GAD-Trainer

Johannes Feil & Gregor Schwarz

feil@in.tum.de & gregor.schwarz@tum.de

http://tutoritum.de

15. September 2013

Das nachfolgende Dokument enthält einige Übungsaufgaben zu ausgewählten Themen der Veranstaltung „Grundlagen Algorithmen und Datenstrukturen“ des Sommersemesters 2013. Die Aufgaben dienen den Studenden als zusätzliche Übung und als Vorbereitung auf die Klausur. Es ist zu beachten, dass dieses Dokument **nicht** zu den offiziellen Unterlagen der Übungsleitung zählt. Für Richtigkeit und Vollständigkeit wird daher ausdrücklich **keine** Haftung übernommen.

Wir weisen ausdrücklich darauf hin, dass die Kapitel 9 und 10 nicht Gegenstand dieses Trainers sind. Das heißt aber im Umkehrschluss natürlich nicht, dass wir diese Kapitel für nicht klausurrelevant erachten.

Aufgabe 0 (Kurzfragen)

		Wahr	Falsch
(a)	GAD ist geil!!! :)	☐	☐
(b)	$f(n) \in o(g(n)) \quad \Rightarrow \quad f(n) \in \mathcal{O}(g(n))$	☐	☐
(c)	Wenn $f(n)$ und $g(n)$ nicht unvergleichbar sind, dann gilt: $f(n) \notin \Omega(g(n)) \quad \Rightarrow \quad f(n) \in \mathcal{O}(g(n))$	☐	☐
(d)	$f(n) \in \Theta(g(n)) \quad \Rightarrow \quad f(n) \in o(g(n))$	☐	☐
(e)	$f(n) \in \mathcal{O}(g(n)) \quad \Rightarrow \quad f(n) \in \Omega(g(n))$	☐	☐
(f)	$f(n) \in \mathcal{O}(g(n)) \quad \Rightarrow \quad f(n) \notin \omega(g(n))$	☐	☐
(g)	Es eixistieren 1-universelle Hashfamilien, die eine Funktion beinhalten, welche alle Keys auf einen einzigen Wert hasht.	☐	☐
(h)	AVL-Bäume wurden von den drei Mathematikern Adelson, Velsky und Landis 1962 am MIT entwickelt.	☐	☐
(i)	Die *locate*-Operation bei Binären Suchbäumen liegt im Worst Case in $\Theta(\log(n))$.	☐	☐
(j)	Die Anzahl der Elemente in einem Binomial-Baum ist immer gleich einer Zweierpotenz.	☐	☐
(k)	Die effiziente *build*-Methode, die man bei Binären Heaps verwendet und die in $\mathcal{O}(n)$ liegt, darf auch bei Binomial-Heaps angewendet werden.	☐	☐
(l)	Jeder vollständige Binärbaum der Tiefe t ist auch ein fast vollständiger Binärbaum der Tiefe t.	☐	☐
(m)	Für das Sortieren von 2 Terabyte Daten eignet sich insbesondere der Merge-Sort-Algorithmus. .	☐	☐
(n)	Die Laufzeit der $find$-Operation bei B^*-Bäumen ist in der Regel besser als die bei (a, b)-Bäumen .	☐	☐
(o)	Ein Fibonaccibaum der Tiefe t gleicht einem AVL-Baum der Tiefe t mit minimal vielen Knoten.	☐	☐
(p)	In einem Fibonaccibaum, bei dem für jeden Knoten der linke Teilbaum tiefer ist als der rechte, haben alle Knoten bis auf die Blätter eine Balancierungszahl von -1.	☐	☐
(q)	Das Ziel von Perfektem Hashing ist es, auch im Worst Case für die $find$-Operation eine konstante Laufzeit zu garantieren.	☐	☐
(r)	Ein dynamisches Array mit $\alpha = 4$ und $\beta = 3$ der Größe $w = 9$ enthält 3 Elemente. Nach dem Löschen eines weiteren Elementes muss die Größe des Arrays halbiert werden.	☐	☐
(s)	Die amortisierte Laufzeit einer Folge von Operationen ist stets eine obere Schranke für die tatsächliche Laufzeit der Folge.	☐	☐

		Wahr	**Falsch**
(t)	MergeSort hat eine Worst Case Laufzeit in $\mathcal{O}(n \log n)$ und eine Best Case Laufzeit in $\Omega(n \log n)$.	☐	☐
(u)	Die Worst Case Laufzeit des QuickSort-Algorithmus liegt in $\mathcal{O}(n \log n)$, vorausgesetzt man wählt des Pivotelement jeweils zufällig.	☐	☐
(v)	Mit Hilfe des Master Theorems kann man die Laufzeit von beliebigen rekursiven Algorithmen abschätzen.	☐	☐
(w)	Binäre Heaps eignen sich, wenn man eine Datenstruktur zur effizienten Suche von Objekten benötigt.	☐	☐
(x)	Ohne den min-Pointer würde die Bestimmung des Minimums in Binomial-Heaps im Worst Case eine Laufzeit in $\Omega(\log n)$ benötigen.	☐	☐
(y)	Eine einzelne decreaseKey Operation von Fibonacci-Heaps hat stets konstante Laufzeit.	☐	☐
(z)	Die Laufzeit eines Algorithmus hängt nicht nur vom zugrunde liegenden Kostenmodell, sondern auch von der Kodierung der Eingabe ab.	☐	☐

		Wahr	**Falsch**
(a)	GAD ist geil!!! :)	☒	☐
(b)	$f(n) \in o(g(n)) \quad \Rightarrow \quad f(n) \in \mathcal{O}(g(n))$	☒	☐
(c)	Wenn $f(n)$ und $g(n)$ nicht unvergleichbar sind, dann gilt: $f(n) \notin \Omega(g(n)) \quad \Rightarrow \quad f(n) \in \mathcal{O}(g(n))$	☒	☐
(d)	$f(n) \in \Theta(g(n)) \quad \Rightarrow \quad f(n) \in o(g(n))$	☐	☒
(e)	$f(n) \in \mathcal{O}(g(n)) \quad \Rightarrow \quad f(n) \in \Omega(g(n))$	☐	☒
(f)	$f(n) \in \mathcal{O}(g(n)) \quad \Rightarrow \quad f(n) \notin \omega(g(n))$	☒	☐
(g)	Es eixistieren 1-universelle Hashfamilien, die eine Funktion beinhalten, welche alle Keys auf einen einzigen Wert hasht.	☒	☐
(h)	AVL-Bäume wurden von den drei Mathematikern Adelson, Velsky und Landis 1962 am MIT entwickelt.	☐	☒
(i)	Die *locate*-Operation bei Binären Suchbäumen liegt im Worst Case in $\Theta(\log(n))$.	☐	☒
(j)	Die Anzahl der Elemente in einem Binomial-Baum ist immer gleich einer Zweierpotenz.	☒	☐
(k)	Die effiziente *build*-Methode, die man bei Binären Heaps verwendet, darf auch bei Binomial-Heaps angewendet werden.	☐	☒
(l)	Jeder vollständige Binärbaum der Tiefe t ist auch ein fast vollständiger Binärbaum der Tiefe t.	☒	☐.
(m)	Für das Sortieren von 2 Terabyte Daten eignet sich insbesondere der Merge-Sort-Algorithmus.	☒	☐
(n)	Die Laufzeit der $find$-Operation bei B^*-Bäumen ist in der Regel besser als die bei (a, b)-Bäumen .	☒	☐
(o)	Ein Fibonaccibaum der Tiefe t gleicht einem AVL-Baum der Tiefe t mit minimal vielen Knoten.	☒	☐
(p)	In einem Fibonaccibaum, bei dem für jeden Knoten der linke Teilbaum tiefer ist als der rechte, haben alle Knoten bis auf die Blätter eine Balancierungszahl von -1.	☒	☐
(q)	Das Ziel von Perfektem Hashing ist es, auch im Worst Case für die $find$-Operation eine konstante Laufzeit zu garantieren.	☒	☐
(r)	Ein dynamisches Array mit $\alpha = 4$ und $\beta = 3$ der Größe $w = 9$ enthält 3 Elemente. Nach dem Löschen eines weiteren Elementes muss die Größe des Arrays halbiert werden.	☐	☒
(s)	Die amortisierte Laufzeit einer Folge von Operationen ist stets eine obere Schranke für die tatsächliche Laufzeit der Folge.	☒	☐

		Wahr	**Falsch**
(t)	MergeSort hat eine Worst Case Laufzeit in $\mathcal{O}(n \log n)$ und eine best case Laufzeit in $\Omega(n \log n)$.	☒	☐
(u)	Die Worst Case Laufzeit des QuickSort-Algorithmus liegt in $\mathcal{O}(n \log n)$, vorausgesetzt man wählt des Pivotelement jeweils zufällig.	☐	☒
(v)	Mit Hilfe des Master Theorems kann man die Laufzeit von beliebigen rekursiven Algorithmen abschätzen.	☐	☒
(w)	Binäre Heaps eignen sich, wenn man eine Datenstruktur zur effizienten Suche von Objekten benötigt.	☐	☒
(x)	Ohne den min-Pointer würde die Bestimmung des Minimums in Binomial-Heaps im Worst Case eine Laufzeit in $\Omega(\log n)$ benötigen.	☒	☐
(y)	Eine einzelne decreaseKey Operation von Fibonacci-Heaps hat stets konstante Laufzeit.	☐	☒
(z)	Die Laufzeit eines Algorithmus hängt nicht nur vom zugrunde liegenden Kostenmodell, sondern auch von der Kodierung der Eingabe ab.	☒	☐

Aufgabe 1 (Landau-Symbole)
Zeigen Sie $\sqrt{n} \in \Omega\big(\ln(n^{\ln(n)})\big)$.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\sqrt{n}}{\ln(n^{\ln(n)})} = \lim_{n\to\infty} \frac{\sqrt{n}}{\ln(n)\,\ln(n)} = \lim_{n\to\infty} \frac{\sqrt{n}}{\ln(n)^2}$$

$$\overset{l'H}{=} \lim_{n\to\infty} \frac{\frac{1}{2}\,n^{-1/2}}{2\,\ln(n)\,n^{-1}} = \lim_{n\to\infty} \frac{\sqrt{n}}{4\,\ln(n)} \overset{l'H}{=} \lim_{n\to\infty} \frac{\frac{1}{2}\,n^{-1/2}}{4\,n^{-1}}$$

$$= \lim_{n\to\infty} \frac{\sqrt{n}}{8} = \infty$$

Da $0 < \lim_{n\to\infty} \frac{\sqrt{n}}{\ln(n^{\ln(n)})} = \infty$, folgt $\sqrt{n} \in \Omega\big(\ln(n^{\ln(n)})\big)$.

Aufgabe 2 (Landau-Symbole)
Zeigen Sie $n! \in o(n^{n+1})$.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n!}{n^{n+1}} = \lim_{n\to\infty} \frac{n\cdot(n-1)\cdot(n-2)\cdot\ldots\cdot 2\cdot 1}{n^{n+1}}$$

$$\leq \lim_{n\to\infty} \frac{n^n}{n^{n+1}} = \lim_{n\to\infty} \frac{1}{n} = 0$$

Da $\lim_{n\to\infty} \frac{n!}{n^{n+1}} = 0$ folgt $n! \in o(n^{n+1})$.

Aufgabe 3 (Landau-Symbole)
Zeigen Sie $\ln(n!) \in \Theta(n \ln(n))$.

Es gilt $f(n) \in \Theta(g(n)) \quad \Leftrightarrow \quad f(n) \in \mathcal{O}(g(n))$ und $f(n) \in \Omega(g(n))$.

$\ln(n!) \in \mathcal{O}(n \ln(n))$:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\ln(n!)}{n \ln(n)} = \lim_{n\to\infty} \frac{\sum_{i=1}^{n} \ln(i)}{n \ln(n)}$$

$$\leq \lim_{n\to\infty} \frac{\sum_{i=1}^{n} \ln(n)}{n \ln(n)} = \lim_{n\to\infty} \frac{n \ln(n)}{n \ln(n)} = 1$$

Wegen $\lim_{n\to\infty} \frac{\ln(n!)}{n \ln(n)} \leq 1 < \infty$ ist $\ln(n!) \in \mathcal{O}(n \ln(n))$.

$\ln(n!) \in \Omega(n \ln(n))$:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\ln(n!)}{n \ln(n)} = \lim_{n\to\infty} \frac{\sum_{i=1}^{n} \ln(i)}{n \ln(n)}$$

$$\geq \lim_{n\to\infty} \frac{\sum_{i=\lfloor \frac{n}{2} \rfloor}^{n} \ln(i)}{n \ln(n)} \geq \lim_{n\to\infty} \frac{\sum_{i=\lfloor \frac{n}{2} \rfloor}^{n} \ln(\lfloor \frac{n}{2} \rfloor)}{n \ln(n)}$$

$$= \lim_{n\to\infty} \frac{\lceil \frac{n}{2} \rceil \ln(\lfloor \frac{n}{2} \rfloor)}{n \ln(n)} \geq \lim_{n\to\infty} \frac{\frac{n}{2} \ln(\lfloor \frac{n}{2} \rfloor)}{n \ln(n)}$$

$$\geq \lim_{n\to\infty} \frac{\frac{n}{2} \ln(\frac{n}{2} - 1)}{n \ln(n)} = \lim_{n\to\infty} \frac{\ln(\frac{n}{2} - 1)}{2 \ln(n)}$$

$$\overset{l'H}{=} \lim_{n\to\infty} \frac{\frac{1}{\frac{n}{2}-1} \cdot \frac{1}{2}}{2 \cdot \frac{1}{n}} = \lim_{n\to\infty} \frac{n}{4} \cdot \frac{1}{\frac{n}{2} - 1}$$

$$= \lim_{n\to\infty} \frac{n}{2n - 4} \overset{l'H}{=} \lim_{n\to\infty} \frac{1}{2} = \frac{1}{2}$$

Wegen $0 < \frac{1}{2} \leq \lim_{n\to\infty} \frac{\ln(n!)}{n \ln(n)}$ ist $\ln(n!) \in \Omega(n \ln(n))$.

Insgesamt ist daher $\ln(n!) \in \Theta(n \ln(n))$.

Aufgabe 4 (Landau-Symbole)
Gegeben seien die folgenden Funktionen von $\mathbb{N}$ nach $\mathbb{R}$:

$f_1 = n^{1+\cos(n \cdot \pi)}$
$f_2 = \log_5(n^2)$
$f_3 = 5^n$
$f_4 = 3\,\log_6(n)$
$f_5 = 1337$
$f_6 = (\ln(\sqrt{n}))^2$
$f_7 = \ln(\ln(\sqrt{n}))$
$f_8 = 3^{n+1}$
$f_9 = 1 \,+\, (1 + (-1)^n)\, n^2$
$f_{10} = 15\,n + (\sin(n \cdot \pi))^2$

Kreuzen Sie in den Zeilen (a) bis (f) jeweils
„$\Delta = o$“ an, wenn $a(n) \in o(b(n))$ gilt,
kreuzen Sie „$\Delta = \mathcal{O}$“ an, wenn $a(n) \in \mathcal{O}(b(n))$ gilt,
kreuzen Sie „$\Delta = \omega$“ an, wenn $a(n) \in \omega(b(n))$ gilt,
kreuzen Sie „$\Delta = \Omega$“ an, wenn $a(n) \in \Omega(b(n))$ gilt,
und kreuzen Sie „Unvergleichbar “ an, wenn die beiden betrachteten Funktionen unvergleichbar sind.
Bergünden Sie Ihre Antworten jeweils mit einem kurzen Beweis.

(a) $f_5 \in \Delta(f_2)$	☐ $\Delta = o$	☐ $\Delta = \mathcal{O}$	☐ $\Delta = \omega$	☐ $\Delta = \Omega$	☐ Unvergleichbar
(b) $f_6 \in \Delta(f_7)$	☐ $\Delta = o$	☐ $\Delta = \mathcal{O}$	☐ $\Delta = \omega$	☐ $\Delta = \Omega$	☐ Unvergleichbar
(c) $f_3 \in \Delta(f_8)$	☐ $\Delta = o$	☐ $\Delta = \mathcal{O}$	☐ $\Delta = \omega$	☐ $\Delta = \Omega$	☐ Unvergleichbar
(d) $f_2 \in \Delta(f_4)$	☐ $\Delta = o$	☐ $\Delta = \mathcal{O}$	☐ $\Delta = \omega$	☐ $\Delta = \Omega$	☐ Unvergleichbar
(e) $f_5 \in \Delta(f_{10})$	☐ $\Delta = o$	☐ $\Delta = \mathcal{O}$	☐ $\Delta = \omega$	☐ $\Delta = \Omega$	☐ Unvergleichbar
(f) $f_1 \in \Delta(f_{10})$	☐ $\Delta = o$	☐ $\Delta = \mathcal{O}$	☐ $\Delta = \omega$	☐ $\Delta = \Omega$	☐ Unvergleichbar
(g) $f_1 \in \Delta(f_9)$	☐ $\Delta = o$	☐ $\Delta = \mathcal{O}$	☐ $\Delta = \omega$	☐ $\Delta = \Omega$	☐ Unvergleichbar

(a) $f_5 \in \Delta(f_2)$	☒ $\Delta = o$	☒ $\Delta = \mathcal{O}$	☐ $\Delta = \omega$	☐ $\Delta = \Omega$	☐ Unvergleichbar
(b) $f_6 \in \Delta(f_7)$	☐ $\Delta = o$	☐ $\Delta = \mathcal{O}$	☒ $\Delta = \omega$	☒ $\Delta = \Omega$	☐ Unvergleichbar
(c) $f_3 \in \Delta(f_8)$	☐ $\Delta = o$	☐ $\Delta = \mathcal{O}$	☒ $\Delta = \omega$	☒ $\Delta = \Omega$	☐ Unvergleichbar
(d) $f_2 \in \Delta(f_4)$	☐ $\Delta = o$	☒ $\Delta = \mathcal{O}$	☐ $\Delta = \omega$	☒ $\Delta = \Omega$	☐ Unvergleichbar
(e) $f_5 \in \Delta(f_{10})$	☒ $\Delta = o$	☒ $\Delta = \mathcal{O}$	☐ $\Delta = \omega$	☐ $\Delta = \Omega$	☐ Unvergleichbar
(f) $f_1 \in \Delta(f_{10})$	☐ $\Delta = o$	☐ $\Delta = \mathcal{O}$	☐ $\Delta = \omega$	☐ $\Delta = \Omega$	☒ Unvergleichbar
(g) $f_1 \in \Delta(f_9)$	☐ $\Delta = o$	☒ $\Delta = \mathcal{O}$	☐ $\Delta = \omega$	☒ $\Delta = \Omega$	☐ Unvergleichbar

(a)

$$\lim_{n\to\infty} \frac{1337}{\log_5(n^2)} = 0 \quad\Rightarrow\quad 1337 \in o(\log_5(n^2)) \quad\Rightarrow\quad 1337 \in \mathcal{O}(\log_5(n^2))$$

(b)

$$\lim_{n\to\infty} \frac{\left(\ln(\sqrt{n})\right)^2}{\ln\left(\ln(\sqrt{n})\right)} = \lim_{n\to\infty} \frac{\left(\frac{1}{2}\ln(n)\right)^2}{\ln\left(\frac{1}{2}\ln(n)\right)} = \lim_{n\to\infty} \frac{\frac{1}{4}\left(\ln(n)\right)^2}{\ln\left(\ln(n)\right) + \ln(\frac{1}{2})}$$

$$\stackrel{l'H}{=} \lim_{n\to\infty} \frac{\frac{1}{2}\ln(n)\,\frac{1}{n}}{\frac{1}{\ln(n)}\,\frac{1}{n}} = \lim_{n\to\infty} \frac{\left(\ln(n)\right)^2}{2} = \infty$$

$$\Rightarrow \left(\ln(\sqrt{n})\right)^2 \in \omega(\ln\left(\ln(\sqrt{n})\right)) \quad\Rightarrow \left(\ln(\sqrt{n})\right)^2 \in \Omega(\ln\left(\ln(\sqrt{n})\right))$$

(c)

$$\lim_{n\to\infty} \frac{5^n}{3^{n+1}} = \lim_{n\to\infty} \frac{5^n}{3\cdot 3^n} = \lim_{n\to\infty} \frac{1}{3}\left(\frac{5}{3}\right)^n = \infty$$

$$\Rightarrow \quad 5^n \in \omega(3^{n+1}) \quad\Rightarrow\quad 5^n \in \Omega(3^{n+1})$$

(d)

$$\lim_{n\to\infty} \frac{\log_5(n^2)}{3\,\log_6(n)} = \lim_{n\to\infty} \frac{2\,\log_5(n)}{3\,\log_6(n)} = \lim_{n\to\infty} \frac{2\,\frac{\log(n)}{\log(5)}}{3\,\frac{\log(n)}{\log(6)}} = \lim_{n\to\infty} \frac{2\,\log(6)}{3\,\log(5)} = \frac{\log(36)}{\log(125)}$$

$$\Rightarrow \quad 0 < \lim_{n\to\infty} \frac{\log_5(n^2)}{3\,\log_6(n)} = \frac{\log(36)}{\log(125)} < \infty \quad\Rightarrow\quad \log_5(n^2) \in \Theta(3\,\log_6(n))$$

(e)

$$\lim_{n\to\infty} \frac{1337}{15\,n + (\sin(n\cdot\pi))^2} = \lim_{n\to\infty} \frac{1337}{15\,n + 0} = 0$$

$$\Rightarrow \quad 1337 \in o(15\,n + (\sin(n\cdot\pi))^2) \quad\Rightarrow\quad 1337 \in \mathcal{O}(15\,n + (\sin(n\cdot\pi))^2)$$

(f) Fall 1: Sei $n = 2k$ mit $k \in \mathbb{N}$.

$$\lim_{n\to\infty} \frac{n^{1+\cos(n\cdot\pi)}}{15\,n + (\sin(n\cdot\pi))^2} = \lim_{n\to\infty} \frac{n^{1+1}}{15\,n + 0} = \lim_{n\to\infty} \frac{n^2}{15\,n} = \lim_{n\to\infty} \frac{n}{15} = \infty$$

$$\Rightarrow n^{1+\cos(n\cdot\pi)} \in \omega(15\,n + (\sin(n\cdot\pi))^2)$$

Fall 2: Sei $n = 2k+1$ mit $k \in \mathbb{N}$.

$$\lim_{n\to\infty} \frac{n^{1+\cos(n\cdot\pi)}}{15\,n + (\sin(n\cdot\pi))^2} = \lim_{n\to\infty} \frac{n^{1+(-1)}}{15\,n + 0} = \lim_{n\to\infty} \frac{n^0}{15\,n} = \lim_{n\to\infty} \frac{1}{15\,n} = 0$$

$$\Rightarrow\ n^{1+\cos(n\cdot\pi)} \in o(15\,n + (\sin(n\cdot\pi))^2)$$

Für gerade n gilt $n^{1+\cos(n\cdot\pi)} \in \omega(15\,n + (\sin(n\cdot\pi))^2)$, für ungerade n gilt jedoch $n^{1+\cos(n\cdot\pi)} \in o(15\,n + (\sin(n\cdot\pi))^2)$. Somit sind die beiden Funktionen unvergleichbar.

(g) Fall 1: Sei $n = 2k$ mit $k \in \mathbb{N}$.

$$\lim_{n\to\infty} \frac{n^{1+\cos(n\cdot\pi)}}{1 + (1+(-1)^n)\,n^2} = \lim_{n\to\infty} \frac{n^{1+1}}{1 + (1+1)\,n^2} = \lim_{n\to\infty} \frac{n^2}{1 + 2\,n^2} = \frac{1}{2}$$

$$\Rightarrow\quad 0 < \lim_{n\to\infty} \frac{n^{1+\cos(n\cdot\pi)}}{1 + (1+(-1)^n)\,n^2} = \frac{1}{2} < \infty \quad\Rightarrow\quad n^{1+\cos(n\cdot\pi)} \in \Theta(1 + (1+(-1)^n)\,n^2)$$

Fall 2: Sei $n = 2k+1$ mit $k \in \mathbb{N}$.

$$\lim_{n\to\infty} \frac{n^{1+\cos(n\cdot\pi)}}{1 + (1+(-1)^n)\,n^2} = \lim_{n\to\infty} \frac{n^{1+(-1)}}{1 + (1+(-1))\,n^2} = \lim_{n\to\infty} \frac{n^0}{1 + 0\,n^2} = 1$$

$$\Rightarrow\quad 0 < \lim_{n\to\infty} \frac{n^{1+\cos(n\cdot\pi)}}{1 + (1+(-1)^n)\,n^2} = 1 < \infty \quad\Rightarrow\quad n^{1+\cos(n\cdot\pi)} \in \Theta(1 + (1+(-1)^n)\,n^2)$$

Für gerade und ungerade n gilt somit stets $n^{1+\cos(n\cdot\pi)} \in \Theta(1 + (1+(-1)^n)\,n^2)$.

Aufgabe 5 (Dynamische Arrays)
Führen Sie folgende Operationen auf einem leeren dynamischen Array (Arraygröße zu Beginn: w=1) aus: pushBack(1), pushBack(2), pushBack(3), pushBack(4), pushBack(5), popBack(), popBack(), popBack(), popBack(), popBack()
Verwenden Sie einmal die Parameter $\alpha = 4, \beta = 2$ und einmal $\alpha = 5, \beta = 3$.

Mit Parametern $\alpha = 4, \beta = 2$:

leeres Array zu Beginn: `[ ]`
pushBack(1): `[1]`
pushBack(2): `[1| ]` `[1|2]`
pushBack(3): `[1|2| | ]` `[1|2|3| ]`
pushBack(4): `[1|2|3|4]`
pushBack(5): `[1|2|3|4| | | | ]` `[1|2|3|4|5| | | ]`
popBack(): `[1|2|3|4| | | | ]`
popBack(): `[1|2|3| | | | | ]`
popBack(): `[1|2| | | | | | ]` `[1|2| | ]`
popBack(): `[1| | | ]` `[1| ]`
popBack(): `[ | ]`

Mit Parametern $\alpha = 5, \beta = 3$:

leeres Array zu Beginn: `[ ]`
pushBack(1): `[1]`
pushBack(2): `[1| | ]` `[1|2| ]`
pushBack(3): `[1|2|3]`
pushBack(4): `[1|2|3| | | | | | ]` `[1|2|3|4| | | | | ]`
pushBack(5): `[1|2|3|4|5| | | | ]`
popBack(): `[1|2|3|4| | | | | ]`
popBack(): `[1|2|3| | | | | | ]`
popBack(): `[1|2| | | | | | | ]`
popBack(): `[1| | | | | | | | ]` `[1| | ]`
popBack(): `[ | | ]`

Aufgabe 6 (Amortisierte Analyse)
Gegeben sei ein anfangs leerer Stack S, der die folgenden Operationen unterstützt:

1. **push(x):** legt Element x auf S
2. **pop():** entfernt oberstes Element von S
3. **multipop(k):** entfernt mit k-maligem **pop()** die obersten k Elemente aus S; falls S weniger als k Elemente enthält, werden alle Elemente aus S entfernt

a) Wir betrachten nun Folgen von m hintereinander auszuführenden Operationen. Die Operationen **push(x)** und **pop()** haben offensichtlich konstante Lauzeit. Bestimmen Sie die Worst Case Laufzeit von **multipop(k)** in Abhängigkeit von m. Geben Sie ausgehend von dieser Laufzeit zusätzlich eine (pessimistische) obere Schranke für die Laufzeit der gesamten Operationsfolge in Abhängigkeit von m an.

b) Zeigen Sie mit Hilfe der Kontomethode, dass die tatsächliche Laufzeit der Operationsfolge in $\mathcal{O}(m)$ ist.

a) Der Worst Case für **multipop(k)** sieht folgendermaßen aus: Es wurden mit den ersten $m-1$ Operationen $m-1$ Elemente mit **push(x)** auf den Stack abgelegt. Diese werden nun mit **multipop(m-1)** allesamt entfernt, was zu einer Worst Case Laufzeit von $\mathcal{O}(m)$ ((m-1)-maliges Ausführen von **pop()**) für eine einzelne multipop Operation führt. Da insgesamt bis zu m solcher Aufrufe von multipop möglich sind, erhalten wir als pessimistische Abschätzung für die gesamte Operationsfolge $m \cdot \mathcal{O}(m) = \mathcal{O}(m^2)$.

b) Idee: Jedes Element, das mit push(x) auf den Stack kommt, zahlt ein Token auf das Konto ein. Dieses Token wird beim Entfernen des Elements durch pop() oder multipop(k) wieder vom Konto abgehoben. So ist auch unmittelbar sichergestellt, dass der Kontostand nie negativ werden kann.

Überprüfen wir nun, ob wir mit diesem Ansatz auch die gewünschten amortisierten Laufzeiten erhalten (l bezeichne wie in den Übungen die tatsächliche Laufzeit der Operation, Δ die Kontobewegung):
$\alpha(push) = l(push) + \Delta(push) = 1 + 1 = 2 \in \mathcal{O}(1)$
$\alpha(pop) = l(pop) + \Delta(pop) = 1 - 1 = 0 \in \mathcal{O}(1)$
$\alpha(multipop) = l(multipop) + \Delta(multipop) = \min(k, |S|) - \min(k, |S|) = 0 \in \mathcal{O}(1)$

Da alle Operationen eine amortisierte Laufzeit in $\mathcal{O}(1)$ haben, ergibt sich eine amortisierte Laufzeit für die gesamte Folge in $m \cdot \mathcal{O}(1) = \mathcal{O}(m)$. Die amortisierte Laufzeit einer Operationsfolge ist eine obere Schranke für die tatsächliche Laufzeit der Folge. Deshalb muss auch die tatsächliche Laufzeit in $\mathcal{O}(m)$ sein. Wir sehen also, dass die amortisierte Analyse eine sehr viel bessere asymptotische Anschätzung liefert als die pessimistische Abschätzung aus Teilaufgabe a).

Aufgabe 7 (Worst Case Analyse)
Das folgende Stück Code berechnet die LR-Zerlegung einer Matrix A. (Anm.: Eine LR-Zerlegung kann dazu verwendet werden, die Lösung eines linearen Gleichungssystems $Ax = b$ für verschiedene rechte Seiten b exakt und effizient zu berechnen.) Bestimmen Sie die asymptotische Worst Case Laufzeit des Algorithmus. Anm.: $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

Algorithm 1: LR-Zerlegung

```
   Input: reguläre Matrix A der Dimension n × n
   Output: Matrix A nach durchgeführter LR-Zerlegung
 1 for i = 1 to n do
 2     for j = i to n do
 3         for k = 1 to i-1 do
 4             A(i,j) −= A(i,k) · A(k,j)
 5         end
 6     end
 7     for j = i+1 to n do
 8         for k = 1 to i-1 do
 9             A(j,i) −= A(j,k) · A(k,i)
10         end
11         A(j,i) /= A(i,i)
12     end
13 end
```

Einmaliges Ausführen der Zeilen 4, 9 und 11 benötigt offensichtlich konstante Laufzeit (konstante Anzahl an Elementaroperationen). Da wir nur an der asymptotischen Laufzeit des Algorithmus interessiert sind, konstante Faktoren also im Laufe der Berechnung sowieso vernachlässigen werden, dürfen wir für diese drei Zeilen gleich Kosten 1 veranschlagen, was die Rechnung etwas vereinfacht.
Die Berechnung gestaltet sich wie in der Vorlesung vorgestellt, d.h. aus jeder for-Schleife wird eine Summe mit entsprechendem Laufindex, mit der alle Kosten aufsummiert werden. Mit diesem Vorgehen erhalten wir folgende Worst Case Laufzeit:

$$
\begin{aligned}
T(n) &= \sum_{i=1}^{n} \Big(\sum_{j=i}^{n} \sum_{k=1}^{i-1} 1 + \sum_{j=i+1}^{n} \Big(1 + \sum_{k=1}^{i-1} 1\Big)\Big) \\
&= \sum_{i=1}^{n} \Big(\sum_{j=i}^{n} (i-1) + \sum_{j=i+1}^{n} i \Big) \\
&= \sum_{i=1}^{n} \Big((i-1) \sum_{j=i}^{n} 1 + i \sum_{j=i+1}^{n} 1 \Big) \\
&= \sum_{i=1}^{n} \Big((i-1)(n-i+1) + i(n-i) \Big) \\
&= \sum_{i=1}^{n} (-2i^2 + (2n+2)i - n - 1) \\
&= -2 \sum_{i=1}^{n} i^2 + (2n+2) \sum_{i=1}^{n} i - n \sum_{i=1}^{n} 1 - \sum_{i=1}^{n} 1 \\
&= -2 \cdot \frac{n(n+1)(2n+1)}{6} + (2n+2) \frac{n(n+1)}{2} - n^2 - n \\
&= -\frac{2}{3} n^3 + n^3 + \mathcal{O}(n^2) = \frac{1}{3} n^3 + \mathcal{O}(n^2) = \mathcal{O}(n^3)
\end{aligned}
$$

Aufgabe 8 (Average Case Analyse)
Wir betrachten in dieser Aufgabe einen Algorithmus, der ein Element x in einem Array a sucht und den Index zurückgibt, an dem sich x befindet (Indizes beginnend bei 0). Wir treffen dazu folgende vereinfachende Grundannahmen: Als Eingabe seien für das Array a nur n-Permutationen zulässig, also Arrays, die die Zahlen 1 bis n in beliebiger Reihenfolge enthalten. Zudem sei $x \in [n]$ und alle Eingaben gleich wahrscheinlich. Bestimmen Sie die exakte Average Case Laufzeit des Algorithmus. Verwenden Sie das in der Vorlesung besprochene Kostenmodell.

```
Function Index(a, x)
  Input: Array a bestehend aus den Integern 1 bis n in beliebiger Reihenfolge; Integer
         x ∈ [n]
  Output: Index i ∈ {0, ..., n − 1}, an dessen Stelle sich x befindet
1 i := 0;
2 while i < n do
3 |  if x = a[i] then
4 |  |  break;
5 |  end
6 |  i++;
7 end
8 return i;
```

Als erstes bestimmen wir die Zeilen, die unabhängig von der Eingabe in jedem Fall ausgeführt werden müssen. Das sind in diesem Fall die Zeilen 1 und 8, also 2 Operationen. Wir betrachten nun ein beliebiges, aber festes Eingabearray a und bestimmen für dieses a den Mittelwert der benötigten Operationen für alle möglichen Eingaben x. Die Anzahl der Operationen hängt vom Index ab, an dem sich x befindet. Es fällt zudem auf, dass jeder Durchlauf der while-Schleife exakt 3 Operationen benötigt: Entweder finden wir das Element und müssen die Zeilen 2, 3 und 4 ausführen oder wir finden es nicht und müssen die Zeilen 2, 3 und 6 ausführen. Wenn x also gleich an Index 0 steht, benötigen wir genau einen Durchlauf der while-Schleife, also 3 zusätzliche Operationen (insgesamt also 2+3 = 5 Operationen), wenn x an Index 1 steht $3 \cdot 2$ zusätzliche Operationen usw. Also allgemein: Steht x an Position i, so benötigen wir $2 + 3 \cdot i$ elementare Operationen. Da alle x als Eingabe gleich wahrscheinlich sind, können wir den Mittelwert der benötigten Operationen folgendermaßen berechnen (Aufaddieren aller Möglichkeiten und Teilen durch die Anzahl aller möglichen Eingaben für x, also durch n):

$$\frac{\sum_{i=1}^{n}(2+3i)}{n} = \frac{2n}{n} + \frac{\sum_{i=1}^{n} 3i}{n} = 2 + \frac{3}{n}\sum_{i=1}^{n} i = 2 + \frac{3}{n} \cdot \frac{n(n+1)}{2} = 1.5n + 3.5 \qquad (1)$$

Da wir immer diese Average Case Laufzeit von $1.5n + 3.5$ Operationen erhalten, egal wie wir unser Array a zu Beginn wählen, ist dies auch gleichzeitig schon die zu bestimmenden Average Case Laufzeit für alle möglichen Eingaben.

Aufgabe 9 (Vereinfachtes Master Theorem)
Schätzen Sie folgende rekursiv definierten Funktionen $T(n)$ mit Hilfe des vereinfachten Master Theorems geeignet ab. $a, c > 0$ seien Konstanten und $T(1) = a$ für alle Teilaufgaben. Lösen Sie Teilaufgabe a) zudem mit der Methode des iterativen Einsetzens. Hierbei dürfen Sie davon ausgehen, dass n eine Zehnerpotenz ist, also dass ein $m \in \mathbb{N}$ existiert mit $n = 10^m$.

a) $T(n) = 10\,T(n/10) + c\,n$

b) $T(n) = 4\,T(n/5) + 200\,n$

c) $T(n) = 4\,T(n/2) + 0.1\,n$

a) Mit Master-Theorem:
$d = b = 10 \Rightarrow T(n) \in \Theta(n \log n)$

Mit iterativem Einsetzen:
$T(n) = 10T(\frac{n}{10}) + cn = 10(10T(\frac{n}{100}) + c \cdot \frac{n}{10}) + cn = 100T(\frac{n}{100}) + 2cn = ... = 10^m T(\frac{n}{10^m}) + m \cdot cn = n \cdot a + log_{10} n \cdot cn \in \Theta(n \log n)$
In der letzten Zeile der Gleichung wird verwendet, dass die Rekursion bei $T(1) = a$ terminiert, zudem dass $n = 10^m \Leftrightarrow m = log_{10} n$.

b) $d = 4$, $b = 5$. $d < b \Rightarrow T(n) \in \Theta(n)$

c) $d = 4$, $b = 2$. $d > b \Rightarrow T(n) \in \Theta(n^{\log_2 4}) = \Theta(n^2)$

Aufgabe 10 (Analyse rekursiver Algorithmen)
Gegeben seien die folgenden beiden Divide-and-Conquer-Algorithmen zur Multiplikation zweier Integer in Binärdarstellung. Wir nehmen der Einfachheit halber an, dass beide Integer aus n Bits bestehen und n eine Zweierpotenz ist (mit einigen Abwandlungen kann man sich einen Algorithmus für den allgemeinen Fall herleiten). Bestimmen Sie jeweils die asymptotische Laufzeit, indem Sie eine geeignete Rekursionsgleichung für die Laufzeit $T(n)$ aufstellen und diese anschließend mit dem (vereinfachten) Master Theorem abschätzen.

Function Multiply1(x,y)

Input: positive ganze Zahlen x und y in Binärdarstellung mit je $n = 2^m$ Bits
Output: Ergebnis des Produkts $x \cdot y$

```
1  n := |x|;                                  // Anzahl der Ziffern von x bzw. y
2  if n = 1 then
3  |  return x · y;                           // Multiplikation der Bits, wie logisches AND
4  end
5  x_L := die linken n/2 Bits von x;
6  x_R := die rechten n/2 Bits von x;
7  y_L := die linken n/2 Bits von y;
8  y_R := die rechten n/2 Bits von y;
9  M_1 := Multiply1(x_L, y_L);
10 M_2 := Multiply1(x_R, y_R);
11 M_3 := Multiply1(x_L, y_R);
12 M_4 := Multiply1(x_R, y_L);
13 return M_1 · 2^n + (M_3 + M_4) · 2^(n/2) + M_2;
```

Function Multiply2(x,y)

Input: positive ganze Zahlen x und y in Binärdarstellung mit je $n = 2^m$ Bits
Output: Ergebnis des Produkts $x \cdot y$

```
1  n := |x|;                                  // Anzahl der Ziffern von x bzw. y
2  if n = 1 then
3  |  return x · y;                           // Multiplikation der Bits, wie logisches AND
4  end
5  x_L := die linken n/2 Bits von x;
6  x_R := die rechten n/2 Bits von x;
7  y_L := die linken n/2 Bits von y;
8  y_R := die rechten n/2 Bits von y;
9  M_1 := Multiply2(x_L, y_L);
10 M_2 := Multiply2(x_R, y_R);
11 M_3 := Multiply2(x_L + x_R, y_L + y_R);
12 return M_1 · 2^n + (M_3 − M_1 − M_2) · 2^(n/2) + M_2;
```

Analyse von Algorithmus 1:
Falls $n = 1$, müssen nur die Zeilen 1-4 durchgeführt werden. Diese haben konstante Laufzeit. Für größere n finden 4 rekursive Aufrufe statt, wobei sich die Problemgröße jeweils von n auf $n/2$ Bits verringert. Die restliche Laufzeit setzt sich zusammen aus der Aufteilung der Bits (Zeilen 5-8), Binäradditionen und Bit-Shifts (Multiplikation mit Zweierpotenzen) in Zeile 13. Diese Operationen können alle in linearer Laufzeit $\mathcal{O}(n)$ durchgeführt werden. Wir erhalten also die folgende Rekursionsgleichung:
$T(n) = 4 \cdot T(n/2) + c \cdot n$, falls $n > 1$ und $T(1) = a$ für positive Konstanten a und c. Das Master-Theorem liefert $T(n) \in \Theta(n^{\log_2 4}) = \Theta(n^2)$.

Analyse von Algorithmus 2:
Die Argumentation erfolgt analog zu Algorithmus 1, mit dem Unterschied dass nun nur noch 3 statt 4 rekursive Aufrufe nötig sind. Als Rekursionsgleichung erhalten wir also nun $T(n) = 3 \cdot T(n/2) + c \cdot n$ für $n > 1$. Das Master-Theorem liefert in diesem Fall $T(n) \in \Theta(n^{\log_2 3})$ bzw. $T(n) \in \mathcal{O}(n^{1.59})$, was eine beträchtliche (asymptotische) Verringerung der Laufzeit im Vergleich zu Algorithmus 1 bringt.

Aufgabe 11 (Hashing with Chaining)
Gegeben sei eine Hash-Funktion $h(k) = 3\,k \mod 7$. Veranschaulichen Sie Hashing with Chaining für die folgenden Operationen:

Insert 2, 0, 9, 16, 4, 3
Delete 2, 4
Insert 2

insert(2)

0	1	2	3	4	5	6
						2

insert(0)

0	1	2	3	4	5	6
0						2

insert(9)

0	1	2	3	4	5	6
0						2
						9

insert(16)

0	1	2	3	4	5	6
0						2
						9
						16

insert(4)

0	1	2	3	4	5	6
0					4	2
						9
						16

insert(3)

0	1	2	3	4	5	6
0		3			4	2
						9
						16

delete(2)

0	1	2	3	4	5	6
0		3			4	9
						16

delete(4)

0	1	2	3	4	5	6
0		3				9
						16

insert(2)

0	1	2	3	4	5	6
0		3				9
						16
						2

Aufgabe 12 (Hashing with Linear Probing)
Gegeben sei eine Hash-Funktion $h(k) = 2\,k \mod 11$. Veranschaulichen Sie Hashing with Linear Probing für die folgenden Operationen:

Insert 2, 5, 16, 6, 11, 0, 7, 12
delete 6, 5, 11
insert 1, 13
delete 7, 12, 13

insert(2)

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

insert(5)

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

insert(16)

0	1	2	3	4	5	6	7	8	9	10
16				2						5

insert(6)

0	1	2	3	4	5	6	7	8	9	10
16	6			2						5

insert(11)

0	1	2	3	4	5	6	7	8	9	10
16	6	11		2						5

insert(0)

0	1	2	3	4	5	6	7	8	9	10
16	6	11	0	2						5

insert(7)

0	1	2	3	4	5	6	7	8	9	10
16	6	11	0	2	7					5

insert(12)

0	1	2	3	4	5	6	7	8	9	10
16	6	11	0	2	7	12				5

delete(6)

0	1	2	3	4	5	6	7	8	9	10
16	11	0	7	2	12					5

delete(5)

0	1	2	3	4	5	6	7	8	9	10
11	0	12	7	2						16

delete(11)

0	1	2	3	4	5	6	7	8	9	10
0		12	7	2						16

insert(1)

0	1	2	3	4	5	6	7	8	9	10
0		12	7	2	1					16

insert(13)

0	1	2	3	4	5	6	7	8	9	10
0		12	7	2	1	13				16

delete(7)

0	1	2	3	4	5	6	7	8	9	10
0		12	1	2	13					16

delete(12)

0	1	2	3	4	5	6	7	8	9	10
0		1		2	13					16

delete(13)

0	1	2	3	4	5	6	7	8	9	10
0		1		2						16

Aufgabe 13 (Double Hashing)

(a) Gegeben sei die Hashfunktion

$$h(k,i) = \big(h_1(k) + i \cdot h_2(k)\big) \mod 11, \text{ wobei}$$

$$h_1(k) = 4k \mod 11 \qquad \text{und}$$

$$h_2(k) = 1 + (2k \mod (10))$$

Veranschaulichen Sie **Double Hashing** für die folgenden Operationen:

Insert 1, 4, 7, 12, 5, 8, 6

(b) In der Vorlesung wurde keine Implementierung der *find(k)*-Operation für Double-Hashing besprochen. Geben Sie in Prosatext eine informelle Beschreibung an, wie Sie eine solche *find(k)*-Operation für Double-Hashing implementieren würden. Sie dürfen davon ausgehen, dass vor Ausführung Ihrer *find*-Operation keine *delete*-Operation ausgeführt wurde.

(c) Warum ist keine effiziente Implementierung der *delete(k)*-Operation für Double Hashing möglich?

a)

insert(1)

$h(k,i) = h(1,0) = 4$ ✓

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

insert(4)

$h(k,i) = h(4,0) = 5$ ✓

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

insert(7)

$h(k,i) = h(7,0) = 6$ ✓

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

insert(12)

$h(k,i) = h(12,0) = 4$ ↯

$h(k,i) = h(12,1) = 9$ ✓

0	1	2	3	4	5	6	7	8	9	10
				1	4	7			12	

insert(5)

$h(k,i) = h(5,0) = 9$ ↯

$h(k,i) = h(5,1) = 10$ ✓

0	1	2	3	4	5	6	7	8	9	10
				1	4	7			12	5

insert(8)

$h(k,i) = h(8,0) = 10$ ↯

$h(k,i) = h(8,1) = 6$ ↯

$h(k,i) = h(8,2) = 2$ ✓

0	1	2	3	4	5	6	7	8	9	10
		8		1	4	7			12	5

insert(6)

$h(k,i) = h(6,0) = 2$ ↯

$h(k,i) = h(6,1) = 5$ ↯

$h(k,i) = h(6,2) = 8$ ✓

0	1	2	3	4	5	6	7	8	9	10
		8		1	4	7		6	12	5

b) Die $find()$-Operation für Double Hashing kann man beispielsweise wie folgt implementieren: Man berechnet zunächst $h(k,0)$. Sollte an dieser Stelle bereits der gesuchte Key stehen, so kann man das entsprechende Element zurückgeben. Sollte kein Element an dem berechneten Hashwert stehen, so gibt man $false$ zurück und die Suche ist beendet. Sollte andererseits ein Key an der untersuchten Position stehen, der nicht der gewünschte Key ist, so muss man als nächstes die Position h(k, 1) in der Hashtabelle untersuchen und fährt auf gleiche Weise fort. Man inkrementiert also so lange das i in h(i,k) bis man entweder das Element mit dem gesuchten Key findet oder eine leere Position, welche impliziert, dass der gesuchte Key nicht in der Hashtabelle ist.

c) Wenn man ein Element aus der Hashtabelle löschen will, dann muss man wie beim Linearen Sondieren sicherstellen, dass zukünftige *find*- und *insert*-Operationen die richtigen Ergebnisse liefern. Wir betrachten unser Beispiel aus Aufgabenteil (a) und stellen uns vor, dass wir nach Ausführung aller *insert*-Operationen das Element mit dem Key 8 (Hashtabellenposition 2) aus der Hashtabelle löschen wollen. Das Lokalisieren und das eigentliche Löschen können wir ohne Probleme durchführen. Wenn nun aber ein weitere $find(6)$-Operation ausgeführt wird, dann liefert unser Algorithmus wie er in (b) beschrieben wurde $false$ zurück, obwohl das Element mit dem gesuchten Key in der Hashtabelle vorhanden ist.
Wenn wir also ein Element aus der Hashtabelle löschen wollen, dann muss sichergestellt werden, dass alle Elemente, die ursprünglich auch an diese Position gehasht hätten werden wollten, an die entsprechende Stelle verschoben werden. Welche Elemente das betrifft, ist aber leider nicht anders herauszufinden, als dass man jedes Element in der Hashtabelle erneut hasht und guckt, ob der betreffende Hashwert für die betrachtete Position auftritt. Verschiebt man dann ein Element, so muss man das gleiche Prozedere natürlich auch für die Position ausführen, auf der das Element ursprünglich stand.

Aufgabe 14 (Perfektes Hashing)
Konstruieren Sie eine statische, perfekte Hashtabelle für die Keys $x = (x_0, x_1)$:

(1,1)	(2,7)	(4,6)
(5,4)	(6,4)	(6,9)
(8,6)	(8,8)	(8,9)

Ihnen stehen hierfür folgende Hashfunktionen zur Verfügung:

$$h_1 = \sum_{i=0}^{1} a_i\, x_i \mod 13 \quad \text{mit} \quad a = (a_0, a_1) = (1, 2)$$

$$h_2 = \sum_{i=0}^{1} a_i\, x_i \mod 7 \quad \text{mit} \quad a = (a_0, a_1) = (3, 2)$$

$$h_3 = \sum_{i=0}^{1} a_i\, x_i \mod 3 \quad \text{mit} \quad a = (a_0, a_1) = (2, 1)$$

$$h_4 = \sum_{i=0}^{1} a_i\, x_i \mod 3 \quad \text{mit} \quad a = (a_0, a_1) = (2, 3)$$

Wir wiederholen kurz noch einmal die Vorgehensweise beim statischen Perfekten Hashing. Das Ziel des statischen Perfekten Hashings ist die Konstruktion einer Hashtabelle, in der keine Kollisionen bezüglich der betrachteten Paare mehr auftreten. Das Verfahren gliedert sich in 2 Stufen:

Stufe 1: Wir hashen die gegeben n Keys mit einer passend gewählten Hashfunktion h in sogenannte Buckets. Dazu ziehen wir so lange eine Hashfunktion $h : Key \to \{0, \ldots, \lceil \sqrt{2} \cdot c \cdot n \rceil\}$ aus einer c-universellen Familie $\mathcal{H}_{\lceil \sqrt{2} \cdot c \cdot n \rceil}$ von Hashfunktionen, bis h maximal $\sqrt{2}n$ Kollisionen aufweist, d.h.

$$C(h) = |\{(x, y) : \; x, y \; \in Key, \; x \neq y, \; h(x) = h(y)\}| \quad \leq \quad \sqrt{2}\, n$$

erfüllt ist. Die Größe der Hashtabelle, in welche unsere Hashfunktion h hasht, hat also gerade die Größe $m = \lceil \sqrt{2} \cdot c \cdot n \rceil$. Die Hashfamilie $\mathcal{H}_{\lceil \sqrt{2} \cdot c \cdot n \rceil}$ enthält daher genau die Hashfunktionen h, die einen Wertebereich von $\{0, \ldots, \lceil \sqrt{2} \cdot c \cdot n \rceil\}$ aufweisen.

Stufe 2: Nach Ausführung der 1. Stufe kann es sein, dass noch Kollisionen in den Buckets B_ℓ mit $\ell \in \{0, \ldots, \lceil \sqrt{2} \cdot c \cdot n \rceil\}$ vorhanden sind. Diese sollen in der 2. Stufe durch Hashen mit einer weiteren Hashfunktion aufgelöst werden. Dazu betrachtet man nacheinander jeden Bucket B_ℓ und wählt aus einer c-universellen Hashfamilie $\mathcal{H}_{m_\ell}$ eine Hashfunktion h aus, welche die in dem Bucket befindlichen Keys **injektiv** abbildet. Hierbei bezeichnet m_ℓ erneut die Größe der Hashtabelle, in welche die gewählte Hashfunktion h hasht. Für m_ℓ gilt zudem $m_\ell = c\, b_\ell\, (b_\ell - 1) + 1$, wobei b_ℓ die Anzahl der Keys im Bucket B_ℓ bezeichnet.

Wir kehren zur eigentlich Aufgabe zurück und starten direkt mit Stufe 1. Zunächst stellen wir fest, dass alle in dieser Aufgabe gegeben Hashfunktionen der Form

$$\sum_{i=0}^{k} a_i\, x_i \mod p$$

sind, wobei p eine Primzahl ist. Aus der Vorlesung wissen wir, dass Hashfunktionen dieser Bauart aus einer 1-universellen Familie von Hashfunktionen entnommen wurden. Daher gilt für den weiteren Verlauf dieser Aufgabe $c = 1$.
Wir beginnen mit der 1. Stufe und müssen daher zunächst eine Hashfunktion h wählen, welche die Keys in eine Hashtabelle der Größe $m = \lceil \sqrt{2} \cdot c \cdot n \rceil = \lceil \sqrt{2} \cdot 1 \cdot 9 \rceil = 13$ hasht. Als einzige Hashfunktion bietet sich daher h_1 an. Wir berechnen die Hashwerte für die gegebenen Keys:

Key	$h_1(Key)$ / Bucket
(1,1)	3
(2,7)	3
(4,6)	3
(5,4)	0
(6,4)	1
(6,9)	11
(8,6)	7
(8,8)	11
(8,9)	0

Wir erhalten Kollisionen in den Buckets 0, 3 und 11. Dennoch ist die Ungleichung

$$C(h) = |\{(x,y) : \; x,y \; \in Key, \; x \neq y, \; h(x) = h(y)\}| \quad \leq \quad \sqrt{2}\,n$$

erfüllt, denn

$$C(h) = 2 + 6 + 2 = 10 \quad \leq \quad 12 \; \leq \; 12.728 \; = \; \sqrt{2} \cdot 9 \; = \; \sqrt{2}\,n.$$

Wir fahren fort mit Stufe 2 und betrachten zunächst den Bucket B_3. Um die Kollisionen aufzulösen benötigen wir eine Hashfunktion h, die injektiv in eine Hashtabelle der Größe $m_\ell = c\,b_\ell\,(b_\ell - 1) + 1 = 1 \cdot 3 \cdot (3-1) + 1 = 7$ abbildet. Da dies nur die Hashfunktion h_2 erfüllt, wählen wir diese und berechnen erneut die Hashwerte für die Elemente in Bucket B_3.

Key	$h_2(Key)$ / Position in Bucket
(1,1)	5
(2,7)	6
(4,6)	3

Erfreut stellen wir fest, dass h_2 die Keys aus Bucket B_3 injektiv abbildet. Wir können uns also nun den Kollisionen in Buckets B_0 und B_{11} zuwenden.

Zur Behebung der Kollisionen in Buckets B_0 und B_{11} benötigen wir jeweils eine Hashfunktion h, die injektiv in eine Hashtabelle der Größe $m_\ell = c\,b_\ell\,(b_\ell - 1) + 1 = 1 \cdot 2 \cdot (2-1) + 1 = 3$ abbildet. Hierfür kommen sowohl die Hashfunktion h_3 als auch die Hashfunktion h_4 in Frage. Wir rechnen daher für beide Buckets die Hashwerte aus, die sich bei Verwendung von h_3 und h_4 ergeben würden.

	Key	$h_3(Key)$/ Position in Bucket
Bucket 0	(5,4)	2
	(8,9)	1
Bucket 11	(6,9)	0
	(8,8)	0

	Key	$h_4(Key)$/ Position in Bucket
Bucket 0	(5,4)	1
	(8,9)	1
Bucket 11	(6,9)	0
	(8,8)	1

Um die Keys aus Bucket B_0 und B_{11} injektiv abbilden zu können, müssen wir daher zur Auflösung der Kollisionen in Bucket B_0 die Hashfunktion h_3 und zur Auflösung der Kollisionen in B_{11} die Hashfunktion h_4 wählen. Insgesamt ergibt sich also folgende Platzierungen der Elemente:

Key	Bucket	Position in Bucket
(1,1)	3	5
(2,7)	3	6
(4,6)	3	3
(5,4)	0	2
(6,4)	1	0
(6,9)	11	0
(8,6)	7	0
(8,8)	11	1
(8,9)	0	1

Aufgabe 15 (c-universelles Hashing)
Wir betrachten die Schlüsselmenge Key = {bernoulli, cantor, euler, gauß, turing} und eine Hashtabelle der Größe $m = 4$. Zudem seien folgende Hashfunktionen gegeben:

h_0:	bernoulli $\to$ 0	cantor $\to$ 0	euler $\to$ 0	gauß $\to$ 0	turing $\to$ 0
h_1:	bernoulli $\to$ 1	cantor $\to$ 1	euler $\to$ 1	gauß $\to$ 3	turing $\to$ 3
h_2:	bernoulli $\to$ 2	cantor $\to$ 2	euler $\to$ 0	gauß $\to$ 0	turing $\to$ 1
h_3:	bernoulli $\to$ 3	cantor $\to$ 1	euler $\to$ 0	gauß $\to$ 3	turing $\to$ 2
h_4:	bernoulli $\to$ 0	cantor $\to$ 2	euler $\to$ 1	gauß $\to$ 2	turing $\to$ 1
h_5:	bernoulli $\to$ 1	cantor $\to$ 3	euler $\to$ 1	gauß $\to$ 2	turing $\to$ 0
h_6:	bernoulli $\to$ 2	cantor $\to$ 0	euler $\to$ 3	gauß $\to$ 1	turing $\to$ 0
h_7:	bernoulli $\to$ 3	cantor $\to$ 2	euler $\to$ 0	gauß $\to$ 1	turing $\to$ 3
h_8:	bernoulli $\to$ 1	cantor $\to$ 2	euler $\to$ 2	gauß $\to$ 3	turing $\to$ 3

(a) Ist die Familie $\mathcal{H}_1 = \{h_1, h_3, h_4, h_7\}$ 1-universell?

(a) Ist die Familie $\mathcal{H}_2 = \{h_0, h_1, h_4, h_5\}$ 1-universell?

(c) Geben Sie eine Hashfamilie $\mathcal{H}_3 \subseteq \{h_0, h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8\}$ an, welche 1-universell ist und in jedem Fall h_0 enthält.

Wir erinnern uns, dass eine Familie $\mathcal{H}$ von Hashfunktionen auf $\{0, \dots, m-1\}$ genau dann c-universell heißt, wenn für jedes Schlüsselpaar $x, y \in Key$ mit $x \neq y$ gilt:

$$|\{h \in \mathcal{H} : h(x) = h(y)\}| \quad \leq \quad \frac{c}{m}\,|\mathcal{H}|$$

Man muss sich hier klarmachen, dass man in der Menge $|\{h \in \mathcal{H} : h(x) = h(y)\}|$ Hashfunktionen zählt, die bezüglich des gerade betrachteten Paares $x, y \in Key$ eine Kollision verursachen.

Paar	h_0	h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8
bernoulli / cantor	x	x	x						
bernoulli / euler	x	x				x			
bernoulli / gauß	x			x					
bernoulli / turing	x							x	
cantor / euler	x	x							x
cantor / gauß	x				x				
cantor / turing	x						x		
euler / gauß	x		x						
euler / turing	x				x				
gauß / turing	x	x							x

(a) Für alle möglichen Paare $x, y \in Key$ mit $x \neq y$ gibt es in $\mathcal{H}_1$ höchstens eine Hashfunktion, die bezüglich des Paares x, y eine Kollision verursacht. Die Ungleichung ist somit erfüllt, wodurch die Familie von Hashfunktionen $\mathcal{H}_1$ wirklich 1-universell ist.

$$|\{h \in \mathcal{H} : h(x) = h(y)\}| \quad \leq \quad \frac{c}{m}\,|\mathcal{H}| \quad \Rightarrow \quad 1 \quad \leq \quad \frac{1}{4} \cdot 4 \quad = \quad 1 \quad \checkmark$$

(b) Für das Paar bernoulli/euler erhalten wir in h_0, h_1 und in h_5 eine Kollision. Die Ungleichung

$$|\{h \in \mathcal{H} : h(x) = h(y)\}| \quad \leq \quad \frac{c}{m}\,|\mathcal{H}| \quad \Rightarrow \quad 3 \quad \leq \quad \frac{1}{4} \cdot 4 \quad = \quad 1 \quad \lightning$$

ist also nicht erüllt, weswegen die Hashfamilie $\mathcal{H}_2$ nicht 1-universell sein kann.

(c) Die Hashfamilie $\mathcal{H}_3 = \{h_0, h_2, h_3, h_4, h_5, h_6, h_7, h_8\}$ erfüllt offenbar die Bedingungen. Bezüglich jedes Paares $x, y \in Key$ mit $x \neq y$ erhalten wir genau 2 Kollisionen. Die Ungleichung

$$|\{h \in \mathcal{H} : h(x) = h(y)\}| \quad \leq \quad \frac{c}{m}\,|\mathcal{H}| \quad \Rightarrow \quad 2 \quad \leq \quad \frac{1}{4} \cdot 8 \quad = \quad 2 \quad \checkmark$$

wird aber dennoch erfüllt, da eine Hashfamilie der Größe 8 betrachtet wird. Somit ist $\mathcal{H}_3$ wirklich 1-universell. Man beachtet, dass wir eine Hashfamilie konstruiert haben, die eine Hashfunktion beinhaltet, welche alle Keys auf den gleichen Hashwert hasht (also maximal schlecht ist), aber unsere Familie $\mathcal{H}_3$ trotzdem 1-universell bleibt.

Aufgabe 16 (Verschiedene Sortierverfahren)
Sortieren Sie das Array A = [42, 34, 21, 92, 59, 67, 15, 1, 78] jeweils mit Hilfe der folgenden Verfahren:

a) MergeSort

b) QuickSort mit der Implementierung aus der Vorlesung (Element ganz rechts in jedem Teilarray ist das Pivotelement)

c) QuickSort mit folgender Implementierung: Bestimme jeweils das drittgrößte Element des betrachteten Teilarrays und vertausche dieses jeweils mit dem Element, das am weitesten rechts im Teilarray steht. Der Rest des Algorithmus soll wie in der Implementierung der Vorlesung ablaufen (mit dem drittgrößten Element als Pivotelement). Für Teilarrays der Größe 2 verwende das rechte Element als Pivotelement.

d) QuickSort mit folgender Implementierung: Bestimme jeweils den Median (das $\lceil n/2 \rceil$-kleinste Element) des betrachteten Teilarrays und vertausche diesen jeweils mit dem Element, das am weitesten rechts im Teilarray steht. Der Rest des Algorithmus soll wie in der Implementierung der Vorlesung ablaufen (mit dem Median als Pivotelement).

e) HeapSort - Zur Erinnerung: Bei HeapSort wird zunächst mit Hilfe von build(A) ein Binär-Heap konstruiert und anschließend n-mal ein deleteMin ausgeführt. Verwenden Sie die effiziente build-Methode für die Binär-Heaps, so wie sie in den Tutorübungen besprochen wurde.

a) MergeSort

[42 34 21 92 59 67 15 1 78]

[42 34 21 92 59] [67 15 1 78]

[42 34 21] [92 59] [67 15] [1 78]

[42 34] [21] [92] [59] [67] [15] [1] [78]

[42] [34] [21] [92] [59] [67] [15] [1] [78]

[34 42] [21] [92] [59] [67] [15] [1] [78]

[21 34 42] [59 92] [15 67] [1 78]

[21 34 42 59 92] [1 15 67 78]

[1 15 21 34 42 59 67 78 92]

b) QuickSort mit rechtem Element als Pivotelement

42	34	21	92	59	67	15	1	78

Pivotelement 78, swap(92, 1), swap(92, 78)

42	34	21	1	59	67	15	78	92

Pivotelement 15, swap(42, 1), swap(34, 15)

1	15	21	42	59	67	34	78	92

Pivotelement 34, swap(42, 34)

1	15	21	34	59	67	42	78	92

Pivotelement 42, swap(59, 42)

1	15	21	34	42	67	59	78	92

Pivotelement 59, swap(67, 59)

1	15	21	34	42	59	67	78	92

1	15	21	34	42	59	67	78	92

c) QuickSort mit drittgrößtem Element als Pivotelement

42	34	21	92	59	67	15	1	78

Pivotelement 67, swap(67, 78)

42	34	21	92	59	78	15	1	67

swap(92, 1), swap(78, 15)

42	34	21	1	59	15	78	92	67

swap(78, 67)

42	34	21	1	59	15	67	92	78

linkes Teilarray: Pivotelement 34, swap(34, 15)
rechtes Teilarray: Pivotelement 78, swap(92, 78)

[42 | 15 | 21 | 1 | 59 | 34] [67] [78] [92]

swap(42, 1), swap(42, 34)

[1 | 15 | 21] [34] [59 | 42] [67] [78] [92]

linkes Teilarray: Pivotelement 1, swap(1, 21), swap(21, 1)
rechtes Teilarray: Pivotelement 42, swap(59, 42)

[1] [15 | 21] [34] [42] [59] [67] [78] [92]

Pivotelement 21, kein swap

[1] [15] [21] [34] [42] [59] [67] [78] [92]

[1 | 15 | 21 | 34 | 42 | 59 | 67 | 78 | 92]

d) QuickSort mit Median als Pivotelement

[42 | 34 | 21 | 92 | 59 | 67 | 15 | 1 | 78]

Pivotelement: 42, swap(42, 78)

[78 | 34 | 21 | 92 | 59 | 67 | 15 | 1 | 42]

swap(78, 1), swap(92, 15), swap(59, 42)

[1 | 34 | 21 | 15] [42] [67 | 92 | 78 | 59]

linkes Teilarray: Pivotelement 15
rechtes Teilarray: Pivotelement 67, swap(67, 59)

[1 | 34 | 21 | 15] [42] [59 | 92 | 78 | 67]

swap(34, 15), swap(92, 67)

[1] [15] [21 | 34] [42] [59] [67] [78 | 92]

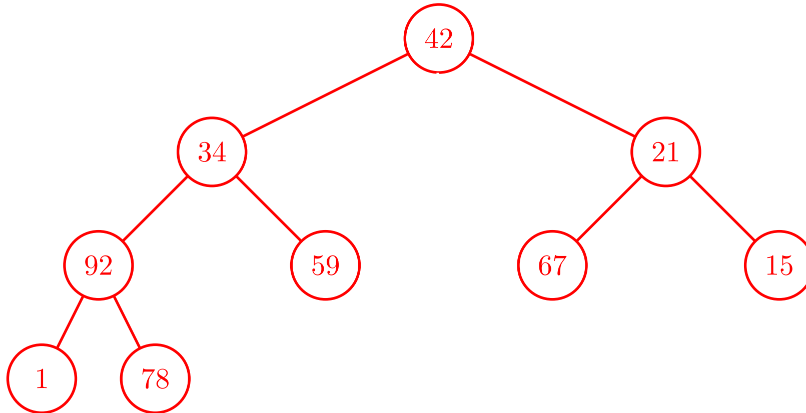
linkes Array: Pivotelement 21, swap(21, 34), swap(34, 21)
rechtes Array: Pivotelement 78, swap(78, 92), swap(92, 78)

[1] [15] [21] [34] [42] [59] [67] [78] [92]

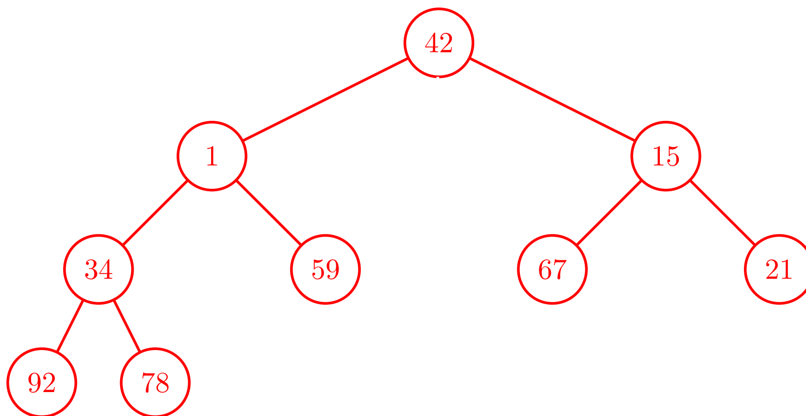
1	15	21	34	42	59	67	78	92

e) HeapSort

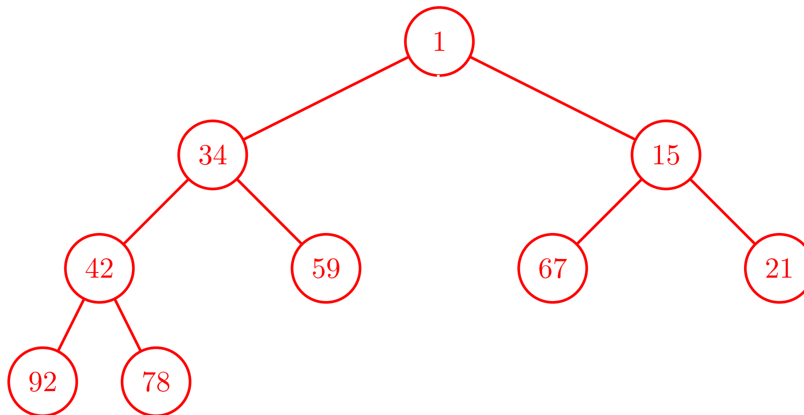
1.Schritt: Konstruktion eines Binär-Heaps mit build(A)



siftDown(92), siftDown(21), siftDown(34)

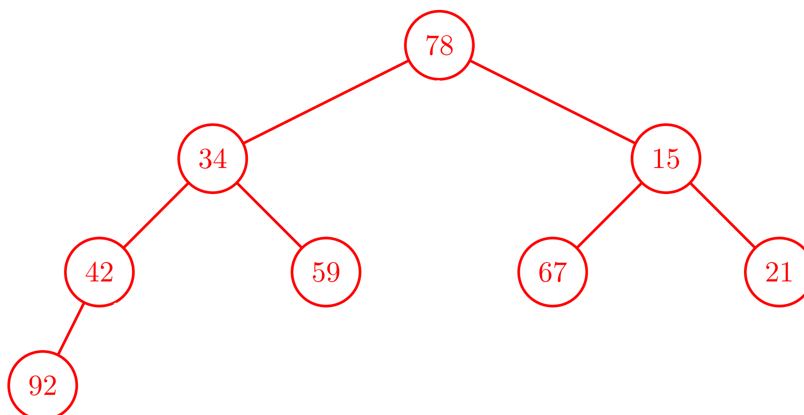


siftDown(42)

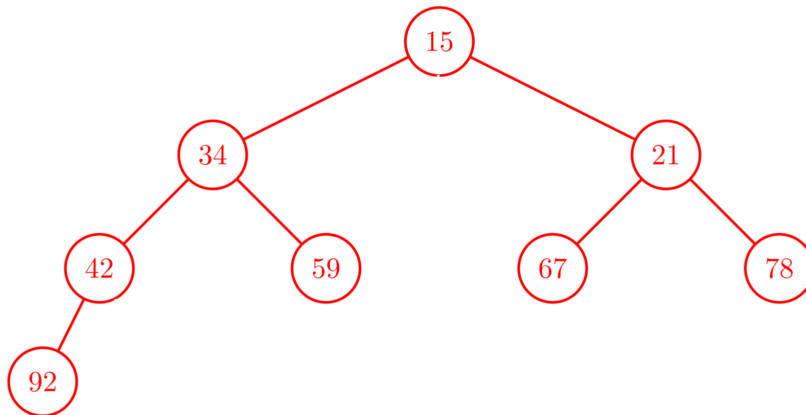


Nun wird auf dem Heap n-mal ein deleteMin ausgeführt. Die erhaltenen Elemente können beispielsweise in ein neues Array A' abgespeichert werden, das am Ende des Verfahrens dann die sortierte Sequenz enthält. [Anm.: Wer speicherschonend arbeiten möchte, kann auch eine in-place-Sortierung implementieren, vgl. dazu die Bemerkungen in den Vorlesungsfolien zu HeapSort.]

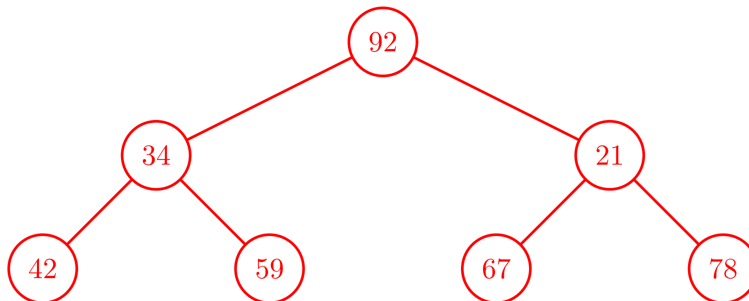
1. deleteMin, A' = [1]



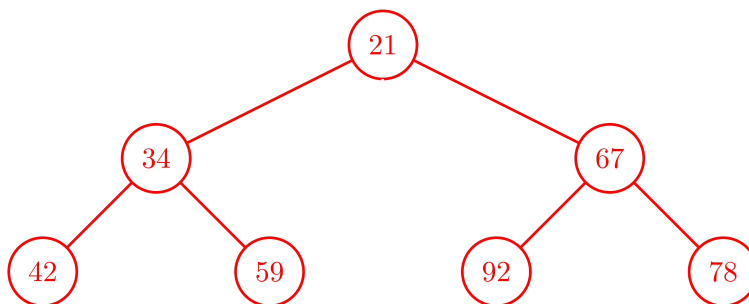
siftDown(78)



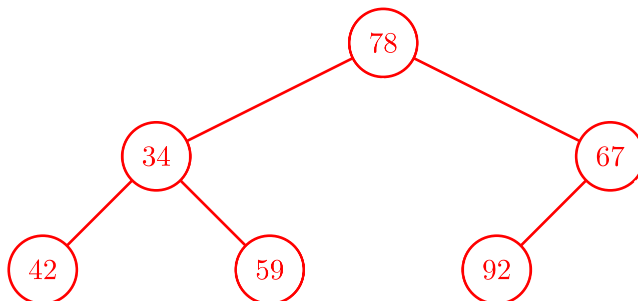
2. deleteMin, A' = [1, 15]



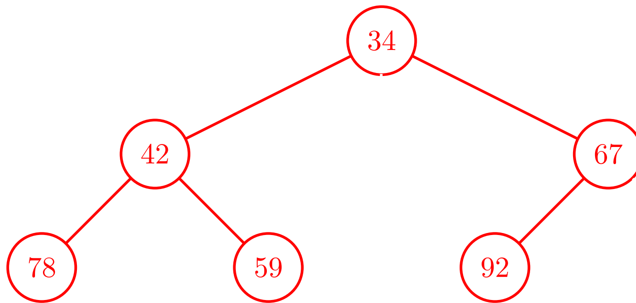
siftDown(92)



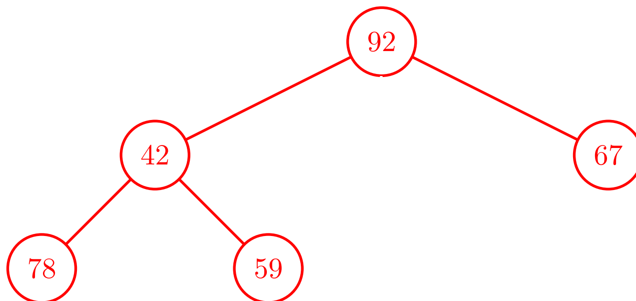
3. deleteMin, A' = [1, 15, 21]



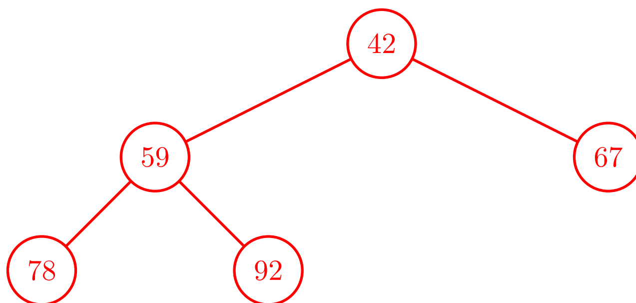
siftDown(78)



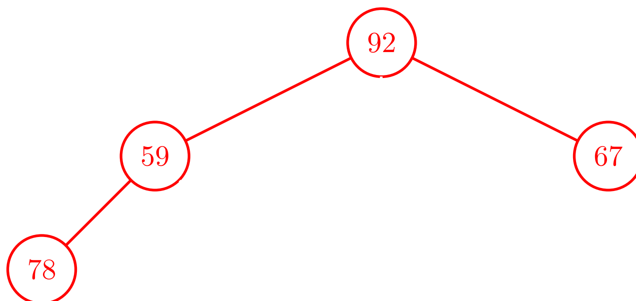
4. deleteMin, A' = [1, 15, 21, 34]



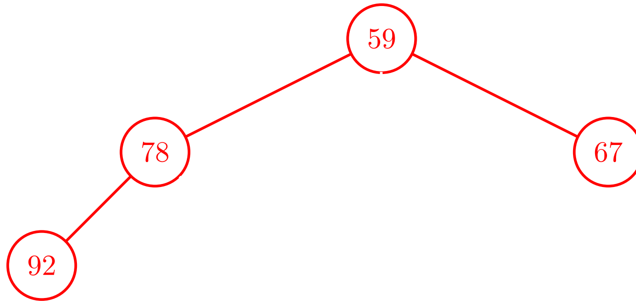
siftDown(92)



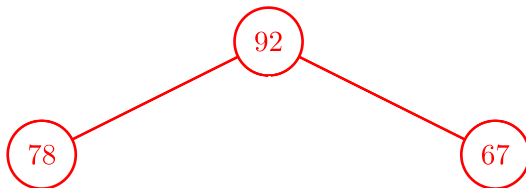
5. deleteMin, A' = [1, 15, 21, 34, 42]



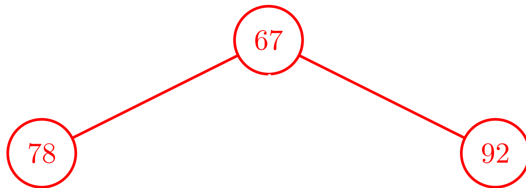
siftDown(92)



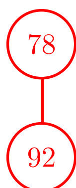
6. deleteMin, A' = [1, 15, 21, 34, 42, 59]



siftDown(92)



7. deleteMin + siftDown(92), A' = [1, 15, 21, 34, 42, 59, 67]



8. und 9. deleteMin, A' = [1, 15, 21, 34, 42, 59, 67, 78, 92]

Aufgabe 17 (RadixSort)
Sortieren Sie das Array A = [4562, 4325, 4050, 7260, 5039, 5700, 756, 1] mit Hilfe des RadixSort Algorithmus.

Zu sortierende Sequenz: 4562, 4325, 4050, 7260, 5039, 5700, 756, 1
Um RadixSort durchführen zu können, müssen die zu sortierenden Zeichenketten gleiche Länge haben. Wir füllen also die Strings 756 und 1 zunächst mit führenden Nullen auf: 4562, 4325, 4050, 7260, 5039, 5700, 0756, 0001

1. Schritt: Sortieren nach dem letzten Zeichen

0	1	2	3	4	5	6	7	8	9
4050	0001	4562			4325	0756		5039	
7260									
5700									

Zwischenstand: 4050, 7260, 5700, 0001, 4562, 4325, 0756, 5039
2. Schritt: Sortieren nach dem vorletzten Zeichen

0	1	2	3	4	5	6	7	8	9
5700		4325	5039		4050	7260			
0001					0756	4562			

Zwischenstand: 5700, 0001, 4325, 5039, 4050, 0756, 7260, 4562
3. Schritt: Sortieren nach dem drittletzten Zeichen

0	1	2	3	4	5	6	7	8	9
0001		7260	4325		4562		5700		
5039							0756		
4050									

Zwischenstand: 0001, 5039, 4050, 7260, 4325, 4562, 5700, 0756
4. Schritt: Sortieren nach dem vordersten Zeichen

0	1	2	3	4	5	6	7	8	9
0001				4050	5039		7260		
0756				4325	5700				
				4562					

Endergebnis: 1, 756, 4050, 4325, 4562, 5039, 5700, 7260

Vorsicht bei RadixSort: Bei Zahlen unterschiedlicher Länge werden **vorne** führende Nullen angehängt, bei unterschiedlich langen Wörtern, die in alphabetischer Ordnung sortiert werden sollen, müssen Leerzeichen am **Ende** des Wortes angehängt werden.

Aufgabe 18 (QuickSelect und Binär-Heaps)
Wir betrachten in dieser Aufgabe zwei verschiedene Möglichkeiten, das k-kleinste Element eines Arrays zu bestimmen.

a) Gegeben sei das Array A = [10, 4, 6, 9, 1, 12, 11, 2, 7, 5, 3, 8]. Bestimmen Sie das drittkleinste Element aus A mit Hilfe des in der Vorlesung vorgestellten QuickSelect-Algorithmus. Verwenden Sie die an QuickSort angelehnte Implementierung. Als Pivotelement soll jeweils das Element mit dem größten Index innerhalb des Teilarrays gewählt werden, also am weitesten rechts stehende Element der betrachteten Teilsequenz.

b) Wandeln Sie das Array nun in einen Binär-Heap um, indem Sie die in den Übungen besprochene Operation build(A) ausführen. Verwenden Sie dazu das unmodifizierte Array A, das zu Beginn gegeben war.

c) In welchen Leveln eines Binär-Heaps kann das k-kleinste Element nur enthalten sein? Beschreiben Sie ausgehend von dieser Beobachtung einen Algorithmus, der das k-kleinste Element des Binär-Heaps in $\mathcal{O}(k^2)$ bestimmt. Begründen Sie kurz, warum Ihr Algorithmus die gewünschte Laufzeit besitzt.

d) Bestimmen Sie nun das drittkleinste Element aus A, indem Sie Ihren in c) entwickelten Algorithmus auf den in b) konstruierten Binär-Heap anwenden.

a) Ausgangssituation:

10	4	6	9	1	12	11	2	7	5	3	8

1. Aufruf von QuickSelect: Pivotelement 8, swap-Operationen: swap(10,3), swap(9,5), swap(12,7), swap(11,2)

3	4	6	5	1	7	2	11	12	9	10	8

Verschieben des Pivotelements mit swap(11,8):

3	4	6	5	1	7	2	8	12	9	10	11

Das gesuchte Element hat Rang 3, das Pivotelement steht an Index 8. Das bedeutet, dass wir Teilarray links des Pivotelements weitersuchen müssen.

3	4	6	5	1	7	2

Neues Pivotelement ist 2, im nächsten Durchlauf müssen wir nur swap(3,1) ausführen:

1	4	6	5	3	7	2

Verschieben des Pivotelements:

1	2	6	5	3	7	4

Rang des gesuchten Elements ist 3, das Pivotelement steht an Index 2. Wir suchen also nun das kleinste Element im Teilarray rechts des Pivot-Elements 2.

6	5	3	7	4

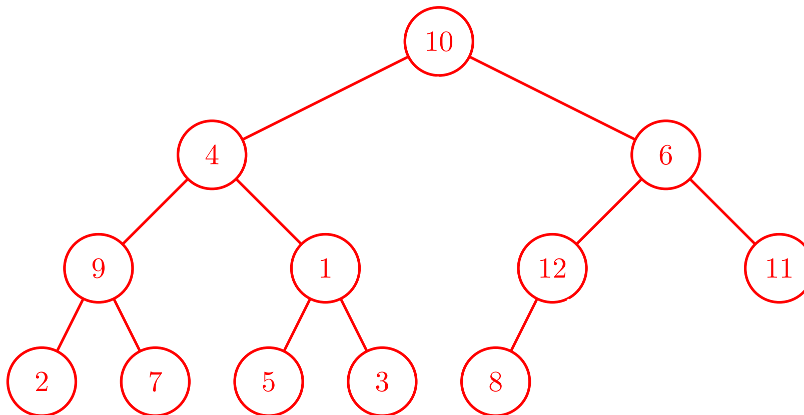
Neues Pivotelement ist 4, im nächsten Durchlauf müssen wir swap(6,3) ausführen und anschließend wieder das Pivotelement mit swap(5,4) an die richtige Position bringen:

3	5	6	7	4

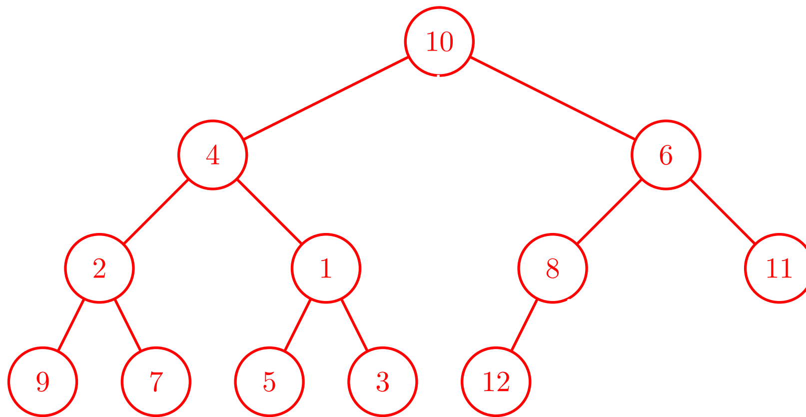
3	4	6	7	5

Das gesuchte Element hat Rang 1 (wir suchen inzwischen das kleinste Element), das Pivotelement ist an Index 2, d.h. wir suchen nun das kleinste Element im linken Teilarray und erhalten damit als Ergebnis 3.

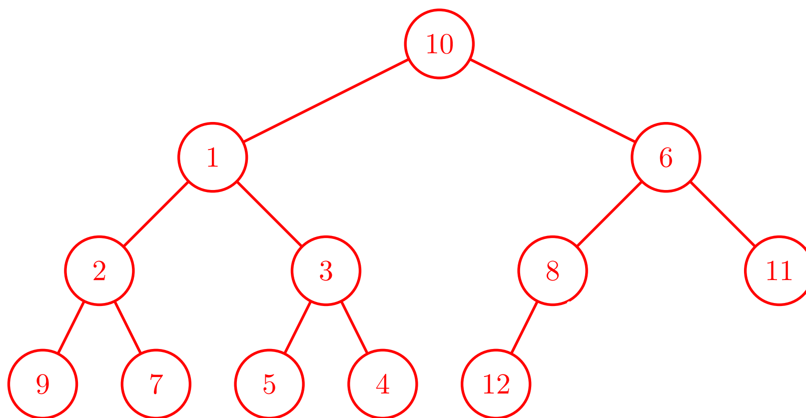
b) Zunächst erstellen wir einen fast vollständigen Binärbaum mit 12 Knoten und tragen die Elemente der Reihe nach ein:



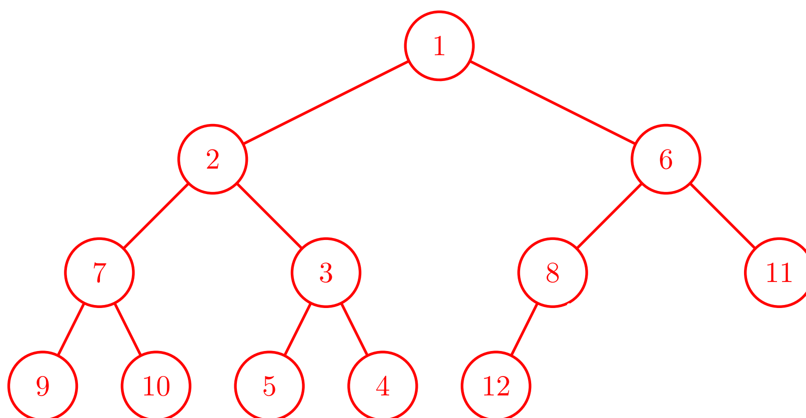
Anschließend muss für alle inneren Knoten ein siftDown ausgeführt werden, und zwar in umgekehrter Reihenfolge: siftDown(12), siftDown(1), ..., siftDown(10). Situation nach siftDown(12), siftDown(1), siftDown(9):



Nach siftDown(6) und siftDown(4):



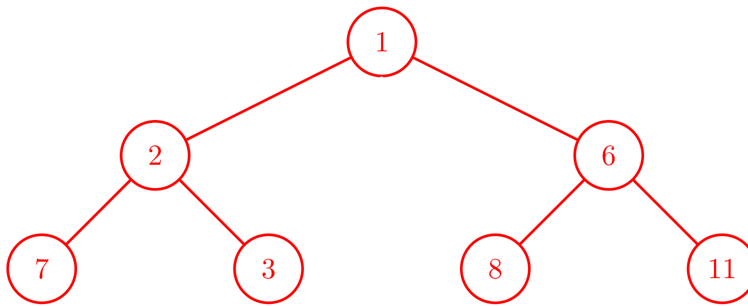
Situation nach siftDown(10) und damit Endergebnis:



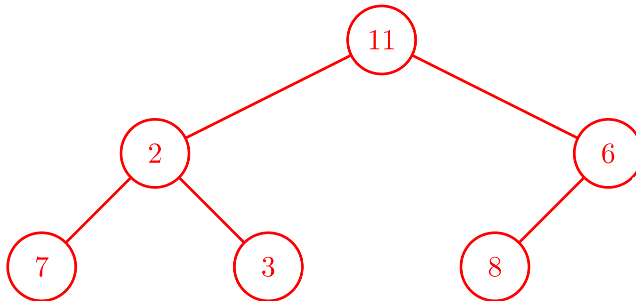
c) Das k-kleinste Element kann sich nur in den obersten k Leveln des Binär-Heaps befinden. Wäre es in Level k+1, dann würden wir aufgrund der zweiten Heap-Invariante (Elternk-

noten haben kleineren Wert als ihre Kindknoten) auf dem Pfad von diesem Element zur Wurzel k kleinere Elemente finden, was im Widerspruch zur Annahme steht. (Analoge Argumentation für höhere Level.)
Aufgrund dieser Tatsache können wir das k-kleinste Element in einem Binär-Heap bestimmen, indem wir nur die oberen k Level des Heaps betrachten und alle anderen Level vernachlässigen. k-maliges Ausführen der deleteMin-Operation liefert uns schließlich das gewünschte Element. Jede der k deleteMin-Operationen ist durch die Laufzeit der siftDown-Operation und damit durch die Tiefe des Heaps, also k, beschränkt. Deshalb erhalten wir eine Gesamtlaufzeit von $k \cdot \mathcal{O}(k) = \mathcal{O}(k^2)$. [Anmerkung: Es gibt einen effizienteren Algorithmus, der das k-kleinste Element sogar in $\mathcal{O}(k \log k)$ bestimmt.]

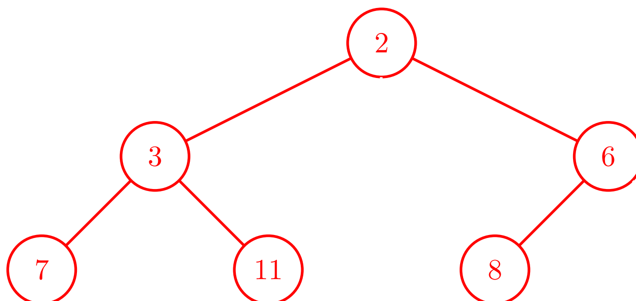
d) Wir reduzieren den in b) erhaltenen Binär-Heap auf die oberen drei Level und führen anschließend dreimal hintereinander ein deleteMin aus.



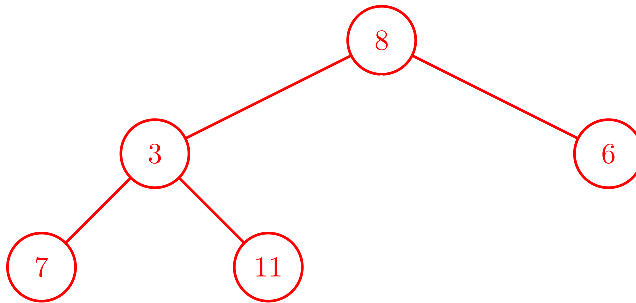
1. deleteMin, 1. Zwischenschritt:



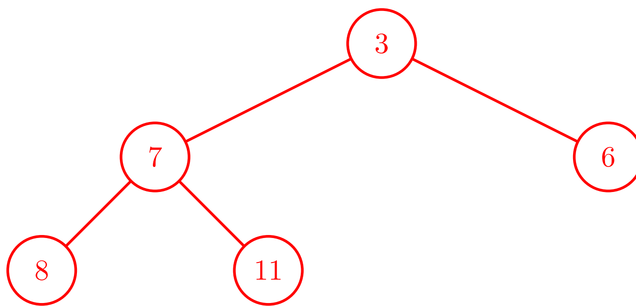
siftDown(11):



2. deleteMin, 1. Zwischenschritt:



siftDown(8):



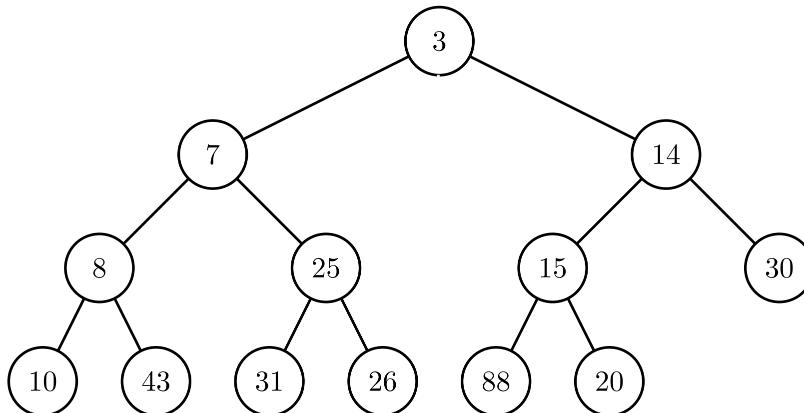
Das dritte deleteMin liefert 3 als Endergebnis.

Aufgabe 19 (Binäre Heaps)

(a) Gegeben Sei eine Arraydarstellung eines Binären Heaps. Zeichnen Sie den dazu enstsprechenden Binären Heap.

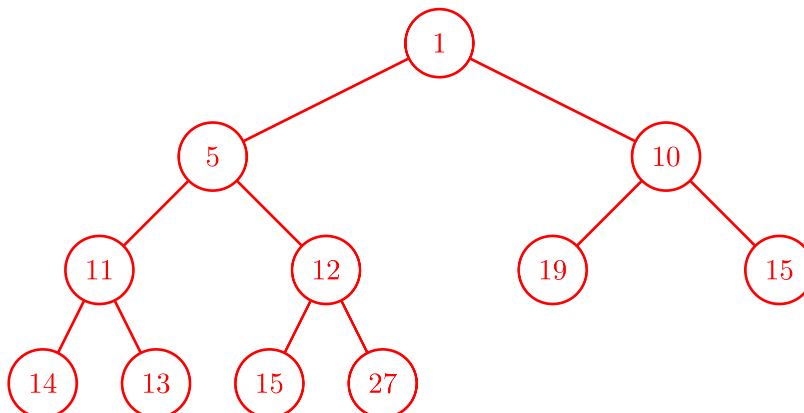
1	5	10	11	12	19	15	14	13	15	27

(b) Gegen sei folgender Binäre Heap. Geben Sie eine passende Arraydarstellung für ihn an.



(c) Wir betrachten den i-ten Einträg einer Arraydarstellung eines Binären Heaps. Geben Sie in Abhängigkeit von i an, an welcher Stelle im Array sich das linke Kind, das rechte Kind und der Vater vom Knoten i befindet.

(a) Der Binäre Heap wird einfach Stufenweise in die Array-Repräsentation gewandelt.



(b)

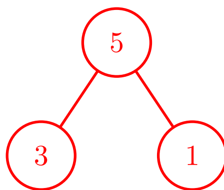
3	7	14	8	25	15	30	10	43	31	26	88	20

(c) Die Kinder von $H[i]$ befinden sich in den Array-Positionen $H[2i+1]$ und $H[2i+2]$. Der Vater Knoten $H[i]$ befindet sich in $H[\lfloor \frac{i-1}{2} \rfloor]$. Sollte man durch diese Rechnungen auf Indizes kommen, die außerhalb des Arrays liegen, so hat der entsprechende Knoten keine Kinder- bzw. keinen Vaterknoten.

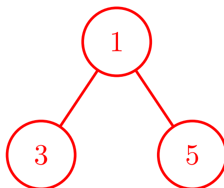
Aufgabe 20 (Binäre Heaps)
Ihr Kommilitone Andi Arbeit behauptet, dass es in der Klausur egal wäre, ob man bei Binären Heaps die effiziente *insert*- Methode oder die naive (n-maliges Inserten) verwendet. Sie können sich noch dunkel an Ihre Tutorübung erinnern, in der Sie ihr Tutor gewarnt hat, dass unter Umständen verschiedene Binäre Heaps entstehen könnten, je nachdem welche *insert*-Methode Sie verwenden. Ihr Kommilitone Andi Arbeit glaubt Ihnen natürlich kein Wort. Können Sie ihn anhand eines (möglichst kleinen Beispiels) von Ihrer Sicht der Dinge überzeugen? Geben Sie zudem die Laufzeiten der effizienten und naiven *build*-Operation an.

Hinweis: Die effiziente build-Operation war wie folgt auszuführen: Füge alle Elemente ohne Beachtung der Priorität nacheinander in den binären Heap ein. Führe für die ersten $\lfloor \frac{n}{2} \rfloor$ Elemente des Heaps in umgekehrter Reihenfolge (also beginnend mit $H[\lfloor \frac{n}{2} \rfloor - 1]$) ein $siftDown(i)$ durch. $H[]$ war hierbei die Array-Repräsentation des Binären Heaps.

Wir betrachten das Beispiel $build(5, 3, 1)$ und verwenden zunächst die effiziente *build*-Methode. Wir fügen also zunächst alle Elemente ohne Beachtung der Priorität nacheinander in einen binären Heap ein:



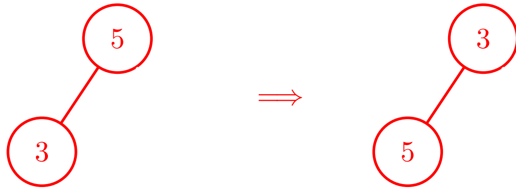
Nun führen wir auf die ersten $\lfloor \frac{n}{2} \rfloor = \lfloor \frac{3}{2} \rfloor = 1$ Elemente - in diesem Fall also nur der 5 - eine $siftDown()$-Operation aus und sind mit der effizienten *build*-Methode fertig.



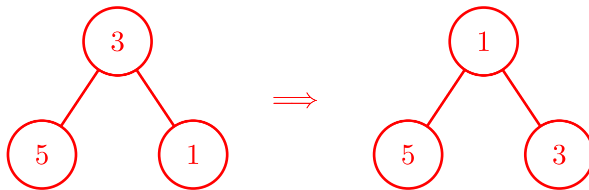
Nun konstruieren wir den Binären Heap mittels der naiven *build*-Methode, das heißt durch n-maliges Inserten der entsprechenden Elemente.

insert(5)

insert(3) mit anschließendem siftUp(3)



insert(1) mit anschließendem siftUp(1)



Wir erhalten also 2 unterschiedliche Binäre Heaps, je nachdem welche *build*()-Operation man verwendet. Andi Arbeit grummelt, aber ist letztendlich überzeugt.

Die Laufzeit der effizienten *build*-Operation liegt in $\mathcal{O}(n)$, während die naive Implementierung Kosten von $\mathcal{O}(n \log(n))$ verursacht.

Aufgabe 21 (Binomial Heaps)

(a) Gegeben sei ein Binomial-Heap mit 222 Elementen. Geben Sie die Ränge der einzelnen Binomial-Bäume an, die in diesem Binomial-Heap enthalten sind.

(b) Der Binomial-Heap aus (a) wird nun durch zeimaligen Aufruf der Merge-Operation zunächst mit einem Binomial-Heap mit 44 Elementen und dann mit einem Binomial-Heap mit 66 Elementen vereinigt. Geben Sie die Ränge der Binomial-Bäume an, die in dem am Ende resultierendem Binomial-Heap enthalten sind.

(a) In jedem Binomial-Heap ist maximal ein Binomial-Baum eines bestimmten Ranges enthalten. Zudem ist die Anzahl der Elemente in einem Binomial-Baum immer genau gleich einer Zweierpotenz. Wir können also die Zahl 222 als Summe von Zweierpotenzen darstellen und erhalten daraus direkt die Ränge von dem Binomial-Bäumen, die in dem Binomial-Heap enthalten sind.

$$\begin{aligned} 222 &= 128 + 94 \\ &= 128 + 64 + 30 \\ &= \dots \\ &= 128 + 64 + 16 + 8 + 4 + 2 \\ &= 2^7 + 2^6 + 2^4 + 2^3 + 2^2 + 2^1 \\ &= 11011110_{(2)} \end{aligned}$$

Ein Binomial-Heap mit 222 Elemeten enthält also Binomial-Bäume mit den Rängen 7, 6, 4, 3, 2 und 1.

(b) Um die Ränge der Binomial-Bäume des resultierendem Binomial-Heaps zu berechnen, gibt es 2 Möglichkeiten. Entweder wandelt man die Zahlen 44 und 64 ins Binärsystem um und addiert die Binärzahlen oder man adiert einfach die Dezimalzahlen und stellt das Ergebnis wieder als Folge von Zweierpotenzen dar.

Variante 1

Analog zu (a) stellen wir die Binomial-Heaps mit 44 bzw. 66 Elementen als Binärzahl dar.

$$\begin{aligned} 44 &= 101100_{(2)} \\ 66 &= 1000010_{(2)} \end{aligned}$$

Eine Mergeoperation entspricht nun genau der Addition der Binärzahlen.

$$\begin{aligned} & 11011110_{(2)} \\ + & \quad 101100_{(2)} \\ + & \quad 1000010_{(2)} \\ = & \quad 101001100_{(2)} \\ = & \quad 2^8 + 2^6 + 2^3 + 2^2 \end{aligned}$$

Variante 2

$$222 + 44 + 66 \;=\; 332 \;=\; 2^8 + 2^6 + 2^3 + 2^2$$

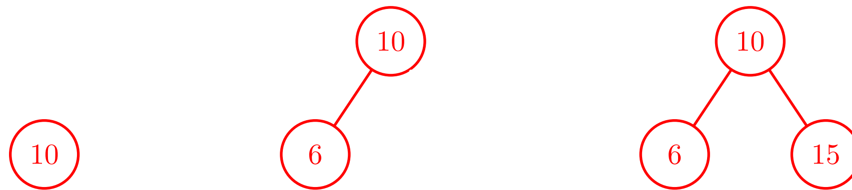
Auf beide Weisen erhalten wir, dass der resultierende Binomial-Heap Binomial-Bäume mit den Rängen 8, 6, 3 und 2 enthält.

Aufgabe 22 (Binäre Suchbäume)

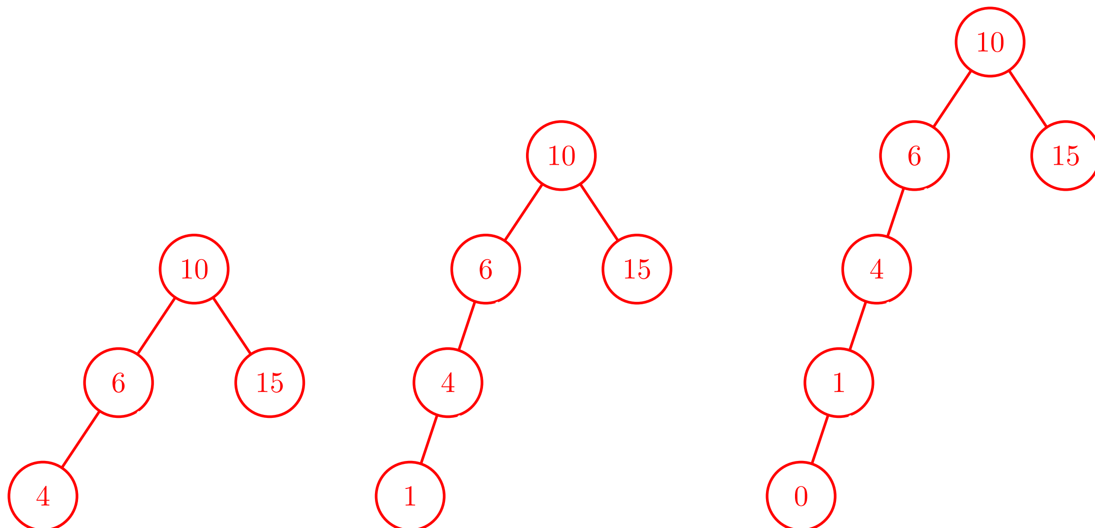
(a) Führen Sie auf einen anfangs leeren Binärbaum folgende Operationen aus.
insert: 10, 6, 15, 4, 1, 0, 5, 9, 8
remove: 1, 9, 10, 6, 5

(b) Was ist ein Nachteil von Binären Suchbäumen gegenüber AVL- und (a,b)-Bäumen?

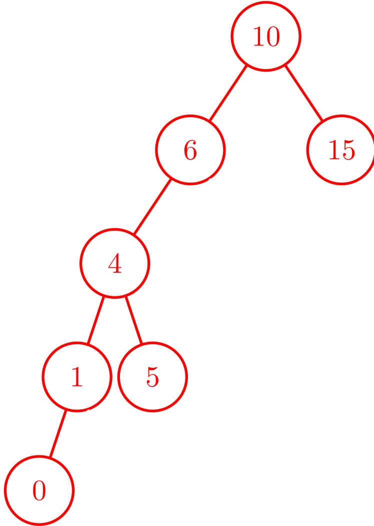
(a) insert(10), insert(6), insert(15)

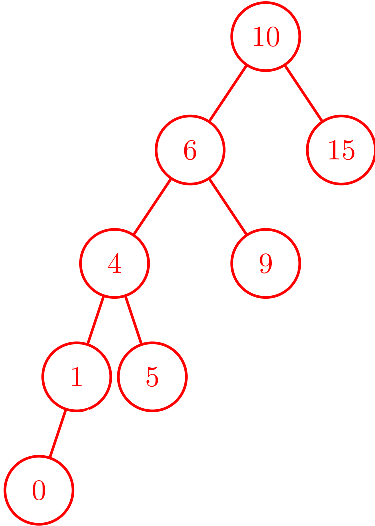


insert(4), insert(1), insert(0)

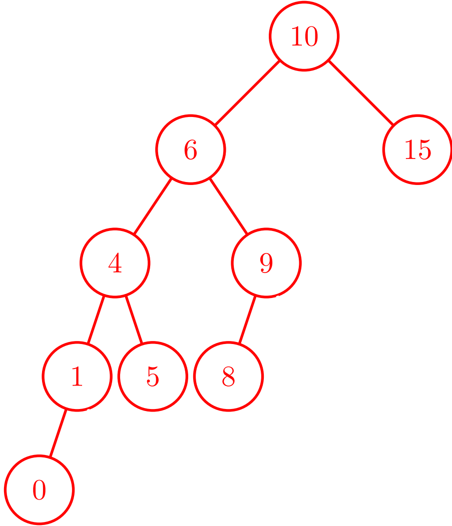


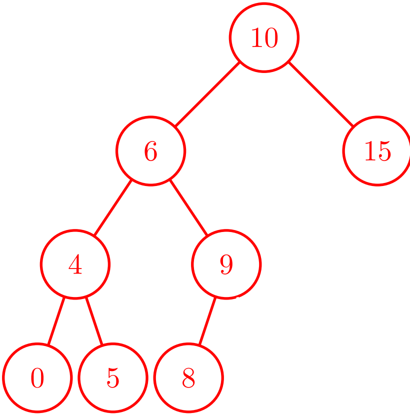
insert(5), insert(9)



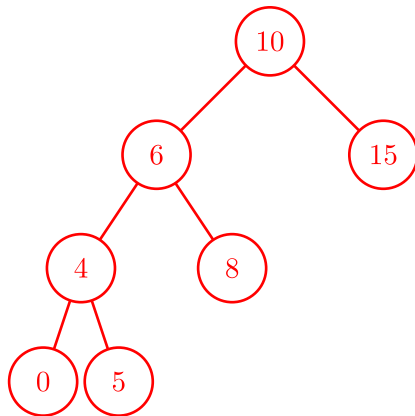


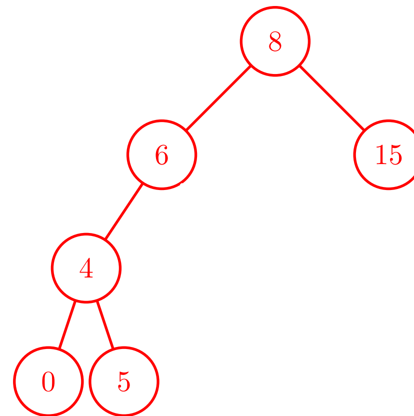
insert(8), remove(1)



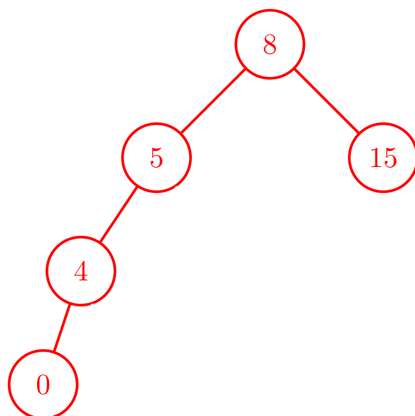


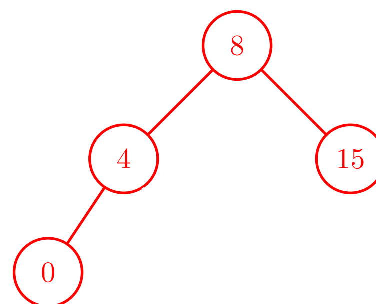
remove(9), remove(10)





remove(6), remove(5)





(b) Binäre Suchbäume können im Worst Case (z.B. beim Einfügen von sortierten Zahlenfolgen) zu einer Liste entarten, wodurch die *locate*-Operation im Worst Case in $\Theta(n)$ liegt. Ein Binärer Suchbaum ist dann in diesem Fall nicht besser als eine verkettete Liste. Bei AVL- und (a,b)-Bäumen umgeht man dieses Problem, indem man durch entsprechende Invarianten die Balancierung des Suchbaumes sicherstellt. Daher liegt die Worst Case-Laufzeit der *locate*-Operation bei AVL- und (a,b)-Bäumen in $\Theta(\log(n))$.

Aufgabe 23 (AVL-Bäume)
Führen Sie auf einen anfangs leeren AVL-Baum folgende Operationen aus.
insert: 10, 6, 15, 4, 1, 0, 5, 9, 8
remove: 1, 9, 10, 6, 5

Wir fassen noch einmal kurz und äußerst informell zusammen, wie man bei AVL-Bäumen eine insert- bzw. eine remove-Operation ausführt. Der Einfachhalt halber definieren wir uns für die Erklärung die folgenden Begriffe:

„Konfliktknoten“:= der tiefste Knoten im Baum mit einer Balancierungszahl von ±2.

„Konfliktpfad“:= der Pfad ausgehend vom Konfliktknoten zu dem Blatt, welches den Konflikt verursacht hat.

„$BZ(A)$“:= bezeichnet die Balancierungszahl von Knoten A.

Vorgehen bei insert(k)

1. Wie bei einem normalen Binärbaum (AVL-Bäume sind nichts anderes als balancierte Binärbäume) hängen wir das das Element an die richtige Stelle im Baum an.
2. Wir schreiben an alle Knoten ihre Balancierungszahl.
3. Sofern kein Knoten eine Balancierungszahl ±2 besitzt, sind wir fertig. Ansonsten markieren wir uns im Baum den Konfliktknoten und den Konfliktpfad.
4. Wir nennen den Konfliktknoten „A“ und seinen Kindknoten, der auf dem Konfliktpfad liegt „B“.
5. Falls $BZ(A) = -2$ und $BZ(B) = +1$ ODER $BZ(A) = +2$ und $BZ(B) = -1$, dann führen wir eine Doppelrotation aus. Ansonsten machen wir eine Einfachrotation.

Einfachrotation

1. Wir schreiben den Knoten B an die Stelle des Knotens A.
2. Der Knoten A wird zu einem linken bzw. rechten Kind vom Knoten B.
3. Falls Knoten A nun ein linkes Kind von B sein sollte, dann fügen wir den ursprünglich linken Teilbaum vom Knoten B als rechtes Kind an Knoten A an. Falls Knoten A ein rechtes Kind von B sein sollte, dann erhält Knoten A den ursprünglich rechten Teilbaum vom Knoten B als linkes Kind.

Doppelrotation

1. Wir benennen das Kind vom Knoten B, das auch auf dem Konflikpfad liegt „C“.
2. Wir schreiben C an die Stelle, wo vorher Knoten A war und geben dem Knoten C die Knoten A und B als Kinder.

3. Den ursprünglich linken und rechten Teilbaum des Knotens C geben wir nun an A bzw. B ab. Ob Knoten A den linken oder rechten Teilbaum von C bekommt, hängt davon ab, ob A ein linkes oder rechtes Kind vom Knoten C ist. Gleiches gilt für den Knoten B.

Nach jeder Rotation prüfen wir, ob der rotierte AVL-Baum immer noch ein Binärbaum ist. Sollte der entstandene Baum kein Binärbaum mehr sein, dann streichen wir das erhaltene Ergebnis durch und fangen vom Neuen an. Es kann durchaus sein, dass bei einer Remove-Operation erst eine Einfach- und anschließend noch eine Doppelrotation ausgeführt werden muss. Allerdings sollte nach Ausführen aller Rotationsoperationen jeder Knoten im AVL-Baum eine der Balancierungszahlen -1, 0 oder +1 aufweisen.

Vorgehen bei remove(k)
Wir unterscheiden bei der remove-Operation 2 verschiedene Fälle:

Fall 1: Der zu löschende Knoten mit Key k ist ein Blatt oder hat nur ein Kind, welches selbst ein Blatt ist.

1. Man lokalisiert den Key k im AVL-Baum.
2. Man löscht k aus dem Baum. Falls k ein Kind hatte, so schreiben wir das Kind an die Stelle wo vorher k stand.
3. Wir überprüfen die Balancierungszahlen vom so entstandenen AVL-Baum und führen unter Umständen eine Einfach- oder Doppelrotation aus.

Fall 2: Der zu löschende Knoten mit Key k hat 2 Kinder.

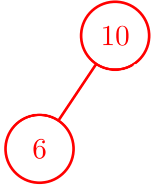
1. Man lokalisiert den Key k im AVL-Baum.
2. Man tauscht den Key k mit dem **rechtesten** Knoten im **linken** Teilbaum von k.
3. Wir führen eine Fall 1-remove-Operation auf den Key k aus.
4. Wir überprüfen die Balancierungszahlen vom so entstandenen AVL-Baum und führen unter Umständen eine Einfach- oder Doppelrotation aus.

Wir kehren zur eigentlich Aufgabe zurück:

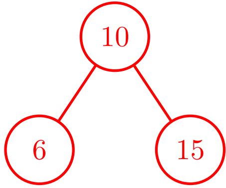
insert(10) ohne Rotation

10

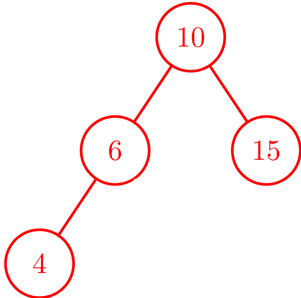
insert(6) ohne Rotation



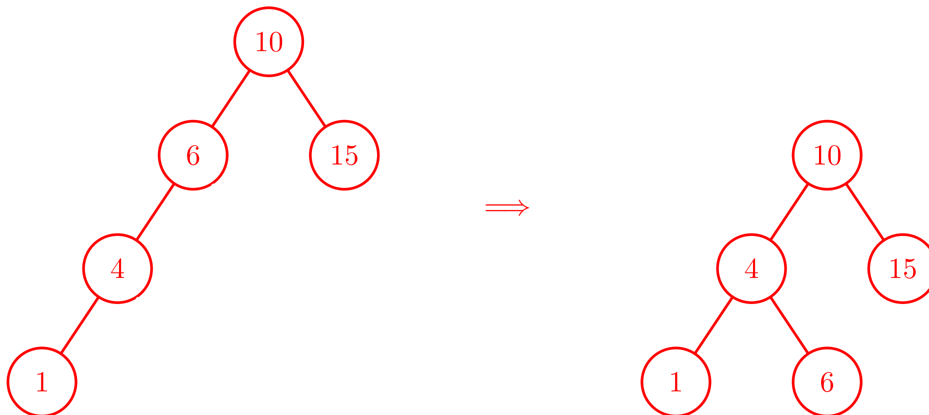
insert(15) ohne Rotation



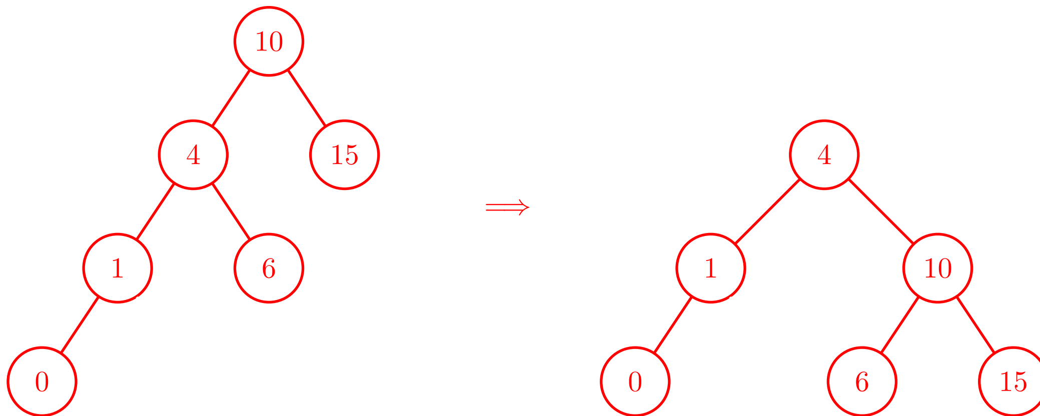
insert(4) ohne Rotation



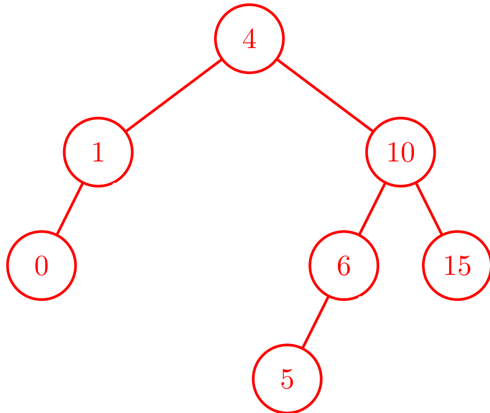
insert(1) mit Einfachrotation an der 6



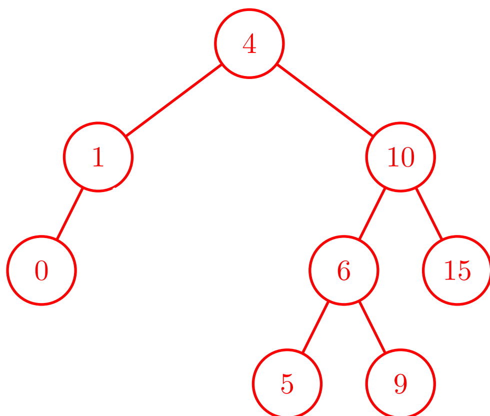
insert(0) mit Einfachrotation an der 10



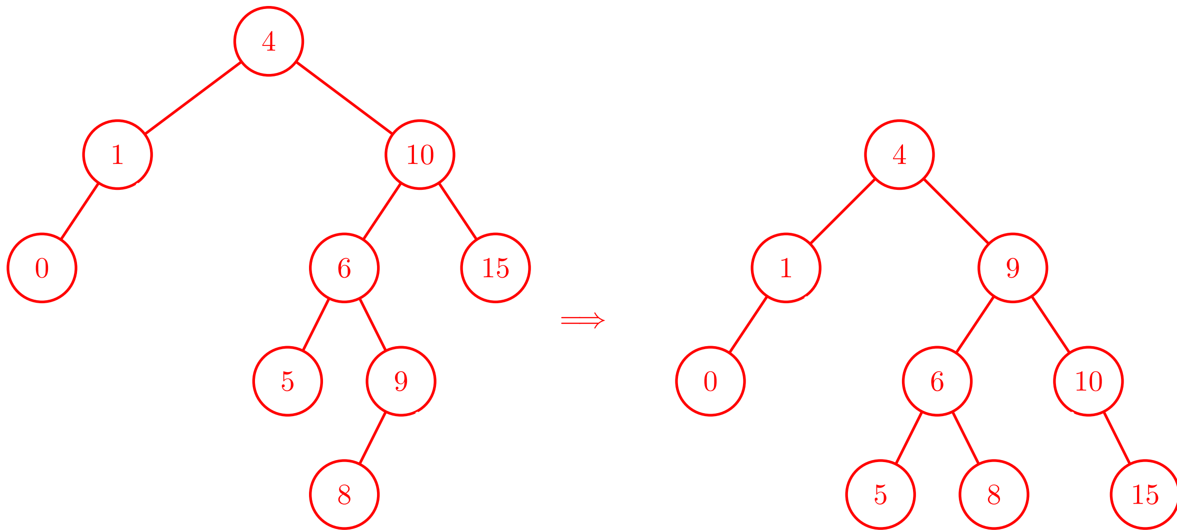
insert(5) ohne Rotation



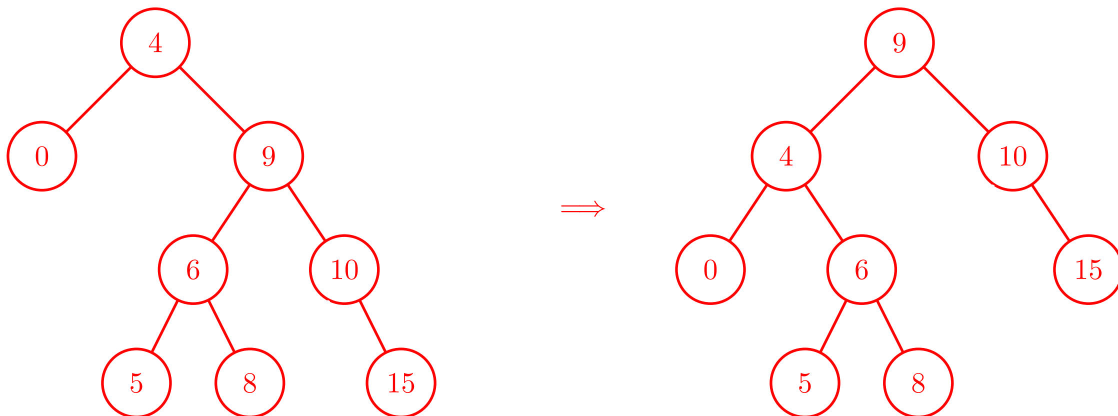
insert(9) ohne Rotation



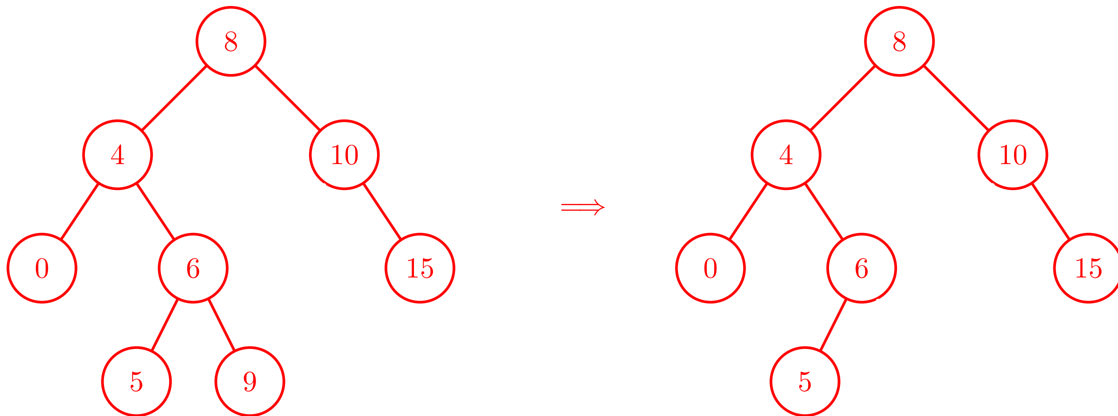
insert(8) mit Doppelrotation an der 10



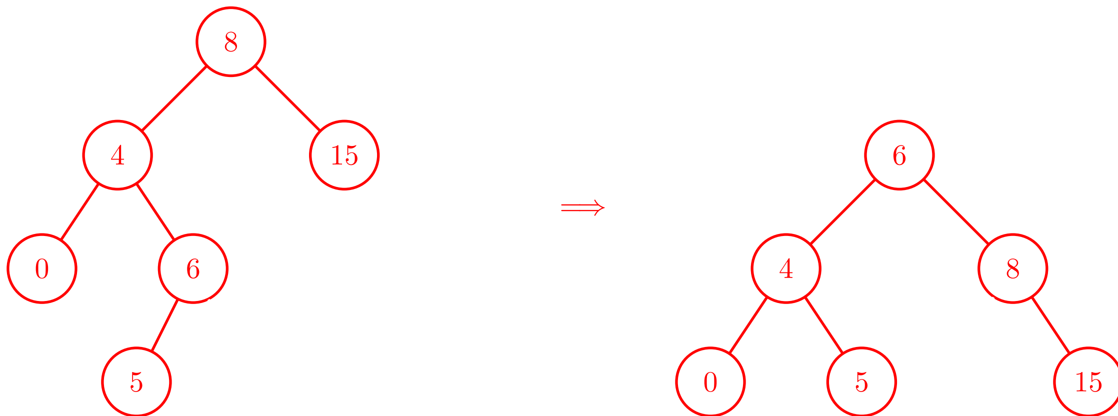
remove(1), Fall 1 - Remove mit anschließender Einfachrotation an der 4



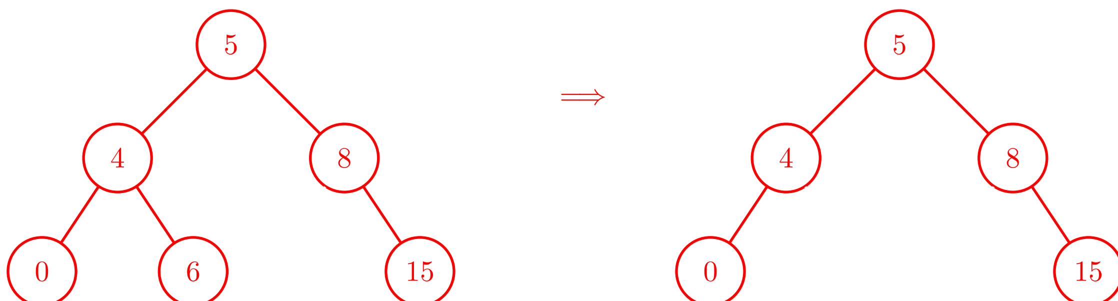
remove(9), Fall 2 - Remove ohne Rotation



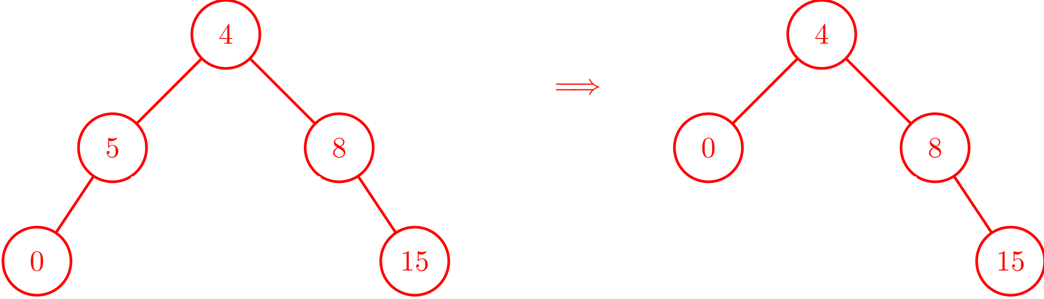
remove(10), Fall 1 - Remove mit Doppelrotation an der 8



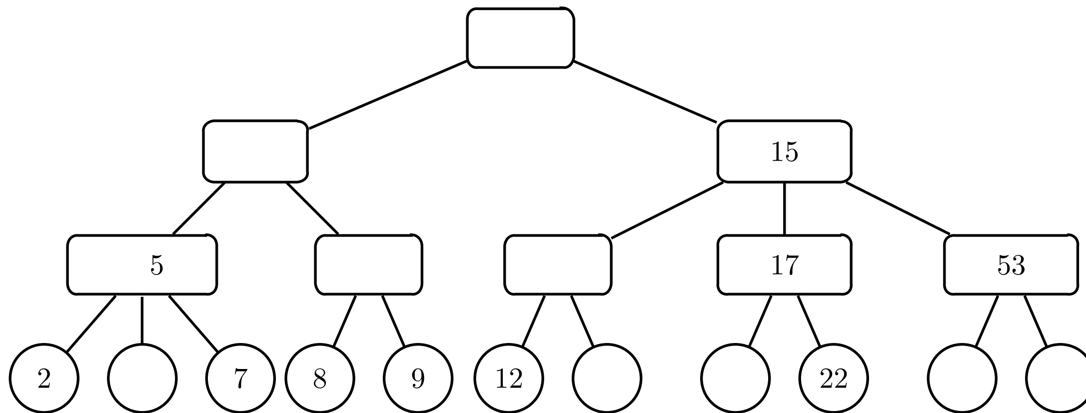
remove(6), Fall 2 - Remove ohne Rotation

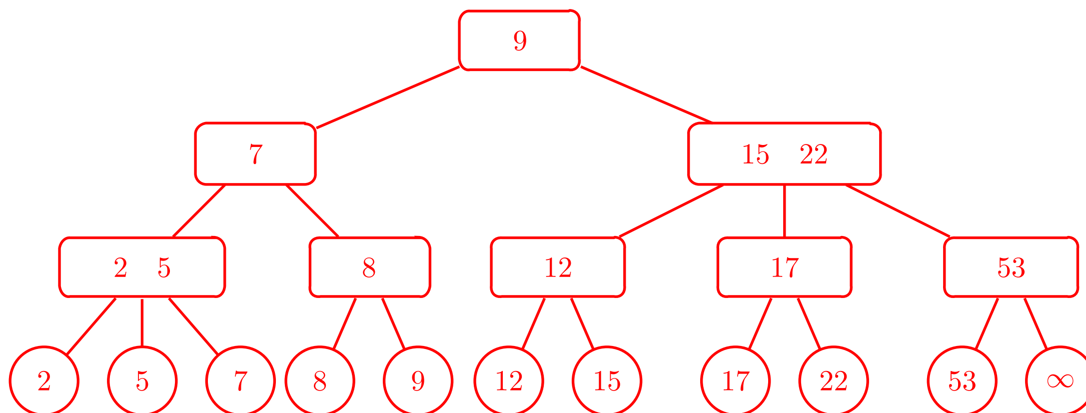


remove(5), Fall 2 - Remove ohne Rotation



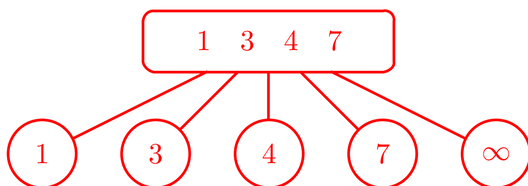
Aufgabe 24 ((a,b) - Bäume)
Andi Arbeit war in der GAD-Tutorstunde mit dem Abschreiben ein wenig zu langsam. Helfen Sie ihm die fehlenden Keys und Splitkeys in den $(2,3)$-Baum einzutragen.



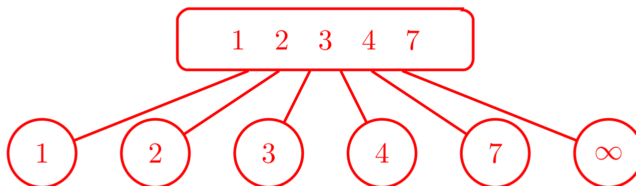


Aufgabe 25 ((a,b) - Bäume)
Ihr Kommilitone Andi Arbeit behauptet, dass er sich zur Übung die Operationen, die bei einem (a, b)-Baum üblich sind, anhand eines $(4, 5)$-Baumes veranschaulicht hätte. Sie wissen natürlich, dass ein (a, b)-Baum nur dann existieren kann, wenn die Ungleichung $b \geq 2a - 1$ erfüllt ist, was bei einem $(4, 5)$-Baum offensichtlich nicht der Fall ist. Obwohl Sie Andi Arbeit darauf hinweisen, beharrt ihr Kommilitone darauf, dass bei seinen Übungen alles immer einwandfrei funktioniert hätte. Zeigen Sie ihm daher durch Wahl eines geeigneten Beispiels an einem beliebigen $(4, 5)$-Baum, welche Probleme durch die Missachtung der oben genannten Ungleichung auftreten können und weshalb ein $(4, 5)$- Baum daher gar nicht existieren kann.

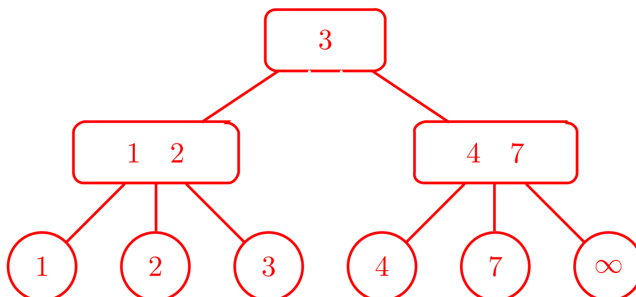
Wir betrachten einen $(4, 5)$-Baum, welcher schon die Elemente mit den Keys 1,3,4 und 7 enthält:



Wir führen eine $insert(2)$-Operation durch:

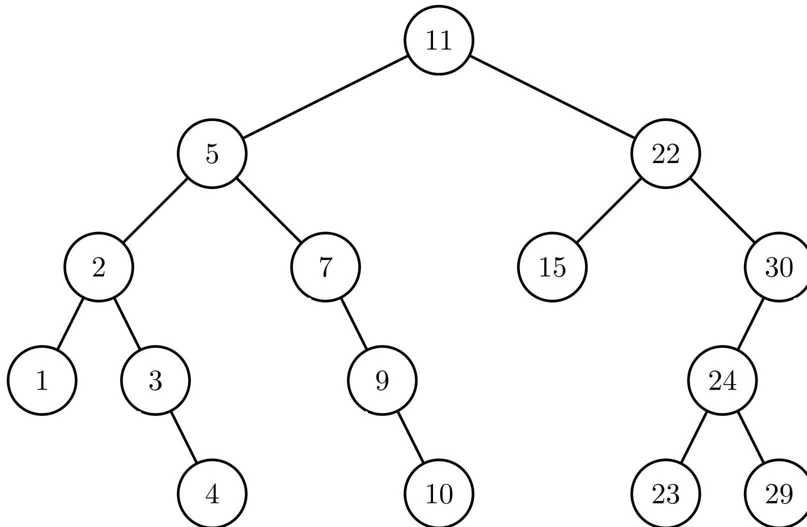


Da der Knoten nun 6 Kinder hat, aber nur $b = 5$ haben darf, müssen wir den Key 3 in eine neue Wurzel ziehen.



Unser $(4, 5)$-Baum enthält nun 2 innere Knoten, die jeweils 3 Kinder besitzen. ↯ ↯ ↯
Dies ist natürlich nicht erlaubt, da ja jeder Knoten in unserem $(4, 5)$-Baum mindestens $a = 4$ Kinder enthalten soll. Mit der Ungleichung $b \geq 2a - 1$ wird also sichergestellt, dass nach einer *insert*- bzw. *remove*− Operation jeder Knoten im Baum mindestens a, aber höchstens b Knoten enthält.

Aufgabe 26 (Graph-Traversierungen)
Gegeben sei der folgende Binärbaum. Geben Sie die PreOrder-, InOrder-, und PostOrder-Traversierung an, sowie die dfs-Nummern und dfs-Finish-Nummern der Knoten an. Was stellen Sie fest?



PreOrder: 11, 5, 2, 1, 3, 4, 7, 9, 10, 22, 15, 30, 24, 23, 29

InOrder: 1, 2, 3, 4, 5, 7, 9, 10, 11, 15, 22, 23, 24, 29, 30

PostOrder: 1, 4, 3, 2, 10, 9, 7, 5, 15, 23, 29, 24, 30, 22, 11

dfs-Nummer: 11, 5, 2, 1, 3, 4, 7, 9, 10, 22, 15, 30, 24, 23, 29

dfs-finish-Nummer: 1, 4, 3, 2, 10, 9, 7, 5, 15, 23, 29, 24, 30, 22, 11

Es fällt auf, dass die PreOrder-Traversierung genau der Anordnung nach dfs-Nummer entspricht, während die PostOrder-Sortierung der dfs-finish-Nummer gleicht. Die InOrder-Traversierung gibt die Knoten in sortierter Reihenfolge aus (siehe dazu auch BinaryTreeSort).