Aufgabe 1

Der initiale Hashwert sowie ggf. die Folge der Kollisionen, kann der folgenden Tabelle entnommen werden:

String	k	k^2	$h_0(k)$	$h_1(k)$	$h_2(k)$	$h_3(k)$	$h_4(k)$
Heidrun	22	484	8				
Anne	29	841	4				
Hartmut	27	729	2				
Christian	29	841	4	5			
Simone	41	1681	8	9			
Frank	25	625	2	3			
Markus	32	1024	2	3	6		
Thomas	43	1849	4	5	8	3	0

Damit ergibt sich die folgende Hashtabelle:

0	1	2	3	4	5	6	7	8	9
Thomas	-	Hartmut	Frank	Anne	Christian	Markus	-	Heidrun	Simone

Aufgabe 2

(a)

algorithm *blbintree*(*A*, *lower*, *upper*)
{ *A* ist ein Array mit aufsteigend sortierten, disjunkten Schlüsselwerten,
lower ist der Index des ersten zu berücksichtigenden Schlüsselwertes in *A*,
upper ist der Index nach dem letzten zu berücksichtigenden Schlüsselwert in *A*,
es gelte stets *lower* ≤ *upper*.
Der Algorithmus gibt einen binären Suchbaum über den indizierten Bereich zurück.
}
if (*upper* = *lower*) **then**
return *empty*;
end if;
if (*upper* - *lower* = 1) **then**
return ((*empty*, *A*[*lower*], *empty*), *A*[*upper*], *empty*)
end if;
if (*upper* - *lower* = 2) **then**
return ((*empty*, *A*[*lower*], *empty*), *A*[*upper* - 1], *empty*);
else
var *mid*;
mid := $lower + \lceil (upper - 1 - lower)/2 \rceil$;

```
    return (blbintree(A, lower, mid), A[mid], blbintree(A, mid + 1, upper));
  end if;
end blbintree.
```

Um aus einem Array A der Länge n einen binären Suchbaum T zu erzeugen, lautet der Aufruf dann T := *blbtree*(A, 1, n + 1).

(b)

Da der Algorithmus rekursiv arbeitet, müssen wir die Rekursionsgleichung finden und lösen. Sei T_n die Laufzeit für einen Aufruf auf n Elementen, gemessen in Zugriffen auf A.

Es gilt offenbar $T_0 = 0$ (**then**-Teil des ersten **if**-Konstrunkts, kein Zugriff auf A), $T_1 = 1$ (**then**-Teil des zweiten **if**-Konstrukts, genau 1 Zugriff auf A) und $T_2 = 2$ (**then**-Teil des dritten **if**-Konstrukts, genau 2 Zugriffe auf A).
Für $n > 2$ (**else**-Teil des dritten **if**-Konstrukts) gilt:

$$T_n = T_{\lfloor n/2 \rfloor} + 1 + T_{\lceil n/2 \rceil}$$
$$T_n \le T_{\lceil n/2 \rceil} + 1 + T_{\lceil n/2 \rceil} = 1 + 2 \cdot T_{\lceil n/2 \rceil}$$

Zur Lösung der Rekursionsgleichung betrachten wir o.B.d.A. die Folge aller Zweierpotenzen. Dann gibt es bei der Berechnung von T_n , $n = 2^m > 2$, gerade m Rekursionsebenen. Auf der i-ten Rekursionsebene erfolgen 2^i direkte Zugriffe auf A und es gilt:

$$\begin{aligned} T_n &= T_{2^m} \\ &\le 1 + \sum_{i=1}^{m} 2^i = 1 + (2 + 4 + 5 \ + 2^m) \\ &= 1 + (2^{m+1} - 1) \\ &= 2^{m+1} = 2 \cdot 2^m = 2n \\ &\le 3n - 1 = S_n, \text{ für } n > 0 \end{aligned}$$

Wir schätzen T_n nach oben großzügig durch S_n ab. Dies können wir unbesorgt tun, da wir lediglich zeigen wollen, dass $T_n = O(n)$ gilt. Dazu müssen wir noch die Korrektheit von $T_n \le S_n$ zeigen. Wir tun dies wiederum für alle Zweierpotenzen $n = 2^i$, $i \in \mathbb{N}$ per vollständiger Induktion über i.

Induktionsanfang:
$T_1 = 1 \le S_1 = 3 \cdot 1 - 1 = 2$,
$T_2 = 2 \le S_2 = 3 \cdot 2 - 1 = 5$.

Induktionsschluss:
Unter der Annahme der Korrektheit für alle $n = 2^i$, $i \le j$, $n > 0$, ergibt sich der Induktionsschritt von $n = 2^i$ nach $m = 2n = 2^{i+1}$:

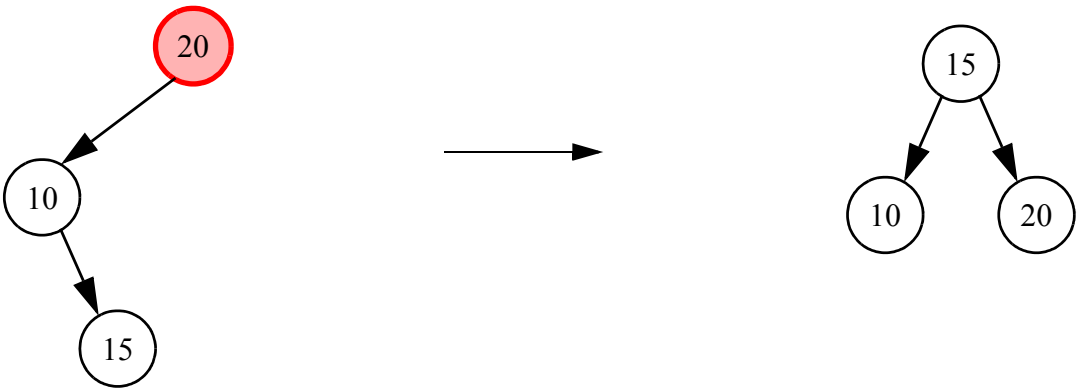
$$T_m = 2 \cdot T_{m/2} + 1 \le 2S_{m/2} + 1 = 2 \cdot (3n/2 - 1) + 1 = 3n - 2 + 1 = 3n - 1$$

Für $n \neq 2^i$, $i \in \mathbb{N}$, ersetzen wir n durch die nächstgrößere Zweierpotenz und erhalten genau 1 weitere Rekursionsebene.
Es gilt also jeweils $T_n \leq S_n = O(n)$.

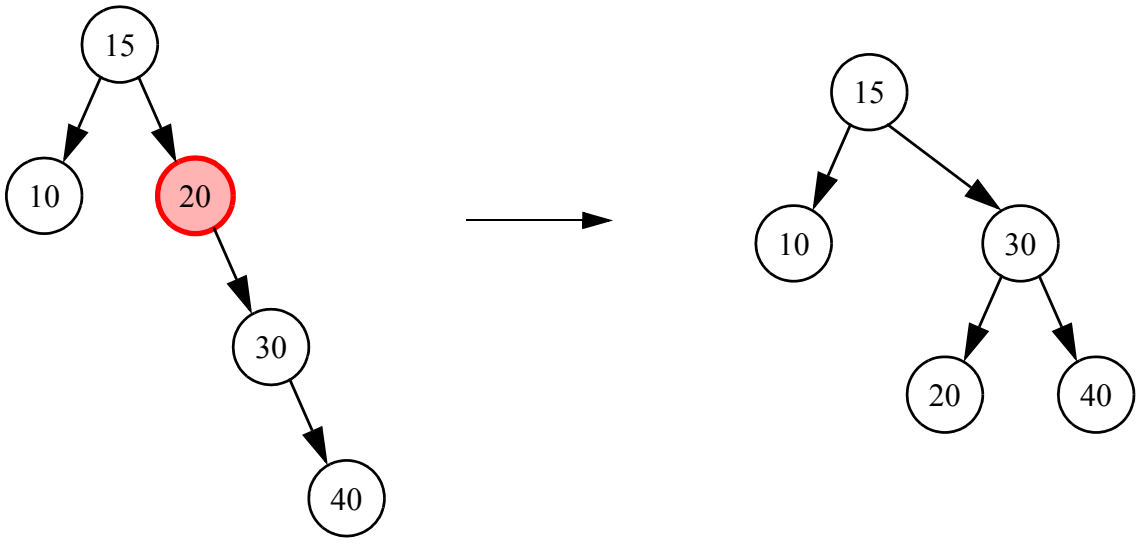
Aufgabe 3

(a)

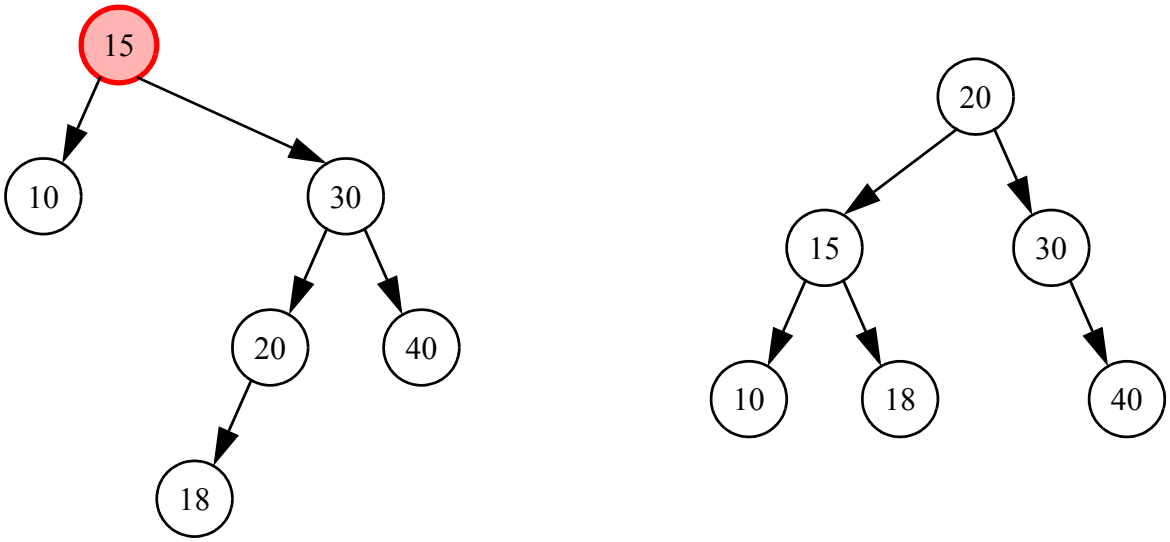
Die erste Doppelrotation ist nach Einfügen der 15 notwendig:



Das Einfügen der 40 erfordert eine einfache Rotation:

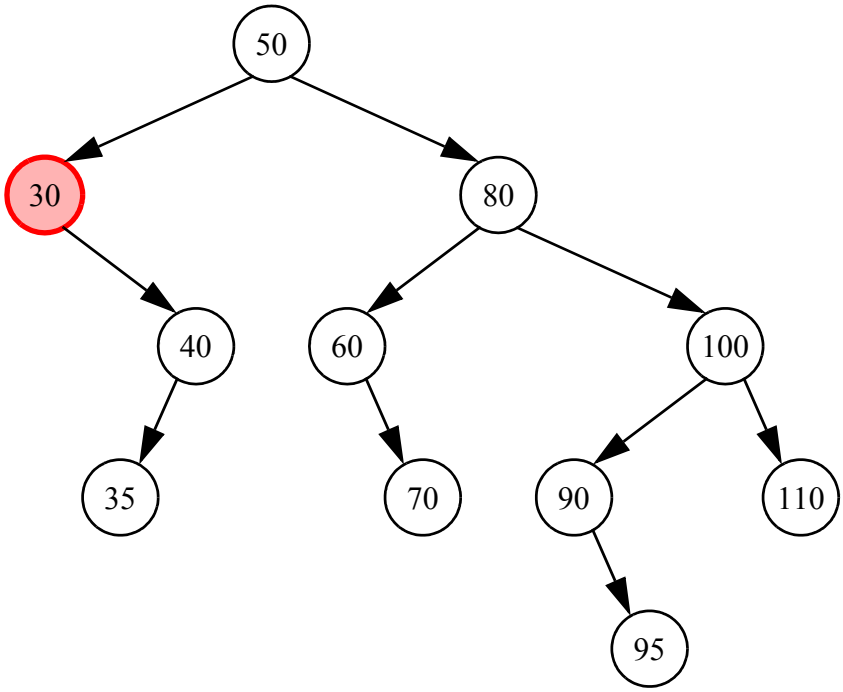


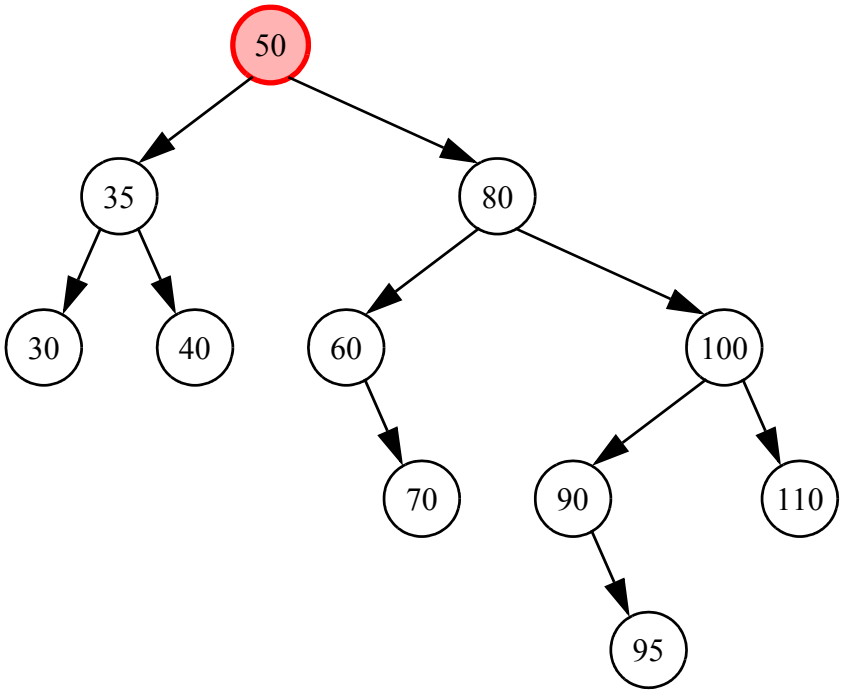
Eine Doppelrotation bringt den durch das Einfügen der 18 aus der Balance geratenen Baum wieder ins Gleichgewicht:

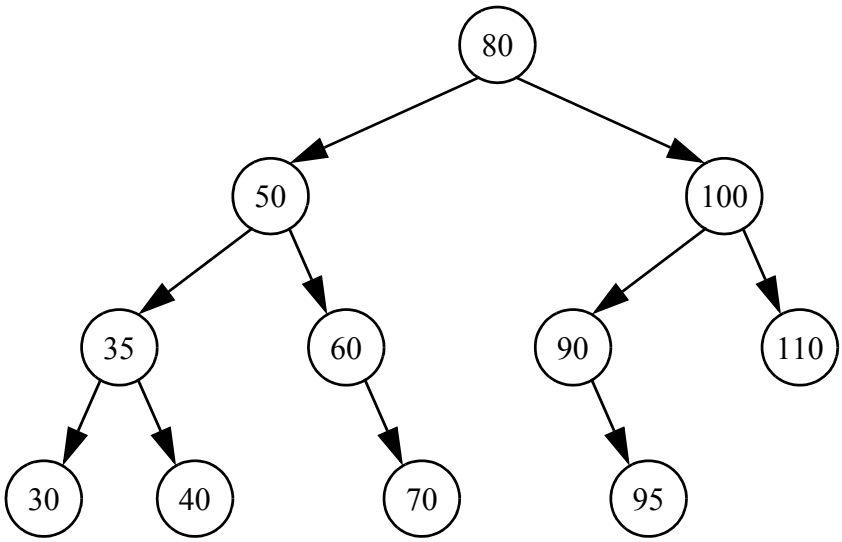


(b)

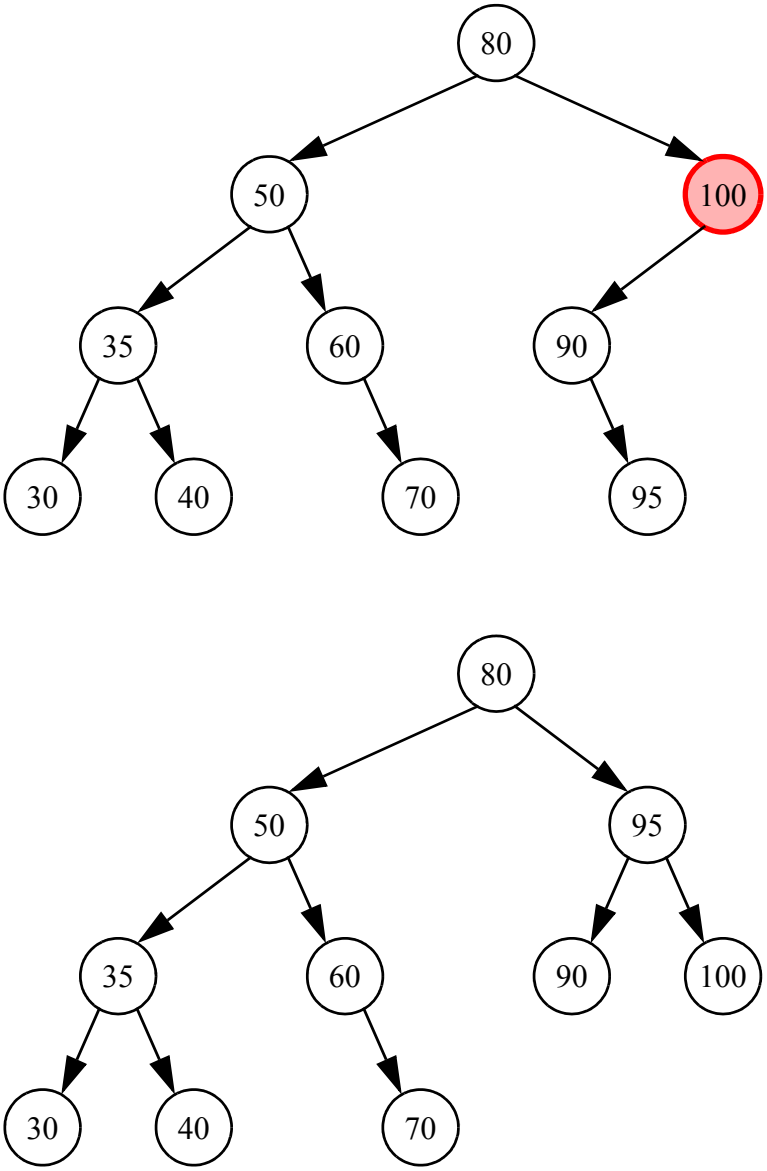
Durch das Löschen des Elements 10 ist der Baum am Knoten mit Label 30 aus dem Gleichgewicht, eine Doppelrotation behebt das Problem. Durch die Verkleinerung dieses Teilbaums ist nun die Wurzel aus der Balance geraten. Eine einfache Rotation stellt wieder einen gültigen AVL-Baum her:




50
35
80
30
40
60
100
70
90
110
95


80
50
100
35
60
90
110
30
40
70
95

Beim Löschen der 110 gerät der Teilbaum mit Wurzel 100 aus dem Gleichgewicht. Eine Doppelrotation korrigiert die Baumstruktur:



(c)

Die Suche nach einem Element in einem binären Suchbaum endet entweder in einem Knoten, der das gesuchte Element enthält oder in einem Knoten, der das nächstgrößere oder nächstkleinere Element beinhaltet. Endet die Suche also in einem Knoten k, dessen Inhalt kleiner ist als der gesuchte Wert, sind wir fertig. In den beiden anderen Fällen ($k.key \geq v$) muss das nächstkleinere Element im Baum bestimmt werden. Hat der Knoten einen linken Sohn s, so ist das nächstkleinere Element das maximale Element im Teilbaum mit Wurzel s. Ansonsten verfolgen wir theoretisch den Suchpfad rückwärts, bis wir auf einen Knoten treffen, bei dem während der Suche der rechte Sohn verwendet wurde. Dieser enthält unseren gesuchten Wert. Praktisch merken wir uns den Wert eines Knotens, wenn der rechte Suchpfad genommen wird. Gibt es keinen

solchen Knoten, sind wir im Baum ausschließlich nach links gegangen und demnach ist *v* kleiner oder gleich dem minimalen Element im Baum und die Suche ist erfolglos.

Zunächst geben wir einen Hilfsalgorithmus an, der zu einem nicht-leeren Teilbaum eines binären Suchbaums dessen Maximum bestimmt:

```
algorithm getMax( t : bintree) : elem
begin
  while t.right ≠ nil do
    t := t.right;
  end while
  return t.key;
end getMax.
```

Um uns den Wert des letzten Knotens zu „merken“, verwenden wir im Hilfsalgorithmus *getNextSmaller*1 einfach einen zusätzlichen Parameter.

```
algorithm getNextSmaller1( t: bintree, v : elem, smallerFather : elem): elem
begin
  if t.key = v then
    if t.left ≠  nil then
      return getMax(t.left);
    else
      return smallerFather;
    end if
  end if
  if t.key < v then
    if t.right = nil then
      return t.key;
    else
      return getNextSmaller(t.right,v, t.key);
    end if
  end if
  if t.key > v then
    if t.left = nil then
      return smallerFather;
    else
      return getNextSmaller(t.left,v,smallerFather);
    end if
  end if
end getNextSmaller1.
```

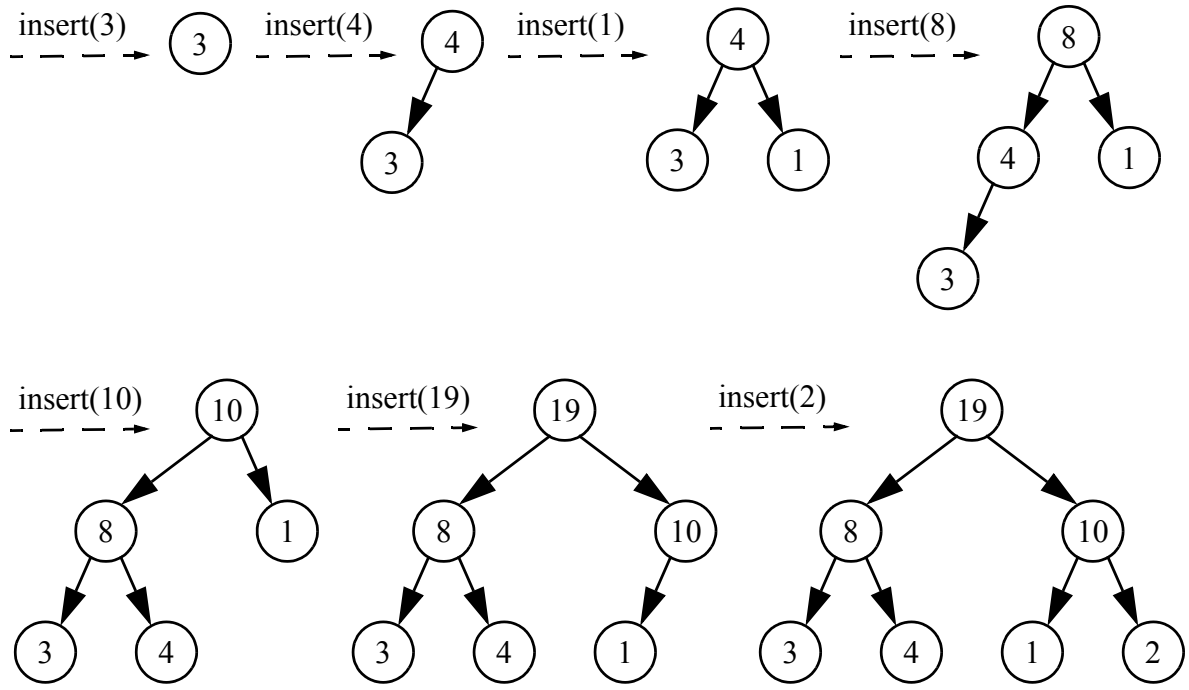
Der Hauptalgorithmus prüft zunächst, ob der übergebene Baum leer ist. Er initialisiert das Argument *smallerFather* des Algorithmus *getNextSmaller*1 mit einem Wert von *v* ist. Ist die Rückgabe von *getNextSmaller*1 gleich *v*, so ist im Baum kein kleinerer Wert als *v* gespeichert und der Algorithmus erzeugt einen Fehler.

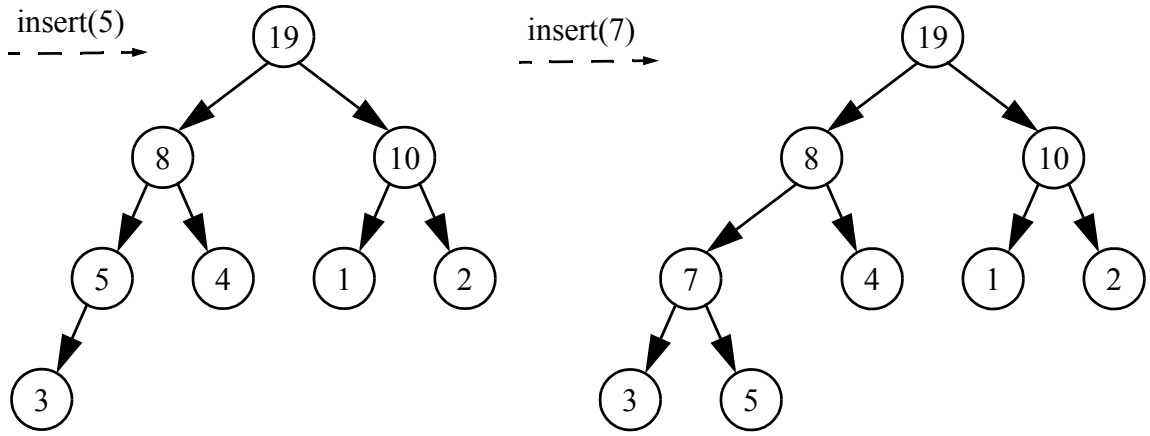
```
algorithm getNextSmaller(t : bintree, v : elem): elem
var r : elem;
begin
  if t = nil then
     return error;
  else
     r := getNextSmaller1(t, v, v);
     if r = v then
        return error;
     else // r < v
        return r;
     end if
  end if
end getNextSmaller.
```

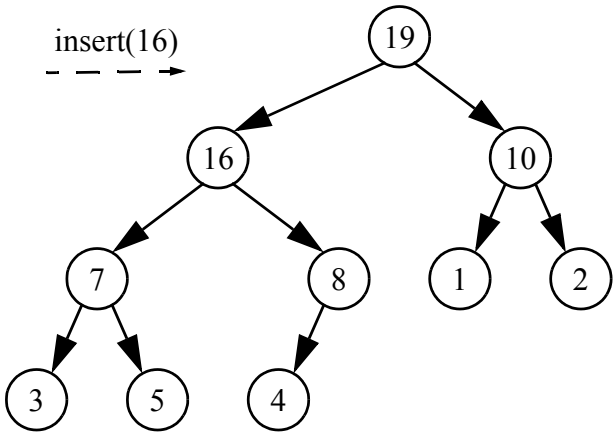
Der Algorithmus verfolgt lediglich einen einzelnen Pfad im Baum. Da die Höhe eines AVL-Baum logarithmisch in der Anzahl seiner Einträge ist, ergibt sich die logarithmische Laufzeitschranke.

Aufgabe 4

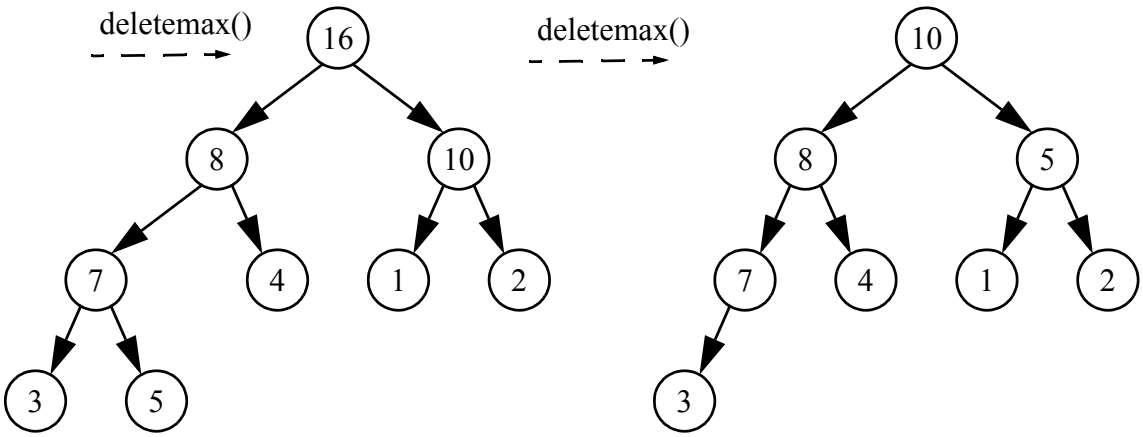
(a)

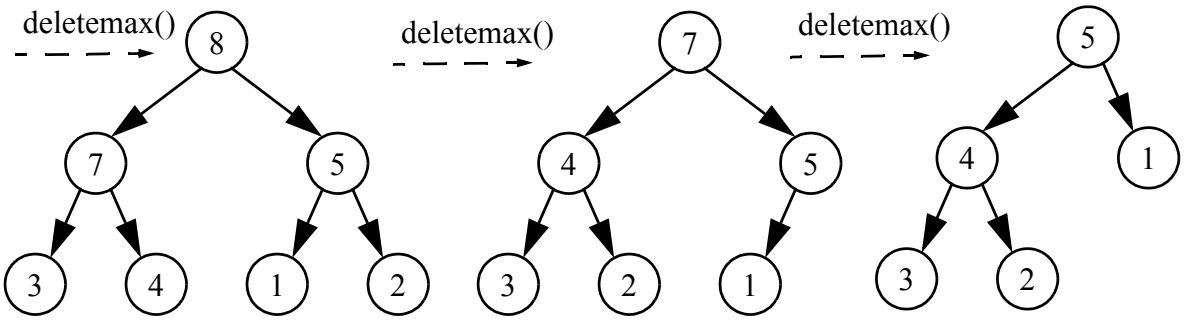



insert(5)
19
8
10
5
4
1
2
3
insert(7)
19
8
10
7
4
1
2
3
5


insert(16)
19
16
10
7
8
1
2
3
5
4

(b)


deletemax()
16
8
10
7
4
1
2
3
5
deletemax()
10
8
5
7
4
1
2
3


deletemax()
8
7
5
3
4
1
2
deletemax()
7
4
5
3
2
1
deletemax()
5
4
1
3
2