Aufgabe 1 ($\mathcal{O}$-Notation): (9 + 4 + 5 = 18 Punkte)

a) Ordnen Sie die folgenden Klassen von Funktionen in aufsteigender Größe an und setzen Sie dabei benachbarte Funktionenmengen mit "$\subset$" (echte Teilmenge) und "=" in Beziehung.

$$\mathcal{O}(n^2), \quad \mathcal{O}(3^n), \quad \mathcal{O}(n!), \quad \mathcal{O}(n+\log_2(n)), \quad \mathcal{O}\left(\sum_{i=0}^{\lceil\sqrt{n}\rceil} i\right),$$
$$\mathcal{O}\left(\frac{1}{n}\right), \quad \mathcal{O}(\log_2(n)), \quad \mathcal{O}(n^n), \quad \mathcal{O}\left(\sum_{i=0}^{n-1} i\right), \quad \mathcal{O}(n)$$

Schreiben Sie beispielsweise

$$\mathcal{O}(f(n)) \subset \mathcal{O}(g(n)) = \mathcal{O}(h(n))$$

wenn $\mathcal{O}(f(n))$ eine **echte Teilmenge** von $\mathcal{O}(g(n))$ ist und $\mathcal{O}(g(n))$ und $\mathcal{O}(h(n))$ **identisch** sind.

b) Beweisen oder widerlegen Sie folgende Aussage: Ist $f(n)$ eine Funktion mit $\mathbf{f(n) > 0}$ **für alle** $\mathbf{n} \in \mathbb{N}$, so gilt $\mathcal{O}(f(n)) = \mathcal{O}(f(n)+c)$ für alle $c \in \mathbb{R}^{>0}$.

c) Beweisen oder widerlegen Sie folgende Aussage: Ist $f(n)$ eine Funktion mit $\mathbf{f(n)} \in \mathbf{\Omega(1)}$, so gilt $\mathcal{O}(f(n)) = \mathcal{O}(f(n)+c)$ für alle $c \in \mathbb{R}^{>0}$.

Lösung:

a)

$$\mathcal{O}\left(\frac{1}{n}\right) \subset \mathcal{O}(\log_2(n)) \subset \mathcal{O}(n) = \mathcal{O}(n+\log_2(n)) = \mathcal{O}\left(\sum_{i=0}^{\lceil\sqrt{n}\rceil} i\right)$$
$$\subset \mathcal{O}(n^2) = \mathcal{O}\left(\sum_{i=0}^{n-1} i\right) \subset \mathcal{O}(3^n) \subset \mathcal{O}(n!) \subset \mathcal{O}(n^n)$$

b) Die Aussage gilt nicht, da $\mathcal{O}(f(n)+c) \not\subseteq \mathcal{O}(f(n))$. Betrachten wir beispielsweise $f(n) = \frac{1}{n}$, $c = 1$ und $g(n) = 1$. Es ist $g(n) \in \mathcal{O}(\frac{1}{n}+1)$, da $1 \leq \frac{1}{n}+1$ für alle $n \geq 1$. Andererseits ist

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{1}{\frac{1}{n}} = \lim_{n\to\infty} n = \infty$$

und deswegen $g(n) \notin \mathcal{O}(f(n))$. Daher gilt $\mathcal{O}(f(n)+c) \not\subseteq \mathcal{O}(f(n))$.

c) Die Aussage gilt. Seien $f(n) \in \Omega(1)$ und $c \in \mathbb{R}^{>0}$.

$\mathcal{O}(f(n)) \subseteq \mathcal{O}(f(n)+c)$: Sei $g(n) \in \mathcal{O}(f(n))$. Dann existiert ein $n_0 \in \mathbb{N}$ und ein $c_1 \in \mathbb{R}^{>0}$ so dass

$$\begin{aligned}
& g(n) \leq c_1 f(n) \text{ für alle } n \geq n_0 \\
\Rightarrow\ & g(n) \leq c_1 f(n) + c_1 c \text{ für alle } n \geq n_0 \\
\Rightarrow\ & g(n) \leq c_1 (f(n) + c) \text{ für alle } n \geq n_0 \\
\Rightarrow\ & g(n) \in \mathcal{O}(f(n)+c)
\end{aligned}$$

$\underline{\mathcal{O}(f(n)+c) \subseteq \mathcal{O}(f(n))}$: Sei $g(n) \in \mathcal{O}(f(n)+c)$. Wegen $f(n) \in \Omega(1)$ existieren ein $n_0^1 \in \mathbb{N}$ und ein $c_1 \in \mathbb{R}^{>0}$ so dass

$$\begin{aligned} & f(n) \geq c_1 \text{ für alle } n \geq n_0^1 \\ \Leftrightarrow & \frac{1}{c_1} f(n) \geq 1 \text{ für alle } n \geq n_0^1 \end{aligned}$$

Wegen $g(n) \in \mathcal{O}(f(n)+c)$ existieren ein n_0^2 und ein $c_2 \in \mathbb{R}^{>0}$ so dass

$$\begin{aligned} & g(n) \leq c_2(f(n)+c) \text{ für alle } n \geq n_0^2 \\ \Rightarrow\ & g(n) \leq c_2(f(n) + \frac{1}{c_1} c f(n)) \text{ für alle } n \geq \max(n_0^2, n_0^1) \\ \Rightarrow\ & g(n) \leq \underbrace{c_2(1+\frac{1}{c_1}c)}_{c^*} f(n) \text{ für alle } n \geq \underbrace{\max(n_0^2, n_0^1)}_{n_0^*} \\ \Rightarrow\ & g(n) \leq c^* f(n) \text{ für alle } n \geq n_0^* \\ \Rightarrow\ & g(n) \in \mathcal{O}(f(n)) \end{aligned}$$

Aufgabe 2 (Sortieren): (3 + 5 + 7 + 3 = 18 Punkte)

a) Sortieren Sie das folgende Array mithilfe von Bubblesort. Geben Sie dazu das Array nach jeder Swap-Operation an. Die vorgegebene Anzahl an Zeilen muss nicht mit der benötigten Anzahl an Zeilen übereinstimmen.

7	5	0	9	2

b) Sortieren Sie das folgende Array mithilfe von Quicksort. Geben Sie dazu das Array nach jeder Partition-Operation an und markieren Sie das jeweils verwendete Pivot-Element. Die vorgegebene Anzahl an Zeilen muss nicht mit der benötigten Anzahl an Zeilen übereinstimmen.

5	9	6	4	1	8	7	2	3

c) Sortieren Sie das folgende Array mithilfe von Heapsort. Geben Sie dazu das Array nach jeder Swap-Operation an. Die vorgegebene Anzahl an Zeilen muss nicht mit der benötigten Anzahl an Zeilen übereinstimmen.

6	7	3	1	8	4

d) Zeigen oder widerlegen Sie: Heapsort ist stabil.

Lösung:

a)

7	5	0	9	2
5	7	0	9	2
5	0	7	9	2
5	0	7	2	9
0	5	7	2	9
0	5	2	7	9
0	2	5	7	9

b)

5	9	6	4	1	8	7	2	3
2	1	**3**	4	9	8	7	5	6
1	2	3	4	9	8	7	5	6
1	2	3	4	5	**6**	7	9	8
1	2	3	4	**5**	6	7	9	8
1	2	3	4	5	6	7	**8**	9

6	7	3	1	8	4
6	7	4	1	8	3
6	8	4	1	7	3
8	6	4	1	7	3
8	7	4	1	6	3
3	7	4	1	6	8
7	3	4	1	6	8
7	6	4	1	3	8
3	6	4	1	7	8
6	3	4	1	7	8
1	3	4	6	7	8
4	3	1	6	7	8
1	3	4	6	7	8
3	1	4	6	7	8

c)

1	3	4	6	7	8

d) **Behauptung:** Heapsort ist nicht stabil.

Beweis:
Betrachte das Eingabearray `[1,1,2]`. Durch Heapsort wird zunächst die Heapeigenschaft hergestellt und dazu das erste und letzte Element vertauscht. Aktuell sind die gleichen Elemente dann in falscher Reihenfolge. Dann wird das erste Element mit dem letzten zurückgetauscht und `heapify` auf dem Array ohne das letzte Element aufgerufen. Da dieses Array schon die Heapeigenschaft erfüllt, geschieht in diesem Schritt nichts weiter und die gleichen Elemente sind wieder in korrekter Reihenfolge. Schließlich werden aber das erste und zweite Element vertauscht und anschließend keine weiteren Operationen mehr ausgeführt, sodass am Ende die gleichen Elemente in falscher Reihenfolge vorliegen. Dies widerlegt die Stabilität von Heapsort.
□

Aufgabe 3 (Rekursionsgleichungen): (7 + 6 + 5 = 18 Punkte)

a) Geben Sie für das Programm

```
int berechne(int n){
    if(n <= 1)
        return 3;

    if(n = 2)
        return berechne(n/2);

    int value = log(n);

    value += 3 * berechne(n/2) + 4 * berechne(value) + 5
    return value;
}

int log(n){
    int res = 0;

    while(n > 1){
        n = n / 2;
        res = res + 1
    }

    return res;
}
```

eine Rekursionsgleichung für die asymptotische Laufzeit des Aufrufes `berechne(n)` an. Die elementaren, also die für die asymptotische Laufzeit relevanten, Operationen sind alle arithmetischen Operationen sowie Vergleiche. Sie brauchen die Basisfälle der Rekursionsgleichung *nicht* anzugeben.

b) Bestimmen Sie für die Rekursionsgleichung

$$T(n) = 125 \cdot T\left(\frac{n}{25}\right) + n^{\sqrt{2}} + \log_2(n) + 4 \cdot n + 3$$

die Komplexitätsklasse Θ mit Hilfe des Master-Theorems. Begründen Sie Ihre Antwort.

Hinweis: $\sqrt{2} \approx 1.414214$

c) Finden Sie mit Hilfe eines Rekursionsbaumes eine exakte Lösung der Rekursionsgleichung

$$T(n) = 3 \cdot T(n-1) + 1, \qquad T(1) = 1\ .$$

Dabei müssen Sie gegebenenfalls auftretende Summen *nicht* auflösen.

Lösung:

a) `log(n)` hat logarithmische Laufzeit im Parameter n, da n in der `while`-Schleife in jedem Durchlauf halbiert wird. Des Weiteren ist das Ergebnis des Aufrufes von `log(n)` durch $\lfloor \log(n) \rfloor$ gegeben. Jeder Aufruf von `berechne(n)` führt zu zwei rekursiven Aufrufen, einer mit $n/2$ und einer mit $\lfloor \log(n) \rfloor$. Somit ergibt sich die folgende asymptotische Laufzeit für den Aufruf:

$$T(n) \ = \ T\left(\frac{n}{2}\right) + T\big(\log(n)\big) + \log(n) \qquad T(0) = T(1) = T(2) = 1$$

b) Es sind $b = 125$, $c = 25$ und $f(n) = n^{\sqrt{2}} + \log_2(n) + 4 \cdot n + 3$. E wird nun wie folgt bestimmt:

$$E \;=\; \frac{\log(125)}{\log(25)} \;=\; \frac{3}{2} \;=\; 1.5$$

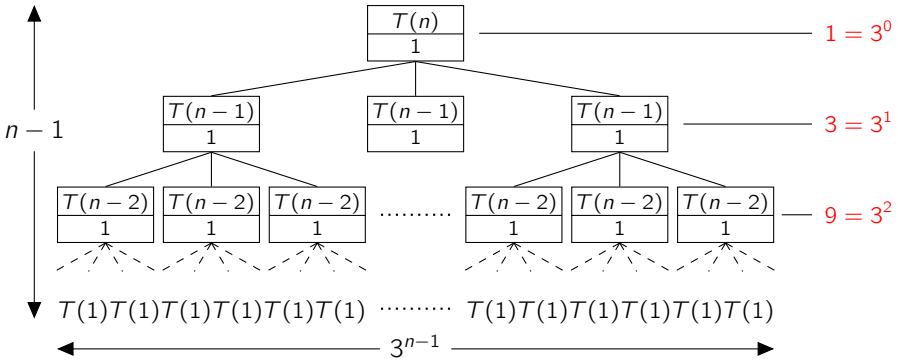
Damit gilt $n^E = n^{1.5}$. Wähle nun $\varepsilon = 1.5 - \sqrt{2} \approx 0.086$. Wir bestimmen nun den folgenden Grenzwert:

$$\begin{aligned}
& \lim_{n\to\infty} \frac{f(n)}{n^{E-\varepsilon}} \\
= \;& \lim_{n\to\infty} \frac{n^{\sqrt{2}} + \log_2(n) + 4 \cdot n + 3}{n^{1.5-(1.5-\sqrt{2})}} \\
= \;& \lim_{n\to\infty} \frac{n^{\sqrt{2}} + \log_2(n) + 4 \cdot n + 3}{n^{\sqrt{2}}} \\
= \;& \lim_{n\to\infty} \frac{n^{\sqrt{2}}}{n^{\sqrt{2}}} + \lim_{n\to\infty} \frac{\log_2(n)}{n^{\sqrt{2}}} + \lim_{n\to\infty} \frac{4 \cdot n}{n^{\sqrt{2}}} + \lim_{n\to\infty} \frac{3}{n^{\sqrt{2}}} \\
\overset{\text{l'Hospital}}{=} \;& \lim_{n\to\infty} \underbrace{1}_{\to 1} + \lim_{n\to\infty} \underbrace{\frac{1}{\ln(2) \cdot n \cdot \sqrt{2} \cdot n^{\sqrt{2}-1}}}_{\to 0} + \lim_{n\to\infty} \underbrace{\frac{4}{n^{\sqrt{2}-1}}}_{\to 0} + \lim_{n\to\infty} \underbrace{\frac{3}{n^{\sqrt{2}}}}_{\to 0} \\
= \;& 1 + 0 + 0 + 0 \;=\; 1
\end{aligned}$$

Damit gilt $f(n) \in \Theta\left(n^{E-\varepsilon}\right)$ und damit insbesondere $f(n) \in \mathcal{O}\left(n^{E-\varepsilon}\right)$. Somit findet der *erste Fall* des Master-Theorems Anwendung und es ergibt sich die Komplexitätsklasse $\Theta\left(n^E\right)$ für $T(n)$, also

$$T(n) \in \Theta\left(n^{1.5}\right) \;.$$

c) Es ergibt sich der folgende Rekursionsbaum:



Aus dem Rekursionsbaum lässt sich die Laufzeit wie folgt ablesen:

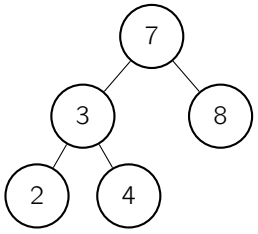
$$T(n) \;=\; \sum_{i=0}^{n-2} \left(3^i\right) + 3^{n-1} \;=\; \sum_{i=0}^{n-1} 3^i$$

Aufgabe 4 (Bäume): (2 + 3 + 2 + 4 + 3 + 4 = 18 Punkte)

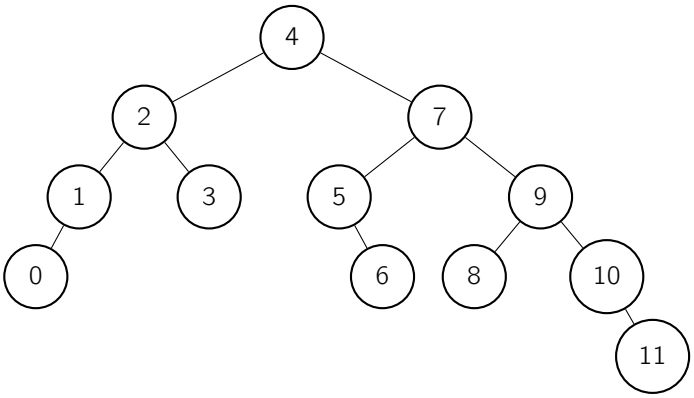
a) Geben Sie einen höhenbalancierten Binärbaum (also die Höhe der Teilbäume unterscheidet sich höchstens um 1 an jedem Knoten) mit den Schlüsselwerten von 0 bis 6 an, der bei *Inorder-Traversierung* die Schlüssel in folgender Reihenfolge ausgibt:

3, 6, 4, 0, 2, 1, 5

b) Fügen Sie den Wert 6 in den folgenden *AVL-Baum* ein und geben Sie die entstehenden Bäume nach jeder *Einfügeoperation* sowie jeder *Rotation* an. Markieren Sie außerdem zu jeder Rotation, welcher Knoten in welche Richtung rotiert wird:



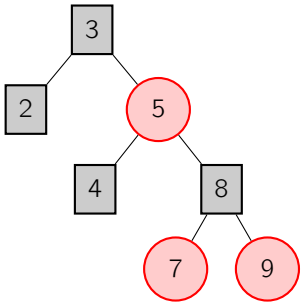
c) Löschen Sie den Wert 4 aus dem folgenden *AVL-Baum* und geben Sie die entstehenden Bäume nach jeder *Löschoperation* sowie jeder *Rotation* an. Markieren Sie außerdem zu jeder Rotation, welcher Knoten in welche Richtung rotiert wird:



d) Fügen Sie den Wert 6 in den folgenden *Rot-Schwarz-Baum* ein und geben Sie die entstehenden Bäume nach

- jeder *Einfügeoperation*,
- jeder *Rotation* sowie
- jeder *Umfärbung* an.

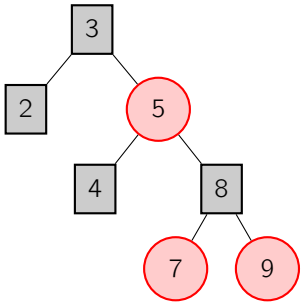
Markieren Sie außerdem zu jeder Rotation, welcher Knoten in welche Richtung rotiert wird. Mehrere Umfärbungen können Sie in einem Schritt zusammenfassen. Beachten Sie, dass rote Knoten rund und schwarze Knoten eckig dargestellt werden.



e) Löschen Sie den Wert 4 aus dem folgenden *Rot-Schwarz-Baum* und geben Sie die entstehenden Bäume nach

- jeder *Löschoperation*,
- jeder *Rotation* sowie
- jeder *Umfärbung* an.

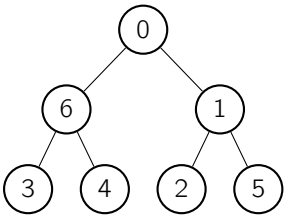
Markieren Sie außerdem zu jeder Rotation, welcher Knoten in welche Richtung rotiert wird. Mehrere Umfärbungen können Sie in einem Schritt zusammenfassen. Beachten Sie, dass rote Knoten rund und schwarze Knoten eckig dargestellt werden.



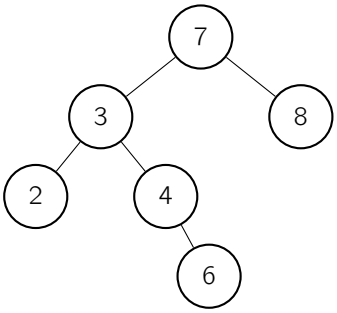
f) Beschreiben Sie stichpunktartig ein Verfahren, mit dem man aus einem aufsteigend sortierten Array einen AVL-Baum mit den gleichen Elementen in gleicher Anzahl konstruieren kann und dessen Worst-Case Laufzeit in $\mathcal{O}(n)$ ist, wobei n die Länge des Eingabearrays ist und die Anzahl an Vergleichen und Rechenoperationen sowie Lese- und Schreibzugriffen als elementare Operationen berücksichtigt werden sollen. Begründen Sie, warum Ihr Verfahren diese Laufzeitschranke einhält.

Lösung:

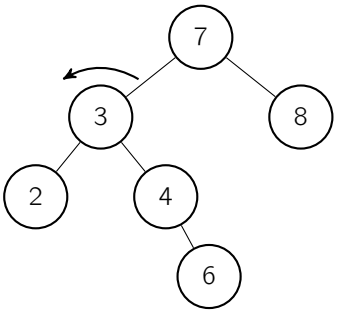
a)

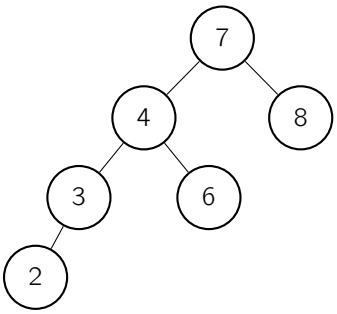


b) füge 6 ein

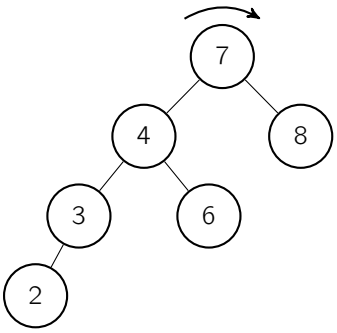


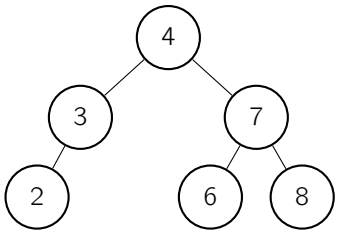
rotiere 3 nach links



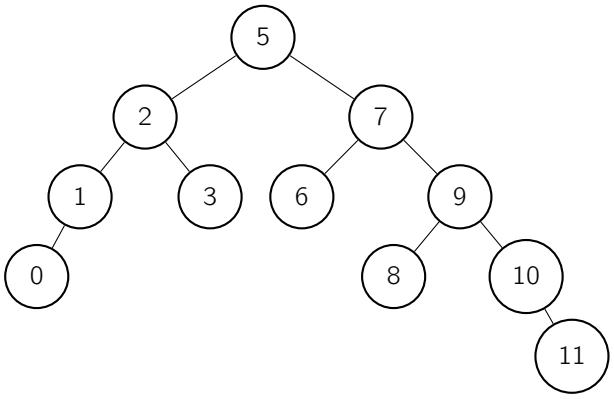


rotiere 7 nach rechts

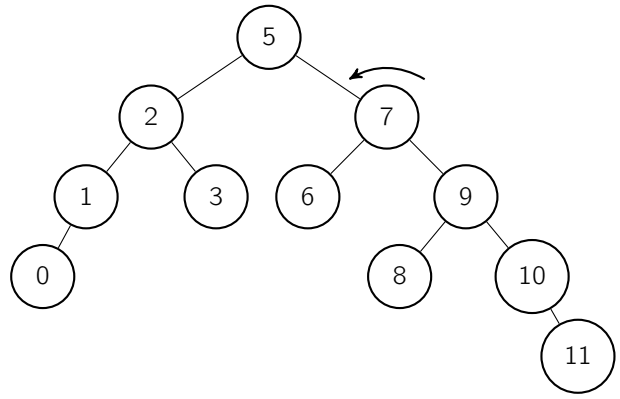


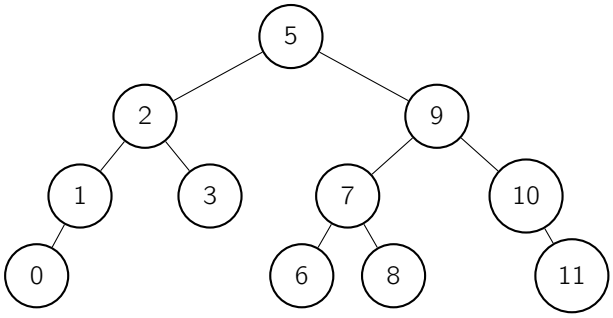


c) entferne 4

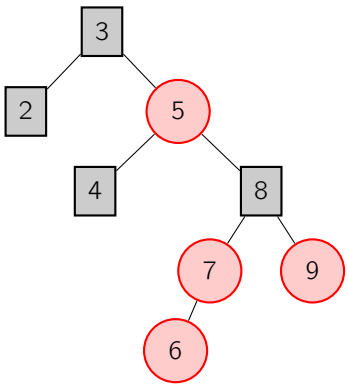


rotiere 7 nach links

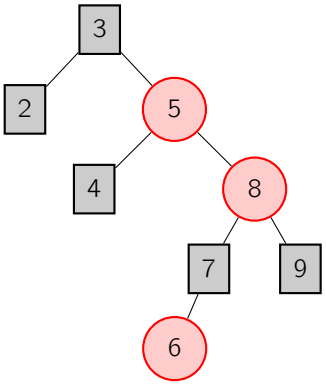




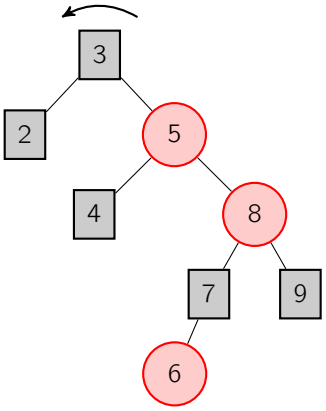
d) füge 6 ein

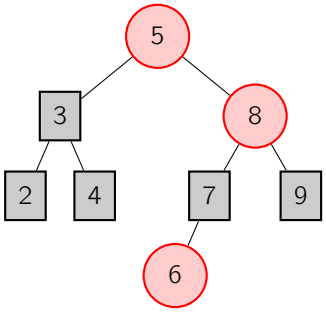


Fall 1: umfärben

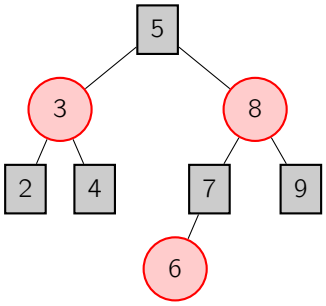


rotiere 3 nach links

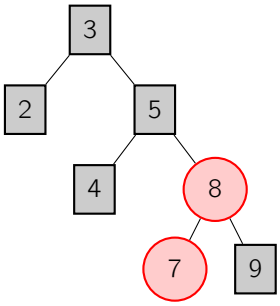




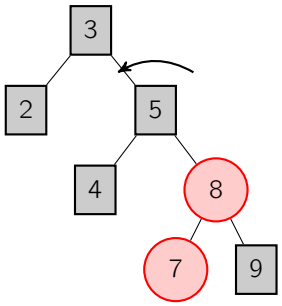
Fall 3: umfärben

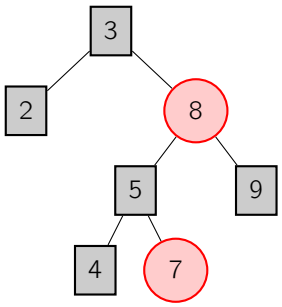


e) Lösche 4:
Fall 4: umfärben

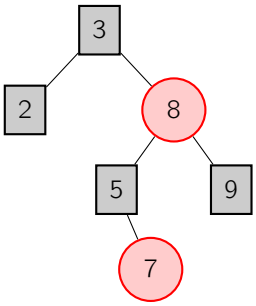


rotiere 5 nach links





ersetze 4 durch rechtes Kind



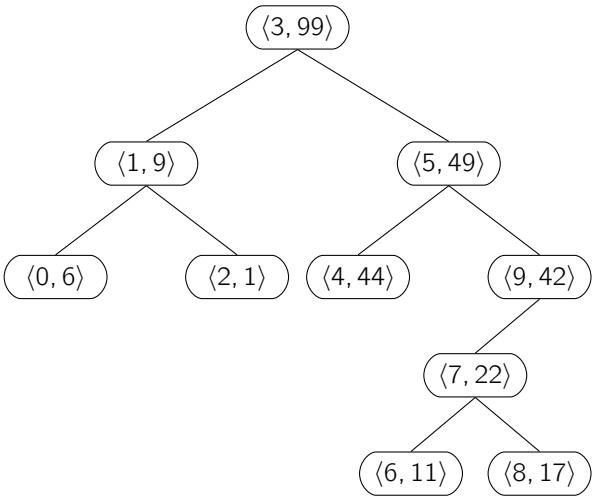
f) Wir verwenden das mittlere Element des aktuell betrachteten Arraybereichs (zu Beginn natürlich das gesamte Array) als aktuellen Knoten und konstruieren die beiden Teilbäume rekursiv aus der linken und rechten Hälfte des übrigen Arraybereichs (bei leerem Arraybereich wird abgebrochen). Auf diese Weise wird jedes Element genau einmal besucht, was insgesamt zu einer linearen Laufzeit führt.

Aufgabe 5 (Datenstruktur): (9 + 9 = 18 Punkte)

Ein **Treap** (auch Balde oder Baufen) t ist ein binärer Baum. Ein Knoten besitzt

- einen Schlüssel **key**,
- eine Priorität **prio**,
- einen Zeiger **parent**, der auf den Elternknoten zeigt,
- einen Zeiger **left**, der auf den linken Teilbaum zeigt,
- sowie einen Zeiger **right**, der auf den rechten Teilbaum zeigt,

Bezüglich der Schlüssel erfüllt t die Suchbaum-Eigenschaft und bezüglich der Prioritäten erfüllt t einen Teil der Max-Heap-Eigenschaft: die Priorität jedes Knotens ist größer oder gleich der Prioritäten seiner Kinder. Ein Beispiel-Treap mit Knoten $\langle$**key**, **prio**$\rangle$ könnte also wie folgt aussehen:



Die Operation `insert(Treap t, int key, int prio)`, die ein neues Element mit dem Schlüssel `key` und der Priorität `prio` in den Treap `t` einfügt, kann wie folgt realisiert werden

```
insert(Treap t, int key, int prio) {
    Node n = new Node(key, prio);
    bstIns(t, n);
    bubble(n);
}
```

wobei

(i) `bstIns(Treap t, Node n)` ein neues Element `n` entsprechend der Suchbaum-Eigenschaft in den Treap `t` einfügt,

(ii) `bubble(Element n)` ein Element `n` solange durch Rotationen bewegt, bis die Max-Heap-Eigenschaft nicht mehr verletzt ist,

(iii) der Konstruktor `Node(int key, int prio)` einen Knoten kreiert, dessen Schlüssel und Priorität mit den gegebenen Werten initialisiert und dessen Elternknoten **parent** und Nachfolger **left**, **right** auf den Wert **null** setzt.

a) Implementieren Sie die Operation `bubble(Node n)`. Beachten Sie hierbei, dass Ihre Implementierung gewährleisten muss, dass `insert(Treap t, int key, int prio)` die Treap-Eigenschaften herstellt.

Hinweis: Sie dürfen die bekannten Operationen `leftRotate` und `rightRotate` aus der Vorlesung benutzen.

b) Gegeben seien n Elemente $\langle$**key**, **prio**$\rangle$ bestehend aus Schlüsseln und ihren Prioritäten, wobei jeder Schlüssel **maximal einmal** und jede Priorität **maximal einmal** vorkommt. Argumentieren Sie (informell), wieso der Treap t, der diese Elemente speichert, **eindeutig** ist. Das bedeutet, es gibt keinen anderen Treap t', der genau diese Elemente speichert.

Lösung:

a) Die Operation `bubble(Node n)` kann iterativ wie folgt implementiert werden.

```
bubble(Node n) {
    while (n.parent != null && n.parent.prio < n.prio) {
        if (n.parent.right == n) {
            leftRotate(n.parent);
        } else {
            rightRotate(n.parent);
        }
    }
}
```

Alternativ funktioniert folgende rekursive Implementierung.

```
bubble(Node n) {
    if (n.parent != null) {
        if (n.parent.right == n) {
            leftRotate(n.parent);
        } else {
            rightRotate(n.parent);
        }
        if (n.parent != null && n.parent.prio < n.prio) {
            bubble(n);
        }
    }
}
```

b) Aus der Max-Heap-Eigenschaft folgt direkt, dass ein Knoten v im Treap t die höchste Priorität in seinem Teilbaum besitzt. Durch die Suchbaumeigenschaft wird die Menge der Knoten V durch einen beliebigen Schlüssel k in zwei disjunkte Teilmengen geteilt, nämlich in die Menge der Elemente, deren Schlüssel kleiner als k ist, sowie die Elemente, deren Schlüssel größer als k ist.

Aus diesen beiden Eigenschaften folgt unmittelbar, dass der Treap, der n gegebene Elemente speichert, eindeutig ist, da die Wurzel jedes Teilbaums das Element mit der größten Priorität sein muss und durch die Suchbaumeigenschaft, die Menge der Elemente, die im linken bzw. rechten Teilbaum gespeichert werden, eindeutig ist.