Datenstrukturen und Algorithmen

Vorlesung 13

Überblick

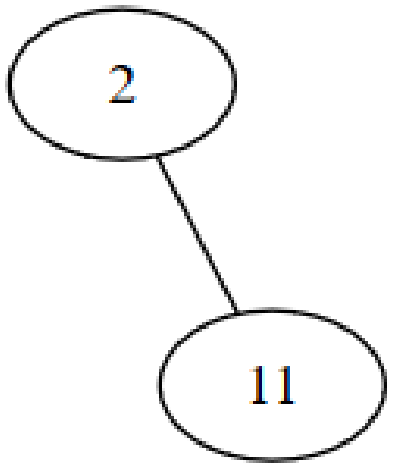
- Vorige Woche:
  - Binärbäume: Traversierungen in Postordnung und Level-Ordnung
  - Binärsuchbäume
  - AVL-Bäume
- Heute betrachten wir:
  - AVL Bäume
  - Huffman Codierung
  - Skip Listen
  - Infix- und Postfixnotation

AVL Rotationen Beispiel

- Wir fangen mit einem leeren AVL Baum an
- Füge 2 ein

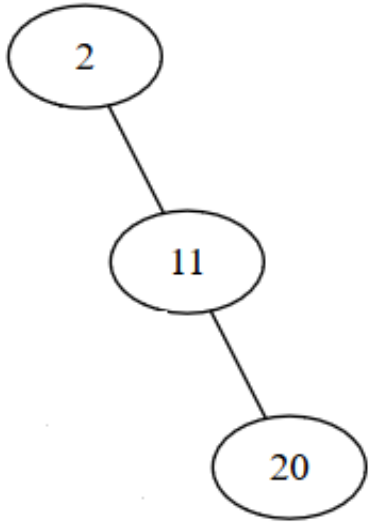
- Brauchen wir eine Rotation?
  - Nein
- Füge 11 ein

AVL Rotationen Beispiel



- Brauchen wir eine Rotation? Falls ja, welche Rotation?
  - Nein
- Füge 20 ein

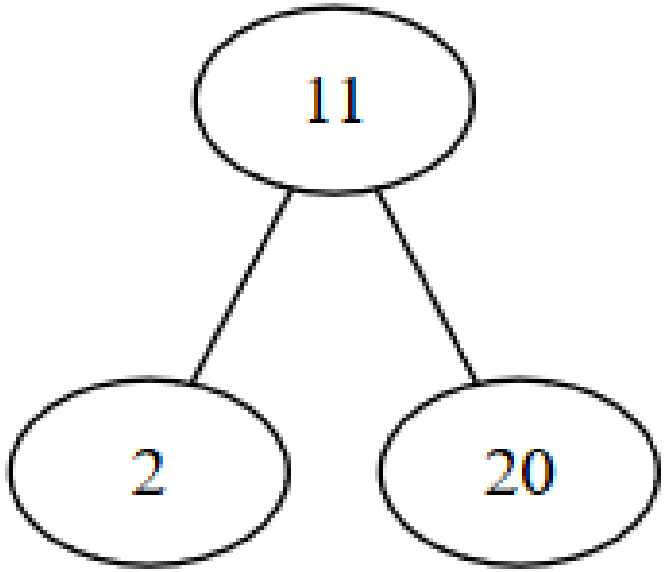
AVL Rotationen Beispiel



- Brauchen wir eine Rotation? Falls ja, welche Rotation?
  - Ja, eine einfache Rotation nach links

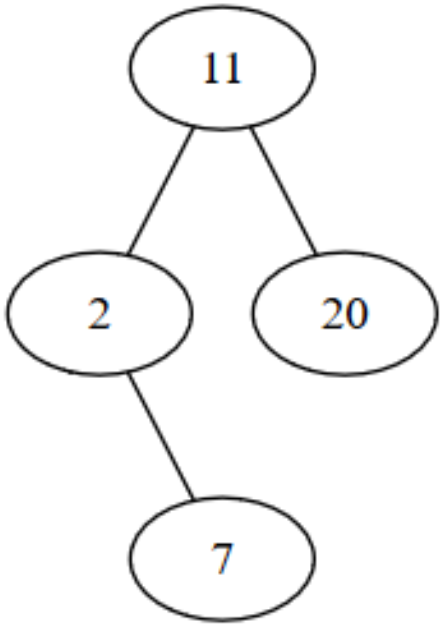
AVL Rotationen Beispiel

- Nach der Rotation:



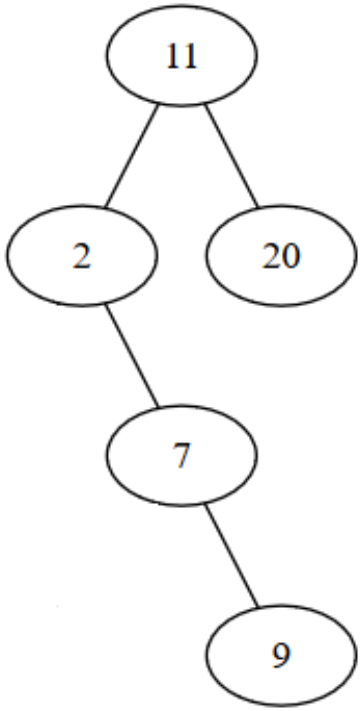
- Füge 7 ein

AVL Rotationen Beispiel



- Brauchen wir eine Rotation? Falls ja, welche Rotation?
  - Nein
- Füge 9 ein

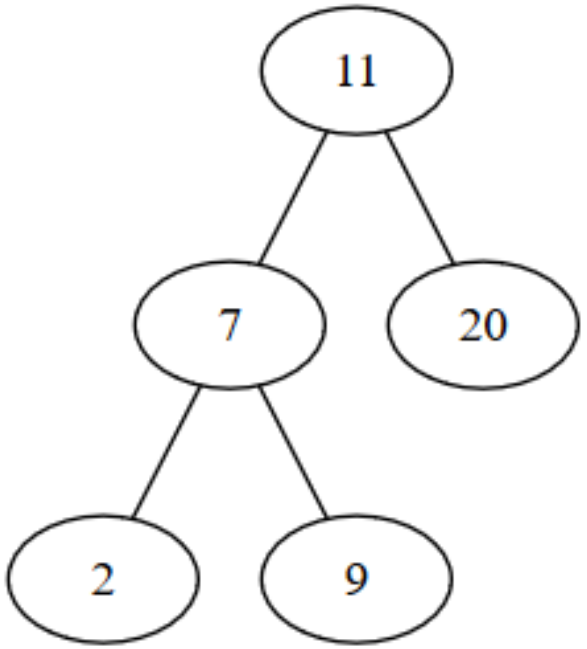
AVL Rotationen Beispiel



- Brauchen wir eine Rotation? Falls ja, welche Rotation?
  - Ja, eine einfache Rotation nach links für den Knoten 2

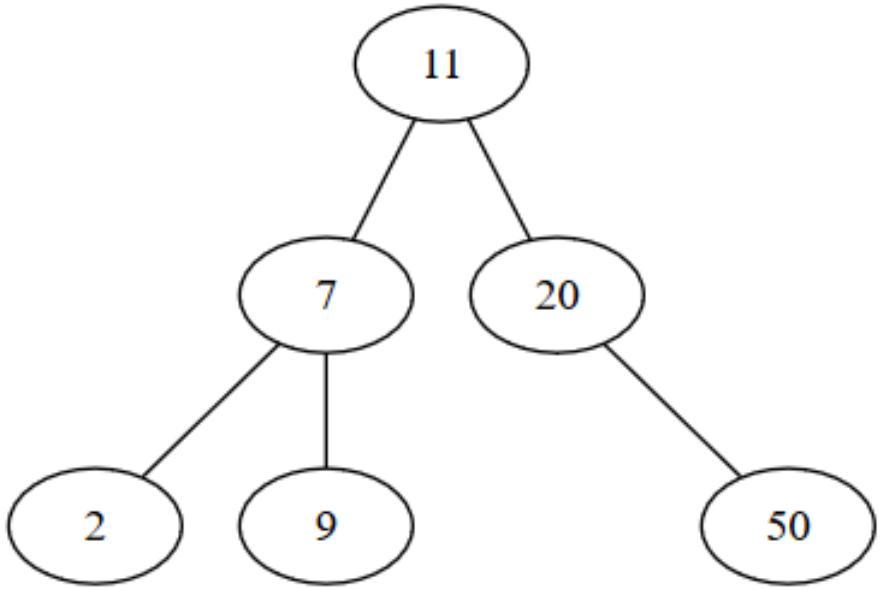
AVL Rotationen Beispiel

- Nach der Rotation:



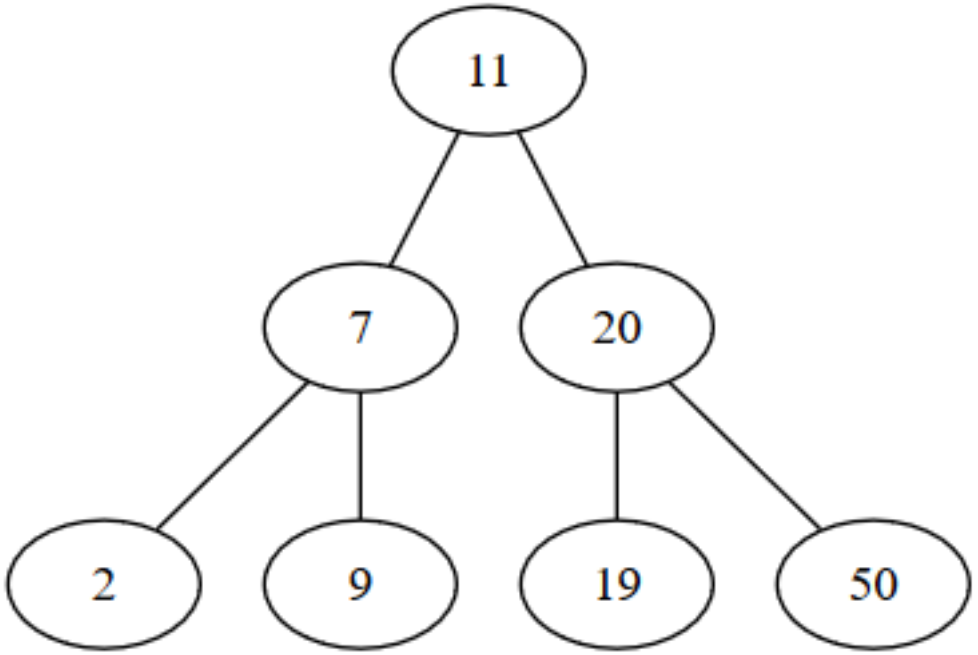
- Füge 50 ein

AVL Rotationen Beispiel



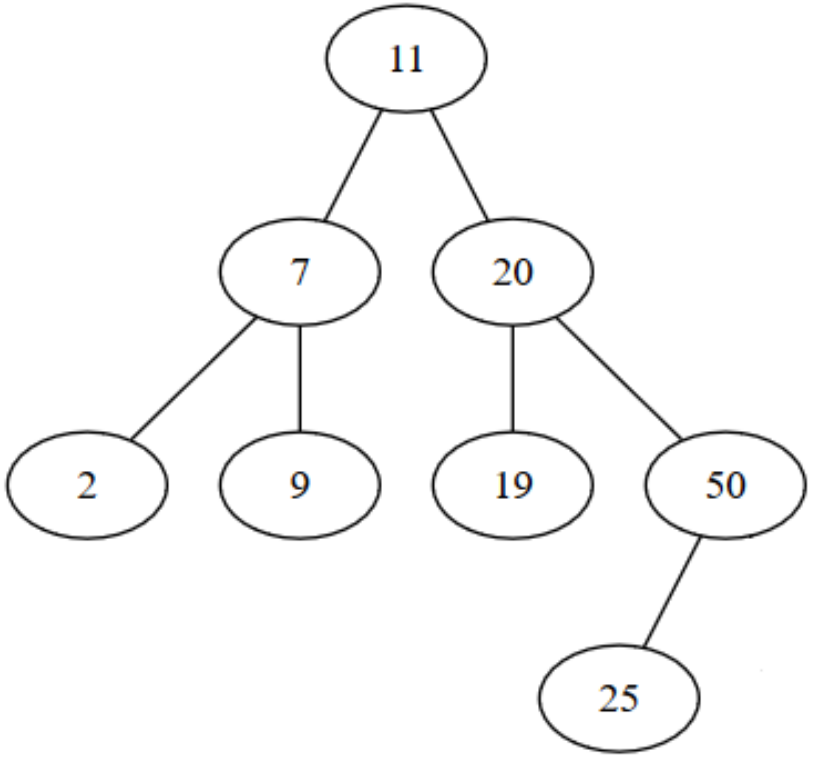
- Brauchen wir eine Rotation? Falls ja, welche Rotation?
  - Nein
- Füge 19 ein

AVL Rotationen Beispiel



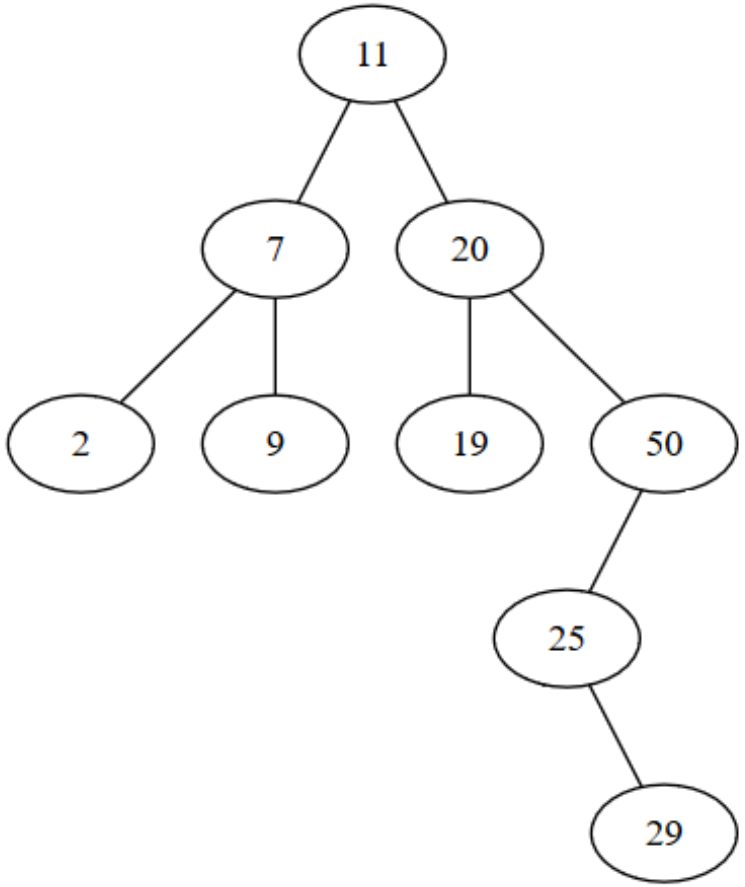
- Brauchen wir eine Rotation? Falls ja, welche Rotation?
  - Nein
- Füge 25 ein

AVL Rotationen Beispiel



- Brauchen wir eine Rotation? Falls ja, welche Rotation?
  - Nein
- Füge 29 ein

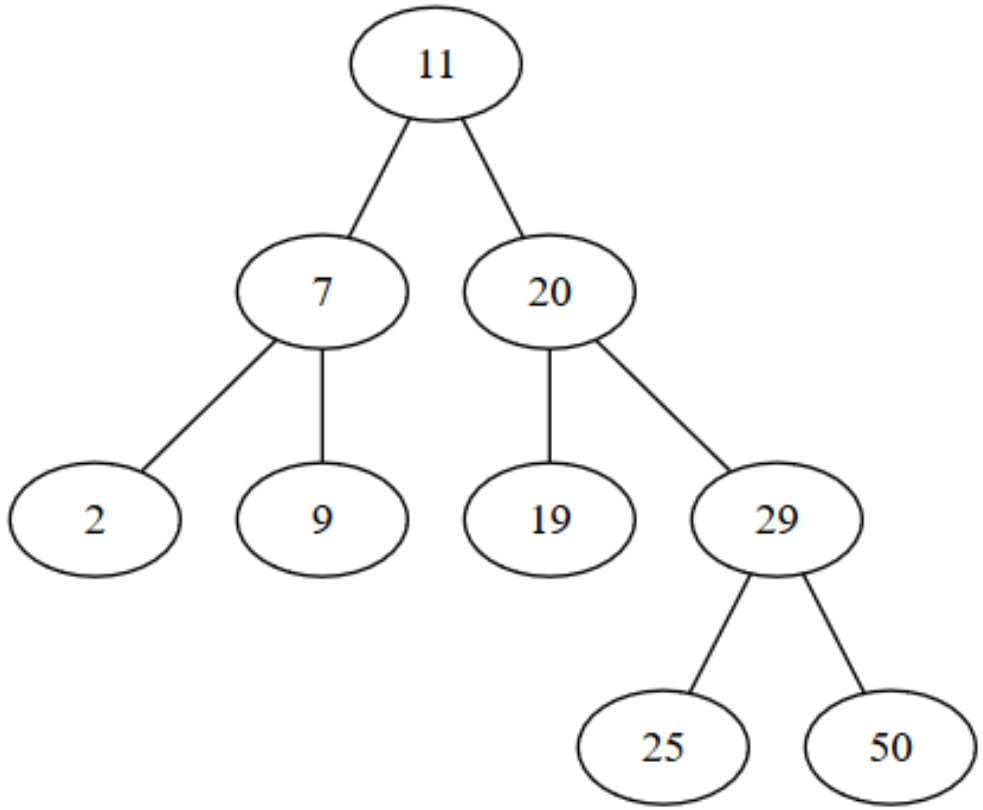
AVL Rotationen Beispiel



- Brauchen wir eine Rotation? Falls ja, welche Rotation?
  - Ja, eine doppelte Rotation nach rechts für den Knoten 50

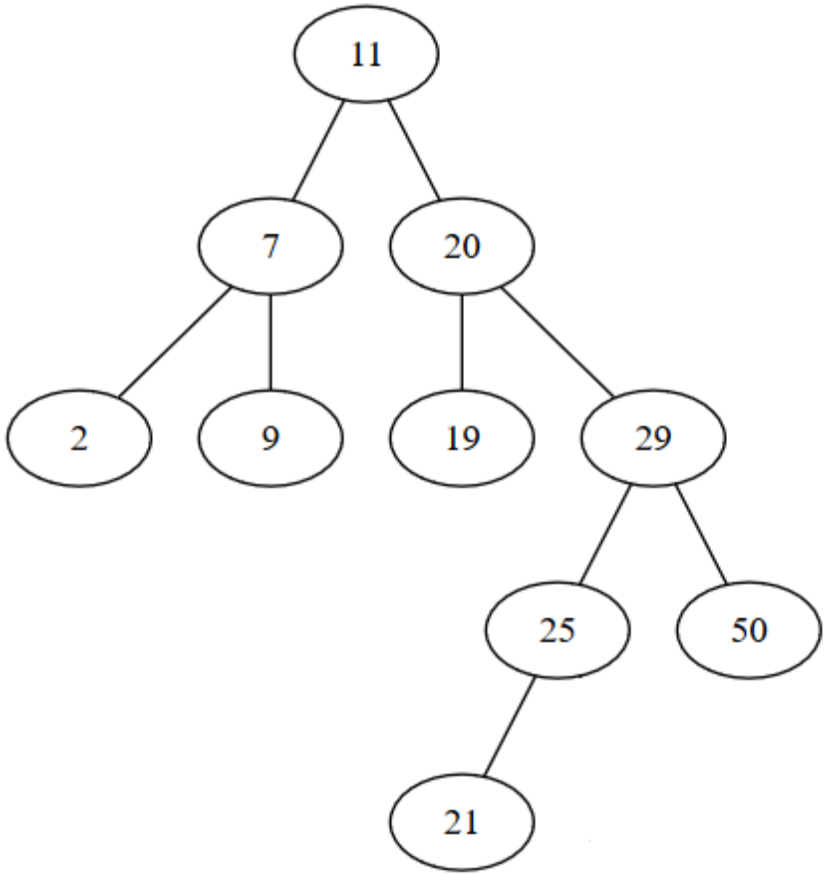
AVL Rotationen Beispiel

- Nach der Rotation:



- Füge 21 ein

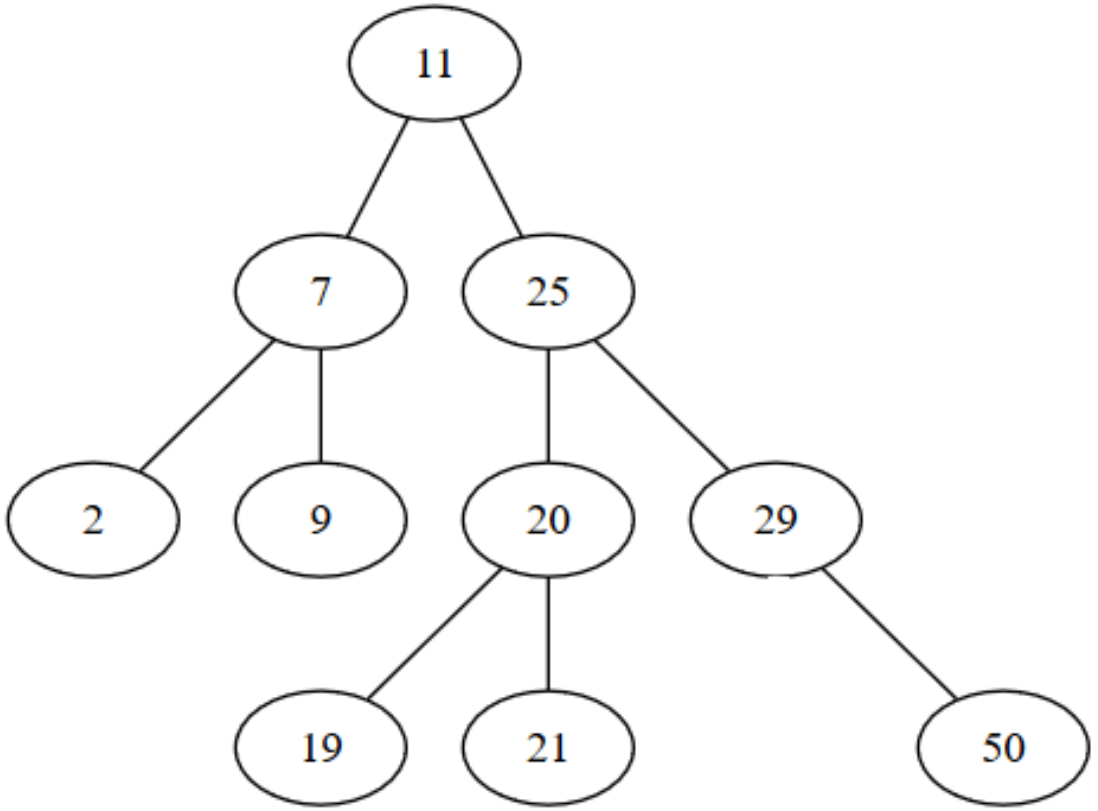
AVL Rotationen Beispiel



- Brauchen wir eine Rotation? Falls ja, welche Rotation?
  - Ja, eine doppelte Rotation nach links für den Knoten 20

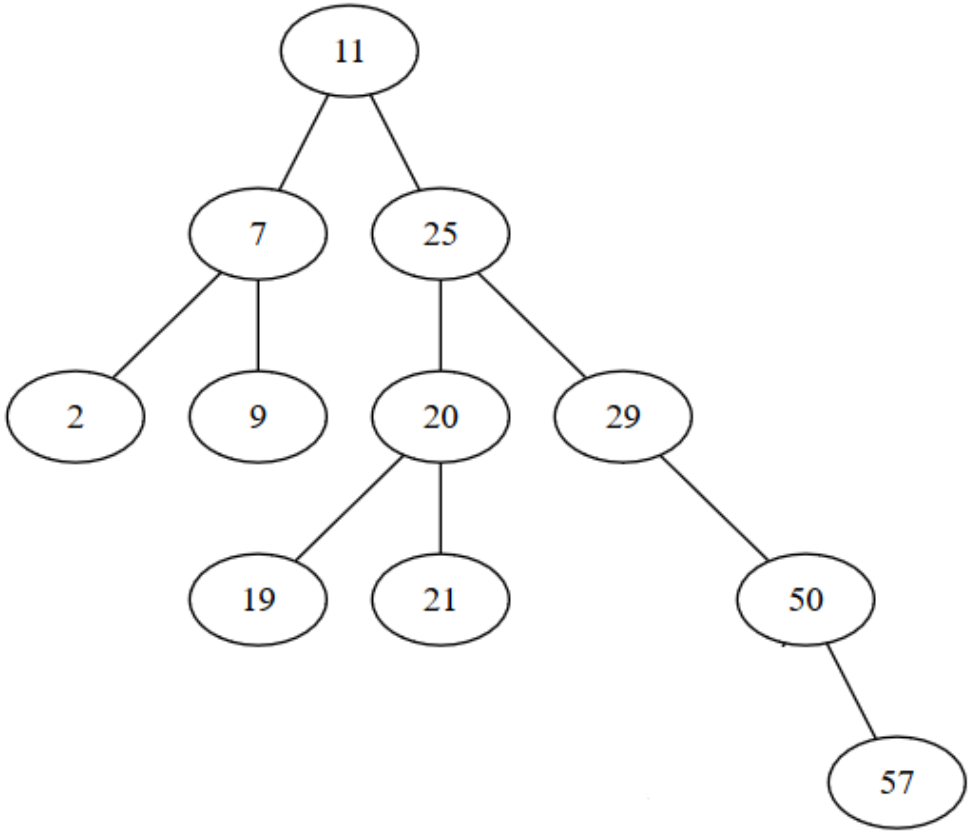
AVL Rotationen Beispiel

- Nach der Rotation:



- Füge 57 ein

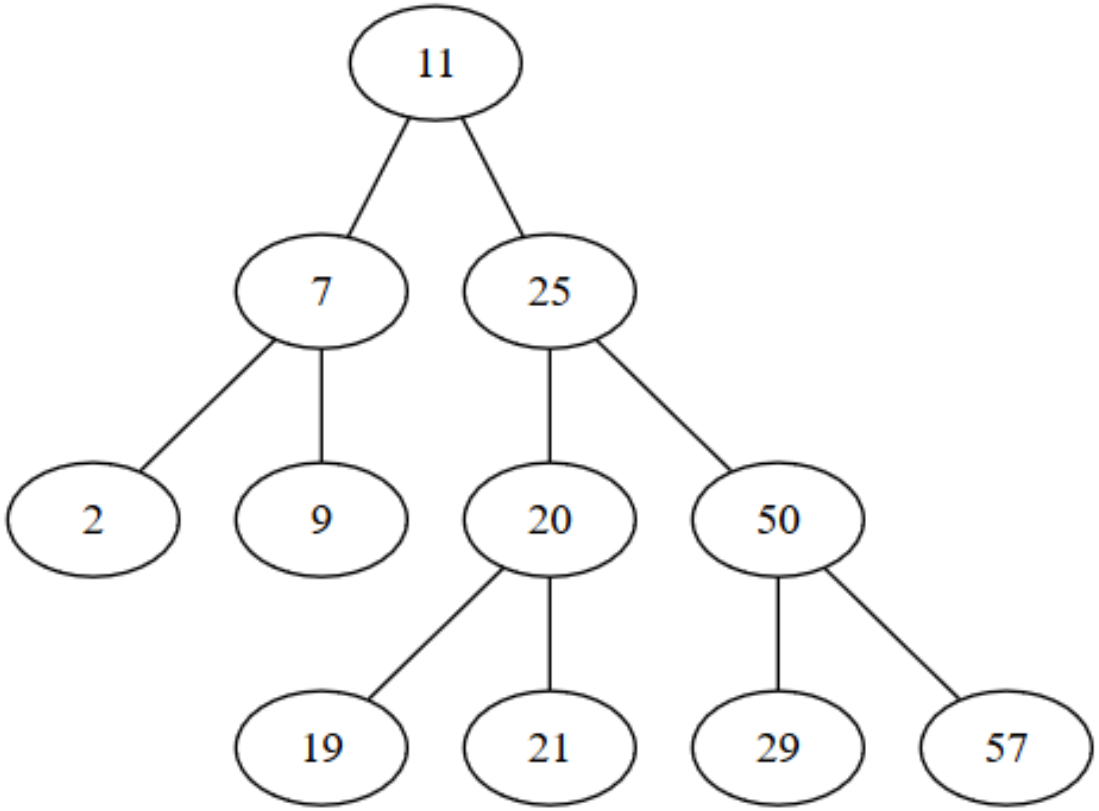
AVL Rotationen Beispiel



- Brauchen wir eine Rotation? Falls ja, welche Rotation?
  - Ja, eine einfache Rotation nach links für den Knoten 29

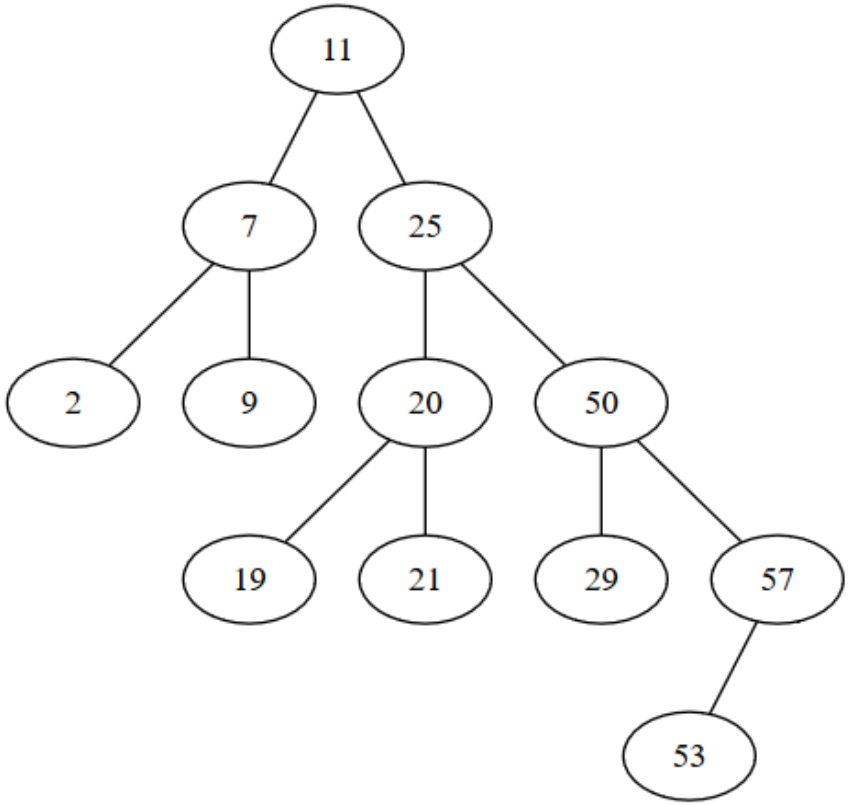
AVL Rotationen Beispiel

- Nach der Rotation:



- Füge 53 ein

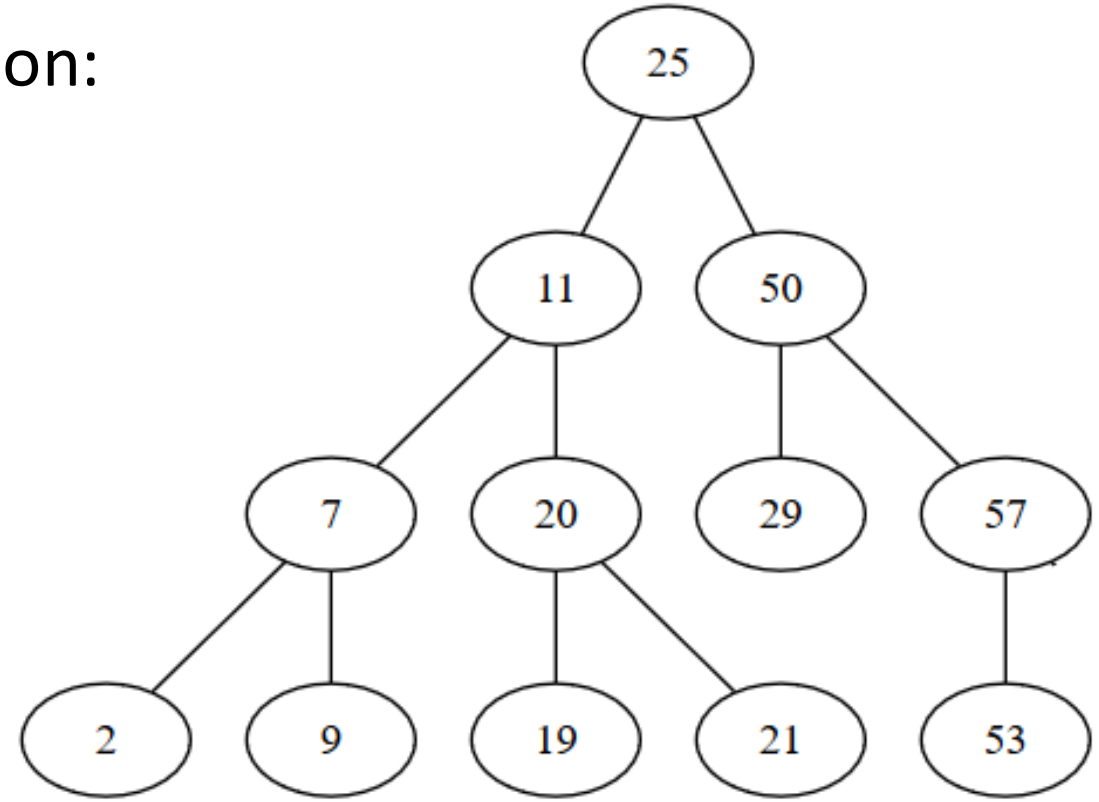
AVL Rotationen Beispiel



- Brauchen wir eine Rotation? Falls ja, welche Rotation?
  - Ja, eine einfache Rotation nach links für den Knoten 11

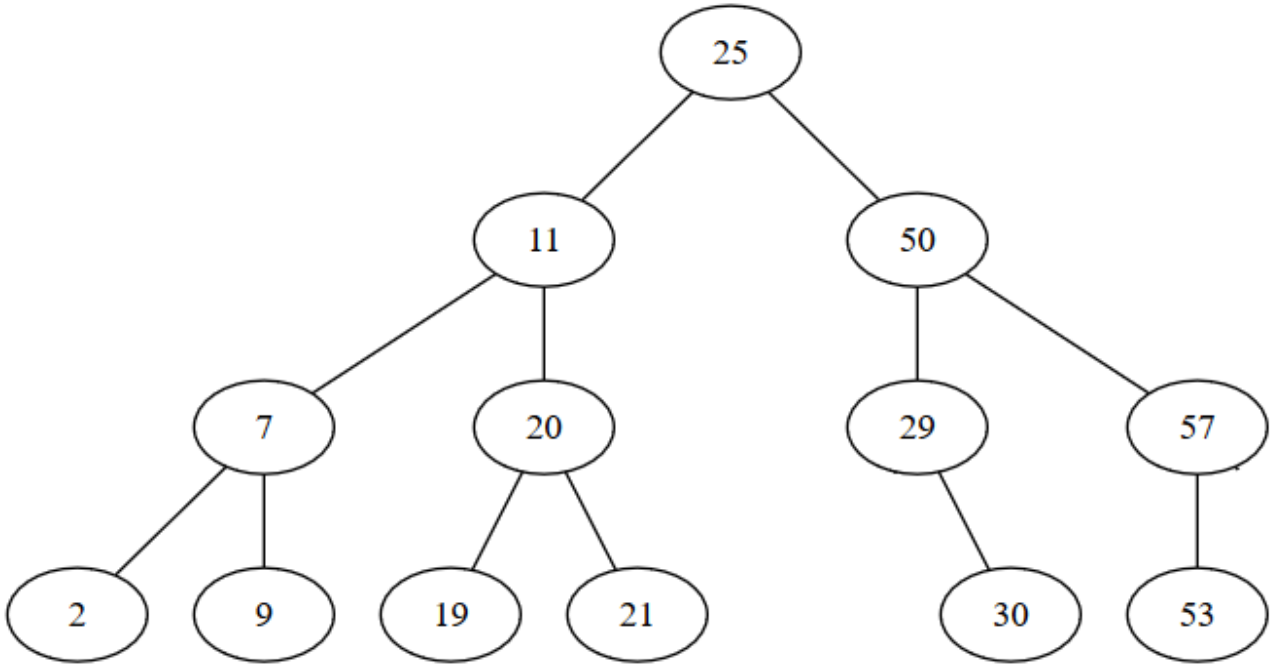
AVL Rotationen Beispiel

- Nach der Rotation:



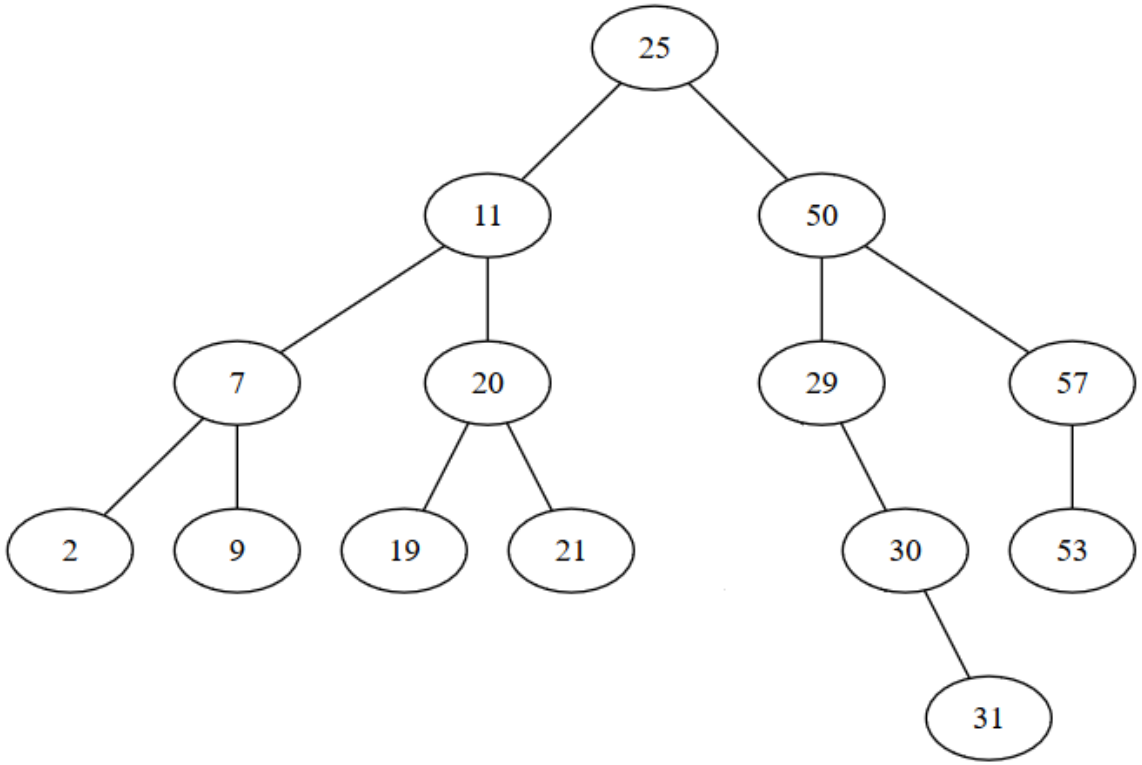
- Füge 30 ein

AVL Rotationen Beispiel



- Brauchen wir eine Rotation? Falls ja, welche Rotation?
  - Nein
- Füge 31 ein

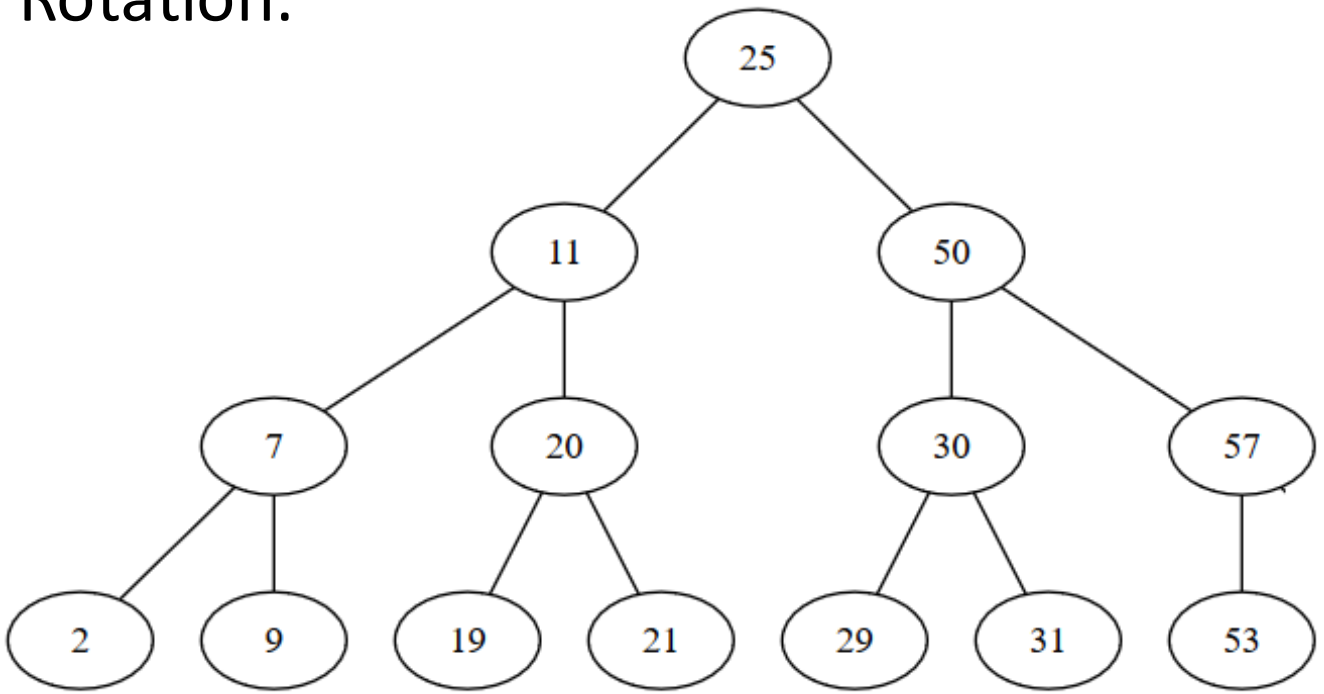
AVL Rotationen Beispiel



- Brauchen wir eine Rotation? Falls ja, welche Rotation?
  - Ja, eine einfache Rotation nach links für den Knoten 29

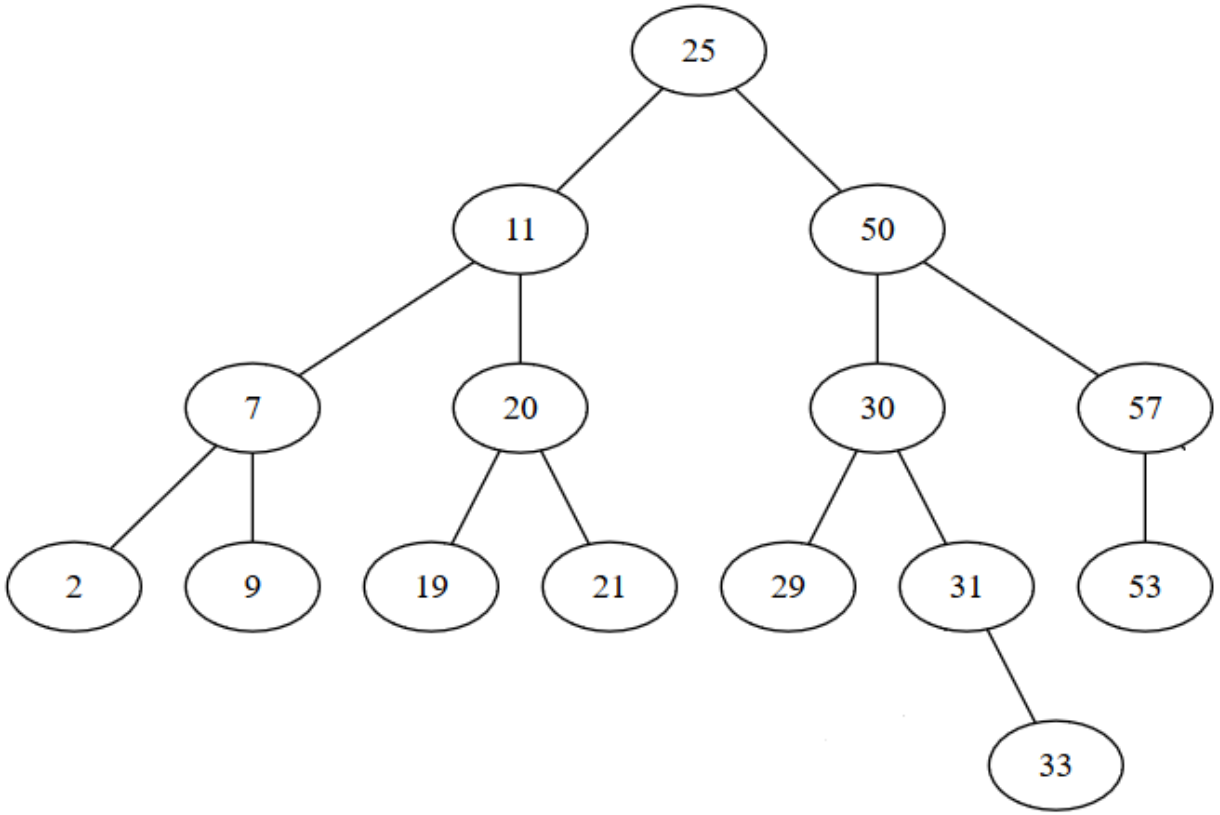
AVL Rotationen Beispiel

- Nach der Rotation:



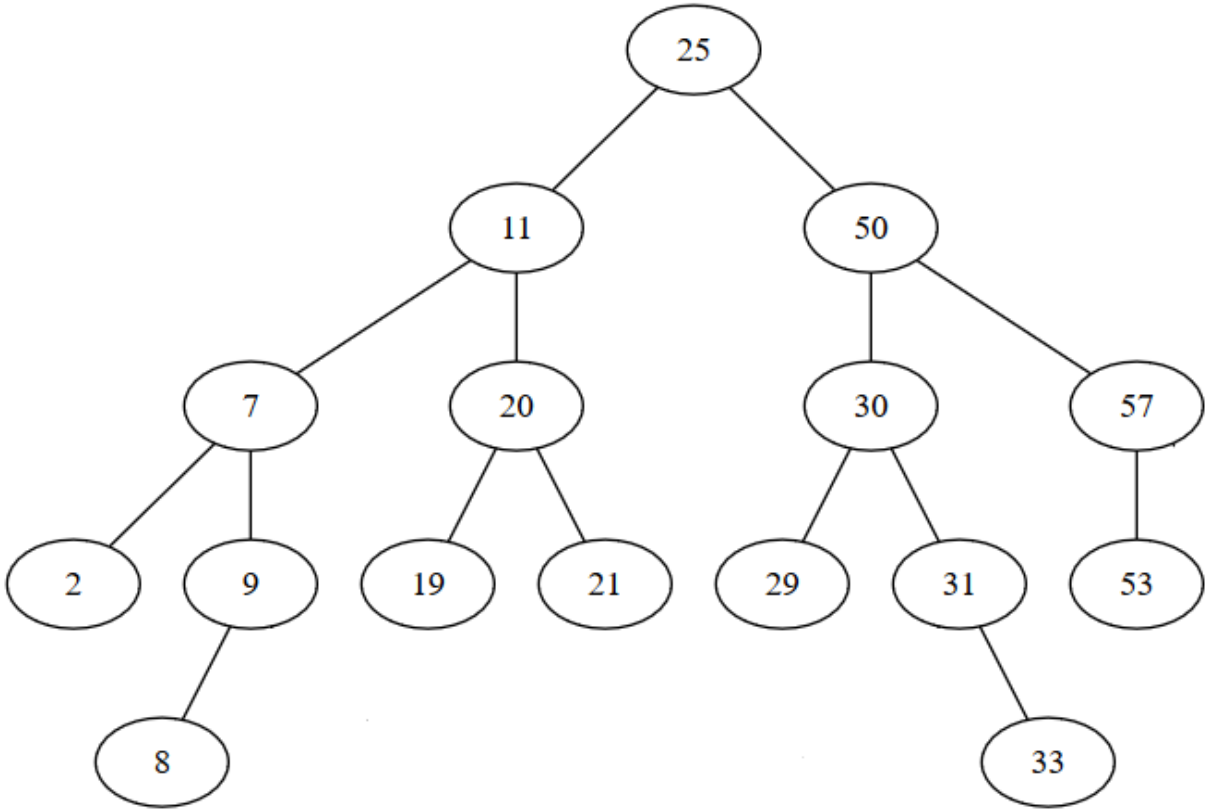
- Füge 33 ein

AVL Rotationen Beispiel



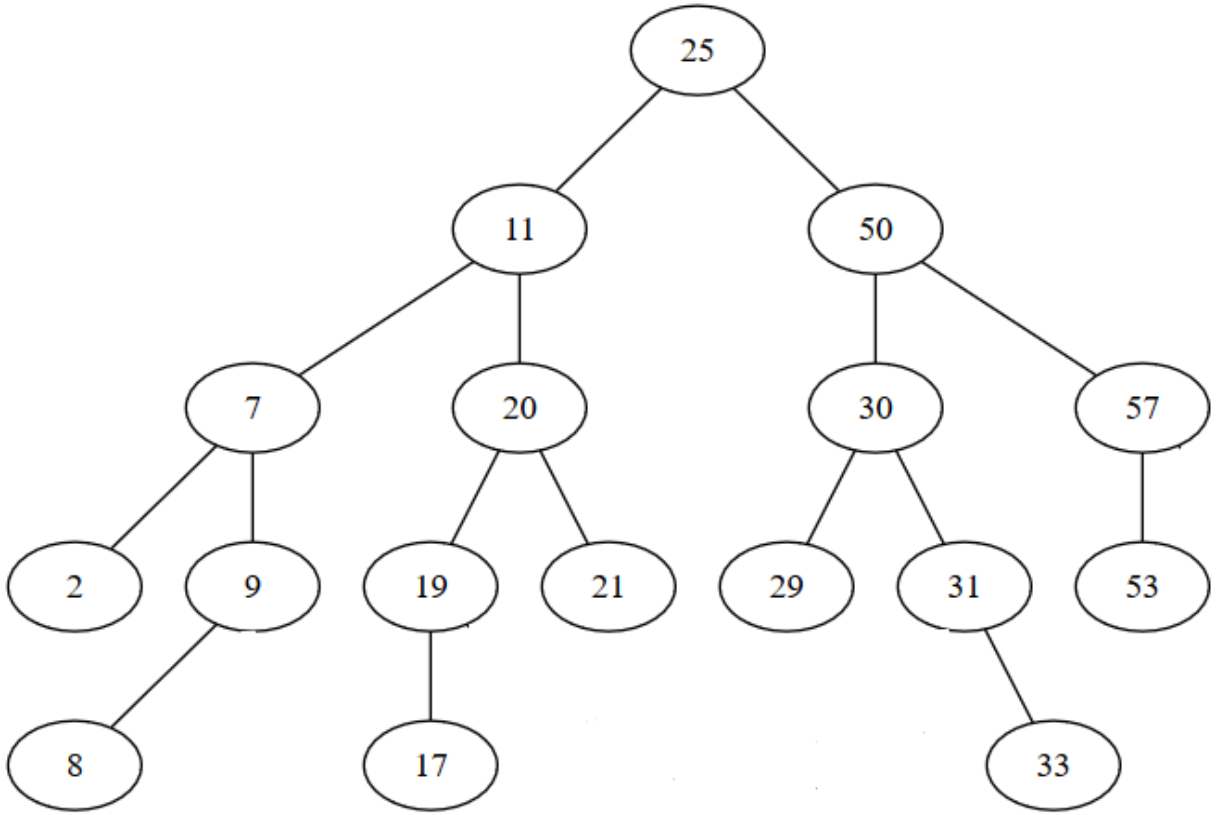
- Brauchen wir eine Rotation? Falls ja, welche Rotation?
  - Nein
- Füge 8 ein

AVL Rotationen Beispiel



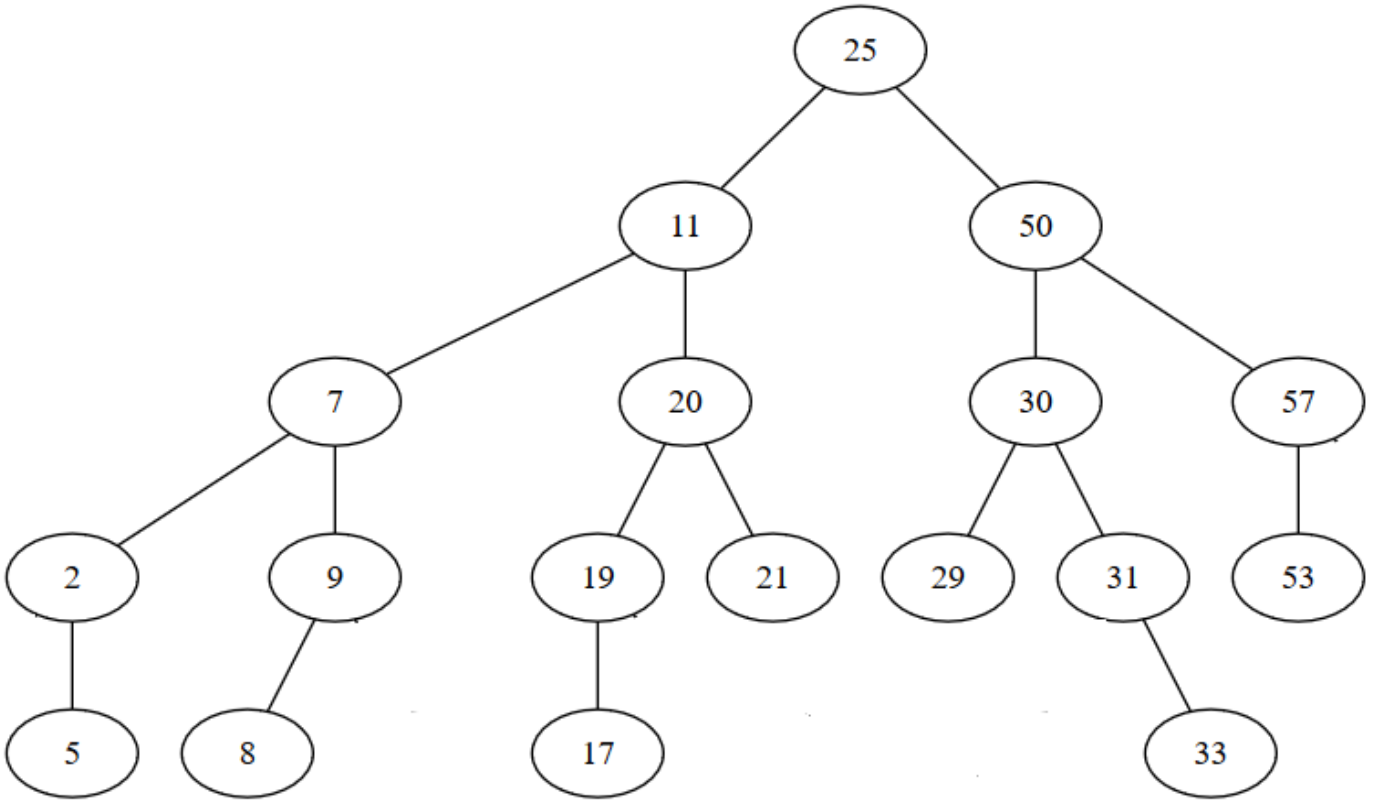
- Brauchen wir eine Rotation? Falls ja, welche Rotation?
  - Nein
- Füge 17 ein

AVL Rotationen Beispiel



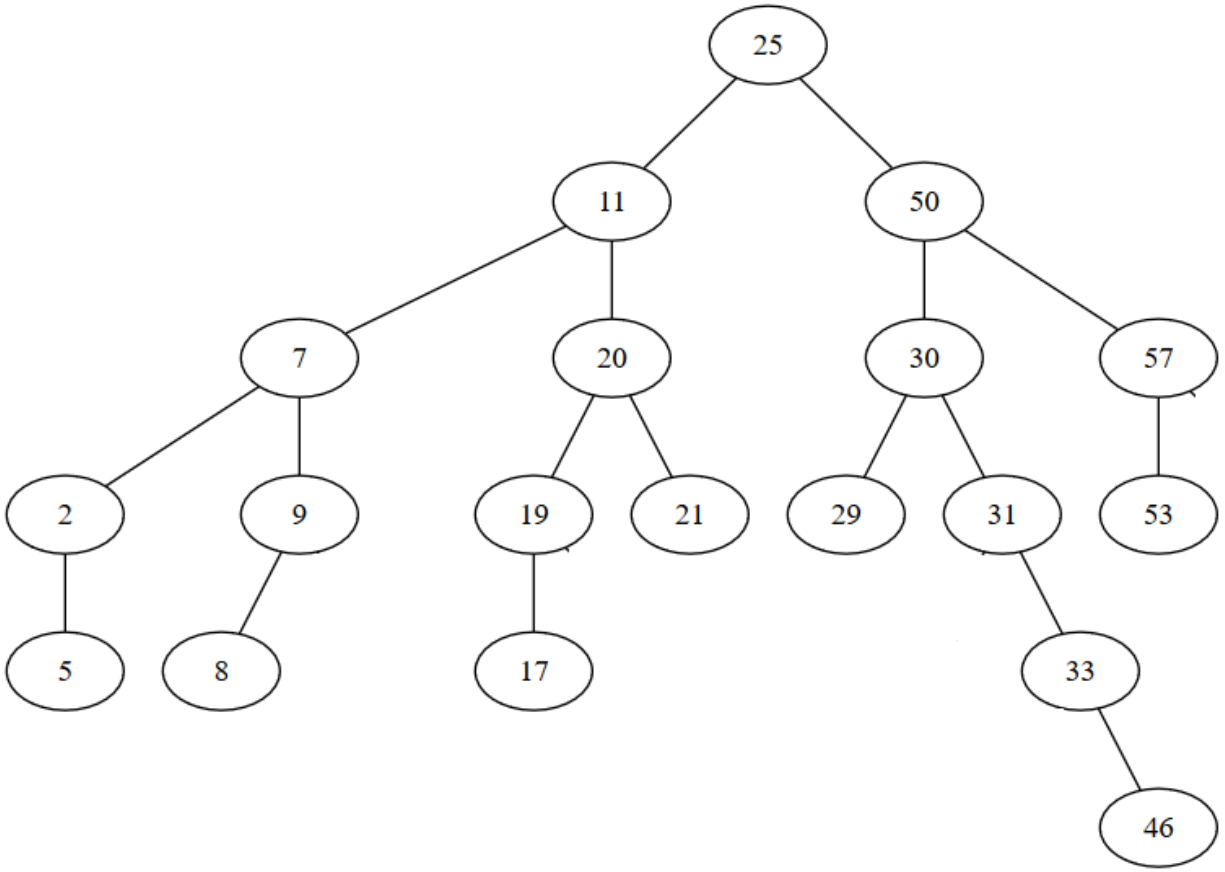
- Brauchen wir eine Rotation? Falls ja, welche Rotation?
  - Nein
- Füge 5 ein

AVL Rotationen Beispiel



- Brauchen wir eine Rotation? Falls ja, welche Rotation?
  - Nein
- Füge 46 ein

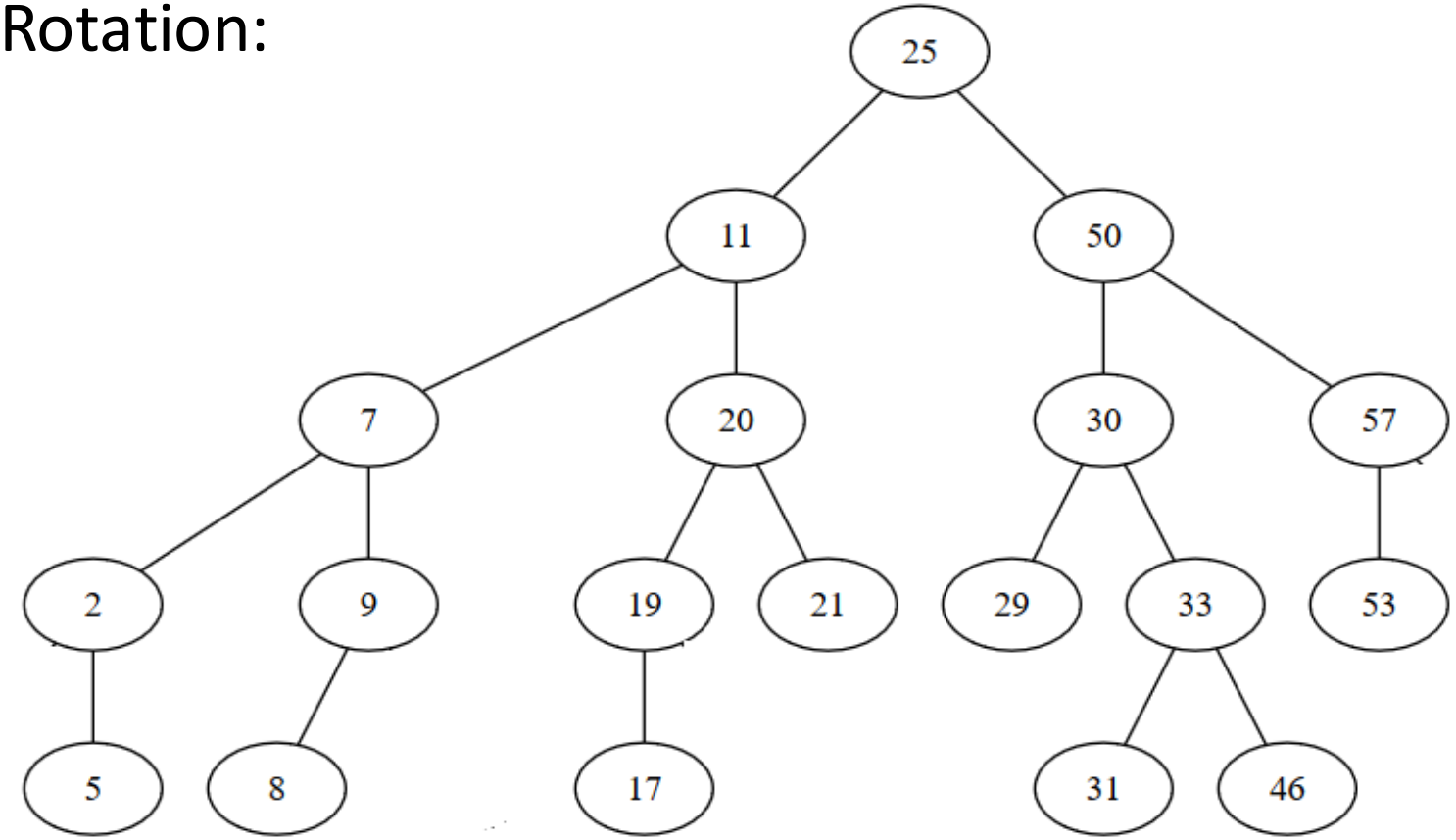
AVL Rotationen Beispiel



- Brauchen wir eine Rotation? Falls ja, welche Rotation?
  - Ja, eine einfache Rotation nach links für den Knoten 31

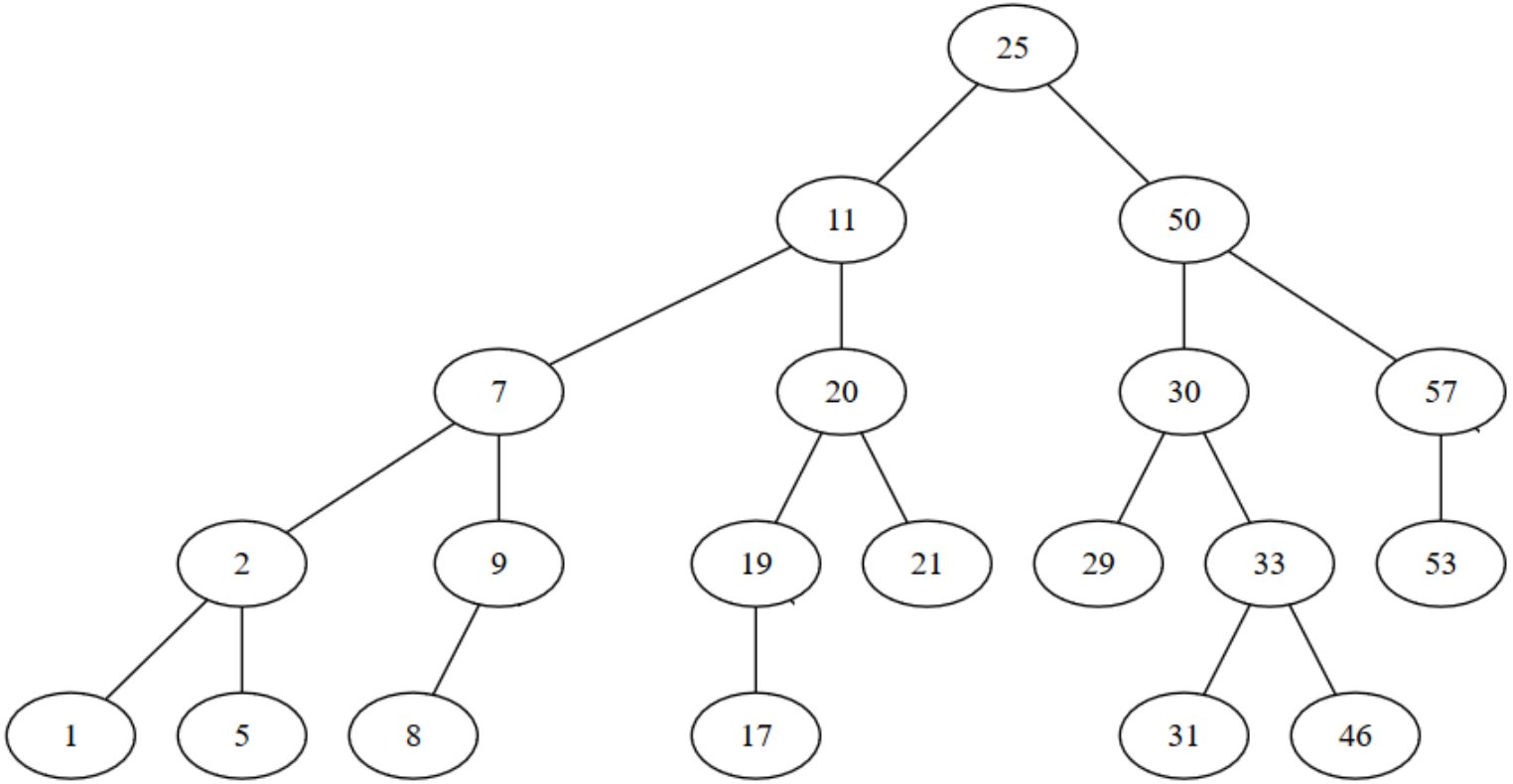
AVL Rotationen Beispiel

- Nach der Rotation:



- Füge 1 ein

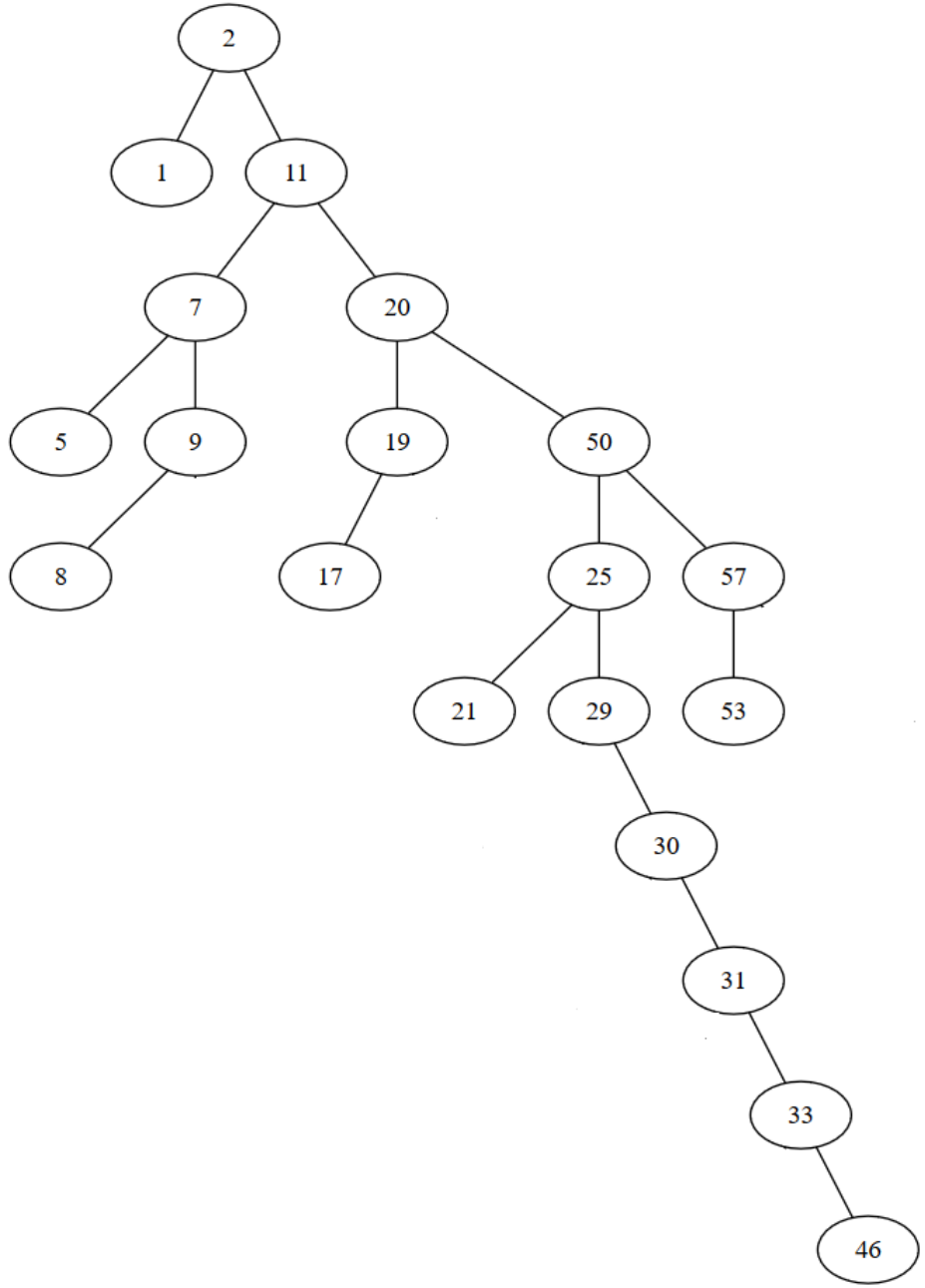
AVL Rotationen Beispiel



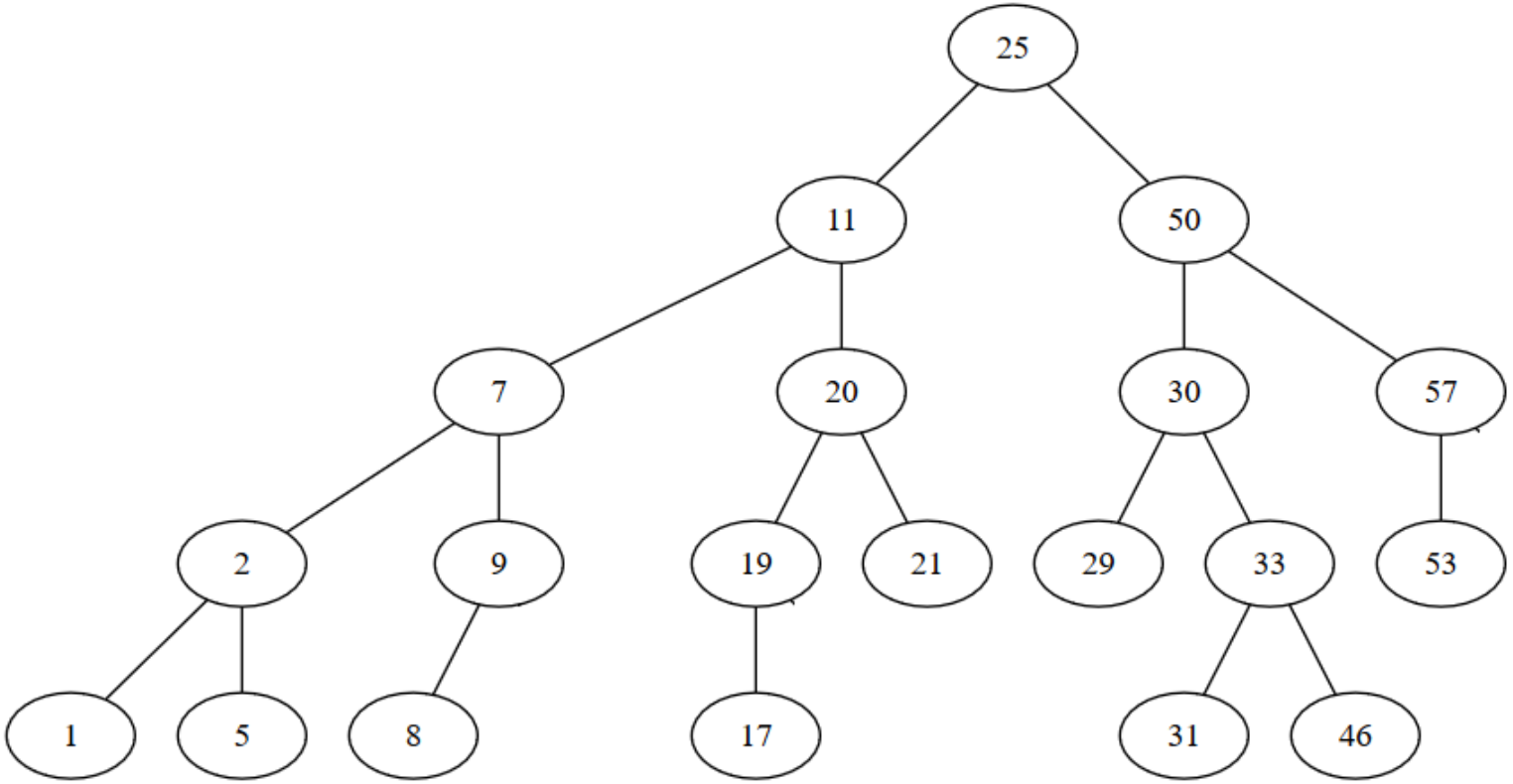
- Brauchen wir eine Rotation? Falls ja, welche Rotation?
  - Nein

Vergleich zwischen AVL und BST

- Wenn wir anstatt den AVL Baum einen Binärsuchbaum benutzten würden, dann sieht der Binärsuchbaum nach den Einfügeoperationen folgendermaßen aus:

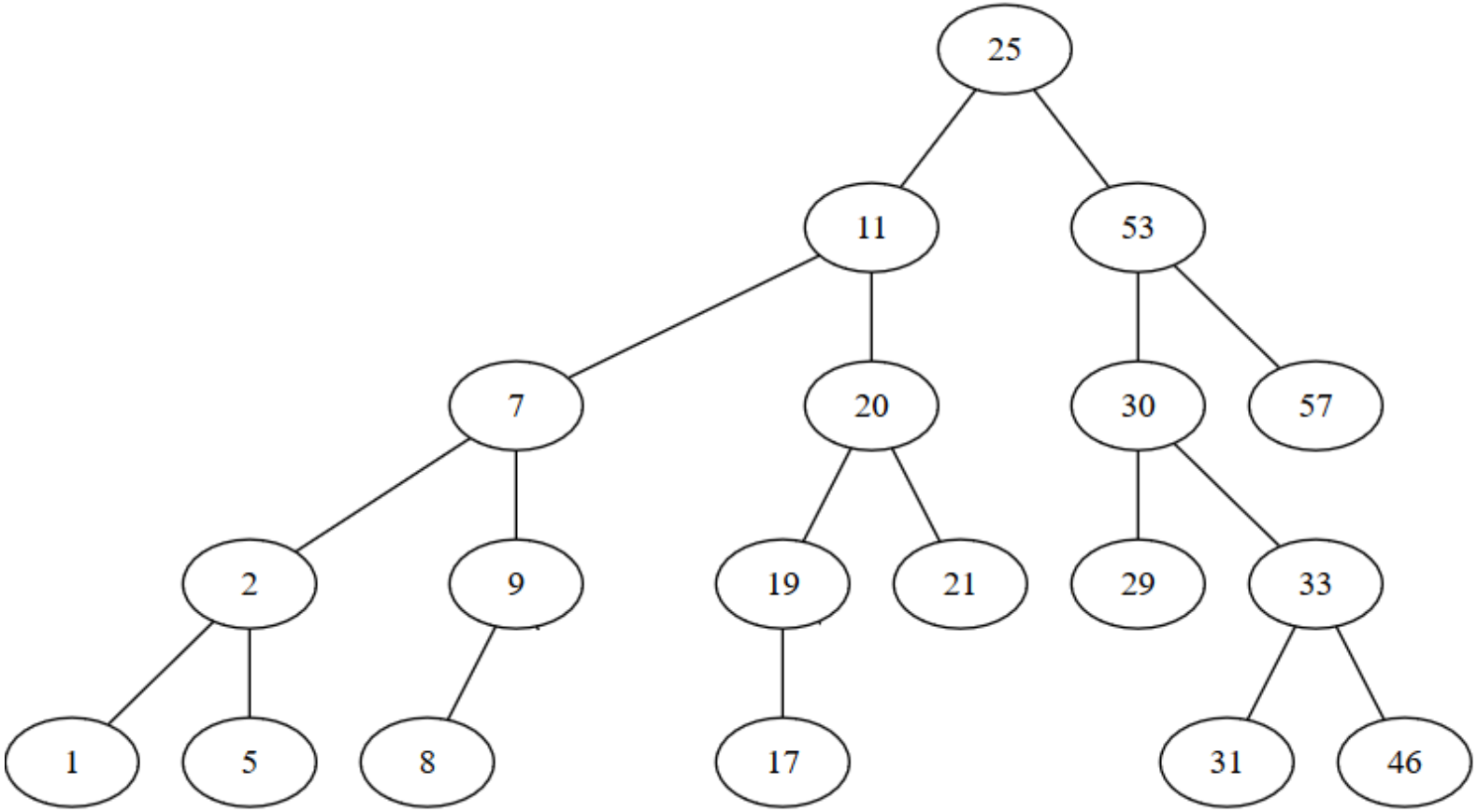

2
1
11
7
20
5
9
19
50
8
17
25
57
21
29
53
30
31
33
46

AVL Rotationen Beispiel



- Lösche 50

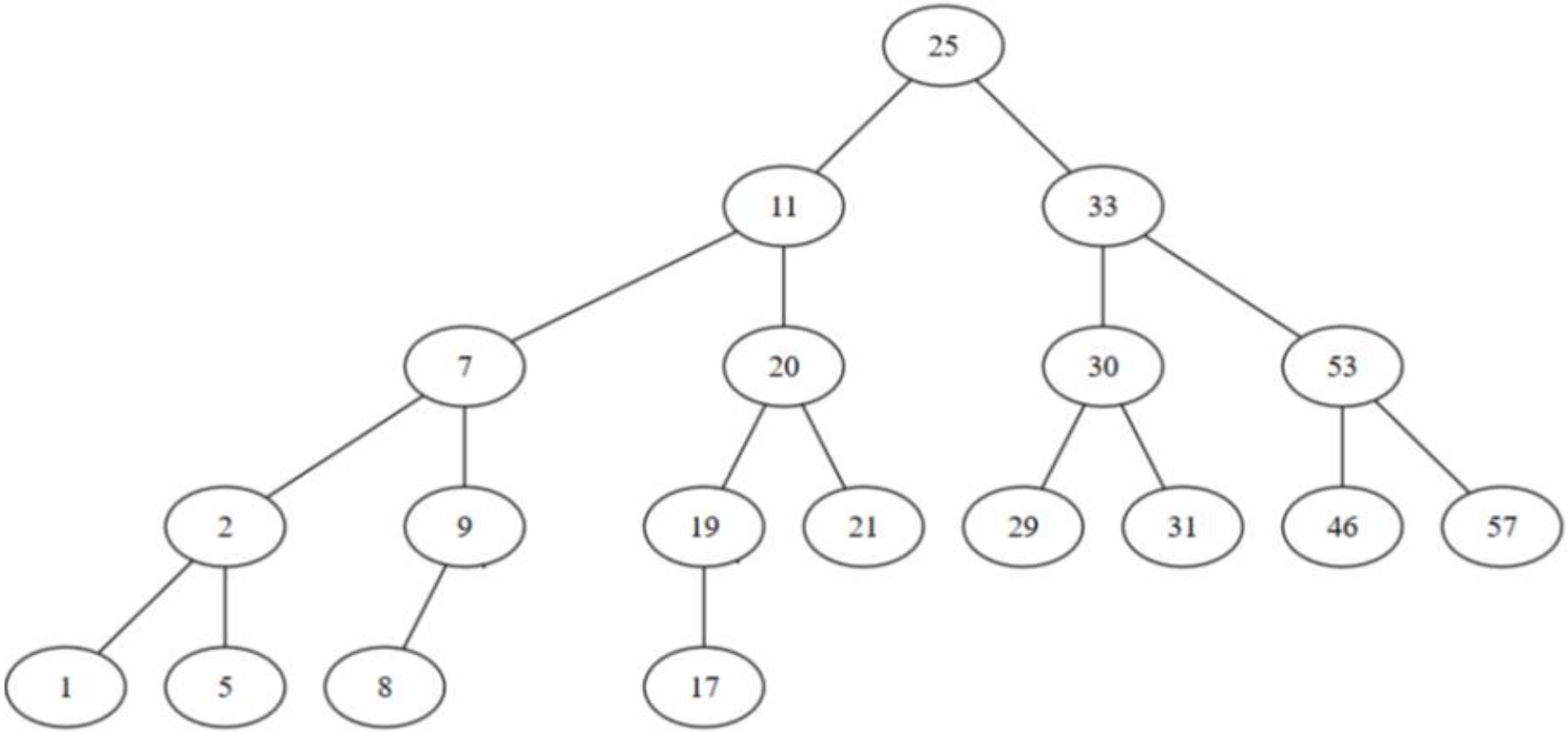
AVL Rotationen Beispiel



- Brauchen wir eine Rotation? Falls ja, welche Rotation?
  - Ja, eine doppelte Rotation nach rechts für den Knoten 53

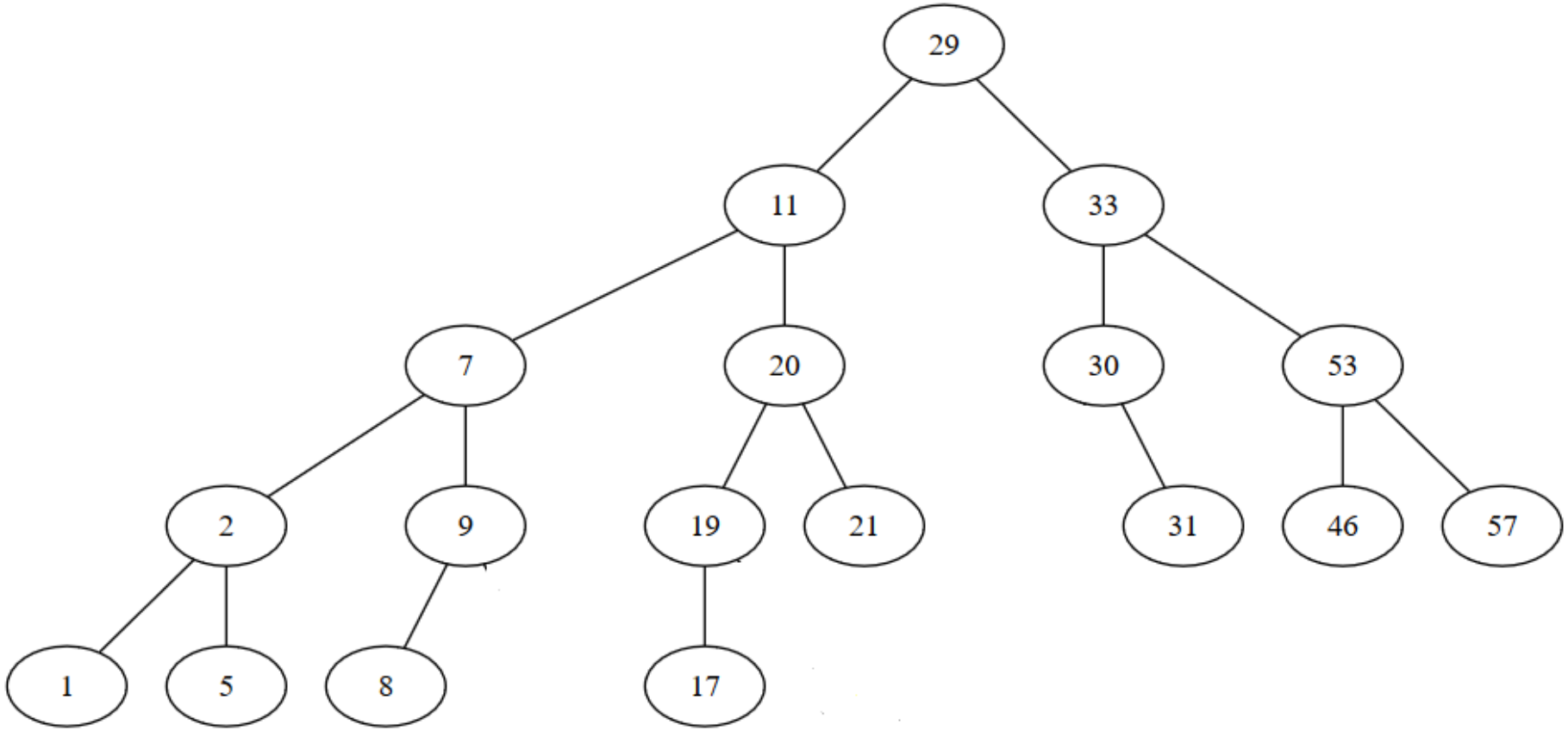
AVL Rotationen Beispiel

- Nach der Rotation:



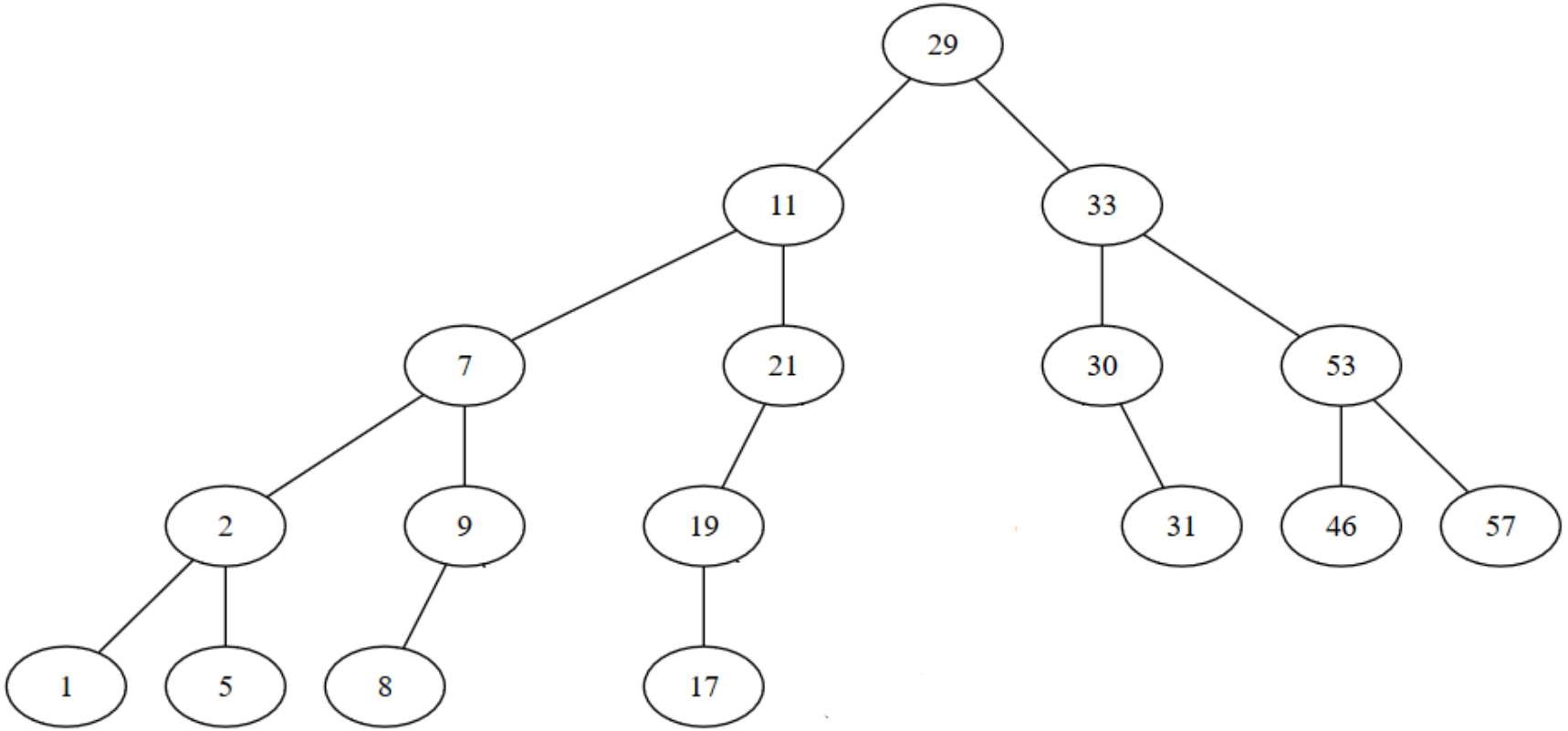
- Lösche 25

AVL Rotationen Beispiel



- Brauchen wir eine Rotation? Falls ja, welche Rotation?
  - Nein
- Lösche 20

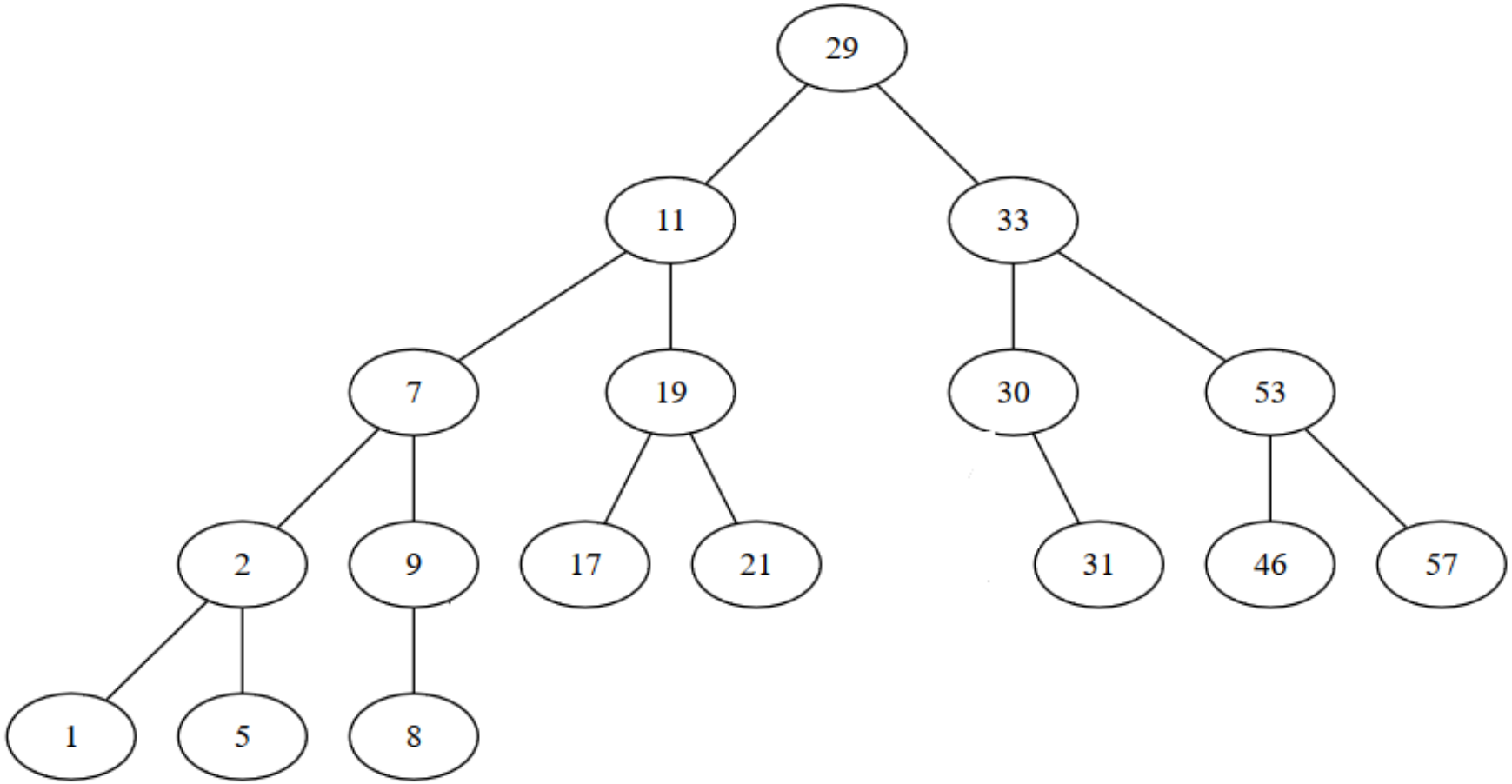
AVL Rotationen Beispiel



- Brauchen wir eine Rotation? Falls ja, welche Rotation?
  - Ja, eine einfache Rotation nach rechts für den Knoten 21

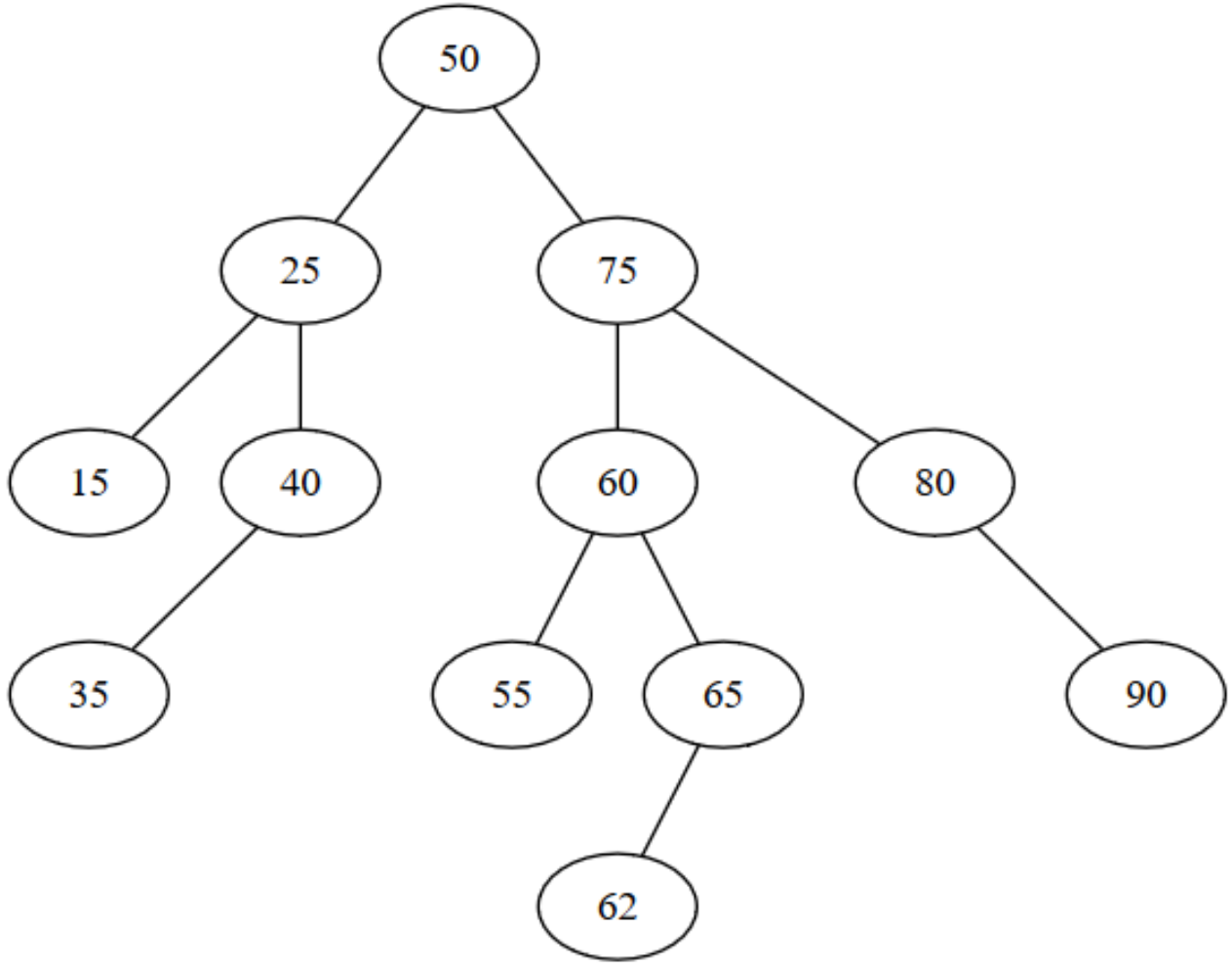
AVL Rotationen Beispiel

- Nach der Rotation:



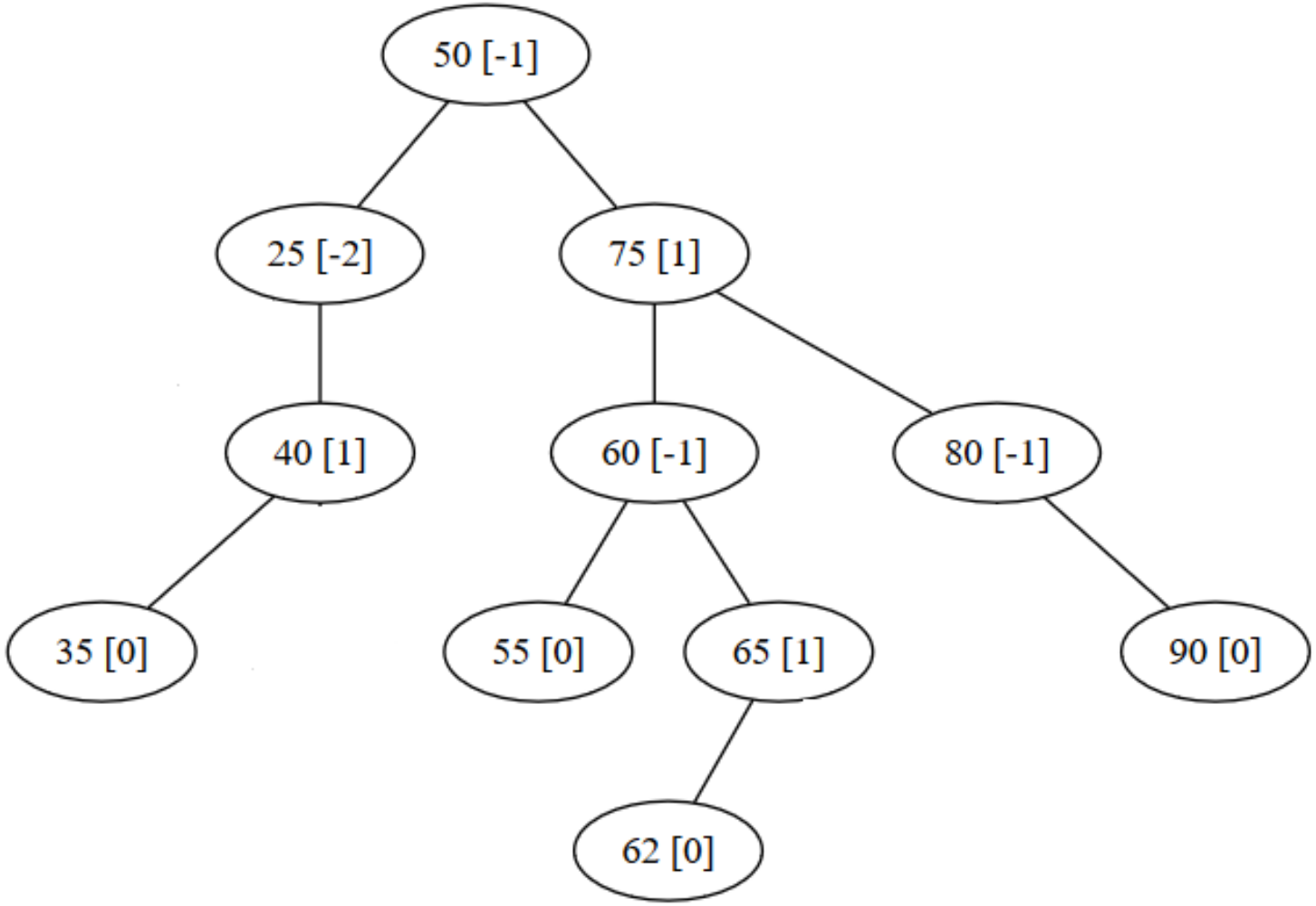
AVL Rotationen Beispiel

- Bem. Wenn wie einen Knoten löschen, kann es sein dass man mehr als 1 Rotation braucht!



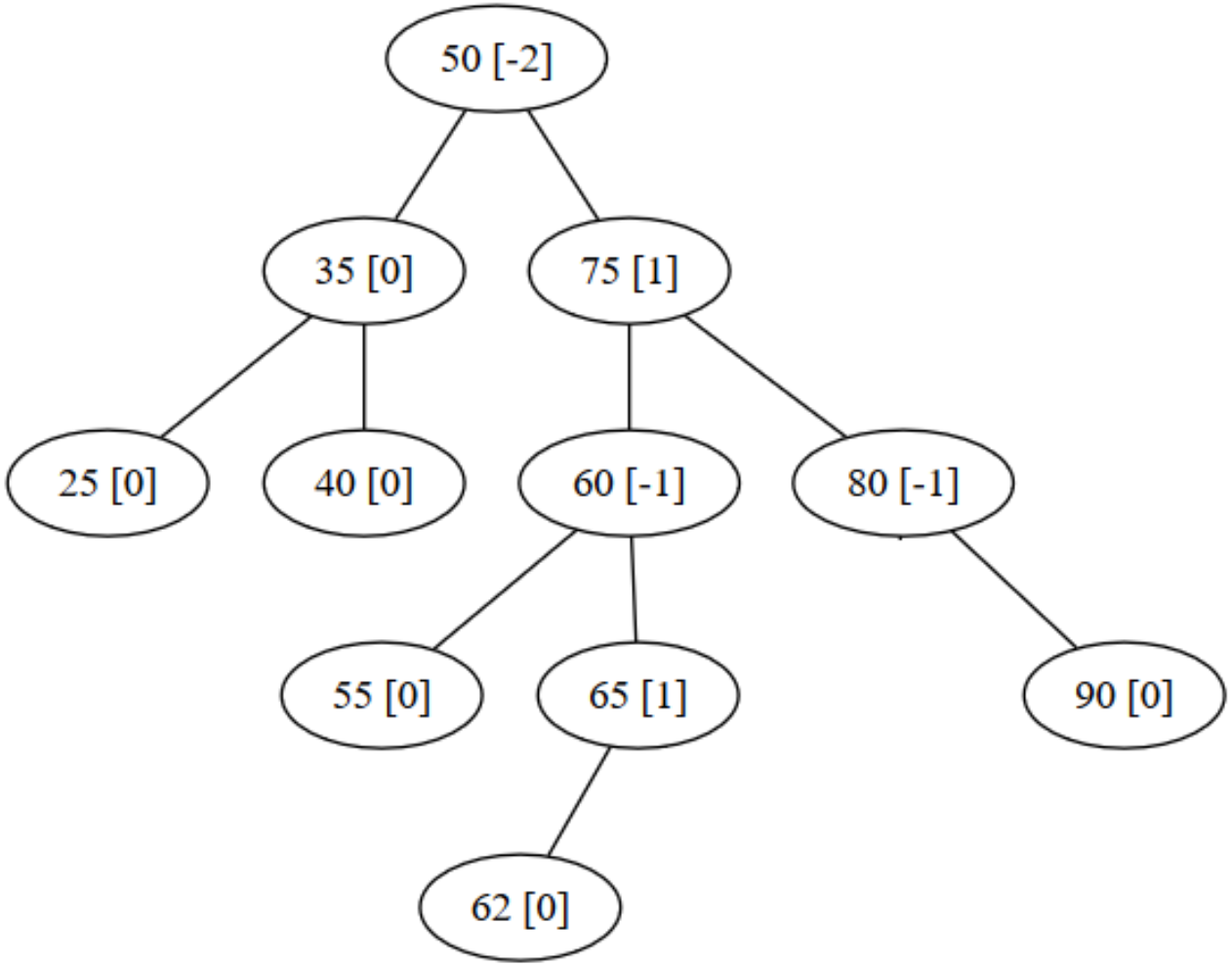
- Lösche 15.

AVL Rotationen Beispiel



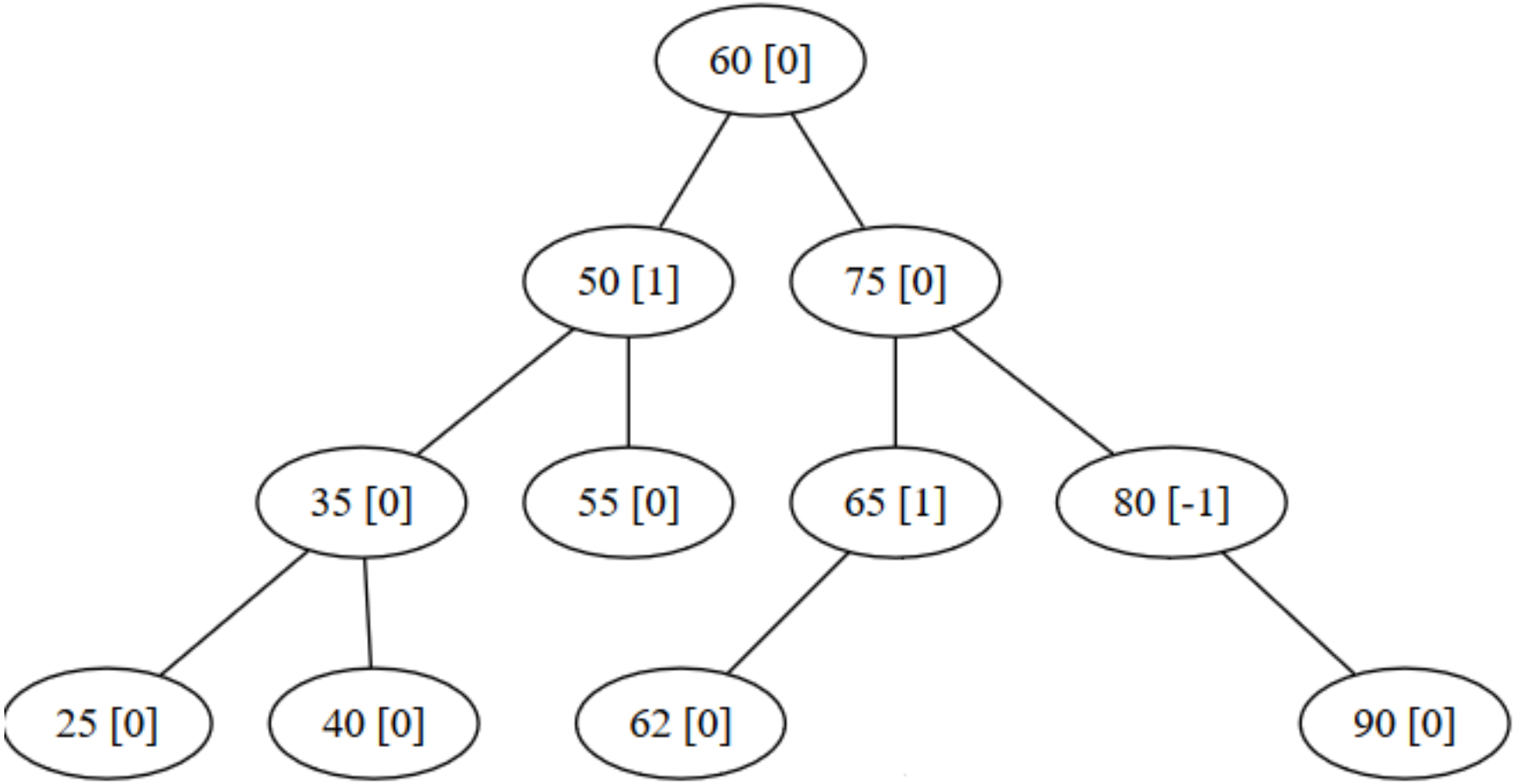
AVL Rotationen Beispiel

- Nach der Rotation:



AVL Rotationen Beispiel

- Nach der zweiten Rotation:



AVL Baum - Repräsentierung

- Welche Datenstrukturen brauchen wir?

AVLNode:
info: TComp
left: ↑ AVLNode
right: ↑ AVLNode
h: Integer //Höhe des Knotens

AVLTree:
root: ↑ AVLNode

AVL Baum – Implementierung

- Wir wollen die Einfügeoperation für den AVL Baum implementieren
- Wir brauchen ein paar Hilfsoperationen für das Einfügen:
  - Ein Algorithmus, der die **Höhe eines Knotens** neu berechnet
  - Ein Algorithmus, der den **Balancierungsfaktor eines Knotens** berechnet
  - Vier Algorithmen für die unterschiedlichen **Rotationen** (wir implementieren einen davon)
- Wir nehmen an, dass es eine Funktion *createNode* gibt, welche einen Knoten mit dem gegebenen Wert erstellt und zurückgibt (mit NIL für das linke und rechte Kind, und Höhe 0)

AVL Baum – Höhe eines Knotens

```
subalgorithm recomputeHeight(node) is:
//pre: node ist ein ↑AVLNode.  Alle Nachkommen von node haben die Höhe (h) auf den
//richtigen Wert gesetzt
//post: falls node ≠ NIL, h von node wird auf den richtigen Wert gesetzt
    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
```

- Komplexität: $\Theta(1)$

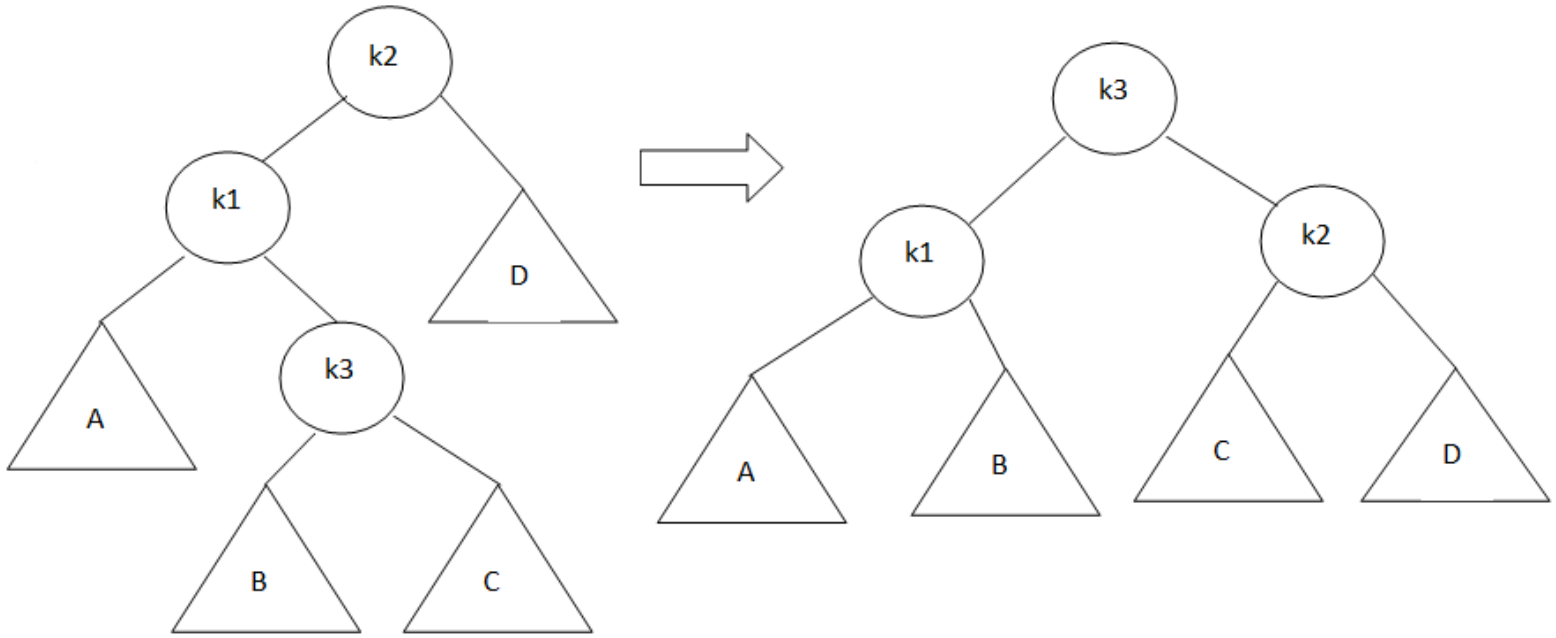
AVL Baum – Höhe eines Knotens

```
function balanceFactor(node) is:
//pre: node ist ein ↑AVLNode.  Alle Nachkommen von node haben die Höhe (h) auf
//den richtigen Wert gesetzt
//post: gibt den Balancierungsfaktor des Knotens zurück
    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
```

- Komplexität: $\Theta(1)$

AVL Baum – Rotationen

- Aus den vier Rotationen implementieren wir eine, die doppelte Rotation nach rechts (DRR)



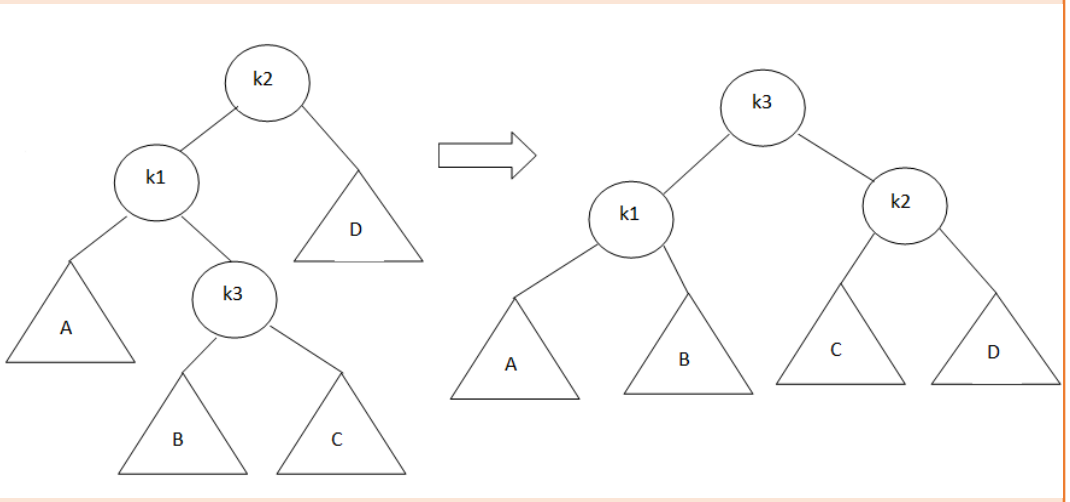
- Die anderen drei können ähnlich implementiert werden (SRR – simple right rotation, SLR – simple left rotation, DLR – double left rotation)

AVL Baum - DRR

function DRR(node) **is**:
//pre: node ist ein ↑AVLNode auf welchem man die doppelte Rotation nach rechts
// ausführt
//post: DRR gibt die neue Wurzel zurück, nach der Rotation

```
    k2 ← node
    k1 ← [node].left
    k3 ← [k1].right
    k3left ← [k3].left
    k3right ← [k3].right
    //man muss die Links umsetzen (reset)
    newRoot ← k3
    [newRoot].left ← k1
    [newRoot].right ← k2
    [k1].right ← k3left
    [k2].left ← k3right
```

//Fortsetzung auf der nächsten Folie



AVL Baum - DRR

```
        //man muss die Höhe der geänderten Knoten neu berechnen
        recomputeHeight(k1)
        recomputeHeight(k2)
        recomputeHeight(newRoot)
        DRR ← newRoot
    end-function
```

- Komplexität: $\Theta(1)$

AVL Baum – insert

```
function insertRec(node, elem) is
//pre: node ist ein ↑AVLNode , elem ist der Wert, den man in dem Teilbaum mit
//   Wurzel node einfügen muss
//post: insertRec gibt die neue Wurzel des Teilbaumes nach dem Einfügen zurück
    if node = NIL then
        insertRec ← createNode(elem)
    else if elem ≤ [node].info then
        [node].left ← insertRec([node].left, elem)
    else
        [node].right ← insertRec([node].right, elem)
    end-if
//Fortsetzung auf der nächsten Folie
```

AVL Baum – insert

```
recomputeHeight(node)
balance ← getBalanceFactor(node)
if balance = -2 then
//der rechte Teilbaum hat die größere Höhe, also man braucht eine Rotation
//nach LINKS
    rightBalance ← getBalanceFactor([node].right)
    if rightBalance < 0 then
    //der rechte Teilbaum des rechten Teilbaums hat die größere Höhe, SLR
        node ← SLR(node)
    else
        node ← DLR(node)
    end-if
//Fortsetzung auf der nächsten Folie
```

AVL Baum – insert

```
else if balance = 2 then
//der linke Teilbaum hat die größere Höhe, also man braucht eine Rotation
//nach RECHTS
    leftBalance ← getBalanceFactor([node].left)
    if leftBalance > 0 then
    // der linke Teilbaum des linken Teilbaums hat die größere Höhe, SRR
        node ← SRR(node)
    else
        node ← DRR(node)
    end-if
end-if
insertRec ← node
end-function
```

AVL Baum – insert

- Komplexität des Algorithmus *insertRec:* $O(log_2 n)$
- Da *insertRec* einen Zeiger zu einem Knoten als Eingabeparamter hat, braucht man eine Wrapper Funktion:

```
subalgorithm insert(tree, elem) is
//pre: tree ist ein AVL Tree, elem ist das Element zum Einfügen
//post: elem wurde in tree eingefügt
    tree.root ← insertRec(tree.root, elem)
end-subalgorithm
```

- Die Löschoperation kann ähnlich implementiert werden (man löscht ein Element wie in einem BST und man führt Rotationen aus, falls nötig)

AVL Baum – Anwendungen

- Dank den Rotationen hat man garantiert höchstens logarithmische Komplexität für die drei Hauptioperationen: Einfügen, Löschen, Suchen
- Dies ist die effizienteste Art zum Bearbeiten von großen **sortierten** Datensätzen
- Es gibt auch andere ähnliche Arten von balancierten Bäumen – in der Praxis wichtig beim Suchen und Indexieren von Datenbanken

- Was ist effizienter: Hashtabellen oder balancierte Bäume? Hängt von der Anwendung ab!

Huffman Codierung

- Huffman Codierung kann verwendet werden um Charakter mit Codes variabler Länge zu codieren

- Um die Gesamtzahl der verwendeten Bits für die Codierung einer Nachricht zu reduzieren, werden für die Charakter, die häufiger vorkommen, kürzere Codes verwendet.

- Da Codes mit variabler Länge verwendet werden, darf kein Code ein Präfix eines anderen Codes sein (wenn Charakter E als 01 und Charakter X als 010011 codiert wird, weiß man bei der Decodierung nicht, ob 01 ein E oder der Beginn der Codierung von X ist)

Huffman Codierung

- Für die Huffman Codierung muss man erst die Frequenz der Charakter berechnen, da die Codes basierend auf Frequenz definiert werden

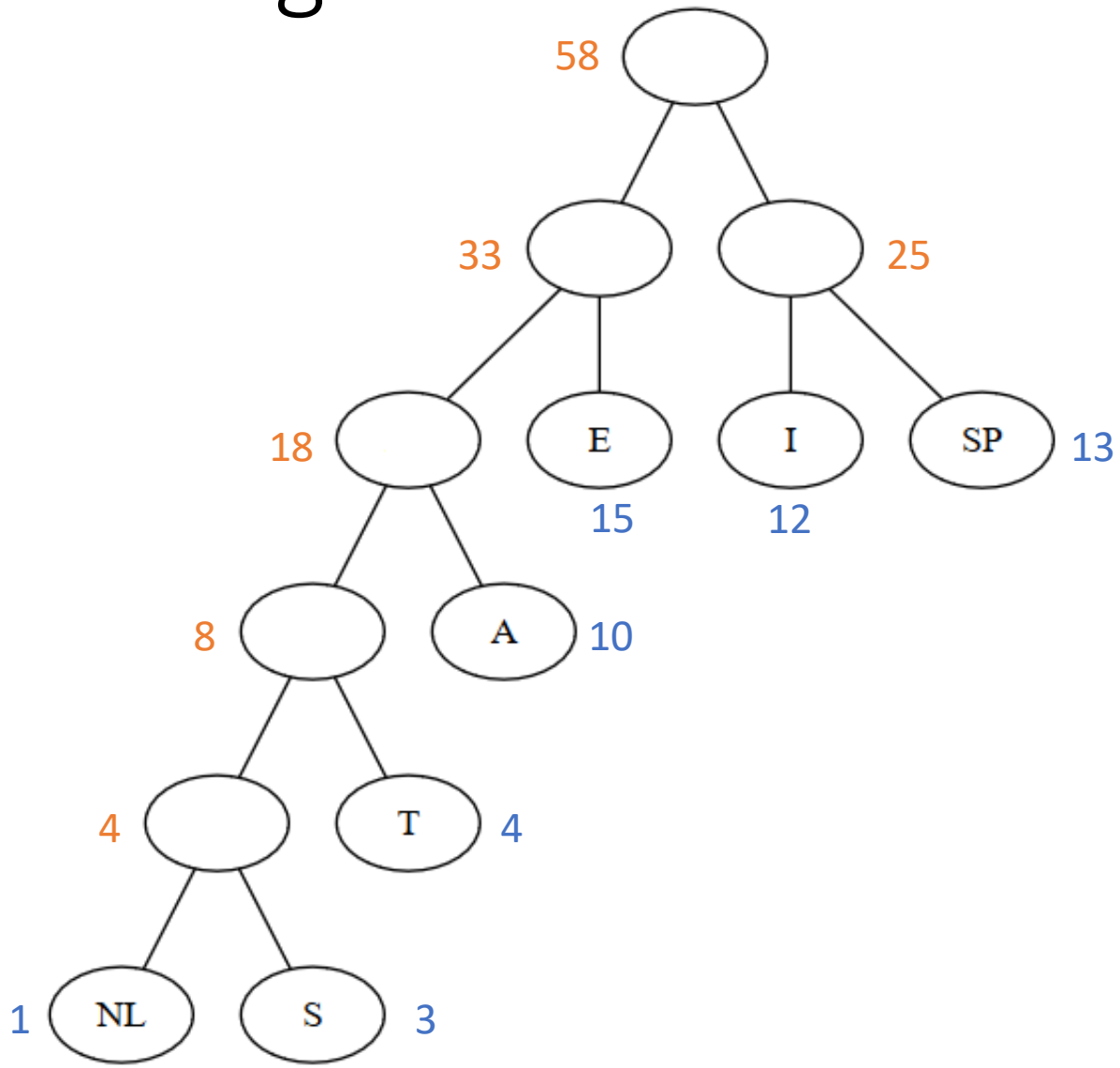
- Nehmen wir an, wir codieren eine Nachricht, welche folgende Charakter mit den entsprechenden Frequenzen enthält:

Charakter	A	E	I	S	T	Leerzeichen SP	Newline NL
Frequenz	10	15	12	3	4	13	1

Huffman Codierung

- Um die Huffman Codierung zu definieren, wird ein Binärbaum folgendermaßen gebildet:
  - Man fängt mit Bäumen an, die jeweils nur eine Wurzel enthalten. Für jeden Charakter gibt es einen Baum, und jeder Baum hat ein **Gewicht**, das der **Frequenz** des Charakters entspricht.
  - Nehme die beiden Bäume mit dem kleinsten Gewicht (bei Gleichstand wähle zufällig), verbinde diese zu einem Baum mit einem Gewicht, das der Summe der zwei Gewichten entspricht.
  - Wiederhole den Vorgang, bis es nur noch einen Baum gibt.

Huffman Codierung

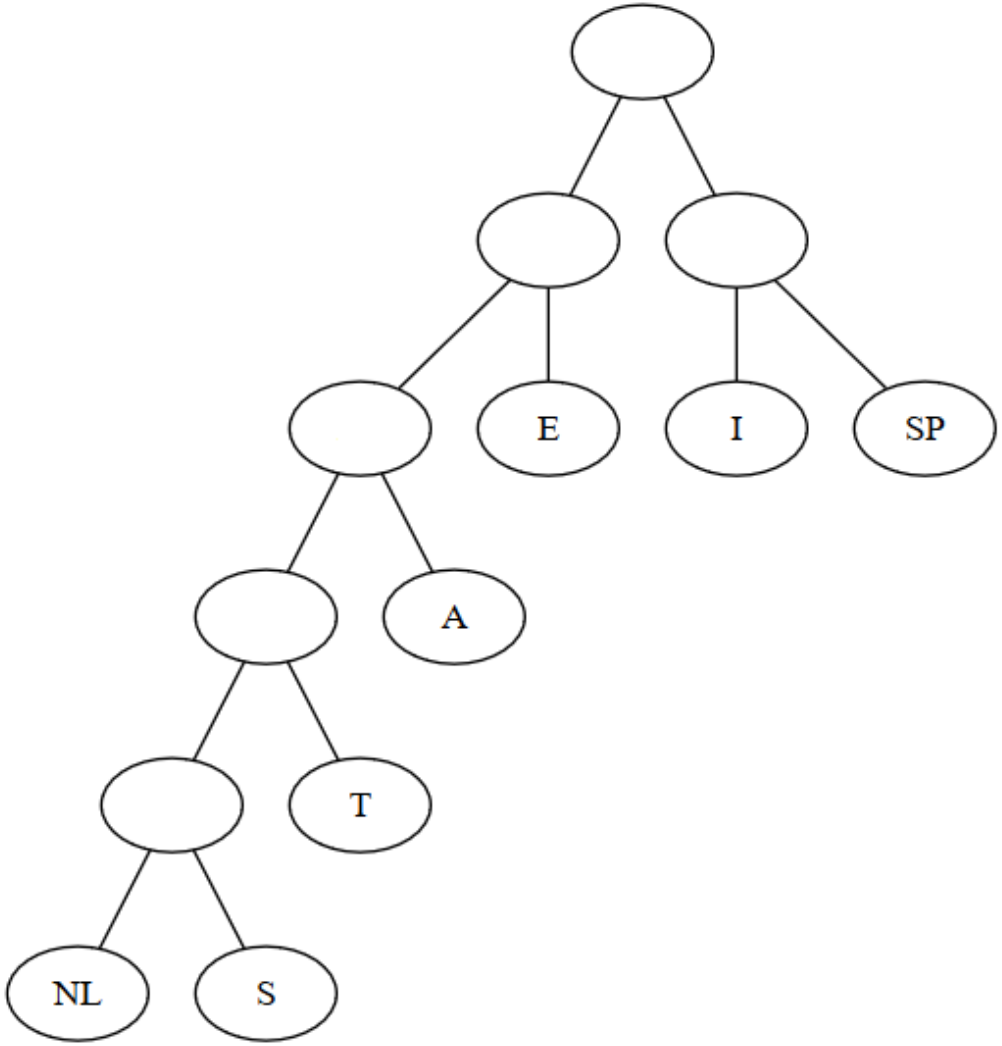


Huffman Codierung

- Der Code für jeden Charakter kann aus dem Baum folgendermaßen abgelesen werden:
  - Starte von der Wurzel und gehe in die Richtung des entsprechenden Blattknotens
  - Jedes Mal, wenn man nach links geht, fügt man der Codierung ein Bit 0 hinzu
  - Jedes Mal, wenn man nach rechts geht, fügt man der Codierung ein Bit 1 hinzu

Huffman Codierung

- Codes für die Charakter:
  - NL – 00000
  - S – 00001
  - T – 0001
  - A – 001
  - E – 01
  - I – 10
  - SP – 11



- Um eine Nachricht zu codieren ersetzt man jeden Charakter durch den entsprechenden Code

Huffman Codierung

- Angenommen, wir haben folgende Codierung, die wir decodieren wollen:

  01101100010001001111001000000

- Man weiß nicht, wo die Codierung für jeden Charakter endet, aber man kann den Baum zur Decodierung verwenden

Huffman Codierung

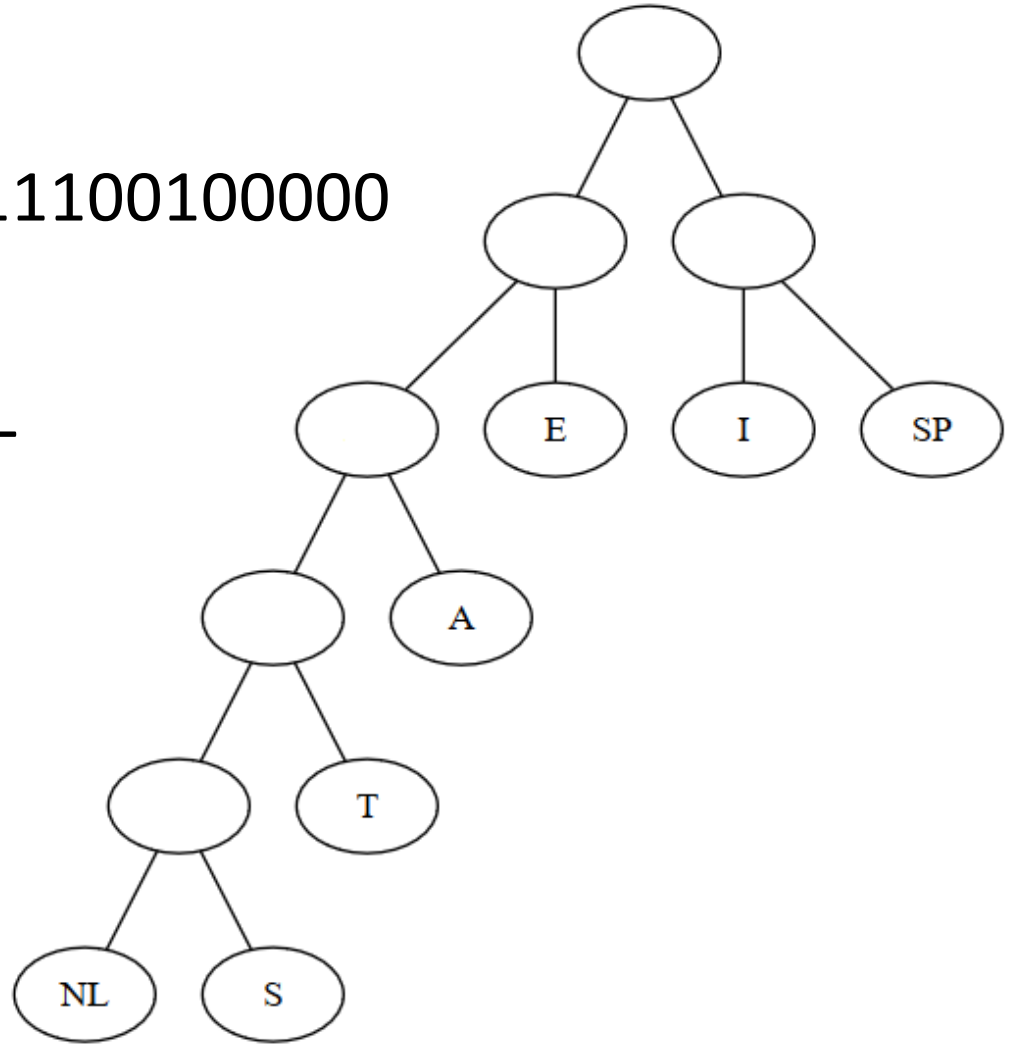
- Man beginnt mit dem Parsen der Codierung und durch den Baum folgendermaßen zu iterieren:
  - Man fängt mit der Wurzel an
  - Wenn der aktuelle Bit 0 ist, geht man zum linken Kind, sonst geht man zum rechten Kind
  - Wenn man zu einem Blatt gelangt, dann hat man einen Charakter decodiert und man fängt wieder bei der Wurzel an

Huffman Codierung

- Codierte Nachricht: 01101100010001001100100000

- Decodierte Nachricht: E I SP T T A SP A NL

Huffman Codierung ist **die** Grundlage
für **alle** Kompressionsalgorithmen
(jpeg, zip, mp3, etc)



Skip Listen

- Nehmen wir an, dass wir eine sortierte Sequenz speichern wollen. Die Elemente können in unterschiedlichen Datenstrukturen gespeichert werden:
  - dynamisches Array
  - verkettete Liste

- Man weiß, dass die Operation, die am meisten gebraucht wird, die Einfüge-Operation ist. Welche Datenstruktur wäre besser?

Skip Liste

- Was für eine Komplexität hat die Einfüge-Operation?
  - Man kann die Einfüge-Operation in zwei Schritte teilen: die **Position suchen** und dann **das neue Element einfügen**

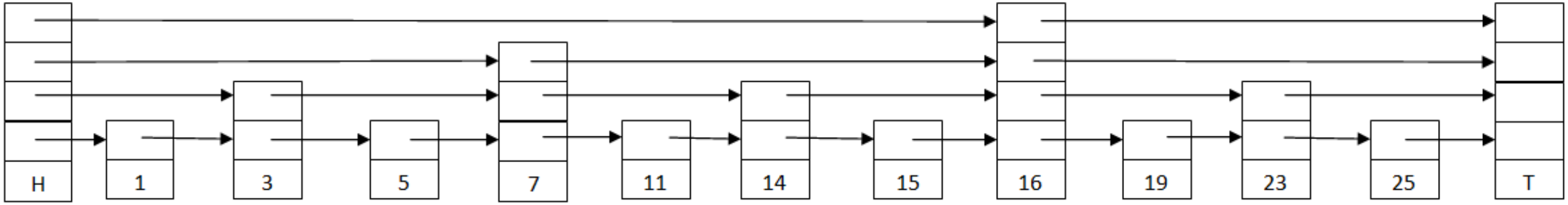
    - Für ein dynamisches Array kann man die Suche optimieren (binäre Suche $O(log_2 n)$), aber das Einfügen ist $O(n)$

    - Für eine verkettete Liste ist das Einfügen optimal ($\Theta(1)$), aber das Suchen der Position ist $O(n)$

Skip Liste

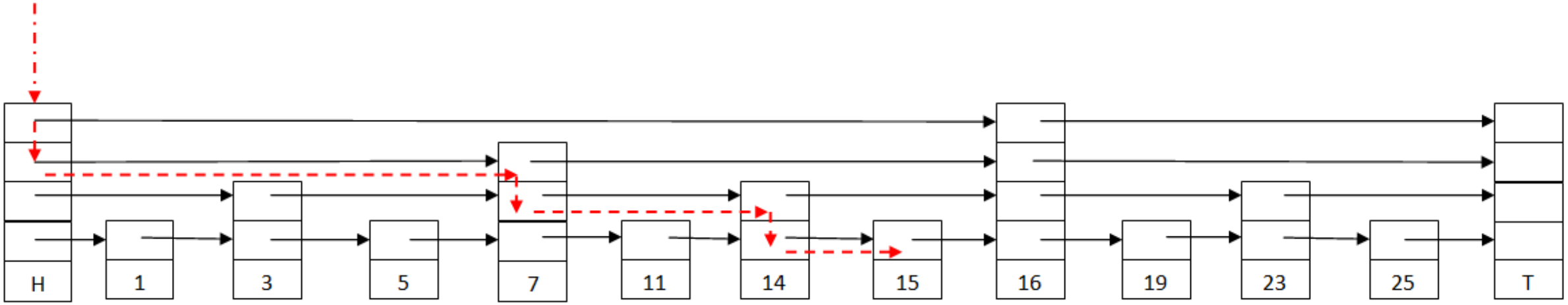
- Die Skip Liste ist eine Datenstruktur, mithilfe deren man in einer sortierten Datensequenz schneller suchen kann
- Die Suche geht offensichtlich schneller, wenn man Elemente überspringen (skip) kann → *Skip Liste*
- Nehmen wir z.B. an, dass es nicht nur von jedem Knoten ein Zeiger auf den nächsten, sondern darüber hinaus auch von jedem zweiten Knoten einen Zeiger auf den übernächsten Knoten gibt
- Man kann dieses Prinzip wiederholen, sodass die Liste eine Zugriffsstruktur ähnlich wie die der binären Bäume bekommt

Skip Liste



- H und T sind zwei spezielle Knoten, die dem Head und Tail entsprechen
- Diese können nicht gelöscht werden, sie bleiben auch wenn die Liste leer ist

Skip Liste



- Man fängt vom Head und von dem höchsten Niveau an
- Wenn möglich, geht man nach rechts weiter
- Wenn man nicht nach rechts gehen kann (das nächste Element ist größer als das gesuchte Element), dann geht man ein Niveau nach unten

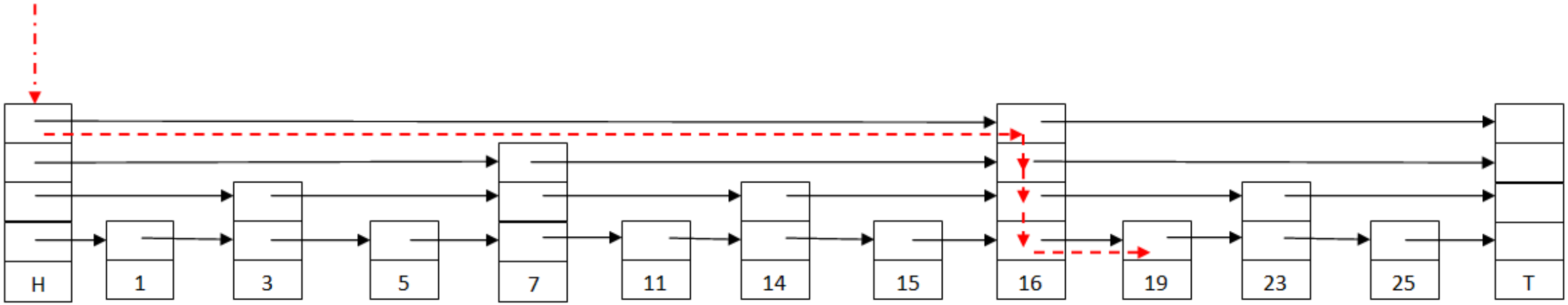
Skip Liste

- Das untere Niveau enthält alle n Elemente
- Das nächste Niveau enthält $\frac{n}{2}$ Elemente
- Das nächste Niveau enthält $\frac{n}{4}$ Elemente, ...

- Es gibt also ungefähr $log_2 n$ Niveaus
- Man überprüft von jedem Niveau höchsten 2 Knoten
- Die Komplexität der Suchoperation: $O(log_2 n)$

Skip Liste – Insert

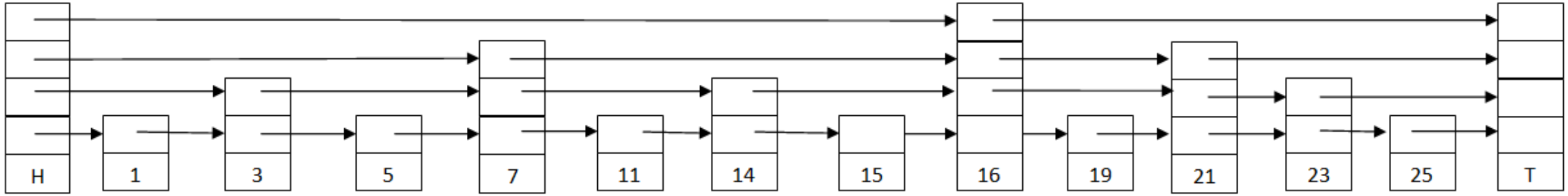
- Füge 21 ein:



- Welches Niveau sollte der neue Knoten haben?

Skip Liste – Insert

- Das Niveau eines Knotens wird zufällig ausgewählt (random), aber so, dass ungefähr die Hälfte der Knoten auf Niveau 2 sind, ein Viertel der Knoten auf Niveau 3, usw.
- Wir nehmen an, dass der neue Knoten das Niveau 3 hat



Skip Liste

- Skip Listen sind probabilistische Datenstrukturen, da die Höhe/Niveau eines neuen Knotens randomisiert berechnet wird
- Es kann ein schlimmster Fall geben, wo jeder Knoten die Höhe 1 hat (also es ist eine normale verkettete Liste)

- In der Praxis funktionieren Skip Listen gut
- Beispiel (link): Speichern von Benutzernamen in Discord-Server

> Internally the Elixir terms stored in the SortedSet are converted to Rust equivalents and stored in a Vector of Vectors. The structure is similar to a skip-list, almost every operation on the SortedSet will perform a linear scan through the buckets to find the bucket that owns the term, then a binary search is done within the bucket to complete the operation.

Einen arithmetischen Ausdruck auswerten

- Aufgabe: entwerfe einen Algorithmus, der das Ergebnis eines arithmetischen Ausdrucks berechnet
- Zum Beispiel:
  - 2+3*4 = 14
  - ((2+4)*7)+3*(9-5) = 54
  - ((((3+1)*3)/((9-5)+2))-((3*(7-4))+6)) = -13
- Ein arithmetischer Ausdruck besteht aus Operatoren (+, -, *, /), Klammern und Operanden (Zahlen)
- Als Vereinfachung betrachten wir Ziffern als Operanden und wir vermuten, dass der Ausdruck korrekt ist

Infix und Postfix Notation

- Die arithmetische Ausdrücke auf der vorigen Folie folgen der *Infixnotation.* Die Infixnotation ist die allgemein gebräuchliche Form der mathematischen Notation, bei der die Operatoren zwischen den Operanden gesetzt werden.
- Menschen arbeiten einfacher mit der Infixnotation, aber für einen Algorithmus ist es nicht so einfach einen Ausdruck in der Infixnotation zu berechnen. Computers arbeiten besser mit der Postfixnotation.
- Bei der *Postfixnotation* werden zunächst die Operanden niedergeschrieben bzw. eingegeben und danach der darauf anzuwendende Operator.

Infix und Postfix Notation

- Beispiele für Audrücke in der Infix- und Postfixnotation:

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+/-*+

- Die Reihenfolge der Operanden in den zwei Notationen ist gleich, nur die Reihenfolge der Operatoren ändert sich
- Die Reihenfolge der Operatoren wird von der Operatorrangfolge und von den Klammern bestimmt

Infix und Postfix Notation

- Die Auswertung eines arithmetischen Ausdruckes kann in zwei Unteraufgaben geteilt werden:
  - Die Infixnotation in die Postfixnotation umwandeln
  - Die Postfixnotation auswerten

- Beide Unteraufgaben werden mit Hilfe von Stacks und Schlangen gelöst

Infixnotation in Postfixnotation umwandeln

- Benutze einen Hilfsstack für die Operatoren und Klammern und eine Schlange für das Ergebnis
- Man analysiert den Ausdruck und:
  - Falls ein **Operand** gefunden wird, dann wird dieser **in die Schlange eingefügt**
  - Falls eine **geöffnete Klammer** gefunden wird, dann wird diese **in den Stack eingefügt**
  - Falls eine **geschlossene Klammer** gefunden wird, dann werden **Elemente von dem Stack gelöscht und in die Schlange eingefügt**, bis man eine geöffnete Klammer findet (die **Klammern werden nicht in die Schlange eingefügt**)

Infixnotation in Postfixnotation umwandeln

- Falls ein **Operator** (*opCurrent*) gefunden wird:
  - Falls der Stack leer ist, dann wird der Operator in den Stack eingefügt
  - Solange der **Top** des Stacks einen **Operator** enthält, **dessen Rang höher** (oder gleich) in der Operatorrangfolge ist als der Rang des aktuellen Operators, ***pop*** **und** ***push*** **der Operator aus dem Stack in die Schlange**. ***Push opCurrent*** **in den Stack** wenn der Stack leer wird, oder wenn der Top eine Klammer oder ein Operator mit niedrigeren Rang ist.
  - Falls der **Top** des Stacks eine **geöffnete Klammer** oder ein **Operator mit niedrigeren Rang** ist, dann ***push opCurrent*** **in den Stack**
- Falls man den ganzen Ausdruck analysiert hat, dann werden alle gebliebene Elemente aus dem Stack gelöscht und in die Schlange eingefügt

Infixnotation in Postfixnotation umwandeln

- Beispiel: 1 + 2 * (3 – 4 / (5 + 6)) + 7

Input	Operation	Stack	Schlange
1	Push to Queue		1
+	Push to Stack	+	1
2	Push to Queue	+	12
*	Überprüfe Top und Push to Stack	+*	12
(	Push to Stack	+*(	12
3	Push to Queue	+*(	123
-	Überprüfe Top und Push to Stack	+*(-	123
4	Push to Queue	+*(-	1234
/	Überprüfe Top und Push to Stack	+*(-/	1234

Infixnotation in Postfixnotation umwandeln

- Beispiel: 1 + 2 * (3 – 4 / (5 + 6)) + 7

Input	Operation	Stack	Schlange
/	Überprüfe Top und Push to Stack	+*(-/	1234
(	Push to Stack	+*(-/(	1234
5	Push to Queue	+*(-/(	12345
+	Überprüfe Top und Push to Stack	+*(-/(+	12345
6	Push to Queue	+*(-/(+	123456
)	Pop from Stack und Push to Queue alles bis zu (	+*(-/	123456+
)	Pop from Stack und Push to Queue alles bis zu (	+*	123456+/-
+	Überprüfe, Pop from Stack zwei mal und dann Push	+	123456+/-*+
7	Push to Queue	+	123456+/-*+7
Ende	Pop from Stack und Push to Queue alles		123456+/-*+7+

Infixnotation in Postfixnotation umwandeln

```
function infixToPostfix(expr) is:
    init(st)
    init(q)
    for elem in expr execute
        if @elem ist ein Operand then
            push(q, elem)
        else if @ elem ist eine geöffnete Klammer then
            push(st, elem)
        else if @elem ist eine geschlossene Klammer then
            while @ top(st) ist keine geöffnete Klammer execute
                op ← pop(st)
                push(q, op)
            end-while
            pop(st) //lösche die geöffnete Klammer
        else //das Element ist ein Operator
//Fortsetzung auf der nächsten Folie
```

Infixnotation in Postfixnotation umwandeln

```
                    while not isEmpty(st) and @ top(st) nicht geöffnete Klammer and
                    @top(st) hat >= Rang als 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
```

- Komplexität: $\Theta(n)$, wobei *n* die Länge der Sequenz ist

Auswertung der Postfixnotation

- Nachdem man die Postfixnotation berechnet hat, kann man den Ausdruck mit Hilfe eines Stacks auswerten
- Hauptidee des Algorithmus:
  - Falls ein Operand gefunden wird, dann wird dieser in den Stack eingefügt
  - Falls ein Operator gefunden wird, dann werden zwei Elemente aus dem Stack gelöscht, die Operation wird ausgeführt und das Ergebnis wird in den Stack eingefügt
  - Am Ende enthält der Stack das Ergebnis

Auswertung der Postfixnotation – Beispiel: 123456+/-*+7+

Pop aus der Schlange	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

Auswertung der Postfixnotation

```
function evaluatePostfix(q) is:
    init(st)
    while not isEmpty(q) execute
        elem ← pop(q)
        if @ elem ist ein Operand then
            push(st, elem)
        else
            op1 ← pop(st)
            op2 ← pop(st)
            @ berechne das Ergebnis der Operation op2 elem op1 in der Variable result
            push(st, result)
        end-if
    end-while
    result ← pop(st)
    evaluatePostfix ← result
end-function
```

- Komplexität: $\Theta(n)$, wobei *n* die Länge der Sequenz ist

Auswertung eines arithmetischen Ausdruckes

- Mit den zwei Funktionen *infixToPostfix* und *evaluatePostfix* kann man einen arithmetischen Ausdruck auswerten

- **Denk darüber nach:**
  - Wie kann man die Infixnotation direkt in einer einzigen Funktion auswerten? (Hinweis: benutze zwei Stacks – einen für die Operatoren und einen für die Operanden)

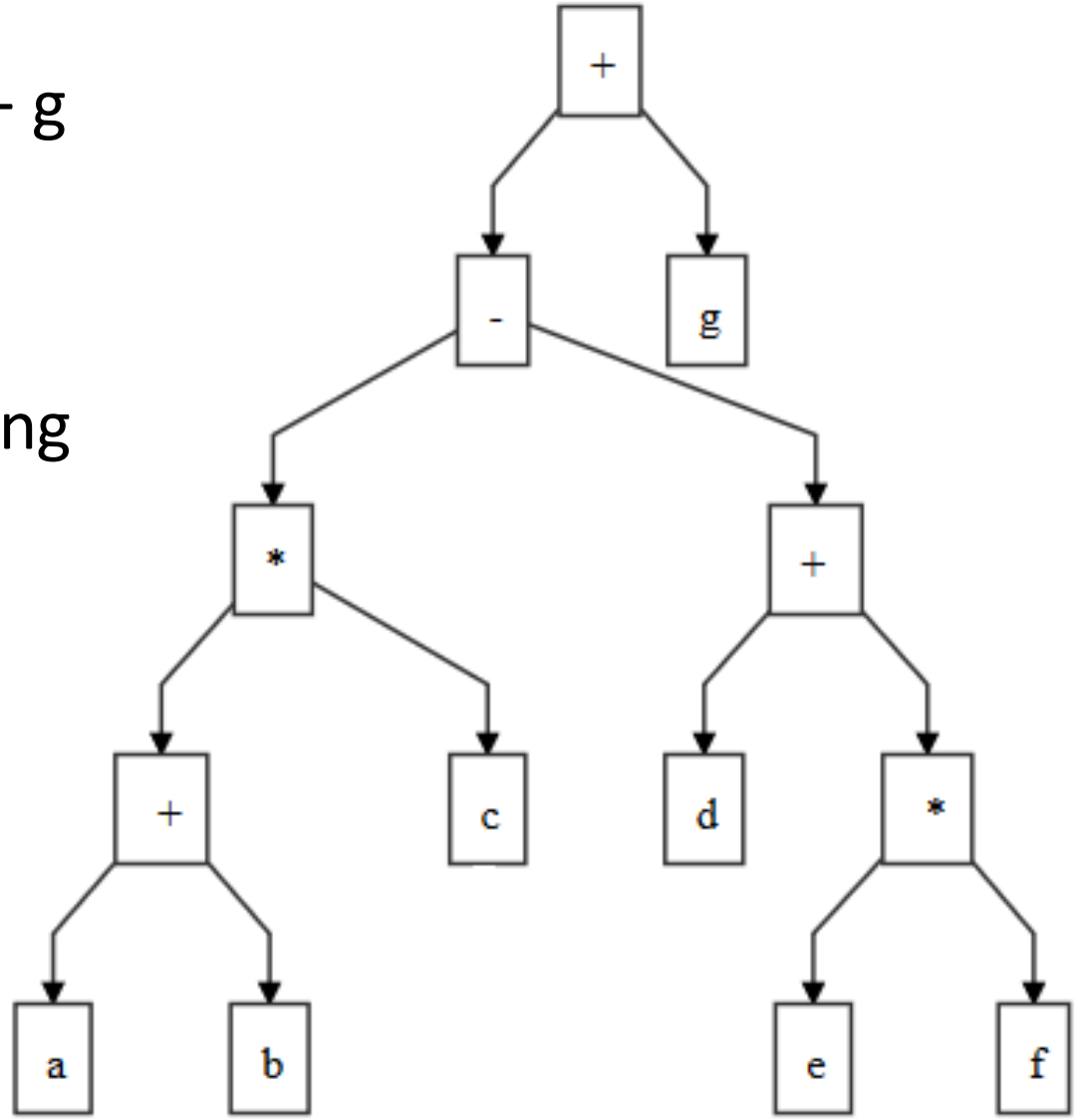
Postfixnotation als Binärbaum

- Erstelle den Binärbaum für die Postfixnotation eines arithmetischen Ausdruckes mit den Operatoren +, -, *,/.
- Bei der *Postfixnotation* werden zunächst die Operanden niedergeschrieben bzw. eingegeben und danach der darauf anzuwendende Operator.
- Implementiere *buildTree (postE, tree)*

Beispiel

- Ausdruck: (a + b) * c - (d + e * f) + g
- Postfix Notation: ab+c*def*+-g+

- Wenn wir den Baum in Postordnung traversieren, dann kriegt man die Postfix Notation



Algorithmus

1. Benutze einen Hilfsstack welcher die Adressen der Knoten aus dem Baum enthält
2. Fange an, den Baum von unten nach oben zu erstellen (bottom up)
3. Analysiere den Postfix Ausdruck
4. Falls man einen Operanden findet → erstelle einen Knoten mit dem Operand als Information und push in den Stack
5. Falls man einen Operator findet →
   a) Pop ein Element aus dem Stack – rechtes Kind
   b) Pop ein Element aus dem Stack – linkes Kind
   c) Erstelle einen Knoten, der den Operator als Information enthält, und die zwei Kinder
   d) Push den neuen Knoten in dem Stack
6. Die Wurzel des Baumes ist das letzte Element aus dem Stack

Repräsentierung des Binärbaumes

- Nehmen wir an, dass wir einen Binärbaum mit dynamisch allokierten Knoten haben
- Repräsentierung:

Node:
e:TElem
left: ↑ Node
right: ↑ Node

BT:
root: ↑ Node

- Der Stack enthält Elemente vom Typ ↑ Node und wir benutzen den Stack-Interface:
  - Init
  - Push
  - Pop
  - Top

Algorithmus *buildTree*

```
Subalgorithm buildTree (postE, tree) is:
        init(s)
        for every e in postE execute:
                if e is an operand then:
                        alocate (newNode)
                        [newNode].e ← e
                        [newNode].left ← NIL
                        [newNode].right ← NIL
                        push (s, newNode)
                else
                        pop(s, p1)
                        pop(s, p2)
                        alocate (newNode)
                        [newNode].e ← e
                        [newNode].left ← p2
                        [newNode].right ← p1
                        push (s, newNode)
                end-if
        end-for
        pop(s, p)
        tree.root ← p
end-subalgorithm
```