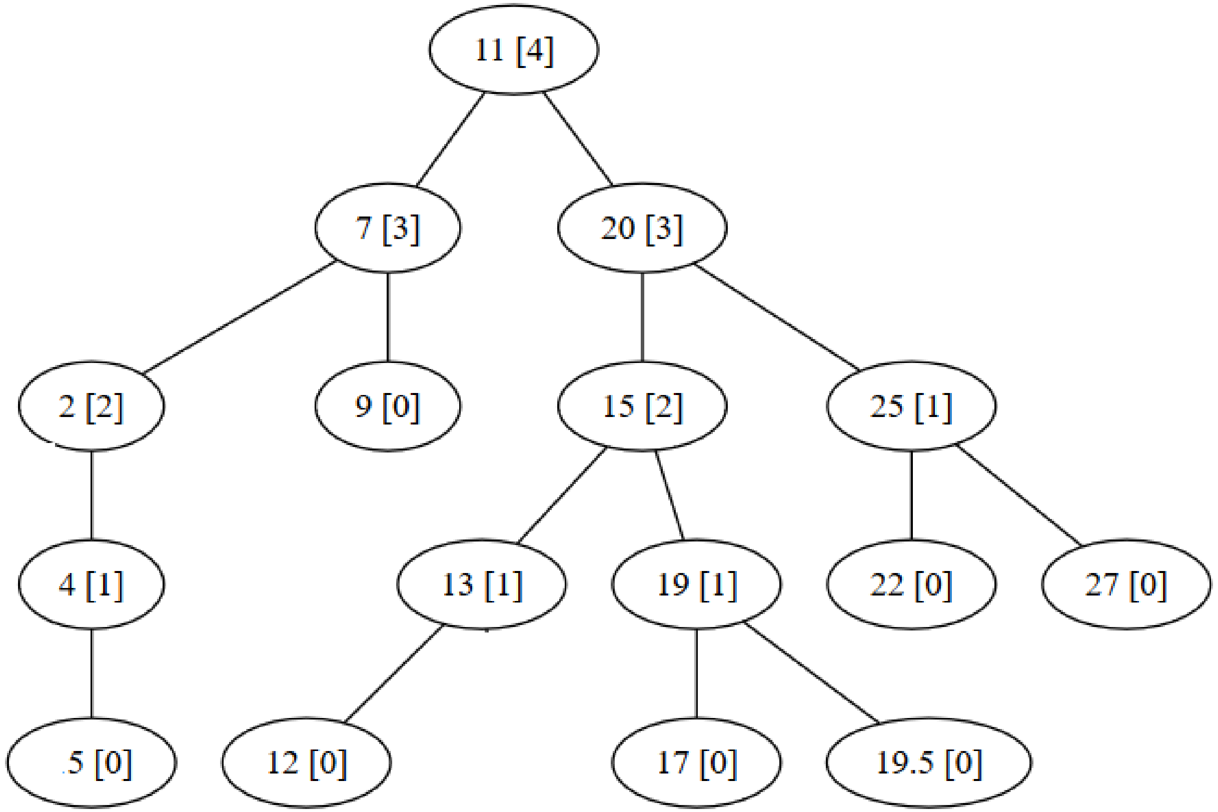


AVL Bäume – Rotationen – Fall 4 - Beispiel



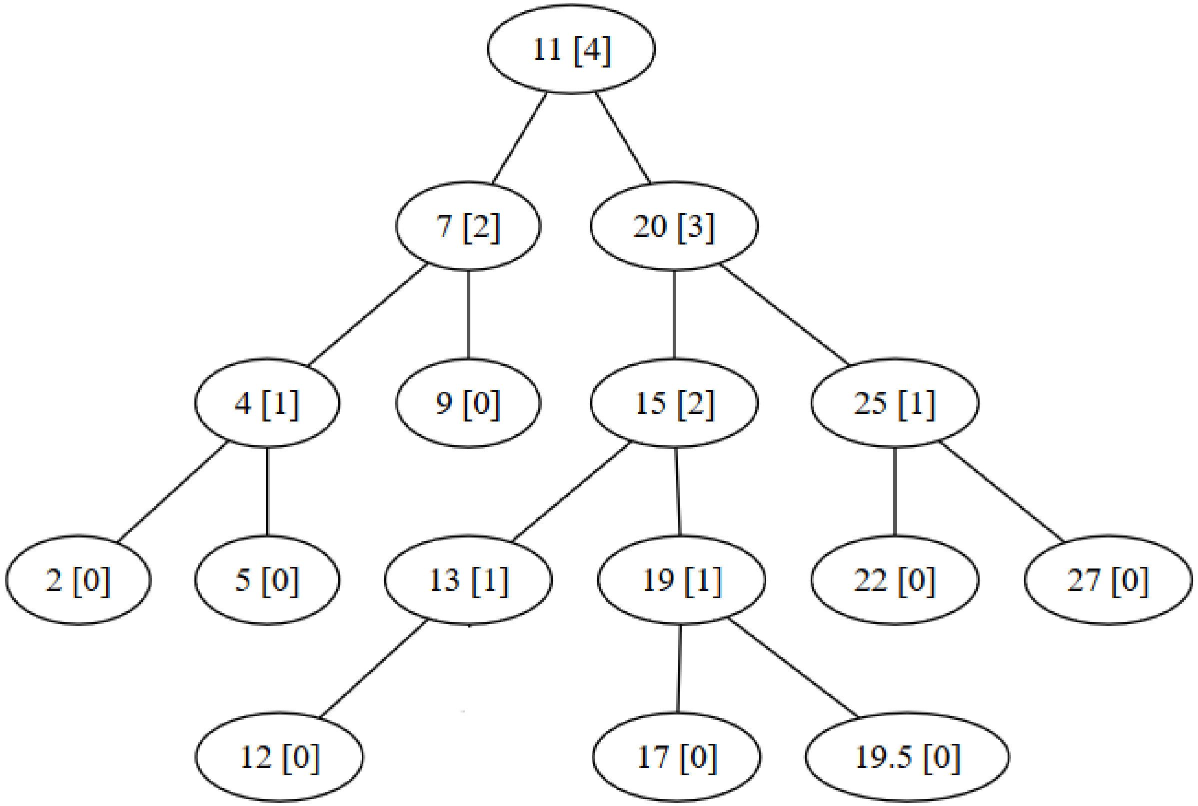
- Füge den Wert 5 ein

AVL Bäume – Rotationen – Fall 4 - Beispiel



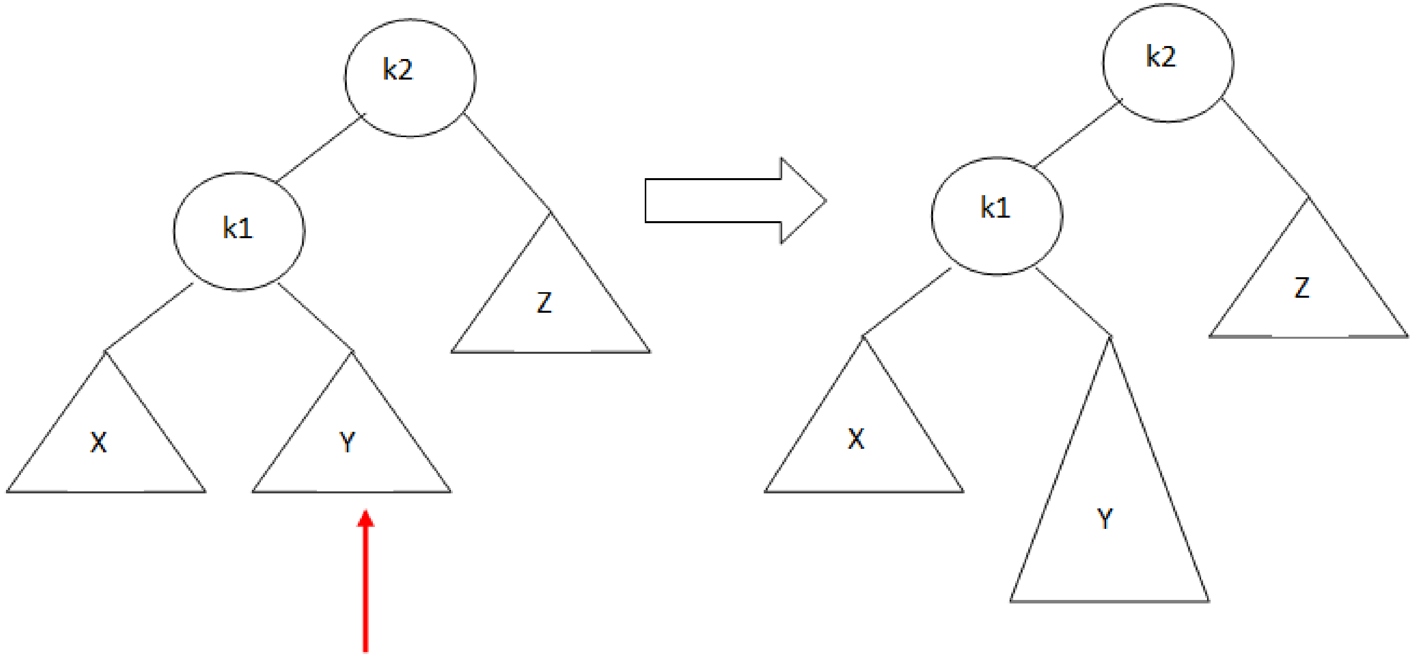
- Knoten 2 ist unbalanciert, da wir den Knoten 5 in dem rechten Teilbaum des rechten Kindes eingefügt haben

AVL Bäume – Rotationen – Fall 4 - Beispiel



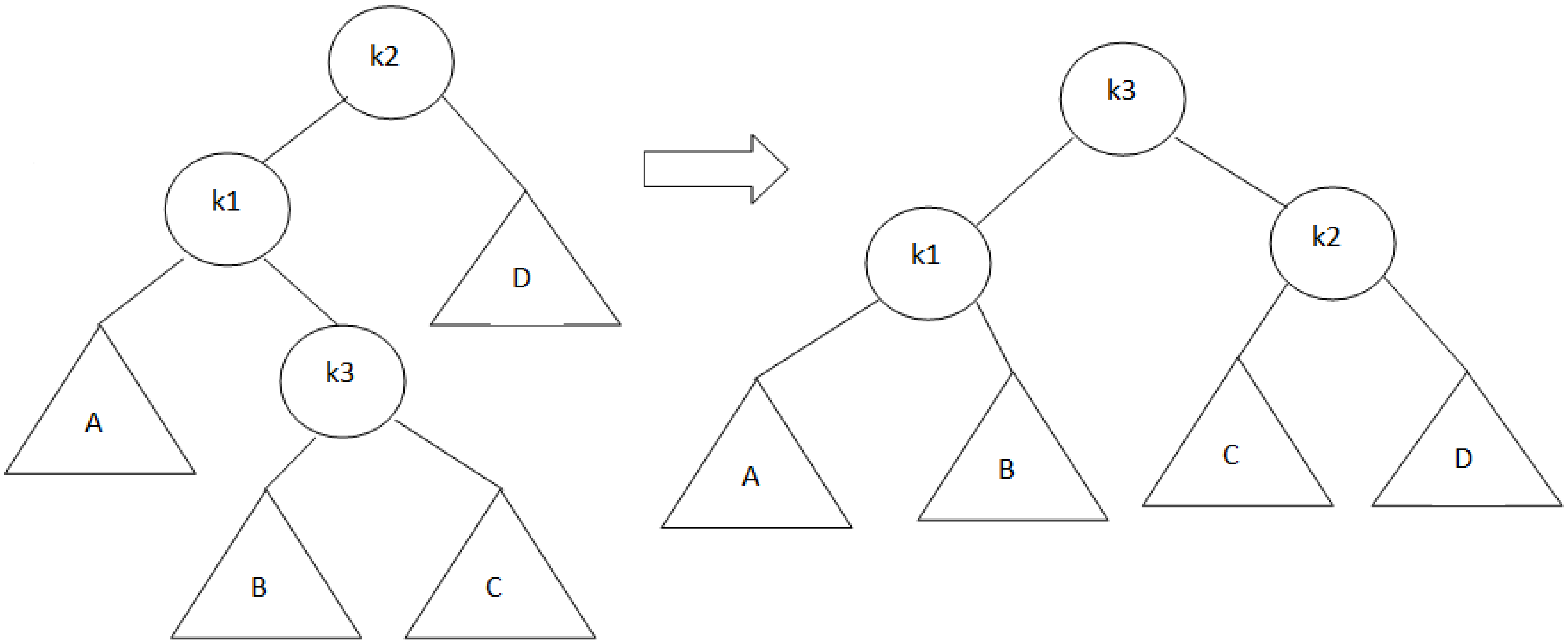
- Lösung: einfache Rotation nach links

AVL Bäume – Rotationen – Fall 2

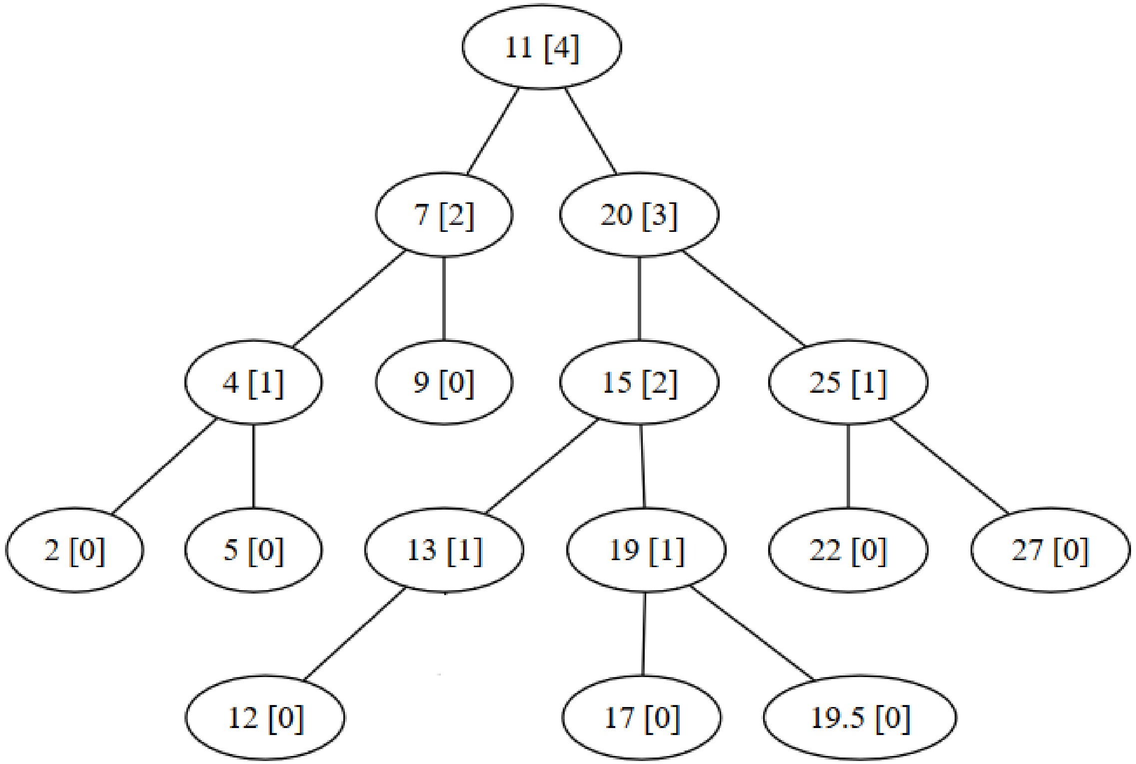


- Lösung: **doppelte Rotation nach rechts**

AVL Bäume – Rotationen – Fall 2

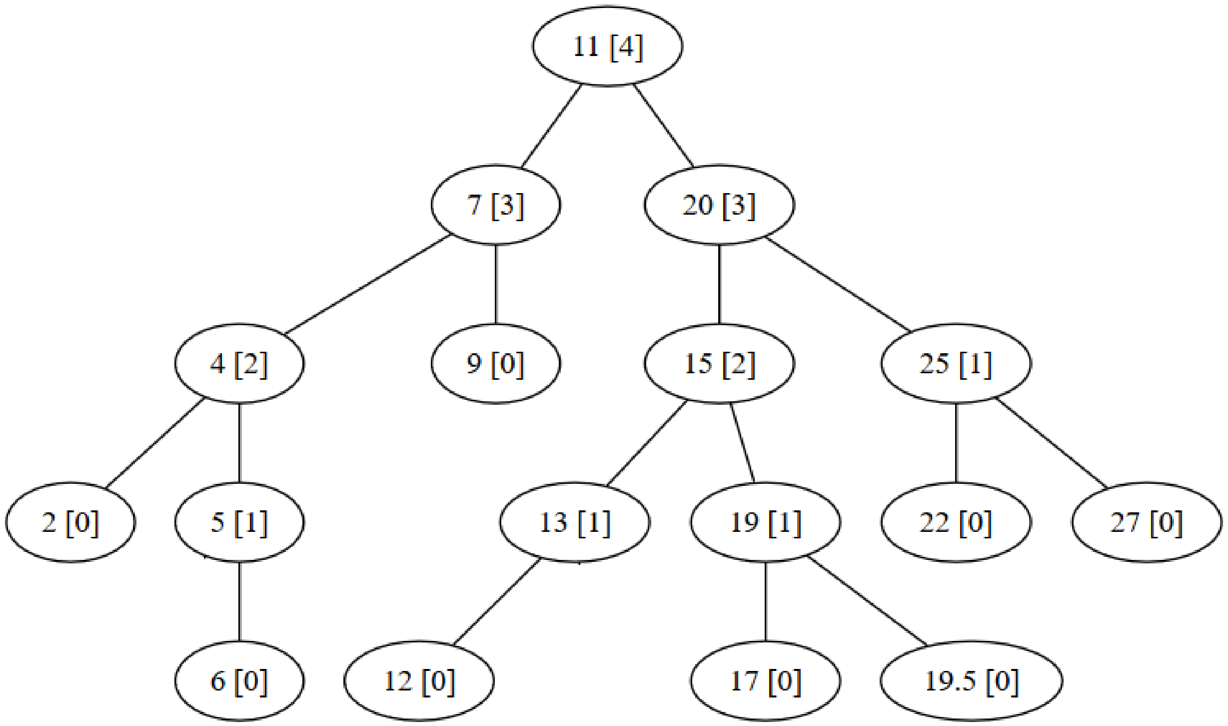


AVL Bäume – Rotationen – Fall 2 - Beispiel



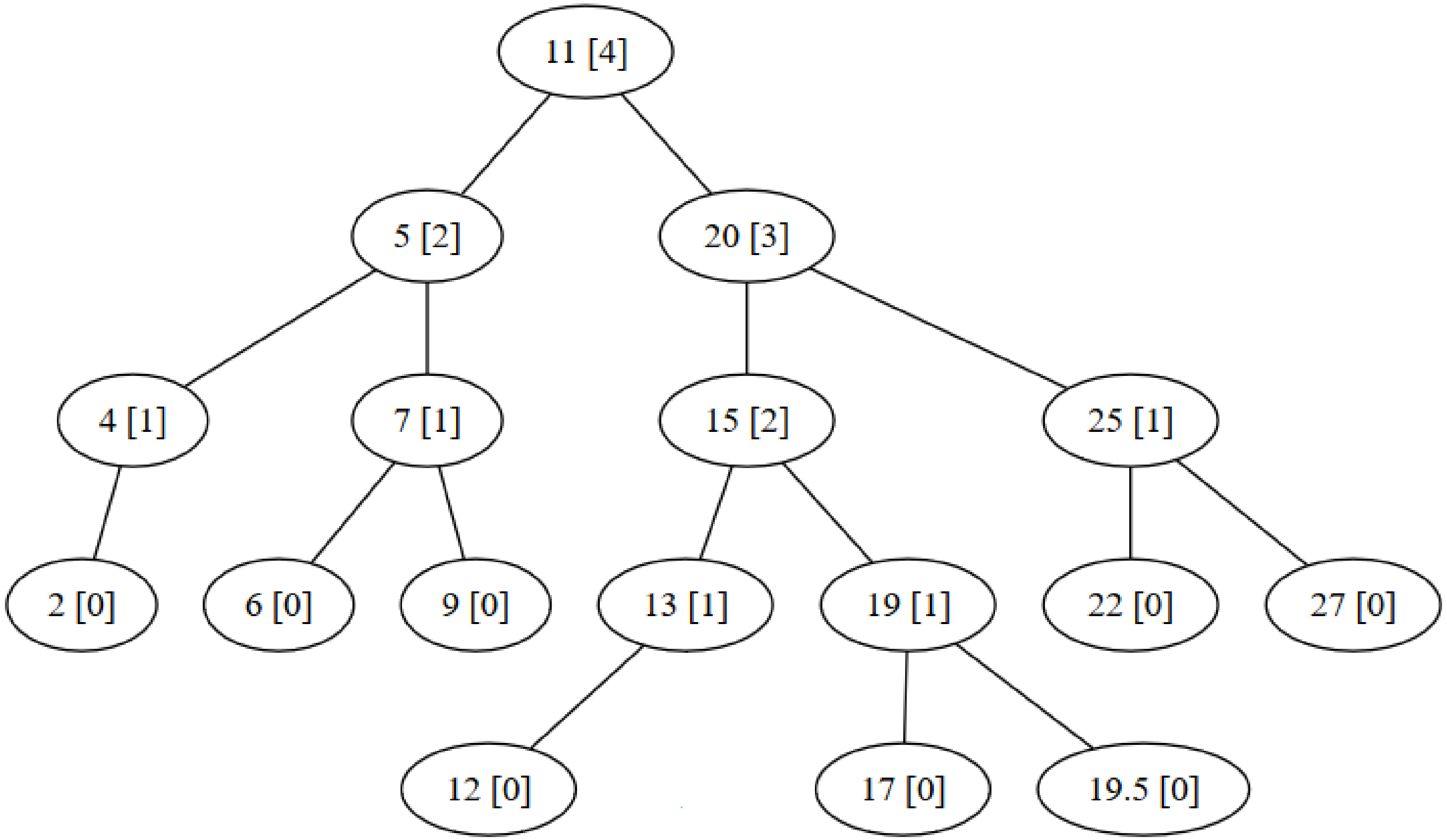
- Füge den Wert 6 ein

AVL Bäume – Rotationen – Fall 2 - Beispiel



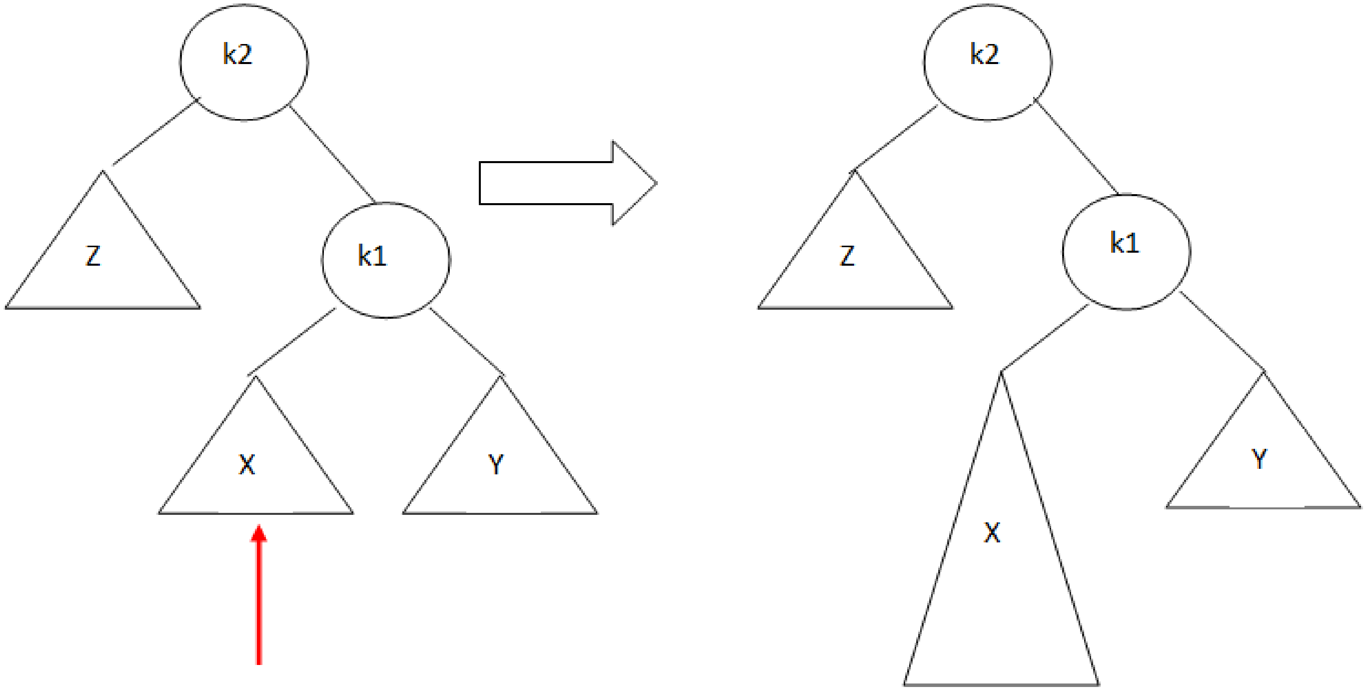
- Knoten 7 ist unbalanciert, da wir den Knoten 6 in dem rechten Teilbaum des linken Kindes eingefügt haben

AVL Bäume – Rotationen – Fall 2 - Beispiel



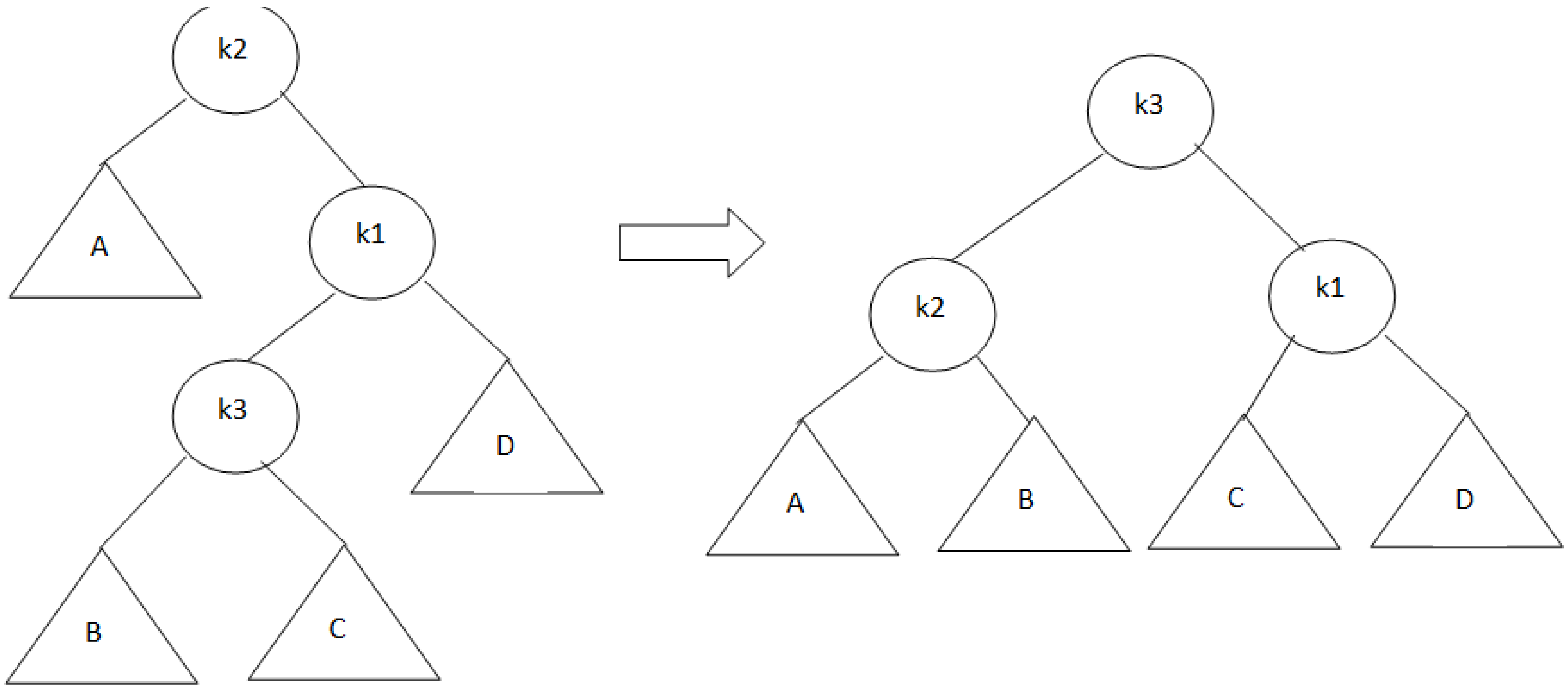
- Lösung: doppelte Rotation nach rechts

AVL Bäume – Rotationen – Fall 3

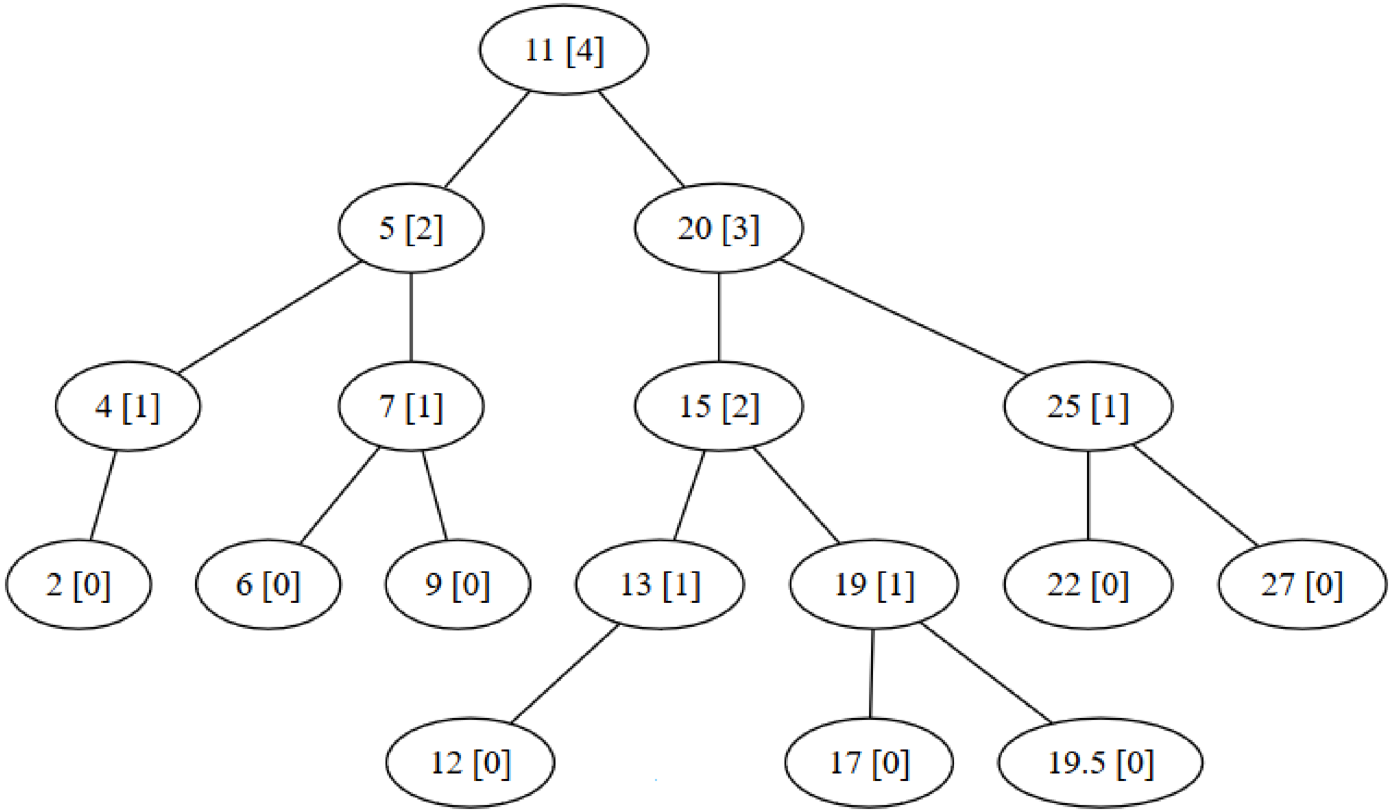


- Lösung: **doppelte Rotation nach links**

AVL Bäume – Rotationen – Fall 3

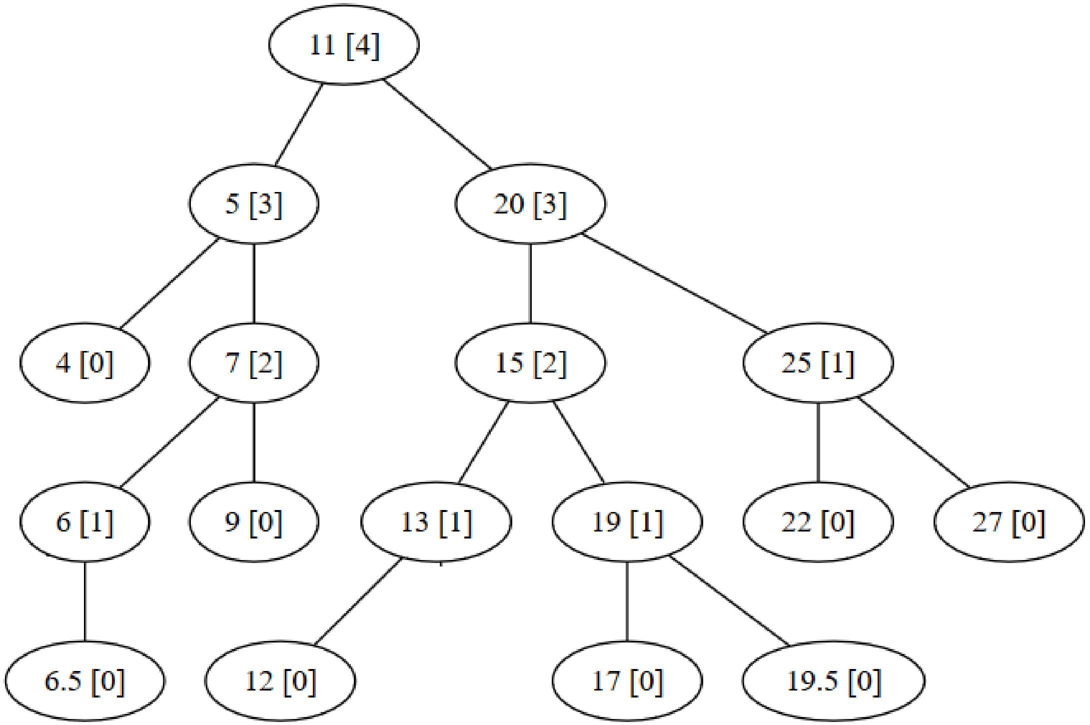


AVL Bäume – Rotationen – Fall 3 - Beispiel



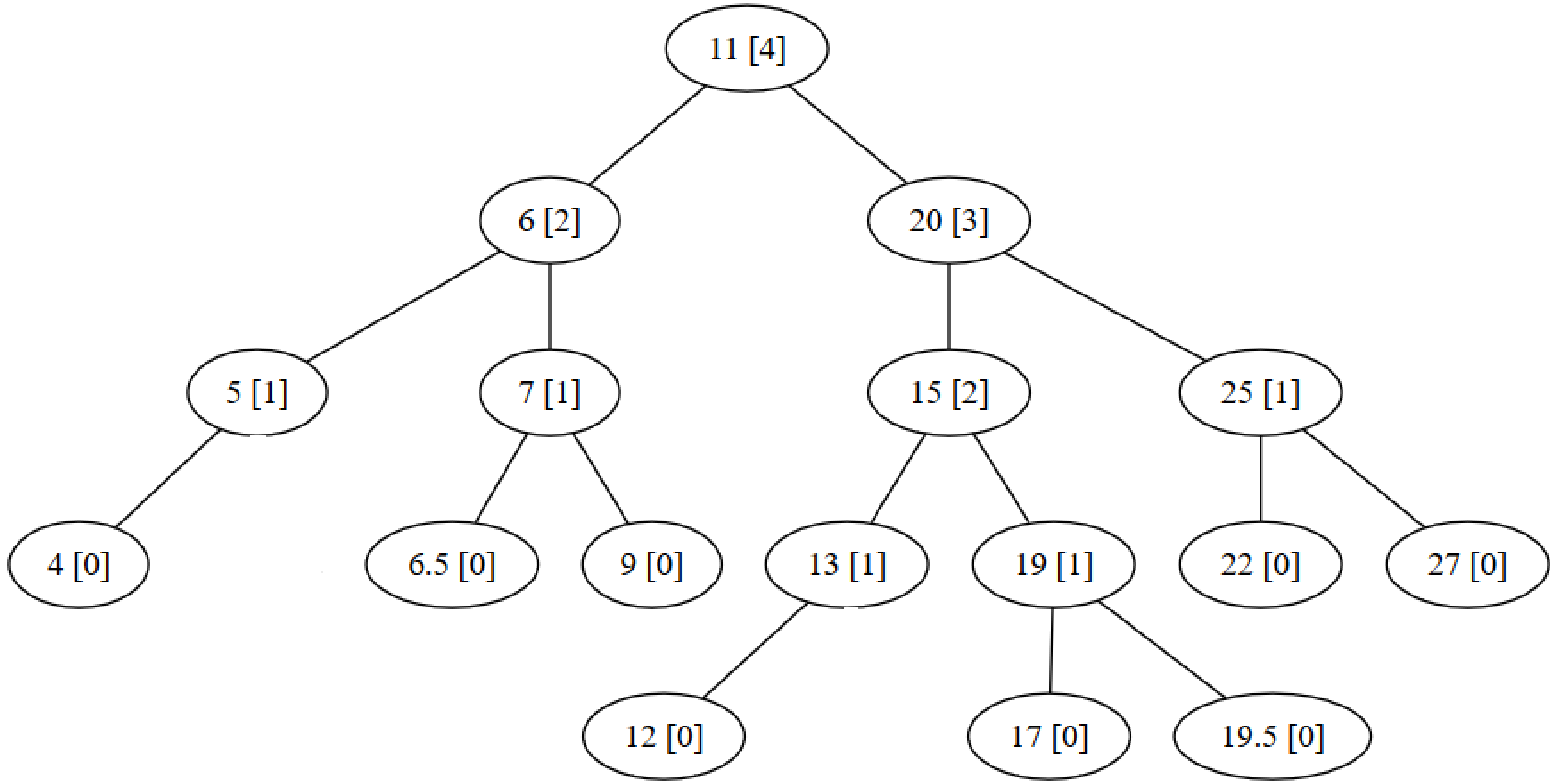
- Lösche den Wert 2 und füge den Wert 6.5 ein

AVL Bäume – Rotationen – Fall 3 - Beispiel



- Knoten 5 ist unbalanciert, da wir den Knoten 6.5 in dem linken Teilbaum des rechten Kindes eingefügt haben

AVL Bäume – Rotationen – Fall 3 - Beispiel



- Lösung: doppelte Rotation nach links