Prof. Dr. Leif Kobbelt
Fabian Emmes, Sven Middelberg,
Michael Kremer

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 1

Abgabe: Montag, **22.04.2013**, 14:00 Uhr

- Die Übungen sollen in Gruppen von zwei bis drei Personen bearbeitet werden.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auf die abgegebenen Lösungen.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auch in die Quellcode-Dateien.
- Geben Sie Ihre Lösungen am **Anfang** der Globalübung, montags, 14:00 Uhr, ab.
- Schicken Sie den jeweiligen Quellcode bitte per **E-Mail** direkt an Ihre/n Tutor/in.
- Geben Sie außerdem den ausgedruckten Quellcode zusammen mit den schriftlichen Lösungen ab.
- Zu spät abgegebene Lösungen werden nicht bewertet.
- Sofern nicht anders gefordert, müssen alle Lösungen und Zwischenschritte kommentiert werden.

Prof. Dr. Leif Kobbelt
Fabian Emmes, Sven Middelberg,
Michael Kremer

Informatik 8
RWTH Aachen

Aufgabe 1 (Fibonacci-Folge [**20 Punkte**])

Im Laufe des Übungsbetriebs, der begleitend zur Vorlesung stattfinden wird, werden Sie verschiedene Algorithmen und Datenstrukturen implementieren. Dafür verwenden wir die Programmiersprache *Java*. In der Regel werden wir Ihnen alle notwendigen Quelldateien zu Beginn der Globalübung zur Verfügung stellen. Dies beinhaltet die Datei `Main.java` sowie Dateien, in der die zum Problem gehörigen Klassen implementiert sind. Diese Klassen werden aus einem Grundgerüst inklusive Methodendeklarationen bestehen, welches dann an den dafür vorgesehenen Stellen durch Ihren Quellcode ergänzt werden soll. Bitte editieren Sie **niemals** die Datei `Main.java`, sondern immer nur die von uns bereitgestellten Problemklassen. Ändern Sie des Weiteren bitte **niemals** von uns bereitgestellte Methodensignaturen (also den Namen oder die Parameter und Rückgabewerte einer Methode). Sie dürfen selbstverständlich immer eigene Variablen und Methoden hinzufügen und benutzen, sofern Sie das von uns vorgegebene Grundgerüst nicht verändern.
Für den Einstieg in die Programmiersprache Java werden Sie in dieser Übung die *Fibonacci-Folge* implementieren. Die Fibonacci-Folge f_i lässt sich durch folgende Gleichung rekursiv definieren:

$$f_i = \begin{cases} 0 & \text{falls } i = 0 \\ 1 & \text{falls } i = 1 \\ f_{i-2} + f_{i-1} & \text{falls } i \geq 2 \end{cases}$$

Wir stellen Ihnen für diese Aufgabe die Klassen `Main.java` und `Fibonacci.java` zur Verfügung. Sofern Sie die Teilaufgaben implementiert haben gibt Ihnen das Programm die Fibonacci-Zahlen f_0 bis f_{40} aus, inklusive der Zeit in Sekunden, welche für die Berechnung der jeweiligen Fibonacci-Zahl benötigt wurde. Für sämtliche Teilaufgaben können Sie davon ausgehen, dass nur valide Werte, also Werte für die die Fibonacci-Folge definiert ist eingegeben werden.

(a) In der Klasse `Fibonacci.java` finden Sie die Methode `fibonacciRecursive`, welche eine Ganzzahl `i` als Parameter erwartet. Ergänzen Sie die Methode so, dass Sie *rekursiv*, also durch Aufrufe von `fibonacciRecursive(i-2)` und `fibonacciRecursive(i-1)`, die `i`-te Fibonacci-Zahl berechnet und zurückgibt. Notieren Sie die Zeiten, welche für die Berechnung der einzelnen Fibonacci-Zahlen benötigt wurde. [**7 Punkte**]

(b) Neben der rekursiven Berechnung, kann die `i`-te Fibonacci-Zahl auch *iterativ* ermittelt werden, indem bei $j = 2$ angefangen die Werte der Fibonacci-Zahlen f_{j-2} und f_{j-1} in Variablen zwischengespeichert und in jeder Iteration aktualisiert werden, solange $j < i$ ist.

Implementieren Sie die Berechnung der `i`-ten Fibonacci-Zahl iterativ, also ohne rekursive Methodenaufrufe, in der Methode `fibonacciIterative`. Notieren Sie erneut die Berechnungszeiten. [**7 Punkte**]

(c) Vergleichen Sie die gemessenen Zeiten aus Aufgabe (a) mit denen aus Aufgabe (b). Interpretieren Sie die Resultate. [**6 Punkte**]

Bitte geben Sie nur die von Ihnen bearbeitete `Fibonacci.java` ab.

Prof. Dr. Leif Kobbelt
Fabian Emmes, Sven Middelberg,
Michael Kremer

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 1

Abgabe: Montag, **22.04.2013**, 14:00 Uhr

- Die Übungen sollen in Gruppen von zwei bis drei Personen bearbeitet werden.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auf die abgegebenen Lösungen.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auch in die Quellcode-Dateien.
- Geben Sie Ihre Lösungen am **Anfang** der Globalübung, montags, 14:00 Uhr, ab.
- Schicken Sie den jeweiligen Quellcode bitte per **E-Mail** direkt an Ihre/n Tutor/in.
- Geben Sie außerdem den ausgedruckten Quellcode zusammen mit den schriftlichen Lösungen ab.
- Zu spät abgegebene Lösungen werden nicht bewertet.
- Sofern nicht anders gefordert, müssen alle Lösungen und Zwischenschritte kommentiert werden.

Prof. Dr. Leif Kobbelt
Fabian Emmes, Sven Middelberg,
Michael Kremer

Informatik 8
RWTH Aachen

Aufgabe 1 (Fibonacci-Folge [**20 Punkte**])

Im Laufe des Übungsbetriebs, der begleitend zur Vorlesung stattfinden wird, werden Sie verschiedene Algorithmen und Datenstrukturen implementieren. Dafür verwenden wir die Programmiersprache *Java*. In der Regel werden wir Ihnen alle notwendigen Quelldateien zu Beginn der Globalübung zur Verfügung stellen. Dies beinhaltet die Datei `Main.java` sowie Dateien, in der die zum Problem gehörigen Klassen implementiert sind. Diese Klassen werden aus einem Grundgerüst inklusive Methodendeklarationen bestehen, welches dann an den dafür vorgesehenen Stellen durch Ihren Quellcode ergänzt werden soll. Bitte editieren Sie **niemals** die Datei `Main.java`, sondern immer nur die von uns bereitgestellten Problemklassen. Ändern Sie des Weiteren bitte **niemals** von uns bereitgestellte Methodensignaturen (also den Namen oder die Parameter und Rückgabewerte einer Methode). Sie dürfen selbstverständlich immer eigene Variablen und Methoden hinzufügen und benutzen, sofern Sie das von uns vorgegebene Grundgerüst nicht verändern.
Für den Einstieg in die Programmiersprache Java werden Sie in dieser Übung die *Fibonacci-Folge* implementieren. Die Fibonacci-Folge f_i lässt sich durch folgende Gleichung rekursiv definieren:

$$f_i = \begin{cases} 0 & \text{falls } i = 0 \\ 1 & \text{falls } i = 1 \\ f_{i-2} + f_{i-1} & \text{falls } i \geq 2 \end{cases}$$

Wir stellen Ihnen für diese Aufgabe die Klassen `Main.java` und `Fibonacci.java` zur Verfügung. Sofern Sie die Teilaufgaben implementiert haben gibt Ihnen das Programm die Fibonacci-Zahlen f_0 bis f_{40} aus, inklusive der Zeit in Sekunden, welche für die Berechnung der jeweiligen Fibonacci-Zahl benötigt wurde. Für sämtliche Teilaufgaben können Sie davon ausgehen, dass nur valide Werte, also Werte für die die Fibonacci-Folge definiert ist eingegeben werden.

(a) In der Klasse `Fibonacci.java` finden Sie die Methode `fibonacciRecursive`, welche eine Ganzzahl `i` als Parameter erwartet. Ergänzen Sie die Methode so, dass Sie *rekursiv*, also durch Aufrufe von `fibonacciRecursive(i-2)` und `fibonacciRecursive(i-1)`, die `i`-te Fibonacci-Zahl berechnet und zurückgibt. Notieren Sie die Zeiten, welche für die Berechnung der einzelnen Fibonacci-Zahlen benötigt wurde. [**7 Punkte**]

(b) Neben der rekursiven Berechnung, kann die `i`-te Fibonacci-Zahl auch *iterativ* ermittelt werden, indem bei $j = 2$ angefangen die Werte der Fibonacci-Zahlen f_{j-2} und f_{j-1} in Variablen zwischengespeichert und in jeder Iteration aktualisiert werden, solange $j < i$ ist.

Implementieren Sie die Berechnung der `i`-ten Fibonacci-Zahl iterativ, also ohne rekursive Methodenaufrufe, in der Methode `fibonacciIterative`. Notieren Sie erneut die Berechnungszeiten. [**7 Punkte**]

(c) Vergleichen Sie die gemessenen Zeiten aus Aufgabe (a) mit denen aus Aufgabe (b). Interpretieren Sie die Resultate. [**6 Punkte**]

Bitte geben Sie nur die von Ihnen bearbeitete `Fibonacci.java` ab.

Lösungsvorschlag

(a) Siehe `Fibonacci.java`

(b) Siehe `Fibonacci.java`

(c) Tabelle 1 zeigt die Zeiten der rekursiven und iterativen Methode gemittelt über 100 Berechnungen. Offensichtlich steigen die Zeiten für die rekursive Methode schnell an, während sie für die iterative Methode nahezu konstant bleiben.

Dies lässt sich durch die Anzahl der Fibonacci-Zahlen, welche insgesamt für die Berechnung der Fibonacci-Zahl f_i ermittelt werden, erklären. Bei der iterativen Methode wird jede Fibonacci-Zahl f_j, $0 \leq j \leq i$, genau einmal berechnet. Also ist die Anzahl der Fibonacci-Zahlen, welche für die Berechnung der Fibonacci-Zahl f_i ermittelt werden, $N_{it}(i) = i + 1$.

Bei der rekursiven Methode ist dies etwas komplizierter. Die Zahl N_{rek} lässt sich rekursiv definieren:

$$N_{rek}(i) = \begin{cases} 1 & \text{falls } i = 0 \\ 1 & \text{falls } i = 1 \\ N_{rek}(i-2) + N_{rek}(i-1) + 1 & \text{falls } i \geq 2 \end{cases} \tag{1}$$

Beispielsweise nimmt N_{rek} für die ersten 10 Fibonacci-Zahlen die folgenden Werte an:

i	f_i	$N_{rek}(i)$
0	0	1
1	1	1
2	1	3
3	2	5
4	3	9
5	5	15
6	8	25
7	13	41
8	21	67
9	34	109

Genaues hinsehen lässt den folgenden Zusammenhang vermuten:

$$N_{rek}(i) = 2 \cdot f_{i+1} - 1 \ . \tag{2}$$

Diese Vermutung lässt sich leicht durch vollständige Induktion beweisen:

Induktionsanfang:
$$N_{rek}(0) = 1 = 2 \cdot f_1 - 1$$

Induktionsschluss:
$$\begin{aligned} N_{rek}(i+1) &\overset{(1)}{=} N_{rek}(i-1) + N_{rek}(i) + 1 \\ &\overset{\text{IV}}{=} 2 \cdot f_i - 1 + 2 \cdot f_{i+1} - 1 + 1 \\ &= 2 \cdot (f_i + f_{i+1}) - 1 \\ &= 2 \cdot f_{i+2} - 1 \end{aligned}$$

□

Dieser Zusammenhang spiegelt sich auch grafisch wider, wenn man die Plots von f_i und die entsprechenden Berechnungszeiten der rekursiven Methode für $0 \leq i \leq 40$ miteinander vergleicht:

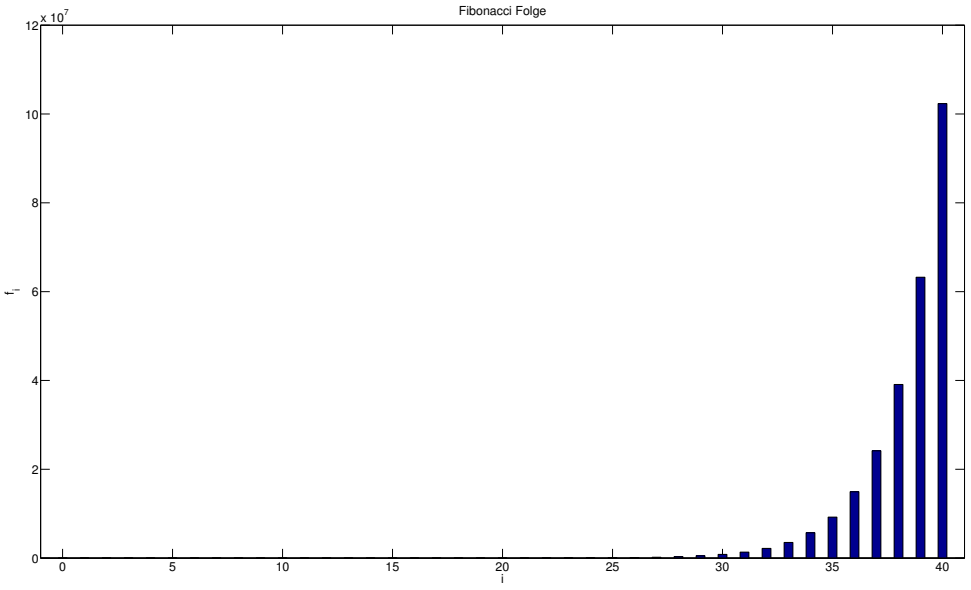


Figure 1: f_i for $0 \leq i \leq 40$.

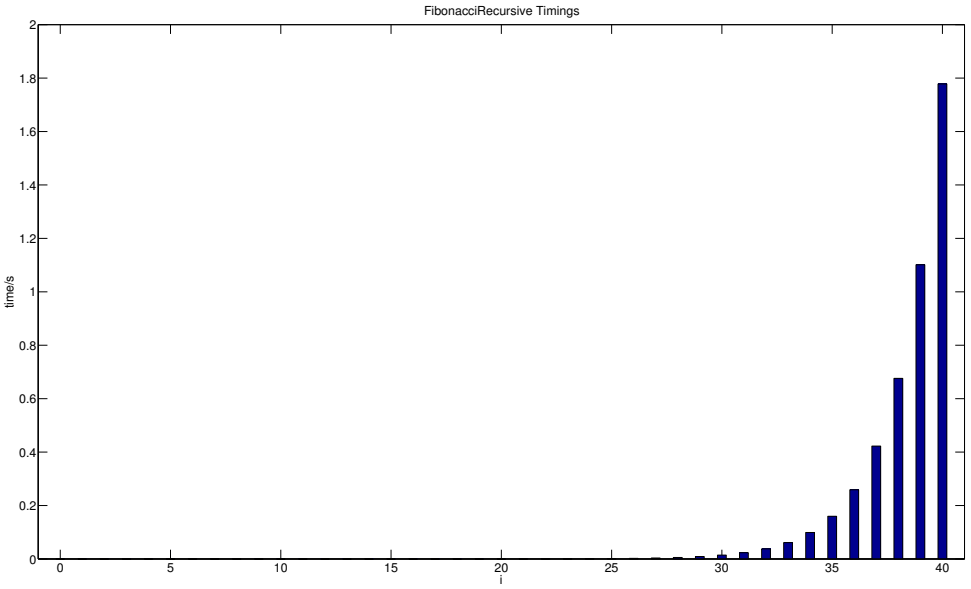


Figure 2: Durchschnittliche Berechnungszeit für f_i, $0 \leq i \leq 40$.

`i`	time(`fibonacciRecursive(i)`)	time(`fibonacciRecursive(i)`)
0	0.00000038	0.00000052
1	0.00000034	0.00000087
2	0.00000153	0.00000037
3	0.00000110	0.00000037
4	0.00000174	0.00000041
5	0.00000288	0.00000046
6	0.00000452	0.00000047
7	0.00000772	0.00000048
8	0.00001370	0.00000171
9	0.00002069	0.00000073
10	0.00000215	0.00000115
11	0.00000172	0.00000076
12	0.00000250	0.00000081
13	0.00000449	0.00000081
14	0.00000716	0.00000083
15	0.00001087	0.00000108
16	0.00001712	0.00000095
17	0.00002849	0.00000102
18	0.00004708	0.00000102
19	0.00007403	0.00000156
20	0.00011960	0.00000157
21	0.00020700	0.00000167
22	0.00036167	0.00000175
23	0.00052614	0.00000169
24	0.00088942	0.00000019
25	0.00133558	0.00000027
26	0.00214821	0.00000023
27	0.00351774	0.00000023
28	0.00555246	0.00000024
29	0.00899743	0.00000023
30	0.01437606	0.00000025
31	0.02344586	0.00000022
32	0.03783847	0.00000023
33	0.06116425	0.00000024
34	0.09915604	0.00000024
35	0.15973625	0.00000023
36	0.25901642	0.00000023
37	0.42260498	0.00000025
38	0.67593269	0.00000020
39	1.10114408	0.00000024
40	1.77799733	0.00000025

Table 1: Berechnungszeiten (in Sekunden) der ersten 41 Fibonacci-Zahlen für die rekursive und iterative Methode

Prof. Dr. Leif Kobbelt
Fabian Emmes, Sven Middelberg,
Michael Kremer

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 2

Abgabe: Montag, **29.04.2013**, 14:00 Uhr

- Die Übungen sollen in Gruppen von zwei bis drei Personen bearbeitet werden.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auf die abgegebenen Lösungen.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auch in die Quellcode-Dateien.
- Geben Sie Ihre Lösungen am **Anfang** der Globalübung, montags, 14:00 Uhr, ab.
- Schicken Sie den jeweiligen Quellcode bitte per **E-Mail** direkt an Ihre/n Tutor/in.
- Geben Sie außerdem den ausgedruckten Quellcode zusammen mit den schriftlichen Lösungen ab.
- Zu spät abgegebene Lösungen werden nicht bewertet.
- Sofern nicht anders gefordert, müssen alle Lösungen und Zwischenschritte kommentiert werden.

Prof. Dr. Leif Kobbelt
Fabian Emmes, Sven Middelberg,
Michael Kremer

Informatik 8
RWTH Aachen

Aufgabe 1 (*Abstrakte Datentypen* [**10 Punkte**])

Gegeben sei folgende Signatur eines abstrakten Datentyps:

Sorten: $\mathbb{R}$, A

Funktionen:

$$\begin{aligned}
\mathrm{c} &: \mathbb{R}\times\mathbb{R}\times\mathbb{R}\times\mathbb{R} \to \mathrm{A}\\
\mathrm{s} &: \mathbb{R}\times\mathrm{A} \to \mathrm{A}\\
+ &: \mathrm{A}\times\mathrm{A} \to \mathrm{A}\\
\times &: \mathrm{A}\times\mathrm{A} \to \mathrm{A}\\
\mathrm{d} &: \mathrm{A} \to \mathbb{R}\\
\mathrm{t} &: \mathrm{A} \to \mathrm{A}
\end{aligned}$$

Axiome: $\forall a \in \mathrm{A}\ \ \forall i,j \in \mathbb{R}$

$$\begin{aligned}
&(1)\quad \mathrm{s}(f,\mathrm{c}(w,x,y,z)) &&= \mathrm{c}(f\cdot w, f\cdot x, f\cdot y, f\cdot z)\\
&(2)\quad +(\mathrm{c}(w,x,y,z),\mathrm{c}(i,j,k,l)) &&= \mathrm{c}(w+i, x+j, y+k, z+l)\\
&(3)\quad \times(\mathrm{c}(w,x,y,z),\mathrm{c}(i,j,k,l)) &&= \mathrm{c}(w\cdot i + x\cdot k,\\
&&&\qquad w\cdot j + x\cdot l,\\
&&&\qquad y\cdot i + z\cdot k,\\
&&&\qquad y\cdot j + z\cdot l)\\
&(4)\quad \mathrm{d}(\mathrm{c}(w,x,y,z)) &&= w\cdot z - x\cdot y\\
&(5)\quad \mathrm{t}(\mathrm{c}(w,x,y,z)) &&= \mathrm{c}(w,y,x,z)
\end{aligned}$$

Die Grundrechenarten können dabei als elementar betrachtet werden.

(a) Beweisen oder widerlegen Sie folgende Behauptung [**2 Punkte**]:

$$\forall i,j,k,l,w,x,y,z \in \mathbb{R} : \times(\mathrm{c}(w,x,y,z),\mathrm{c}(i,j,k,l)) = \times(\mathrm{c}(i,j,k,l),\mathrm{c}(w,x,y,z))$$

(b) Beweisen oder widerlegen Sie folgende Behauptung [**3 Punkte**]:

$$\begin{aligned}
\forall w,x,y,z,f \in \mathbb{R} :\ & \mathrm{s}\Big(f,\mathrm{s}\Big(f,\mathrm{s}\Big(\mathrm{d}\big(\mathrm{t}(\mathrm{c}(w,x,y,z))\big),\mathrm{c}\big(1,0,0,1\big)\Big)\Big)\Big)\\
&= \mathrm{s}\Big(\mathrm{d}\Big(\mathrm{s}\big(f,\mathrm{c}(w,x,y,z)\big)\Big),\mathrm{c}(1,0,0,1)\Big)
\end{aligned}$$

(c) Beschreiben Sie den Datentyp und dessen Funktionen mit eigenen Worten. [**2 Punkte**]

(d) Definieren Sie einen ADT $\mathrm{STACK_A}$, welcher einen Stack für Objekte des Datentyps A darstellt. Neben den üblichen Stackoperationen soll der Datentyp des Weiteren eine Funktion $\mathrm{contains_d}$ bereitstellen, welche prüft, ob der Stack für einen Skalar $f \in \mathbb{R}$ ein Objekt $a \in \mathrm{A}$ enthält mit $\mathrm{d}(a) = f$. Geben Sie sämtliche Sorten, Funktionen und Axiome an, welche für die Definition des ADT $\mathrm{STACK_A}$ notwendig sind. [**3 Punkte**]

Aufgabe 2 (*Labyrinth* [**10 Punkte**])

In der Vorlesung wurde ein iterativer Algorithmus zum Lösen von Labyrinthen vorgestellt, der nur unter der Annahme funktioniert, dass das Labyrinth keine Zyklen enthält.
Erweitern Sie den Algorithmus, so dass er auch auf Labyrinthen mit Zyklen funktioniert.

(a) Geben Sie Ihren erweiterten Algorithmus in Pseudocode an und beschreiben Sie die Idee Ihrer Erweiterung knapp. [**5 Punkte**]

(b) In der auf der Homepage bereitgestellten Datei `Labyrinth.java` findet sich eine Implementierung des Algorithmus der Vorlesung und ein beispielhaftes Labyrinth. Das Labyrinth enthält allerdings einen Zyklus, daher terminiert der ursprüngliche Algorithmus hier nicht. Erweitern Sie die Methode `sucheIt`, welche den Algorithmus der Vorlesung implementiert, um Ihre Idee. Testen Sie, ob der Algorithmus nun das Ziel findet. [**5 Punkte**]

Bitte geben Sie nur die von Ihnen bearbeitete Datei `Labyrinth.java` ab.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 2

Abgabe: Montag, **29.04.2013**, 14:00 Uhr

- Die Übungen sollen in Gruppen von zwei bis drei Personen bearbeitet werden.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auf die abgegebenen Lösungen.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auch in die Quellcode-Dateien.
- Geben Sie Ihre Lösungen am **Anfang** der Globalübung, montags, 14:00 Uhr, ab.
- Schicken Sie den jeweiligen Quellcode bitte per **E-Mail** direkt an Ihre/n Tutor/in.
- Geben Sie außerdem den ausgedruckten Quellcode zusammen mit den schriftlichen Lösungen ab.
- Zu spät abgegebene Lösungen werden nicht bewertet.
- Sofern nicht anders gefordert, müssen alle Lösungen und Zwischenschritte kommentiert werden.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Aufgabe 1 (*Abstrakte Datentypen* [**10 Punkte**])

Gegeben sei folgende Signatur eines abstrakten Datentyps:

Sorten: $\mathbb{R}$, A

Funktionen:

$$\begin{array}{lll} \mathrm{c} & : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} & \to \mathrm{A} \\ \mathrm{s} & : \mathbb{R} \times \mathrm{A} & \to \mathrm{A} \\ + & : \mathrm{A} \times \mathrm{A} & \to \mathrm{A} \\ \times & : \mathrm{A} \times \mathrm{A} & \to \mathrm{A} \\ \mathrm{d} & : \mathrm{A} & \to \mathbb{R} \\ \mathrm{t} & : \mathrm{A} & \to \mathrm{A} \end{array}$$

Axiome: $\forall a \in \mathrm{A}\ \forall i, j \in \mathbb{R}$

$$\begin{array}{lll} (1) & \mathrm{s}(f, \mathrm{c}(w,x,y,z)) & = \mathrm{c}(f \cdot w, f \cdot x, f \cdot y, f \cdot z) \\ (2) & +(\mathrm{c}(w,x,y,z), \mathrm{c}(i,j,k,l)) & = \mathrm{c}(w+i, x+j, y+k, z+l) \\ (3) & \times(\mathrm{c}(w,x,y,z), \mathrm{c}(i,j,k,l)) & = \mathrm{c}(w \cdot i + x \cdot k, \\ & & \quad w \cdot j + x \cdot l, \\ & & \quad y \cdot i + z \cdot k, \\ & & \quad y \cdot j + z \cdot l) \\ (4) & \mathrm{d}(\mathrm{c}(w,x,y,z)) & = w \cdot z - x \cdot y \\ (5) & \mathrm{t}(\mathrm{c}(w,x,y,z)) & = \mathrm{c}(w,y,x,z) \end{array}$$

Die Grundrechenarten können dabei als elementar betrachtet werden.

(a) Beweisen oder widerlegen Sie folgende Behauptung [**2 Punkte**]:

$$\forall i,j,k,l,w,x,y,z \in \mathbb{R} : \times(\mathrm{c}(w,x,y,z), \mathrm{c}(i,j,k,l)) = \times(\mathrm{c}(i,j,k,l), \mathrm{c}(w,x,y,z))$$

(b) Beweisen oder widerlegen Sie folgende Behauptung [**3 Punkte**]:

$$\begin{aligned} \forall w,x,y,z,f \in \mathbb{R} : \ & \mathrm{s}\Big(f, \mathrm{s}\Big(f, \mathrm{s}\Big(\mathrm{d}\big(\mathrm{t}(\mathrm{c}(w,x,y,z))\big), \mathrm{c}\big(1,0,0,1\big)\Big)\Big)\Big) \\ = \ & \mathrm{s}\Big(\mathrm{d}\Big(\mathrm{s}\big(f, \mathrm{c}(w,x,y,z)\big)\Big), \mathrm{c}(1,0,0,1)\Big) \end{aligned}$$

(c) Beschreiben Sie den Datentyp und dessen Funktionen mit eigenen Worten. [**2 Punkte**]

(d) Definieren Sie einen ADT $\mathrm{STACK_A}$, welcher einen Stack für Objekte des Datentyps A darstellt. Neben den üblichen Stackoperationen soll der Datentyp des Weiteren eine Funktion $\mathrm{contains_d}$ bereitstellen, welche prüft, ob der Stack für einen Skalar $f \in \mathbb{R}$ ein Objekt $a \in \mathrm{A}$ enthält mit $\mathrm{d}(a) = f$. Geben Sie sämtliche Sorten, Funktionen und Axiome an, welche für die Definition des ADT $\mathrm{STACK_A}$ notwendig sind. [**3 Punkte**]

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Lösungsvorschlag

(a) Anwendung von Axiom (3) auf beiden Seiten führt zu:

$$\begin{aligned} & \mathrm{c}(w \cdot i + x \cdot k, w \cdot j + x \cdot l, y \cdot i + z \cdot k, y \cdot j + z \cdot l) \\ = \ & \mathrm{c}(i \cdot w + j \cdot y, i \cdot x + j \cdot z, k \cdot w + l \cdot y, k \cdot x + l \cdot z) \ . \end{aligned} \tag{1}$$

Sei nun z. B. $w = y = z = i = j = l = 0$ und $x = k = 1$. Einsetzen in Gleichung (1) liefert:

$$\mathrm{c}(1,0,0,0) \ = \ \mathrm{c}(0,0,0,1) \ .$$

Da dies offensichtlich falsch ist, ist die Behauptung widerlegt. □

(b) Anwendung der Axiome auf beiden Seiten führt zu:

$$\begin{aligned}
& \mathrm{s}\Big(f, \mathrm{s}\Big(f, \mathrm{s}\Big(\mathrm{d}\big(\mathrm{t}(\mathrm{c}(w,x,y,z))\big), \mathrm{c}\big(1,0,0,1\big)\Big)\Big)\Big) \\
= \ & \mathrm{s}\Big(\mathrm{d}\Big(\mathrm{s}\big(f, \mathrm{c}(w,x,y,z)\big)\Big), \mathrm{c}(1,0,0,1)\Big) \\
\Rightarrow \quad & \mathrm{s}\Big(f, \mathrm{s}\Big(f, \mathrm{s}\Big(\mathrm{d}\big(\mathrm{c}(w,y,x,z)\big), \mathrm{c}\big(1,0,0,1\big)\Big)\Big)\Big) && \text{Axiom (5)} \\
= \ & \mathrm{s}\Big(\mathrm{d}\Big(\mathrm{c}(f \cdot w, f \cdot x, f \cdot y, f \cdot z)\Big), \mathrm{c}(1,0,0,1)\Big) && \text{Axiom (1)} \\
\Rightarrow \quad & \mathrm{s}\Big(f, \mathrm{s}\Big(f, \mathrm{s}\Big(w \cdot z - x \cdot y, \mathrm{c}\big(1,0,0,1\big)\Big)\Big)\Big) && \text{Axiom (4)} \\
= \ & \mathrm{s}\Big(f^2 \cdot (w \cdot z - x \cdot y), \mathrm{c}(1,0,0,1)\Big) && \text{Axiom (4)} \\
\Rightarrow \quad & \mathrm{s}\Big(f, \mathrm{s}\Big(f, \mathrm{c}(w \cdot z - x \cdot y, 0, 0, w \cdot z - x \cdot y)\Big)\Big) && \text{Axiom (1)} \\
= \ & \mathrm{c}(f^2 \cdot (w \cdot z - x \cdot y), 0, 0, f^2 \cdot (w \cdot z - x \cdot y)) && \text{Axiom (1)} \\
\Rightarrow \quad & \mathrm{s}\Big(f, \mathrm{c}(f \cdot (w \cdot z - x \cdot y), 0, 0, f \cdot (w \cdot z - x \cdot y))\Big) && \text{Axiom (1)} \\
= \ & \mathrm{c}(f^2 \cdot (w \cdot z - x \cdot y), 0, 0, f^2 \cdot (w \cdot z - x \cdot y)) \\
\Rightarrow \quad & \mathrm{c}(f^2 \cdot (w \cdot z - x \cdot y), 0, 0, f^2 \cdot (w \cdot z - x \cdot y)) && \text{Axiom (1)} \\
= \ & \mathrm{c}(f^2 \cdot (w \cdot z - x \cdot y), 0, 0, f^2 \cdot (w \cdot z - x \cdot y))
\end{aligned}$$

Damit ist die Behauptung bewiesen. □

(c) Der Datentyp repräsentiert 2×2 Matrizen, wobei die Funktionen den folgenden Matrixoperatoren entsprechen:

c(w, x, y, z) Konstruiert die 2×2 Matrix $\begin{pmatrix} w & x \\ y & z \end{pmatrix}$.

s(f, M) Skaliert die Matrix M uniform mit Faktor f.

$+(M_1, M_2)$ Addiert die Matrizen M_1 und M_2.

$\times(M_1, M_2)$ Multipliziert die Matrizen M_1 und M_2.

d(M) Berechnet die Determinante der Matrix M.

t(M) Transponiert die Matrix M.

(d) Definition des Datentyps STACK_A:

Sorten: $\mathbb{R}$, BOOL, A, STACK_A

Funktionen:

create	:		$\rightarrow \text{STACK}_\text{A}$
push	:	$\text{A} \times \text{STACK}_\text{A}$	$\rightarrow \text{STACK}_\text{A}$
pop	:	STACK_A	$\rightarrow \text{STACK}_\text{A}$
top	:	STACK_A	$\rightarrow$ A
empty	:	STACK_A	$\rightarrow$ BOOL
contains_d	:	$\mathbb{R} \times \text{STACK}_\text{A}$	$\rightarrow$ BOOL

Axiome: $\forall a \in \text{A} \;\; \forall i, j \in \mathbb{R}$

(1)	pop(create())	$=$ error
(2)	pop(push(a, s))	$= s$
(3)	top(create())	$=$ error
(4)	top(push(a, s))	$= a$
(5)	empty(create())	$=$ true
(6)	empty(push(a, s))	$=$ false
(7)	$\text{contains}_\text{d}(f, \text{create}())$	$=$ false
(8)	$\text{contains}_\text{d}(f, \text{push}(a, s))$	$\Leftrightarrow (\text{d}(a) = f \;\vee \text{contains}_\text{d}(f, s))$

Aufgabe 2 (*Labyrinth* [**10 Punkte**])

In der Vorlesung wurde ein iterativer Algorithmus zum Lösen von Labyrinthen vorgestellt, der nur unter der Annahme funktioniert, dass das Labyrinth keine Zyklen enthält.
Erweitern Sie den Algorithmus, so dass er auch auf Labyrinthen mit Zyklen funktioniert.

(a) Geben Sie Ihren erweiterten Algorithmus in Pseudocode an und beschreiben Sie die Idee Ihrer Erweiterung knapp. [**5 Punkte**]

(b) In der auf der Homepage bereitgestellten Datei `Labyrinth.java` findet sich eine Implementierung des Algorithmus der Vorlesung und ein beispielhaftes Labyrinth. Das Labyrinth enthält allerdings einen Zyklus, daher terminiert der ursprüngliche Algorithmus hier nicht. Erweitern Sie die Methode `sucheIt`, welche den Algorithmus der Vorlesung implementiert, um Ihre Idee. Testen Sie, ob der Algorithmus nun das Ziel findet. [**5 Punkte**]

Bitte geben Sie nur die von Ihnen bearbeitete Datei `Labyrinth.java` ab.

Lösungsvorschlag

(a) Eine mögliche Lösung ist es, den Algorithmus um ein Array mit Booleschen Werten zu ergänzen, welche markieren, ob ein Feld schon besucht wurde.

Wir führen ein Hilfsprädikat $\mathsf{V} : \mathsf{W} \times \mathsf{R} \times \mathsf{Bool}^{\mathsf{m}\times\mathsf{n}} \to \mathsf{Bool}$ ein ((m, n) ist die Größe des Labyrinths), so dass $\mathsf{V}(p, r, v)$ genau dann wahr ist, wenn das Nachbarfeld von Feld p in Richtung r in v als besucht markiert ist.

Der folgende Pseudocode zeigt den erweiterten Algorithmus. Die ergänzten Stellen sind hierbei unterstrichen.

```
P ← Pbegin
S ← Create()
r ← 'N'
v ← Bool[n][m] /* Boolesches Array, initialisiert mit false. */
while P ≠ Pend and ((L(P,r) and V(P,r,v))
                    or r < 'W' or not Empty(S)) do
    while P ≠ Pend and L(P,r) and V(P,r,v) and
                        (S = ∅ or r ≠ −Top(S)) do
        S = Push(r,S)
        v[P.x][P.y] ← true
        P ← Go(P,r)
        r ← 'N'
    if P ≠ Pend then
        while r = 'W' and not Empty(S) do
            r = Top(S)
            P ← Go(P,−r)
            S ← Pop(S)
```

```
if r < 'W' then
  r ← next(r)
```

(b) Die Datei `Labyrinth.java` finden Sie zum Download auf der Homepage.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 3

Abgabe: Montag, **06.05.2013**, 14:00 Uhr

- Die Übungen sollen in Gruppen von zwei bis drei Personen bearbeitet werden.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auf die abgegebenen Lösungen.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auch in die Quellcode-Dateien.
- Geben Sie Ihre Lösungen am **Anfang** der Globalübung, montags, 14:00 Uhr, ab.
- Schicken Sie den jeweiligen Quellcode bitte per **E-Mail** direkt an Ihre/n Tutor/in.
- Geben Sie außerdem den ausgedruckten Quellcode zusammen mit den schriftlichen Lösungen ab.
- Zu spät abgegebene Lösungen werden nicht bewertet.
- Sofern nicht anders gefordert, müssen alle Lösungen und Zwischenschritte kommentiert werden.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

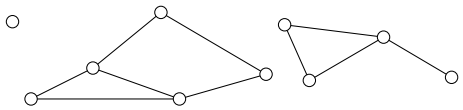
Informatik 8
RWTH Aachen

Aufgabe 1 (Graphen [**10 Punkte**])

1. Stellen Sie die folgende Adjazenzliste als Adjazenzmatrix A sowie als Diagramm dar. Berechnen Sie $(A+I)^2$ und $(A+I)^3$ wobei I die Einheitsmatrix ist. [4 Punkte]

   1: 3, 4
   2: 3
   3: 1, 2
   4: 1, 5
   5: 4

2. Gegeben sei ein ungerichteter, planarer Graph mit v Knoten, e Kanten, f Facetten und c Zusammenhangskomponenten. Die Zahl der Facetten eines planaren Graphen ist die Zahl der von Kanten begrenzten Gebiete in einer beliebigen planaren Darstellung des Graphen, die keine anderen Kanten enthalten. Das außerhalb des Graphen liegende Gebiet zählt hierbei nicht als Facette. Für den untenstehenden Graph ist z. B. $v = 10$, $e = 10$, $f = 3$, $c = 3$.

   Für solche Graphen gilt die Euler-Formel

   $$v - e + f = c.$$

   Diese Formel soll im Folgenden hergeleitet werden.

   (a) Zeigen Sie zunächst: Die Euler-Formel gilt für Bäume (hier: Baum = zusammenhängender, ungerichteter, zyklenfreier Graph). [2 Punkte]

   (b) Beweisen Sie die allgemeine Euler-Formel für zusammenhängende Graphen und $e \geq 1$ durch Induktion über e. Verwenden sie im Induktionsschritt Teilaufgabe (a). [2 Punkte]

   (c) Beweisen Sie die Behauptung unter Verwendung von Teil (b) für nichtzusammenhängende Graphen. [2 Punkte]

Aufgabe 2 (Programmierung: Heap [**10 Punkte**])

1. Implementieren Sie die Datenstruktur Heap (Warteschlange, die die Heap-Bedingung erfüllt) in Java. Verwenden Sie dazu die effiziente Implementierung auf einem Array (Vorlesung 1.4, Folien 11 ff.) Die Implementierung soll insbesondere die Funktionen `Enq` zum Hinzufügen eines Elements zum Heap, `Deq` zum Entfernen des obersten Elements und `Get` zum Zugriff auf das erste Element enthalten. In dem bereitgestellten Code ist die Warteschlange bereits als Integer-Array

   ```
   private int[] S;
   ```

   als Member-Variable der Klasse `Heap` deklariert. Diese wird im Konstruktor mit der festen Größe von 10 Elementen angelegt:

   ```
   this.S = new int[10];
   ```

   Vervollständigen Sie die im Code deklarierten Methoden `Enq`, `Deq`, sowie `Get`. Die zu bearbeitenden Stellen sind bereits im Code markiert. Sie dürfen alle existierenden Methoden in der Klasse `Heap` benutzen. [8 Punkte]

2. In der von uns bereitgestellten `main()`-Methode wird eine Instanz der Klasse `Heap` erzeugt und nacheinander einige Testwerte der Warteschlange hinzugefügt und wieder herausgenommen.

   Vervollständigen Sie die Funktion `printQueue` in der Klasse `Heap` so, dass sie die Prioritätsschlange mit Elementen x_0 bis x_{n-1} in der folgenden Form ausgibt:

   $$[x_0, x_1, x_2, \ldots, x_{n-1}]$$

   Beim Ausführen der `main()`-Methode erscheint nun jeweils das Zwischenergebnis jeder Operation. Verifizieren Sie, dass Ihre Implementierung stets das richtige Ergebnis produziert und die ausgegebene Prioritätsschlange zu jeder Zeit die Heap-Bedingung erfüllt. Führen Sie hierzu die von uns bereitgestellte `main()`-Methode aus. Editieren Sie die Klasse `MainClass` *nicht*! Sie brauchen zu diesem Aufgabenteil nichts aufschreiben. Drucken Sie lediglich die produzierte Ausgabe aus und geben Sie den Ausdruck zusammen mit dem Ausdruck des Quellcodes der Klasse `Heap` ab. [2 Punkte]

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 3

Abgabe: Montag, **06.05.2013**, 14:00 Uhr

- Die Übungen sollen in Gruppen von zwei bis drei Personen bearbeitet werden.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auf die abgegebenen Lösungen.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auch in die Quellcode-Dateien.
- Geben Sie Ihre Lösungen am **Anfang** der Globalübung, montags, 14:00 Uhr, ab.
- Schicken Sie den jeweiligen Quellcode bitte per **E-Mail** direkt an Ihre/n Tutor/in.
- Geben Sie außerdem den ausgedruckten Quellcode zusammen mit den schriftlichen Lösungen ab.
- Zu spät abgegebene Lösungen werden nicht bewertet.
- Sofern nicht anders gefordert, müssen alle Lösungen und Zwischenschritte kommentiert werden.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

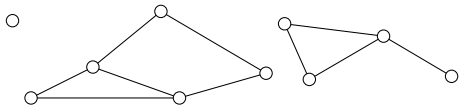
Informatik 8
RWTH Aachen

Aufgabe 1 (Graphen [**10 Punkte**])

1. Stellen Sie die folgende Adjazenzliste als Adjazenzmatrix A sowie als Diagramm dar. Berechnen Sie $(A+I)^2$ und $(A+I)^3$ wobei I die Einheitsmatrix ist. [4 Punkte]

   1: 3, 4
   2: 3
   3: 1, 2
   4: 1, 5
   5: 4

2. Gegeben sei ein ungerichteter, planarer Graph mit v Knoten, e Kanten, f Facetten und c Zusammenhangskomponenten. Die Zahl der Facetten eines planaren Graphen ist die Zahl der von Kanten begrenzten Gebiete in einer beliebigen planaren Darstellung des Graphen, die keine anderen Kanten enthalten. Das außerhalb des Graphen liegende Gebiet zählt hierbei nicht als Facette. Für den untenstehenden Graph ist z. B. $v = 10$, $e = 10$, $f = 3$, $c = 3$.

   Für solche Graphen gilt die Euler-Formel

   $$v - e + f = c.$$

   Diese Formel soll im Folgenden hergeleitet werden.

   (a) Zeigen Sie zunächst: Die Euler-Formel gilt für Bäume (hier: Baum = zusammenhängender, ungerichteter, zyklenfreier Graph). [2 Punkte]

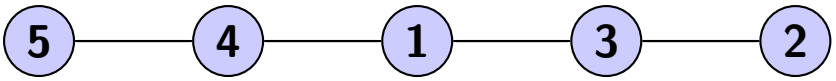
   (b) Beweisen Sie die allgemeine Euler-Formel für zusammenhängende Graphen und $e \geq 1$ durch Induktion über e. Verwenden sie im Induktionsschritt Teilaufgabe (a). [2 Punkte]

   (c) Beweisen Sie die Behauptung unter Verwendung von Teil (b) für nichtzusammenhängende Graphen. [2 Punkte]

Lösungsvorschlag

1.

$$A := \begin{pmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$



$$A+I = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

$$(A+I)^2 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \end{pmatrix}$$

$$(A+I)^3 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

2. (a) Da Bäume zyklenfrei sind, haben sie auch keine Facetten. Somit gilt es, $v-e=1$ zu zeigen. Wir zeigen die Aussage induktiv über die Anzahl der Knoten v im Baum. Ein Baum mit nur einem Knoten erfüllt die Formel offensichtlich. Ein Baum mit mehr als einem Knoten hat mindestens ein Blatt. Entfernen wir dieses Blatt, entfällt auch genau eine Kante. Nach Induktionshypothese gilt $(v-1)-(e-1)=1$ und somit auch $v-e=1$.

   (b) Für die Hinzunahme einer Kante zu einem bestehenden Graphen G, für den $v-e+f=1$ gilt, existieren zwei Fälle, zwischen denen unterschieden werden muss:

   (1) Die neue Kante e' ist inzident zu einem in G bereits existierenden Knoten und einem neuen Knoten v'.

   (2) Die neue Kante wird zwischen zwei in G existierenden Knoten eingefügt.

   Sei G nun ein solcher Graph und sei G' der Graph nach Hinzunahme einer neuen Kante. v', e' und f' seien die in G' hinzugekommenen Knoten, Kanten und Facetten. Wir betrachten nun vier verschiedene Fälle.

   - Sei G ein Baum und erfülle e Fall (1). Es gilt, wie in Aufgabenteil (a) bereits gezeigt, $(v+v')-(e+e')=1$.
   - Sei G ein Baum und erfülle e' Fall (2). Offensichtlich wird damit ein Zyklus (Facette) erzeugt. Es gilt

   $$\begin{aligned} v-(e+e')+f' &= 1 \\ v-e+1-1 &= 1 \quad \text{da } f'=e'=1. \end{aligned}$$

   - Sei G ein zyklischer Graph und erfülle e' Fall (1). Es gilt

   $$\begin{aligned} (v+v')-(e+e')+f &= 1 \\ v-e+f+1-1 &= 1 \quad \text{da } v'=e'=1 \\ v-e+f &= 1. \end{aligned}$$

– Sei G ein zyklischer Graph und erfülle e' Fall (2). Da per Voraussetzung gilt, dass G zusammenhängend ist, wird mit Hinzunahme von e' ein neuer Zyklus (Facette) in G' erzeugt. Es gilt

$$\begin{aligned} v - (e + e') + (f + f') &= 1 \\ v - e + f - 1 + 1 &= 1 \quad \text{da } e' = f' = 1 \\ v - e + f &= 1. \end{aligned}$$

(c) Für $c = 1$ folgt die Aussage aus (b). Sei $c > 1$, dann läßt sich jeder Knoten, jede Kante und jede Facette genau einer Komponente zuordnen. Wähle eine beliebige Komponente in dem Graphen. Dann gilt nach (b) für diese einzelne Komponente $v' - e' + f' = 1$ (wobei v', e' und f' jeweils die Zahl der Knoten, Kanten und Facetten der Komponente sind). Für den Teil des Graphen ohne diese Komponente gilt nach Induktionshypothese $v - e + f = c$. Somit folgt für den gesamten Graphen $(v + v') - (e + e') + (f + f') = c + 1$, bei $v + v'$ Knoten, $e + e'$ Kanten, $f + f'$ Facetten und $c + 1$ Komponenten.

Aufgabe 2 (Programmierung: Heap [**10 Punkte**])

1. Implementieren Sie die Datenstruktur Heap (Warteschlange, die die Heap-Bedingung erfüllt) in Java. Verwenden Sie dazu die effiziente Implementierung auf einem Array (Vorlesung 1.4, Folien 11 ff.) Die Implementierung soll insbesondere die Funktionen `Enq` zum Hinzufügen eines Elements zum Heap, `Deq` zum Entfernen des obersten Elements und `Get` zum Zugriff auf das erste Element enthalten. In dem bereitgestellten Code ist die Warteschlange bereits als Integer-Array

   ```
   private int[] S;
   ```

   als Member-Variable der Klasse `Heap` deklariert. Diese wird im Konstruktor mit der festen Größe von 10 Elementen angelegt:

   ```
   this.S = new int[10];
   ```

   Vervollständigen Sie die im Code deklarierten Methoden `Enq`, `Deq`, sowie `Get`. Die zu bearbeitenden Stellen sind bereits im Code markiert. Sie dürfen alle existierenden Methoden in der Klasse `Heap` benutzen. [8 Punkte]

2. In der von uns bereitgestellten `main()`-Methode wird eine Instanz der Klasse `Heap` erzeugt und nacheinander einige Testwerte der Warteschlange hinzugefügt und wieder herausgenommen.

   Vervollständigen Sie die Funktion `printQueue` in der Klasse `Heap` so, dass sie die Prioritätsschlange mit Elementen x_0 bis x_{n-1} in der folgenden Form ausgibt:

   $$[x_0, x_1, x_2, \ldots, x_{n-1}]$$

   Beim Ausführen der `main()`-Methode erscheint nun jeweils das Zwischenergebnis jeder Operation. Verifizieren Sie, dass Ihre Implementierung stets das richtige Ergebnis produziert und die ausgegebene Prioritätsschlange zu jeder Zeit die Heap-Bedingung erfüllt. Führen Sie hierzu die von uns bereitgestellte `main()`-Methode aus. Editieren Sie die Klasse `MainClass` *nicht*! Sie brauchen zu diesem Aufgabenteil nichts aufschreiben. Drucken Sie lediglich die produzierte Ausgabe aus und geben Sie den Ausdruck zusammen mit dem Ausdruck des Quellcodes der Klasse `Heap` ab. [2 Punkte]

Lösungsvorschlag

1. Der in der Vorlesung vorgestellte Binärbaum, der die Heap-Eigenschaft erfüllt, kann in effizienter Art und Weise als Array implementiert werden. Dies geht, weil der Baum stets vollständig (linksseitig aufgefüllt ist). Da uns in einer Prioritätsschlange lediglich das *größte* Element interessiert, ist es belanglos, dass die jeweiligen Kindknoten im Baum keiner speziellen Ordnung untereinander unterliegen. Die Implementierung funktioniert so, dass der Wurzelknoten immer an Array-Position 1 (bzw. bei null-indizierten Arrays an Position 0) ist.

Die Kindknoten eines Knoten $i \geq 1$ sind dann an Array-Position $2i$ und $2i+1$. Dementsprechend befindet sich der Elternknoten eines Knoten i immer an Array-Position $\lfloor \frac{i}{2} \rfloor$. In unserem Fall beschränken wir die Länge des Arrays auf 10 Elemente.

Beim Einfügen und Herausnehmen eines Elements muss nun jeweils darauf geachtet werden, dass die Heap-Eigenschaft des Baumes erfüllt ist, d.h., dass

(a) der Baum immer linksseitig aufgefüllt ist und

(b) der Wert jedes Elternknoten stets größer als oder gleich der seiner Kinder ist.

In der gewählten Array-Implementierung ist der erste Punkt trivial, da wir per Konstruktion den Baum immer von links auffüllen. Es muss nur sichergestellt werden, dass der Back-Pointer immer auf die Array-Position *hinter* der letzten besetzten Position zeigt (in einer leeren Schlange auf Position 1). Somit gilt: Wenn der Back-Pointer auf Position 1 zeigt, ist die Schlange leer.

Die zweite Bedingung kann beim Einfügen oder Herausnehmen von Elementen verletzt werden und muss ggf. wiederhergestellt werden. Beide Funktionen werden im Folgenden beschrieben.

Einfügen Zeigt der Back-Pointer auf die elfte Stelle im Array (Länge + 1), ist die Schlange voll und ein Fehler wird zurückgegeben (hier eine Java-Exception). Ansonsten wird das neue Element an die letzte Position—also die Position, auf die der Back-Pointer zeigt—geschrieben und solange mit seinem Elternknoten vertauscht, bis man eine Position gefunden hat, an der der Elternknoten *größer als oder gleich* dem neu eingefügten Element ist. Der Back-Pointer wird um eins erhöht.

Herausnehmen Zeigt der Back-Pointer auf die erste Position im Array, wird ein Fehler ausgegeben. Nun wird der Wurzelknoten, also das Element an Array-Position 1, entfernt und durch das letzte Element im Array, also das Element an Position Back-Pointer - 1, ersetzt. Dies verletzt in der Regel die Heap-Eigenschaft, da das letzte Element mit sehr hoher Wahrscheinlichkeit kleiner ist als der Wurzelknoten (es sei denn, alle Elemente haben denselben Wert). Nun wird der neue Wurzelknoten solange mit dem *größeren* seiner beiden Kinder vertauscht, bis eine Position gefunden ist, an der beide Kindknoten einen kleineren oder maximal gleich großen Wert haben. Die Position des Back-Pointers wird um eins verringert.

Das Implementieren der Funktion `Get` ist trivial, da dort lediglich das erste Element im Array zurückgegeben werden muss. Bei einem leeren Array sollte dies einen Fehler produzieren—in dem Falle wieder eine Java-Exception.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 4

Abgabe: Montag, **13.05.2013**, 14:00 Uhr

- Die Übungen sollen in Gruppen von zwei bis drei Personen bearbeitet werden.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auf die abgegebenen Lösungen.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auch in die Quellcode-Dateien.
- Geben Sie Ihre Lösungen am **Anfang** der Globalübung, montags, 14:00 Uhr, ab.
- Schicken Sie den jeweiligen Quellcode bitte per **E-Mail** direkt an Ihre/n Tutor/in.
- Geben Sie außerdem den ausgedruckten Quellcode zusammen mit den schriftlichen Lösungen ab.
- Zu spät abgegebene Lösungen werden nicht bewertet.
- Sofern nicht anders gefordert, müssen alle Lösungen und Zwischenschritte kommentiert werden.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Aufgabe 1 (Algorithmische Analyse [**10 Punkte**])

1. Beweisen Sie die folgenden Eigenschaften der Komplexitätsklassen O, Ω und Θ: Für alle $f, g, h : \mathbb{N} \to \mathbb{R}^+$ gilt [3 Punkte]

   - **Transitivität:** $f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$, analog für Ω und Θ
   - **Reflexivität:** $f \in O(f)$, $f \in \Omega(f)$, $f \in \Theta(f)$
   - **Symmetrie:** $f \in \Theta(g) \Rightarrow g \in \Theta(f)$
   - **Antisymmetrie:** $f \in O(g) \wedge g \in O(f) \Rightarrow f \in \Theta(g)$ und analog $f \in \Omega(g) \wedge g \in \Omega(f) \Rightarrow f \in \Theta(g)$

2. Beschreiben Sie das asymptotische Verhalten folgender Funktionen möglichst genau:

   - $T(n) = 2T(n/2) + n^3$
   - $T(n) = 16T(n/4) + n^2$
   - $T(n) = 3T(n/4) + n \log n$
   - $T(3n) = 3T(2n) + \log n$

   Hierbei sei stets $T(1) = 1$. [3 Punkte]

3. Zeigen oder widerlegen Sie [1 Punkt]:

$$f \in \Theta(g) \wedge \lim_{n\to\infty} \frac{h(n)}{g(n)} = 0 \quad \Rightarrow \quad f + h \in \Theta(g).$$

4. Zeigen oder widerlegen Sie [1 Punkt]:

$$f \in O(n \log_a n) \quad \Leftrightarrow \quad f \in O(n \log_b n).$$

5. Lösen Sie die folgende Rekursionsgleichung so auf, dass sie keine Rekursion mehr enthält, d.h. nur noch von n, nicht aber von $T(n)$ abhängig ist. Die Ermittlung der Komplexitätsklasse ist nicht ausreichend. [2 Punkte]:

$$\begin{aligned} T(0) &= 5 \\ 2T(n) &= nT(n-1) + 3 \cdot n! \end{aligned}$$

Aufgabe 2 (Rekursionsgleichungen [**10 Punkte**])

Gegeben sei folgende Java-Methode (der Einfachheit halber kann angenommen werden, dass der Parameter `i` stets positiv oder null ist):

```
public static int func(int i) {

  if(i == 0 || i == 1) return i;

  int f1 = func((i+1)/2);
  int f2 = func((i+1)/2 - 1);

  if(i % 2 == 1) {
    return f1 * f1 + f2 * f2;
  }

  return f1 * f1 + 2 * f1 * f2;
}
```

Beachten Sie, dass bei der Ganzzahldivision immer auf die nächstniedrigere Ganzzahl *abgerundet* wird.

(a) Bestimmen Sie die Rekursionsgleichung für die Funktion. [4 Punkte]

(b) Schätzen Sie die asymptotische Komplexität mit Hilfe des Master-Theorems ab. [4 Punkte]

(c) Beschreiben Sie, was die Funktion berechnet. [2 Punkte]

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 4

Abgabe: Montag, **13.05.2013**, 14:00 Uhr

- Die Übungen sollen in Gruppen von zwei bis drei Personen bearbeitet werden.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auf die abgegebenen Lösungen.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auch in die Quellcode-Dateien.
- Geben Sie Ihre Lösungen am **Anfang** der Globalübung, montags, 14:00 Uhr, ab.
- Schicken Sie den jeweiligen Quellcode bitte per **E-Mail** direkt an Ihre/n Tutor/in.
- Geben Sie außerdem den ausgedruckten Quellcode zusammen mit den schriftlichen Lösungen ab.
- Zu spät abgegebene Lösungen werden nicht bewertet.
- Sofern nicht anders gefordert, müssen alle Lösungen und Zwischenschritte kommentiert werden.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Aufgabe 1 (*Algorithmische Analyse* [**10 Punkte**])

1. Beweisen Sie die folgenden Eigenschaften der Komplexitätsklassen O, Ω und Θ: Für alle $f, g, h : \mathbb{N} \to \mathbb{R}^+$ gilt

   - **Transitivität:** $f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$, analog für Ω und Θ
   - **Reflexivität:** $f \in O(f)$, $f \in \Omega(f)$, $f \in \Theta(f)$
   - **Symmetrie:** $f \in \Theta(g) \Rightarrow g \in \Theta(f)$
   - **Antisymmetrie:** $f \in O(g) \wedge g \in O(f) \Rightarrow f \in \Theta(g)$ und analog $f \in \Omega(g) \wedge g \in \Omega(f) \Rightarrow f \in \Theta(g)$

2. Beschreiben Sie das asymptotische Verhalten folgender Funktionen möglichst genau:

   - $T(n) = 2T(n/2) + n^3$
   - $T(n) = 16T(n/4) + n^2$
   - $T(n) = 3T(n/4) + n \log n$
   - $T(3n) = 3T(2n) + \log n$

   Hierbei sei stets $T(1) = 1$.

3. Zeigen oder widerlegen Sie:

$$f \in \Theta(g) \wedge \lim_{n\to\infty} \frac{h(n)}{g(n)} = 0 \quad \Rightarrow \quad f + h \in \Theta(g).$$

4. Zeigen oder widerlegen Sie:

$$f \in O(n \log_a n) \quad \Leftrightarrow \quad f \in O(n \log_b n).$$

5. Lösen Sie die folgende Rekursionsgleichung so auf, dass sie keine Rekursion mehr enthält, d.h. nur noch von n, nicht aber von $T(n)$ abhängig ist. Die Ermittlung der Komplexitätsklasse ist nicht ausreichend.

$$\begin{aligned} T(0) &= 5 \\ 2T(n) &= nT(n-1) + 3 \cdot n! \end{aligned}$$

Lösungsvorschlag

1. • **Transitivität:**

$$f \in O(g) \Rightarrow \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : f(n) \leq c \cdot g(n)$$
$$g \in O(h) \Rightarrow \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : g(n) \leq c \cdot h(n)$$

Somit existieren Konstanten c_1, n_1, c_2, n_2 für welche gilt:

$$\forall n \geq n_1 : f(n) \leq c_1 \cdot g(n)$$
$$\forall n \geq n_2 : g(n) \leq c_2 \cdot h(n)$$

Es folgt:

$$\begin{aligned} & \forall n \geq \max(n_1, n_2) : f(n) \leq c_1 \cdot g(n) \leq c_1 \cdot c_2 \cdot h(m) \\ \Rightarrow\ & \forall n \geq \max(n_1, n_2) : f(n) \leq c_1 \cdot c_2 \cdot h(m) \\ \Rightarrow\ & \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : f(n) \leq c \cdot h(n) \\ \Rightarrow\ & f \in O(h) \end{aligned}$$

□

$$f \in \Omega(g) \Rightarrow \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : f(n) \geq c \cdot g(n)$$
$$g \in \Omega(h) \Rightarrow \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : g(n) \geq c \cdot h(n)$$

Somit existieren Konstanten c_1, n_1, c_2, n_2 für welche gilt:

$$\forall n \geq n_1 : f(n) \geq c_1 \cdot g(n)$$
$$\forall n \geq n_2 : g(n) \geq c_2 \cdot h(n)$$

Es folgt:

$$\begin{aligned} & \forall n \geq \max(n_1, n_2) : f(n) \geq c_1 \cdot g(n) \geq c_1 \cdot c_2 \cdot h(m) \\ \Rightarrow\ & \forall n \geq \max(n_1, n_2) : f(n) \geq c_1 \cdot c_2 \cdot h(m) \\ \Rightarrow\ & \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : f(n) \geq c \cdot h(n) \\ \Rightarrow\ & f \in \Omega(h) \end{aligned}$$

□

$$\begin{aligned} & f \in \Theta(g) \Rightarrow f \in O(g) \wedge f \in \Omega(g) \\ & g \in \Theta(h) \Rightarrow g \in O(h) \wedge g \in \Omega(h)) \\ \Rightarrow\ & f \in O(h) \wedge f \in \Omega(h) \\ \Rightarrow\ & f \in \Theta(h) \end{aligned}$$

□

• **Reflexivität:**
Trivialerweise gilt:

$$\begin{aligned} & \forall n \geq 1 : f(n) \leq f(n) \\ \Rightarrow\ & \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : f(n) \leq c \cdot f(n) \\ \Rightarrow\ & f \in O(f) \end{aligned}$$

□

Analog:

$$\begin{aligned} & \forall n \geq 1 : f(n) \geq f(n) \\ \Rightarrow\ & \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : f(n) \geq c \cdot f(n) \\ \Rightarrow\ & f \in \Omega(f) \end{aligned}$$

□

Es folgt:

$$f \in O(f) \wedge f \in \Omega(f) \ \Rightarrow\ f \in \Theta(f)$$

□

- **Symmetrie:**

$$f \in \Theta(g) \Rightarrow \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : \frac{f(n)}{c} \leq g(n) \leq c \cdot f(n)$$

Damit folgt:

$$\begin{aligned} & \forall n \geq n_0 : \frac{f(n)}{c} \leq g(n) \\ \Rightarrow\ & \forall n \geq n_0 : f(n) \leq c \cdot g(n) \\ & \forall n \geq n_0 : g(n) \leq c \cdot f(n) \\ \Rightarrow\ & \forall n \geq n_0 : \frac{g(n)}{c} \leq f(n) \end{aligned}$$

Und somit:

$$\forall n \geq n_0 : \frac{g(n)}{c} \leq f(n) \leq c \cdot g(n) \ \Rightarrow\ g \in \Theta(f)$$

□

- **Antisymmetrie:**

$$\begin{aligned} f \in O(g) \Rightarrow \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : f(n) \leq c \cdot g(n) \\ g \in O(f) \Rightarrow \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : g(n) \leq c \cdot f(n) \end{aligned}$$

Somit existieren Konstanten c_1, n_1, c_2, n_2 für welche gilt:

$$\begin{aligned} \forall n \geq n_1 : f(n) \leq c_1 \cdot g(n) \\ \forall n \geq n_2 : g(n) \leq c_2 \cdot f(n) \end{aligned}$$

Es folgt:

$$\begin{aligned} & \forall n \geq \max(n_1, n_2) : f(n) \leq c_1 \cdot c_2 \cdot g(n) \\ & \forall n \geq \max(n_1, n_2) : g(n) \leq c_1 \cdot c_2 \cdot f(n) \\ \Rightarrow\ & \forall n \geq \max(n_1, n_2) : \frac{g(n)}{c_1 \cdot c_2} \leq f(n) \leq c_1 \cdot c_2 \cdot g(n) \ \Rightarrow\ f \in \Theta(g) \end{aligned}$$

□

$$f \in \Omega(g) \Rightarrow \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : f(n) \geq c \cdot g(n)$$
$$g \in \Omega(f) \Rightarrow \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : g(n) \geq c \cdot f(n)$$

Somit existieren Konstanten c_1, n_1, c_2, n_2 für welche gilt:

$$\forall n \geq n_1 : f(n) \geq c_1 \cdot g(n)$$
$$\forall n \geq n_2 : g(n) \geq c_2 \cdot f(n)$$

Es folgt:

$$\begin{aligned}
&\forall n \geq \max(n_1, n_2) : f(n) \geq c_1 \cdot c_2 \cdot g(n) \\
&\forall n \geq \max(n_1, n_2) : g(n) \geq c_1 \cdot c_2 \cdot f(n) \\
\Rightarrow\ &\forall n \geq \max(n_1, n_2) : \frac{g(n)}{c_1 \cdot c_2} \geq f(n) \geq c_1 \cdot c_2 \cdot g(n) \\
\Rightarrow\ &\forall n \geq \max(n_1, n_2) : \frac{g(n)}{c_1^{-1} \cdot c_2^{-1}} \leq f(n) \leq c_1^{-1} \cdot c_2^{-1} \cdot g(n) \ \Rightarrow\ f \in \Theta(g)
\end{aligned}$$

□

2.
- $T(n) = 2T(n/2) + n^3 = 2T(n/2) + O(n^3)$
  Master Theorem: $a = b = 2$, $p = 3 \Rightarrow a = 2 < b^p = 8$ (1. Fall), und somit $T(n) = O(n^3)$.
- $T(n) = 16T(n/4) + n^2 = 16T(n/4) + O(n^2)$
  Master Theorem: $a = 16$, $b = 4$, $p = 2 \Rightarrow a = 16 = b^p$ (2. Fall), und somit $T(n) = O(n^2 \cdot \log n)$.
- $T(n) = 3T(n/4) + n \log n = 3T(n/4) + O(n^2)$
  Master Theorem: $a = 3$, $b = 4$, $p = 2 \Rightarrow a = 3 < b^p = 16$ (1. Fall), und somit $T(n) = O(n^2)$.
- $T(3n) = 3T(2n) + \log n$
  Substitutiere $m = 3n$: $T(m) = 3T(\frac{2}{3}m) + \log \frac{m}{3} = 3T(m/1.5) + O(m)$
  Master Theorem: $a = 3$, $b = 1.5$, $p = 1 \Rightarrow a = 3 > b^p = 1.5$ (3. Fall), und somit $T(m) = O(m^{\log_{1.5} 3})$. Rücksubstitution liefert $T(3n) = O((3n)^{\log_{1.5} 3}) = O(n^{\log_{1.5} 3})$

3.
$$\lim_{n \to \infty} \frac{h(n)}{g(n)} = 0 \ \Rightarrow\ h \in O(g) \ \Rightarrow\ \exists c_1 : h(n) \leq c_1 \cdot g(n) \ \ \forall n \geq n_1$$
$$f \in \Theta(g) \ \Rightarrow\ \exists c_2 : \frac{1}{c_2} g(n) \leq f(n) \leq c_2 \cdot g(n) \ \ \forall n \geq n_2$$

Dann:

$$\frac{1}{c_1 + c_2} g(n) \leq \frac{1}{c_2} g(n) \leq f(n) \leq f(n) + h(n) \leq c_2 g(n) + c_1 g(n) = (c_1 + c_2) g(n)$$

Es folgt $f + h \in \Theta(g)$ mit $c := c_1 + c_2$ und $n_0 := \max\{n_1, n_2\}$.

4.
$$
\begin{aligned}
f \in O(n \log_a n) \quad &\Leftrightarrow \quad f(n) \leq c \cdot n \log_a n, \forall n \geq n_0 \\
&\Leftrightarrow f(n) \leq c \cdot n \cdot \frac{\ln n}{\ln a} \\
&= c \cdot n \cdot \frac{\ln b}{\ln b} \cdot \frac{\ln n}{\ln a} \\
&= c \cdot \frac{\ln b}{\ln a} \cdot n \cdot \frac{\ln n}{\ln b} \\
&= c \cdot \log_a b \cdot n \cdot \log_b n
\end{aligned}
$$

Damit folgt $f \in O(n \log_b n)$ mit $c' := c \cdot \log_a b$ und $n_0' = n_0$.

5. Multipliziere beide Seiten mit $\frac{2^{n-1}}{n!}$:

$$
\frac{2^n}{n!} T(n) = \frac{2^{n-1}}{(n-1)!} T(n-1) + 3 \cdot 2^{n-1}
$$

$$
S(n) := 2^n \frac{T(n)}{n!}
$$

Damit folgt

$$
\begin{aligned}
S(n) &= S(n-1) + 3 \cdot 2^{n-1} \\
&= S(n-2) + 3 \cdot 2^{n-2} + 3 \cdot 2^{n-1} \\
&\vdots \\
&= S(0) + \sum_{i=0}^{n-1} 3 \cdot 2^i \\
&= 5 + 3(2^n - 1) \\
&= 3 \cdot 2^n + 2
\end{aligned}
$$

und es ist

$$
\begin{aligned}
T(n) &= \frac{n!}{2^n} S(n) = \frac{n!}{2^n}(3 \cdot 2^n + 2) \\
&= 3(n!) + \frac{n!}{2^{n-1}}
\end{aligned}
$$

Aufgabe 2 (Rekursionsgleichungen [**10 Punkte**])

Gegeben sei folgende Java-Methode (der Einfachheit halber kann angenommen werden, dass der Parameter `i` stets positiv oder null ist):

```
public static int func(int i) {

  if(i == 0 || i == 1) return i;

  int f1 = func((i+1)/2);
  int f2 = func((i+1)/2 - 1);

  if(i % 2 == 1) {
    return f1 * f1 + f2 * f2;
  }

  return f1 * f1 + 2 * f1 * f2;
}
```

Beachten Sie, dass bei der Ganzzahldivision immer auf die nächstniedrigere Ganzzahl *abgerundet* wird.

(a) Bestimmen Sie die Rekursionsgleichung für die Funktion. [4 Punkte]

(b) Schätzen Sie die asymptotische Komplexität mit Hilfe des Master-Theorems ab. [4 Punkte]

(c) Beschreiben Sie, was die Funktion berechnet. [2 Punkte]

Lösungsvorschlag

(a) Wenn $i = 0$ oder $i = 1$, dann steigt der Funktionsaufruf in den ersten `if`-Zweig ab. Für die Vergleichsoperation ist der Aufwand c konstant. Für die Summe aller Vergleiche und arithmetischer Operationen im Falle $i > 1$ ist der Aufwand d ebenfalls konstant. Im zweiten Fall wird in jedem Aufruf die Funktion `func` zweimal rekursiv aufgerufen. Dabei wird der Parameter i stets für beide Aufrufe halbiert (und ggf. um eins verringert). Dies führt uns zu folgender Rekursionsgleichung:

$$\begin{aligned} T(1) &= c \\ T(n) &= 2 \cdot T(n/2) + d \\ &= 2 \cdot T(n/2) + O(n^0) \end{aligned}$$

(b) Mit Hilfe der zuvor bestimmten Darstellung lassen sich die im Master-Theorem verwendeten Koeffizienten leicht bestimmen:

$$a = 2, b = 2, p = 0$$

$$\Rightarrow 2 = a > b^p = 1$$

$$\Rightarrow T(n) \in O(n^{\log_2 2}) = O(n)$$

Dass sich die Komplexität des beschriebenen Algorithmus linear in der Eingabegröße verhält, ist auch durch Substitution, ohne Hilfe des Master-Theorems, ersichtlich. Sei der Einfachheit halber $c = d = 1$ (das können wir machen, da wir ja wissen, dass ein konstanter Vorfaktor keinen Einfluss auf die asymptotische Komplexität einer Funktion hat). Dann gilt

$$\begin{aligned}
T(n) &= 2 \cdot T(n/2) + 1 \\
&= 2 \cdot \big[2 \cdot T(n/4) + 1\big] + 1 \\
&= 4 \cdot T(n/4) + 2 + 1 \\
&= 8 \cdot T(n/8) + 4 + 2 + 1 \\
&\vdots \\
&= 2^k \cdot T(n/2^k) + \sum_{i=0}^{k-1} 2^i \\
&\vdots \\
&= n \cdot T(1) + n - 1 \quad (\text{da } n = 2^k) \\
\\
&\Rightarrow T(n) = 2 \cdot n - 1 \in O(n)
\end{aligned}$$

(c) Die Funktion berechnet rekursiv die Folge der Fibonacci-Zahlen in linearem Zeitaufwand.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 5

Abgabe: Montag, **27.05.2013**, 14:00 Uhr

- Die Übungen sollen in Gruppen von zwei bis drei Personen bearbeitet werden.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auf die abgegebenen Lösungen.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auch in die Quellcode-Dateien.
- Geben Sie Ihre Lösungen am **Anfang** der Globalübung, montags, 14:00 Uhr, ab.
- Schicken Sie den jeweiligen Quellcode bitte per **E-Mail** direkt an Ihre/n Tutor/in.
- Geben Sie außerdem den ausgedruckten Quellcode zusammen mit den schriftlichen Lösungen ab.
- Zu spät abgegebene Lösungen werden nicht bewertet.
- Sofern nicht anders gefordert, müssen alle Lösungen und Zwischenschritte kommentiert werden.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Aufgabe 1 (Dynamische Programmierung [**10 Punkte**])

In rekursiven Programmen werden Werte oft unnötigerweise mehrfach berechnet, so z.B. bei der naiven Implementierung einer Funktion zur Berechnung der Fibonacci-Zahlen:

```
fibonacci(n)
    if (n < 2) return n;
    return fibonacci(n-1) + fibonacci(n-2)
```

Man mache sich das an einem Beispiel klar! Bei der dynamischen Programmierung geht man daher umgekehrt vor: Ausgehend vom Basisfall der Rekursion berechnet man weitere Werte und speichert diese für spätere Verwendung ab:

```
fib[0] = 0; // Basisfall der
fib[1] = 1; // Rekursion
for (i = 2; i <= n; ++i)
    fib[i] = fib[i-1] + fib[i-2]
// Ergebnis steht in fib[n]
```

Überzeugen sie sich, dass hier jede Fibonacci-Zahl nur einmal berechnet wird! In der folgenden Aufgabe soll diese Technik nochmals eingeübt werden.
Gegeben seien n verschiedene, positive Münzwerte $a_1, \ldots, a_n \in N^+$ und ein Betrag $m \in N_0$. Gesucht ist die Zahl der möglichen n-Tupel $(k_1, \ldots, k_n)$ mit

$$\sum_{l=1}^{n} k_l a_l = m$$

also die Zahl der Möglichkeiten, m durch die gegebenen Münzwerte darzustellen. Hierbei wollen wir optimistischerweise annehmen, dass von jeder Münzsorte beliebig viele zur Verfügung stehen. Außerdem vereinbaren wir folgenden Sonderfall: Der Betrag $m = 0$ lässt sich immer genau einmal durch 0 Münzen darstellen.
Wir verallgemeinern zunächst das Problem und fragen stattdessen nach der Zahl der Möglichkeiten einen Wert $0 \leq j \leq m$ durch nur $0 \leq i \leq n$ Münzwerte darzustellen. Diese Zahl nennen wir s_{ij}. Offensichtlich erhalten wir dann die Lösung zum ursprünglichen Problem in s_{nm}. Die Werte s_{ij} lassen sich bequem in einer Tabelle speichern deren Zeilen mit i und deren Spalten mit j nummeriert werden.

1. Welchen Wert hat s_{0j} für $0 \leq j \leq m$, d.h. welche Einträge stehen in der ersten Zeile der Tabelle? (Beachten Sie den Sonderfall für $j = 0$.) [1 Punkt]

2. Welchen Wert hat s_{i0} für $1 \leq i \leq n$, d.h. welche Einträge stehen in der ersten Spalte der Tabelle? [1 Punkt]

3. Es seien $n = 3, m = 10, a_1 = 2, a_2 = 5$ und $a_3 = 3$. Stellen Sie die zugehörige Tabelle auf und geben Sie die verschiedenen Möglichkeiten, den Betrag 10 darzustellen an. [2 Punkte]

4. Überlegen Sie sich eine Rekursions-Formel für den (i, j)-ten Eintrag der Tabelle. Was können Sie als Basis-Fall der Rekursion verwenden? [4 Punkte]

5. Skizzieren Sie einen Algorithmus der mittels der Rekursionsformel die Tabelle vollständig ausfüllt. Welche Zeit- und Platzkomplexität hat dieser Algorithmus in Abhängigkeit von n und m? [2 Punkte]

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Aufgabe 2 (Sortieralgorithmen [**10 Punkte**])

(a) In der Vorlesung wurde das *Insertion-Sort*-Verfahren vorgestellt (Foliensatz 2.3.2, Folie 11 ff.). In dem zu dieser Übungsaufgabe bereitgestellten Quellcode finden Sie die Java-Klasse `InsertionSort.java`, die eine leere Methode `sort()` beinhaltet. Beim Instanziieren der Klasse wird das zu sortierende Array übergeben und in das Member-Array `int[] A` kopiert. Implementieren Sie die Methode `sort()` so, dass das Array `A` mit Hilfe des Insertion-Sort Algorithmus sortiert wird. Überprüfen Sie Ihre Ergebnisse durch Ausführen der von uns in der Klasse `MainClass` bereitgestellten `main`-Methode. [2 Punkte]

(b) Insertion-Sort hat eine *Worst-Case*- und *Average-Case*-Komplexität von $O(n^2)$, wobei n die Eingabegröße ist. Wie könnte man die Laufzeit des Algorithmus verbessern? Begründen Sie ihre Antwort und analysieren Sie die *Best*-, *Worst*- und *Average-Case*-Laufzeit. Was fällt Ihnen bezüglich der *Best-Case*-Laufzeit im Vergleich zur normalen Version des Algorithmus auf? [3 Punkte]

(c) Ein Sortieralgorithmus ist "stabil", wenn er die Reihenfolge zweier Elemente mit demselben Sortierschlüsselwert beim Sortieren erhält. Erklären Sie, warum Quick-Sort diese Eigenschaft nicht besitzt und erweitern Sie den Algorithmus so, dass er stabil ist. Sie brauchen hierzu nichts implementieren, sondern lediglich die notwendigen Erweiterungen in Worten erklären und ggf. in Form von Pseudocode notieren. [5 Punkte]

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 5

Abgabe: Montag, **27.05.2013**, 14:00 Uhr

- Die Übungen sollen in Gruppen von zwei bis drei Personen bearbeitet werden.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auf die abgegebenen Lösungen.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auch in die Quellcode-Dateien.
- Geben Sie Ihre Lösungen am **Anfang** der Globalübung, montags, 14:00 Uhr, ab.
- Schicken Sie den jeweiligen Quellcode bitte per **E-Mail** direkt an Ihre/n Tutor/in.
- Geben Sie außerdem den ausgedruckten Quellcode zusammen mit den schriftlichen Lösungen ab.
- Zu spät abgegebene Lösungen werden nicht bewertet.
- Sofern nicht anders gefordert, müssen alle Lösungen und Zwischenschritte kommentiert werden.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Aufgabe 1 (Dynamische Programmierung [**10 Punkte**])

In rekursiven Programmen werden Werte oft unnötigerweise mehrfach berechnet, so z.B. bei der naiven Implementierung einer Funktion zur Berechnung der Fibonacci-Zahlen:

```
fibonacci(n)
    if (n < 2) return n;
    return fibonacci(n-1) + fibonacci(n-2)
```

Man mache sich das an einem Beispiel klar! Bei der dynamischen Programmierung geht man daher umgekehrt vor: Ausgehend vom Basisfall der Rekursion berechnet man weitere Werte und speichert diese für spätere Verwendung ab:

```
fib[0] = 0; // Basisfall der
fib[1] = 1; // Rekursion
for (i = 2; i <= n; ++i)
    fib[i] = fib[i-1] + fib[i-2]
// Ergebnis steht in fib[n]
```

Überzeugen sie sich, dass hier jede Fibonacci-Zahl nur einmal berechnet wird! In der folgenden Aufgabe soll diese Technik nochmals eingeübt werden.
Gegeben seien n verschiedene, positive Münzwerte $a_1, \ldots, a_n \in N^+$ und ein Betrag $m \in N_0$. Gesucht ist die Zahl der möglichen n-Tupel $(k_1, \ldots, k_n)$ mit

$$\sum_{l=1}^{n} k_l a_l = m$$

also die Zahl der Möglichkeiten, m durch die gegebenen Münzwerte darzustellen. Hierbei wollen wir optimistischerweise annehmen, dass von jeder Münzsorte beliebig viele zur Verfügung stehen. Außerdem vereinbaren wir folgenden Sonderfall: Der Betrag $m = 0$ lässt sich immer genau einmal durch 0 Münzen darstellen.
Wir verallgemeinern zunächst das Problem und fragen stattdessen nach der Zahl der Möglichkeiten einen Wert $0 \leq j \leq m$ durch nur $0 \leq i \leq n$ Münzwerte darzustellen. Diese Zahl nennen wir s_{ij}. Offensichtlich erhalten wir dann die Lösung zum ursprünglichen Problem in s_{nm}. Die Werte s_{ij} lassen sich bequem in einer Tabelle speichern deren Zeilen mit i und deren Spalten mit j nummeriert werden.

1. Welchen Wert hat s_{0j} für $0 \leq j \leq m$, d.h. welche Einträge stehen in der ersten Zeile der Tabelle? (Beachten Sie den Sonderfall für $j = 0$.) [1 Punkt]

2. Welchen Wert hat s_{i0} für $1 \leq i \leq n$, d.h. welche Einträge stehen in der ersten Spalte der Tabelle? [1 Punkt]

3. Es seien $n = 3, m = 10, a_1 = 2, a_2 = 5$ und $a_3 = 3$. Stellen Sie die zugehörige Tabelle auf und geben Sie die verschiedenen Möglichkeiten, den Betrag 10 darzustellen an. [2 Punkte]

4. Überlegen Sie sich eine Rekursions-Formel für den (i, j)-ten Eintrag der Tabelle. Was können Sie als Basis-Fall der Rekursion verwenden? [4 Punkte]

5. Skizzieren Sie einen Algorithmus der mittels der Rekursionsformel die Tabelle vollständig ausfüllt. Welche Zeit- und Platzkomplexität hat dieser Algorithmus in Abhängigkeit von n und m? [2 Punkte]

Lösungsvorschlag

1. Es ist
$$s_{0j} = \begin{cases} 1 & \text{falls} \quad j = 0 \quad \text{(der Sonderfall)} \\ 0 & \text{sonst} \end{cases}$$
denn ohne Münzen kann man auch keinen positiven Betrag darstellen!

2. Ohne Münzen läßt sich nur die 0 darstellen, also $s_{i0} = 1$.

3. Wir haben

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

und es ist
$$\begin{aligned} 10 &= 2+2+2+2+2 \\ &= 2+3+5 \\ &= 2+2+3+3 \\ &= 5+5 \end{aligned}$$

4. Für $i > 0$ ist
$$s_{ij} = s_{i-1,j} + \begin{cases} s_{i,j-a_i} & \text{falls} \quad j - a_i \geq 0 \\ 0 & \text{sonst} \end{cases}$$
Basisfall ist $s_{00} = 1$ und $s_{0j} = 0, j > 0$, also die oberste Zeile der Tabelle.

5. Der Algorithmus berechnet zunächst die oberste Zeile der Tabelle. Dann werden die übrigen Einträge mit Hilfe der Rekursionsformel von links nach rechts und von oben nach unten ("Leserichtung") aufgefüllt. Offensichtlich ist Zeitkomplexität = Platzkomplexität = Größe der Tabelle = $O(nm)$.

Aufgabe 2 (Sortieralgorithmen [**10 Punkte**])

(a) In der Vorlesung wurde das *Insertion-Sort*-Verfahren vorgestellt (Foliensatz 2.3.2, Folie 11 ff.). In dem zu dieser Übungsaufgabe bereitgestellten Quellcode finden Sie die Java-Klasse `InsertionSort.java`, die eine leere Methode `sort()` beinhaltet. Beim Instanziieren der Klasse wird das zu sortierende Array übergeben und in das Member-Array `int[] A` kopiert. Implementieren Sie die Methode `sort()` so, dass das Array `A` mit Hilfe des Insertion-Sort Algorithmus sortiert wird. Überprüfen Sie Ihre Ergebnisse durch Ausführen der von uns in der Klasse `MainClass` bereitgestellten `main`-Methode. [2 Punkte]

(b) Insertion-Sort hat eine *Worst-Case*- und *Average-Case*-Komplexität von $O(n^2)$, wobei n die Eingabegröße ist. Wie könnte man die Laufzeit des Algorithmus verbessern? Begründen Sie ihre Antwort und analysieren Sie die *Best-*, *Worst-* und *Average-Case*-Laufzeit. Was fällt Ihnen bezüglich der *Best-Case*-Laufzeit im Vergleich zur normalen Version des Algorithmus auf? [3 Punkte]

(c) Ein Sortieralgorithmus ist "stabil", wenn er die Reihenfolge zweier Elemente mit demselben Sortierschlüsselwert beim Sortieren erhält. Erklären Sie, warum Quick-Sort diese Eigenschaft nicht besitzt und erweitern Sie den Algorithmus so, dass er stabil ist. Sie brauchen hierzu nichts implementieren, sondern lediglich die notwendigen Erweiterungen in Worten erklären und ggf. in Form von Pseudocode notieren. [5 Punkte]

Lösungsvorschlag

(a) Siehe Quellcode.

(b) Das Suchen der Stelle innerhalb der sortierten Liste am Anfang des Arrays, an der das jeweils nächste Element eingefügt werden soll, geschieht in linearem Zeitaufwand, also $O(n)$. Dies ist leicht nachzuvollziehen: Die sortierte Liste am Anfang des Arrays hat zu Beginn die Länge eins und wächst mit jedem Durchlauf der äußeren `while`-Schleife um ein Element. Dementsprechend hat sie in Durchlauf k die Größe k.

Im besten Falle (*Best-Case*) ist das zu sortierende Array bereits sortiert und das einzufügende Element muss lediglich mit einem, dem in der sortierten Liste am weitesten rechts stehenden Element verglichen werden und wird dort direkt einsortiert. Dies hat offensichtlich konstante Laufzeitkomplexität, sprich $O(1)$. Bei n Elementen kommen wir in dem Falle also auf eine Laufzeit von $O(n)$. Im schlimmsten Falle (*Worst-Case*) ist die Liste invers sortiert. Dann muss das einzusortierende Element in Durchlauf k mit allen k Elementen in der sortierten Liste verglichen werden. Bei n zu sortierenden Elementen ergibt sich folgende Komplexität:

$$\sum_{i=1}^{n} i = \frac{(n+1)\cdot n}{2} \leq c \cdot n^2 \Rightarrow \sum_{i=1}^{n} i \in O(n^2).$$

Das Suchen der Stelle in der sortierten Liste, an der ein Element einsortiert werden soll, könnte nun beschleunigt werden, indem man anstatt einer linearen Suche eine Binärsuche verwendet. Die Binärsuche funktioniert wie folgt:

```
function BinarySearch(l, r, h)          ▷ Suche zwischen Position l und r
    if r < l then
        return Error
    end if
    if l = r − 1 then
        if Wert[l] < h then
            return r                                  ▷ Position gefunden
        else
            return l                                  ▷ Position gefunden
        end if
    end if
    m ← l + ⌊(r−l)/2⌋
    if Wert[m] < h then
        return BinarySearch(m, r, h)
    end if
    return BinarySearch(l, m, h)                      ▷ Wert[m] ≥ h
end function
```

In dem Fall wird der Suchraum mit jedem rekursiven Aufruf halbiert, bis das gesuchte Element (die gesuchte Position) gefunden ist. Allerdings bestünde jetzt das Problem, dass beim Einfügen des einzusortierenden Elements an der gewünschten im Array alle nachfolgenden Elemente um eine Stelle nach hinten kopiert werden müssten. Um dies zu vermeiden, könnte man überlegen, eine (doppelt) verkettete Liste als zugrunde liegende Datenstruktur zu verwenden. Allerdings kann man in dem Fall nicht in konstantem Zeitaufwand an eine beliebige Stelle in der Folge springen, da man immer den Links zum jeweiligen nächsten/vorherigen Element folgend muss, um an eine bestimmte Stelle zu gelangen. Von daher benutzen wir eine *zusätzliche* Datenstruktur, die die guten Eigenschaften beider Ansätze verbindet: wir fügen die Elemente der Reihe nach in einen **binären Suchbaum** ein. Der Speicheraufwand verdoppelt sich nun offensichtlich und beträgt $2n \in O(n)$. Der Zeitaufwand für das Einfügen aller Elemente in den Suchbaum beträgt

$$\sum_{i=0}^{n-1} \log_2 i \in O(n \log n),$$

Da der Baum zu Beginn Größe 0 hat und mit jeder Einfüge-Operation um ein Element wächst. Jedes Element, das in einen Suchbaum eingefügt wird, wird als Blatt an den Baum gehängt. Aus diesem Grund gilt die o.g. Laufzeitabschätzung für das Einfügen von Elementen in jedem Fall, unabhängig von der Sortierung der Elemente in der ursprünglichen Folge. Vergleichen wir unsere Erweiterung mit dem *Best-Case* der Originalimplementierung, so sehen wir, dass die Originalmethode eine bessere Laufzeit hat, nämlich $O(n)$. In allen anderen Fällen hat die erweiterte Variante eine bessere Laufzeit.

(c) In jedem Partitioniervorgang wird die interne Reihenfolge zweier gleichgroßer Elemente invertiert, da die beiden Zeiger jeweils von außen nach innen laufen und Elemente vertauschen, wenn das des linken Zeigers größer und das des rechten Zeigers kleiner ist, als das Pivotelement. Das bedeutet, dass die Stabilität des Algorithmus maßgeblich von der Anzahl der Partitioniervorgänge abhängt. Darauf haben wir allerdings keinen Einfluss. Aus diesem Grund könnte man den Quick-Sort-Algorithmus wie folgt erweitern: Man führe einen zweiten Sortierschlüssel ein, der die Position eines jeden Elements in der initialen Folge enthält. Nach dem eigentlichen Sortiervorgang wird auf jedes Intervall in der sortierten Folge, das aus Elementen desselben Wertes besteht, ein weiterer Sortierdurchlauf angewendet. Für diesen Vorgang wählen wir nun den neu eingeführten Sortierschlüssel für die Vergleiche. Im Worst-Case besteht die zu sortierende Folge aus n Elementen desselben Wertes. In dem Fall wird der Quick-Sort-Algorithmus genau zweimal auf die gesamte Folge angewendet: Einmal zum Sortieren nach dem Primärschlüssel und ein weiteres Mal zum Sortieren nach dem neu eingeführten Sekundärschlüssel. Es ergibt sich eine Gesamtkomplexität von $2 \cdot O(n \log n) = O(n \log n)$. Im besten Fall gibt es in der Folge keine Duplikate und wir sind nach dem initialen Sortiervorgang bereits fertig.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 6

Abgabe: Montag, **10.06.2013**, 14:00 Uhr

- Die Übungen sollen in Gruppen von zwei bis drei Personen bearbeitet werden.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auf die abgegebenen Lösungen.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auch in die Quellcode-Dateien.
- Geben Sie Ihre Lösungen am **Anfang** der Globalübung, montags, 14:00 Uhr, ab.
- Schicken Sie den jeweiligen Quellcode bitte per **E-Mail** direkt an Ihre/n Tutor/in.
- Geben Sie außerdem den ausgedruckten Quellcode zusammen mit den schriftlichen Lösungen ab.
- Zu spät abgegebene Lösungen werden nicht bewertet.
- Sofern nicht anders gefordert, müssen alle Lösungen und Zwischenschritte kommentiert werden.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Aufgabe 1 (*Summe von Array Werten* [**10 Punkte**])

In dieser Aufgabe geht es darum, einen Algorithmus zu entwerfen, der in einem Byte-Array zwei Werte sucht, deren Summe einen bestimmten Betrag hat.
Sei A ein Byte-Array der Länge n, und sei m der gesuchte Betrag.
Entwerfen Sie einen Algorithmus, der linear viele Schritte benötigt, um zu entscheiden, ob in dem Array zwei Indizes i und j existieren, so dass $A[i] + A[j] = m$ gilt. Falls solche Indizes existieren, soll der Algorithmus **True** zurückgeben, ansonsten **False**.

(a) Beschreiben Sie Ihre Idee für den Algorithmus und geben Sie den Algorithmus in Pseudocode an. [6 Punkte]

(b) Beweisen Sie, dass Ihr Algorithmus die Laufzeitkomplexität $O(n)$ hat. [4 Punkte]

Aufgabe 2 (*Merge-Sort* [**10 Punkte**])

Der klassische Merge-Sort Algorithmus sortiert ein Array nach dem Divide-and-Conquer Prinzip, indem er das Array in zwei (optimalerweise) gleich große, kleinere Arrays aufteilt, diese rekursiv sortiert, und danach die beiden sortierten Teilarrays in linearer Zeit zu einem sortierten Array zusammenfügt.
Eine Variation dieses Algorithmus ist der *ternäre Merge-Sort*, welcher das Array nicht in zwei, sondern in drei kleinere Arrays aufteilt. In dieser Aufgabe soll der klassische binäre Merge-Sort hinsichtlich der Laufzeitkomplexität mit dem ternären Merge-Sort verglichen werden.

(a) Stellen Sie Rekursionsgleichungen für die Anzahl der Vergleiche zweier Array-Elemente des binären und des ternären Merge-Sort sowohl im *Worst-Case* als auch im *Best-Case* auf. Finden Sie explizite (nicht-rekursive) Darstellungen der Rekursionsgleichungen und vergleichen Sie. Welche Variante würden Sie vorziehen? [4 Punkte]

(b) In dem von uns bereitgestellten Code finden Sie zwei Klassen. Die Klasse `MergeSort.java` stellt alle notwendigen Methoden zur Verfügung, um ein Array `A` mit dem binären oder ternären Merge-Sort zu sortieren. Ihre Aufgabe ist es, die Rümpfe der Methoden `mergeSortBinary`, `mergeBinary`, `mergeSortTernary` und `mergeTernary` zu ergänzen.

Die Methode `mergeSortBinary(int l, int r)` sortiert das Teilarray `A[l..r]` rekursiv mit der binären Merge-Sort Variante. Sie verwendet die Methode `mergeBinary(int l, int m, int r)`, um die beiden rekursiv sortierten Teilarrays `A[l..m-1]` und `A[m..r-1]` zu einem sortierten Gesamtarray zusammenzufügen.

Analog dazu sortiert die Methode `mergeSortTernary(int l, int r)` das Teilarray `A[l..r]` rekursiv mit der ternären Merge-Sort Variante. Sie verwendet die Methode `mergeTernary(int l, int m1, int m2, int r)`, um die drei rekursiv sortierten Teilarrays `A[l..m1-1]`, `A[m1..m2-1]` und `A[m2..r-1]` zu einem sortierten Gesamtarray zusammenzufügen.

Die Klasse `MainClass.java` erstellt ein Array mit 5 Millionen zufallsgenerierten Elementen, sortiert es einmal mit der binären Merge-Sort und einmal mit der ternären Merge-Sort-Variante. Anschließend gibt sie aus, wie lange die jeweilige Variante für die Sortierung des Arrays benötigt hat und ob das Array korrekt sortiert wurde. [4 Punkte]

(c) Messen Sie wiederholt die Zeiten ihrer Implementierung aus Teilaufgabe (b). Erklären Sie die gemessenen Zeitunterschiede und vergleichen Sie diese mit Ihren Resultaten aus Teilaufgabe (a). **Hinweis**: Stellen Sie Rekursionsgleichungen für die Anzahl der Aufrufe der jeweiligen `merge`-Methoden auf. [2 Punkte]

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 6

Abgabe: Montag, **10.06.2013**, 14:00 Uhr

- Die Übungen sollen in Gruppen von zwei bis drei Personen bearbeitet werden.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auf die abgegebenen Lösungen.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auch in die Quellcode-Dateien.
- Geben Sie Ihre Lösungen am **Anfang** der Globalübung, montags, 14:00 Uhr, ab.
- Schicken Sie den jeweiligen Quellcode bitte per **E-Mail** direkt an Ihre/n Tutor/in.
- Geben Sie außerdem den ausgedruckten Quellcode zusammen mit den schriftlichen Lösungen ab.
- Zu spät abgegebene Lösungen werden nicht bewertet.
- Sofern nicht anders gefordert, müssen alle Lösungen und Zwischenschritte kommentiert werden.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Aufgabe 1 (Summe von Array Werten [**10 Punkte**])

In dieser Aufgabe geht es darum, einen Algorithmus zu entwerfen, der in einem Byte-Array zwei Werte sucht, deren Summe einen bestimmten Betrag hat.
Sei A ein Byte-Array der Länge n, und sei m der gesuchte Betrag.
Entwerfen Sie einen Algorithmus, der linear viele Schritte benötigt, um zu entscheiden, ob in dem Array zwei Indizes i und j existieren, so dass $A[i] + A[j] = m$ gilt. Falls solche Indizes existieren, soll der Algorithmus **True** zurückgeben, ansonsten **False**.

(a) Beschreiben Sie Ihre Idee für den Algorithmus und geben Sie den Algorithmus in Pseudocode an.

(b) Beweisen Sie, dass Ihr Algorithmus die Laufzeitkomplexität $O(n)$ hat.

Lösungsvorschlag

(a)
```
function A1(A, m)
    M ← Boolean[256]  ▷ M markiert, welche Werte bisher gesehen wurden.
    for i ← 0, 255 do
        M[i] ← False
    end for
    for i ← 0, n − 1 do
        M[A[i]] ← True        ▷ Markieren, dass der Wert A[i] gesehen wurde.
        j ← m − A[i]
        if 0 ≤ j ≤ 255 ∧ M[j] then
            return True
        end if
    end for
    return False
end function
```

(b) Die erste Schleife im Algorithmus $A1$ hat eine konstante Zahl an Durchläufen. Die zweite Schleife hat n Durchläufe, die jeweils maximal eine konstante Zahl an Schritten benötigen. Somit benötigt der Algorithmus $O(n)$ viele Schritte zur Berechnung.

Aufgabe 2 (*Merge-Sort* [**10 Punkte**])

Der klassische Merge-Sort Algorithmus sortiert ein Array nach dem Divide-and-Conquer Prinzip, indem er das Array in zwei (optimalerweise) gleich große, kleinere Arrays aufteilt, diese rekursiv sortiert, und danach die beiden sortierten Teilarrays in linearer Zeit zu einem sortierten Array zusammenfügt.
Eine Variation dieses Algorithmus ist der *ternäre Merge-Sort*, welcher das Array nicht in zwei, sondern in drei kleinere Arrays aufteilt. In dieser Aufgabe soll der klassische binäre Merge-Sort hinsichtlich der Laufzeitkomplexität mit dem ternären Merge-Sort verglichen werden.

(a) Stellen Sie Rekursionsgleichungen für die Anzahl der Vergleiche zweier Array-Elemente des binären und des ternären Merge-Sort sowohl im *Worst-Case* als auch im *Best-Case* auf. Finden Sie explizite (nicht-rekursive) Darstellungen der Rekursionsgleichungen und vergleichen Sie. Welche Variante würden Sie vorziehen? [4 Punkte]

(b) In dem von uns bereitgestellten Code finden Sie zwei Klassen. Die Klasse `MergeSort.java` stellt alle notwendigen Methoden zur Verfügung, um ein Array `A` mit dem binären oder ternären Merge-Sort zu sortieren. Ihre Aufgabe ist es, die Rümpfe der Methoden `mergeSortBinary`, `mergeBinary`, `mergeSortTernary` und `mergeTernary` zu ergänzen.

Die Methode `mergeSortBinary(int l, int r)` sortiert das Teilarray `A[l..r]` rekursiv mit der binären Merge-Sort Variante. Sie verwendet die Methode `mergeBinary(int l, int m, in tr)`, um die beiden rekursiv sortierten Teilarrays `A[l..m-1]` und `A[m..r-1]` zu einem sortierten Gesamtarray zusammenzufügen.

Analog dazu sortiert die Methode `mergeSortTernary(int l, int r)` das Teilarray `A[l..r]` rekursiv mit der ternären Merge-Sort Variante. Sie verwendet die Methode `mergeTernary(int l, int m1, int m2, int r)`, um die drei rekursiv sortierten Teilarrays `A[l..m1-1]`, `A[m1..m2-1]` und `A[m2..r-1]` zu einem sortierten Gesamtarray zusammenzufügen.

Die Klasse `MainClass.java` erstellt ein Array mit 5 Millionen zufallsgenerierten Elementen, sortiert es einmal mit der binären Merge-Sort und einmal mit der ternären Merge-Sort-Variante. Anschließend gibt sie aus, wie lange die jeweilige Variante für die Sortierung des Arrays benötigt hat und ob das Array korrekt sortiert wurde. [4 Punkte]

(c) Messen Sie wiederholt die Zeiten ihrer Implementierung aus Teilaufgabe (b). Erklären Sie die gemessenen Zeitunterschiede und vergleichen Sie diese mit Ihren Resultaten aus Teilaufgabe (a). **Hinweis**: Stellen Sie Rekursionsgleichungen für die Anzahl der Aufrufe der jeweiligen `merge`-Methoden auf. [2 Punkte]

Lösungsvorschlag

(a) *Worst-Case binär:*

$$\begin{aligned}
T(1) &= 0 \\
T(n) &= 2T(n/2) + n - 1 \\
&= 2\left[2T(n/4) + n/2 - 1\right] + n - 1 \\
&= 4T(n/4) + 2n - 1 - 2 \\
&\quad\vdots \\
&= n\log_2 n - \sum_{i=0}^{\log_2 n - 1} 2^i \\
&= n\log_2 n - n + 1
\end{aligned}$$

Best-Case binär:

$$\begin{aligned}
T(1) &= 0 \\
T(n) &= 2T(n/2) + n/2 \\
&= 2\left[2T(n/4) + n/4\right] + n/2 \\
&= 4T(n/4) + n \\
&\quad\vdots \\
&= \frac{n}{2}\log_2 n
\end{aligned}$$

Worst-Case ternär:

$$\begin{aligned}
T(1) &= 0 \\
T(n) &= 3T(n/3) + 2(n-2) + 1 \\
&= 3\left[3T(n/9) + 2(n/3 - 2) + 1\right] + 2n - 4 + 1 \\
&= 9T(n/9) + (2n + 2n) - (4 + 12) + (1 + 3) \\
&\quad\vdots \\
&= 2n\log_3 n - 4\sum_{i=0}^{\log_3 n - 1} 3^i + \sum_{i=0}^{\log_3 n - 1} 3^i \\
&= 2n\log_3 n - 3\sum_{i=0}^{\log_3 n - 1} 3^i \\
&= 2n\log_3 n - 3\frac{n-1}{2} \\
&= 2n\log_3 n - 1.5n + 1.5
\end{aligned}$$

Best-Case ternär:

$$\begin{aligned} T(1) &= 0 \\ T(n) &= 3T(n/3) + 2n/3 + n/3 \\ &= 3\left[3T(n/9) + 2n/9 + n/9\right] + n \\ &= 9T(n/9) + n + n \\ &\vdots \\ &= n \log_3 n \end{aligned}$$

Im Best-Case sind bei der ternären Variante

$$\frac{n \log_3 n}{n/2 \log_2 n} = 2\frac{\ln n / \ln 3}{\ln n / \ln 2} = 2\frac{\ln 2}{\ln 3} \approx 1.26$$

mal soviele Vergleiche notwendig wie bei der binären Variante.

Im Worst-Case sind bei der ternären Variante für $n \to \infty$

$$\begin{aligned} &\lim_{n\to\infty} \frac{2n \log_3 n - 1.5n + 1.5}{n \log_2 n - n + 1} \\ \overset{\text{L'H.}}{=} &\lim_{n\to\infty} \frac{2 \log_3 n + 2/\ln 3 - 1.5}{\log_2 n + 1/\ln 2 - 1} \\ \overset{\text{L'H.}}{=} &\lim_{n\to\infty} 2\frac{\ln 2}{\ln 3} \approx 1.26 \end{aligned}$$

mal soviele Vergleiche notwendig wie bei der binären Variante.

Daher wäre aufgrund der geringeren Anzahl der Vergleiche die binäre Variante vorzuziehen.

(b) Siehe `MergeSort.java`.

(c) Nach 100 Wiederholungen benötigte der binäre Merge Sort durschnittlich 2.54 Sekunden, der ternäre Merge Sort benötigte durchschnittlich 2.07 Sekunden. Dies steht im Widerspruch zu den Resultaten aus Teilaufgabe (a), welche vermuten ließen, dass die binäre der ternären Variante überlegen ist.

Der Grund dafür ist, dass bei der ternären Variante deutlich weniger Aufrufe der `merge`-Methode stattfinden. Dies lässt sich auch durch Aufstellen von Rekursionsgleichungen für die Anzahl der Funktionsaufrufe verifizieren:

Funktionsaufrufe binär:

$$\begin{aligned} T(1) &= 0 \\ T(n) &= 2T(n/2) + 1 \\ &= 2\left[2T(n/4) + 1\right] + 1 \\ &= 4T(n/4) + 2 + 1 \\ &\vdots \\ &= \sum_{i=0}^{\log_2 n - 1} 2^i \\ &= n - 1 \end{aligned}$$

Funktionsaufrufe ternär:

$$\begin{aligned} T(1) &= 0 \\ T(n) &= 3T(n/3) + 1 \\ &= 3\left[3T(n/9) + 1\right] + 1 \\ &= 9T(n/9) + 3 + 1 \\ &\vdots \\ &= \sum_{i=0}^{\log_3 n - 1} 3^i \\ &= \frac{n-1}{2} \end{aligned}$$

Die binäre Variante benötigt also doppelt so viele Aufrufe der `merge` Funktion wie die ternäre Variante. In jedem dieser Aufrufe wird Speicher für das Array `B` allokiert und zwei mal über den betrachteten Arraybereich iteriert (einmal beim zusammenfügen der beiden Teilarrays in das Array `B` und einmal beim Kopieren des Arrays `B` in das Array `A`). Einen noch detaillierteren Vergleich beider Varianten erhält man, wenn man die Gesamtanzahl der Arrayzugriffe ermittelt. Dies lässt sich erneut durch Aufstellen und Lösen von Rekursionsgleichungen (hier exemplarisch für den Best-case) bewerkstelligen:

Arrayzugriffe binär (Best-case):

$$\begin{aligned} T(1) &= 0 \\ T(n) &= 2T(n/2) + \frac{n}{2} \cdot 4 + \frac{n}{2} \cdot 2 + 2n \\ &= 2T(n/2) + 5n \\ &\vdots \\ &= 5n \cdot \log_2 n \end{aligned}$$

Arrayzugriffe ternär (Best-case):

$$\begin{aligned} T(1) &= 0 \\ T(n) &= 3T(n/3) + \frac{n}{3} \cdot 6 + \frac{n}{3} \cdot 4 + \frac{n}{3} \cdot 2 + 2n \\ T(n) &= 3T(n/3) + 6n \\ &\vdots \\ &= 6n \cdot \log_3 n \end{aligned}$$

Damit sind bei der binären Variante $\frac{5 \ln 3}{6 \ln 2} \approx 1.32$ mal soviele Arrayzugriffe notwendig wie bei der ternären, wodurch die längere Laufzeit zu erklären ist.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 7

Abgabe: Montag, **17.06.2013**, 14:00 Uhr

- Die Übungen sollen in Gruppen von zwei bis drei Personen bearbeitet werden.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auf die abgegebenen Lösungen.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auch in die Quellcode-Dateien.
- Geben Sie Ihre Lösungen am **Anfang** der Globalübung, montags, 14:00 Uhr, ab.
- Schicken Sie den jeweiligen Quellcode bitte per **E-Mail** direkt an Ihre/n Tutor/in.
- Geben Sie außerdem den ausgedruckten Quellcode zusammen mit den schriftlichen Lösungen ab.
- Zu spät abgegebene Lösungen werden nicht bewertet.
- Sofern nicht anders gefordert, müssen alle Lösungen und Zwischenschritte kommentiert werden.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Aufgabe 1 (*Hashing* [**10 Punkte**])

(a) Güte von Hash-Funktionen
Betrachten Sie die folgenden Funktionen, die ein Wort $s = a_1 \dots a_n$ auf einen Hash-Wert zwischen 0 und m abbilden. Dabei sind die a_i als ASCII-Werte gegeben, also $0 \leq a_i \leq 127$ für alle i.

- $h_1(s) = n \mod m$
- $h_2(s) = (\sum_{k=1}^{n} a_k) \mod m$
- $h_3(s) = (\sum_{k=1}^{n} k \cdot a_k^2) \mod m$
- $h_4(s) = ((h_{good}(s)^{p-1} \mod p) \mod m)$

wobei $h_{good}(s)$ eine Hash-Funktion ist, die alle wichtigen an eine Hash-Funktion gestellten Eigenschaften (einfache Berechenbarkeit, Surjektivität und Streuung) erfüllt und p eine Primzahl ist. Erläutern Sie, inwiefern die Funktionen h_i ($1 \leq i \leq 4$) die drei genannten Eigenschaften einer guten Hash-Funktion erfüllen. [2 Punkte]

Hinweis: Wenn Sie sich nicht sicher sind, wie sich die Hash-Funktionen verhalten, ist es ratsam, sie auf ein paar Beispiel-Wörter anzuwenden. Für die vierte Hash-Funktion können Sie dabei beispielsweise die drei anderen Hash-Funktionen anstelle von h_{good} verwenden. Dies dient jedoch lediglich Ihrer Intuition und ist für die Bearbeitung dieser Aufgabe nicht zwingend erforderlich.

(b) Einfaches Hashing
Gegeben sei eine Hash-Tabelle der Größe 19 und die folgenden beiden Hash-Funktionen über der Universalmenge $U = \{0, \dots, 99\}$:

- $h_1(x) =$ Quersumme von x
- $h_2(x) = x \mod 19$

1. Fügen Sie die Werte $12, 99, 21, 76, 23, 30$ sowohl mit h_1 als auch h_2 mittels
   (i) Hashing mit Verkettung
   (ii) Hashing mit linearem Sondieren

   in jeweils eine Tabelle ein (es sind also 4 Tabellen zu erstellen). Geben Sie die nicht-leeren Teile der Tabellen nach jedem Einfügeschritt an. [2 Punkte]
2. Löschen Sie anschließend nacheinander die Werte 21, 12 und 76 aus allen Tabellen aus dem vorigen Aufgabenteil. Geben Sie die nicht-leeren Teile der Tabellen nach jedem Löschschritt an. Erläutern Sie, welches Vorgehen dafür jeweils nötig ist. [2 Punkte]
3. Suchen Sie in den Tabellen, die aus der letzten Teilaufgabe (nach dem Löschen) resultierten, nach dem Wert 12. Erläutern Sie, welches Vorgehen dafür jeweils nötig ist. [1 Punkt]

(c) Doppeltes Hashing
In der Vorlesung haben Sie unter anderem auch doppeltes Hashing kennen gelernt, bei dem eine zusätzliche Hash-Funktion verwendet wird, um Sondierungsschritte durchzuführen. In dieser Teilaufgabe sollen Sie dieses Verfahren und eine Variante davon implementieren. Dazu sollen Sie den von uns bereit gestellten Code vervollständigen. Letzterer enthält drei Klassen. Die Datei `MainClass.java` enthält die Hauptklasse, die zum Testen genutzt werden kann und nicht modifiziert werden soll. Ebenso soll die Datei `Hash.java` nicht modifiziert werden. Diese stellt eine einfache Datenstruktur für Hash-Werte mit einer zusätzlichen Markierung bereit, um angeben zu können, ob ein Hash-Eintrag frei (free), belegt (used) oder entfernt worden (deleted) ist. In der Datei `Hashing.java` soll von Ihnen die Methode `insert2` vervollständigt werden. Die Methode `insert1` implementiert bereits eine Einfüge-Operation nach doppeltem Hashing mit der Sondierungs-Funktion $h(k, 0) = h_1(k) \mod m$ und $h(k, i) = (h_1(k) + h_2(k) + i - 1) \mod m$ für $i > 0$, wobei h_1 und h_2 Hash-Funktionen sind (**Achtung: Die Folien zum doppelten Hashing wurden geändert – die frühere Version** $h(k, i) = (h_1(k) + h_2(k) \times i) \mod m$ **führt zu Nicht-Terminierung der Kollisionsbehandlung, falls** $h_2(k) = 0$). Die von Ihnen zu vervollständigende Methode soll im Gegensatz dazu bei einer Kollision die nächste Sondierungsposition sowohl für das neu einzufügende als auch für das bereits enthaltene Element an der Kollisionsposition berechnen. Sollte das alte Element an seiner neuen Position ohne weitere Sondierungen einzufügen sein, während das neue Element weitere Sondierungen benötigen würde, wird das alte Element an seine neue Position verschoben und das neue Element an der ursprünglichen Position des alten Elementes eingefügt. Ansonsten wird rekursiv versucht, das neue Element an der nächsten Sondierungsposition einzufügen.

Etwas formaler ausgedrückt funktioniert die von Ihnen zu implementierende Hashing-Variante für ein Element k_1 im i-ten Sondierungsschritt wie folgt:

- Ist $h(k_1, i)$ frei, so füge k_1 an dieser Position ein.
- Ist $h(k_1, i)$ belegt mit einem Element k_2, $h(k_1, i + 1)$ ebenfalls belegt, k_2 wurde mit j Sondierungsschritten eingefügt (d. h. $h(k_1, i) = h(k_2, j)$) und $h(k_2, j + 1)$ ist frei, so füge k_2 an der Position $h(k_2, j + 1)$ ein und k_1 an der bisherigen Position von k_2.
- Ansonsten fahre mit der Sondierung für k_1 und $i + 1$ fort.

Die von Ihnen zu ergänzenden Teile sind im Code durch Kommentare markiert. [2 Punkte]

(d) Experimenteller Vergleich
Die Testklasse gibt für das bereits vorgegebene und das von Ihnen ergänzte Hash-Verfahren aus, wie viele Kollisionen beim Einfügen von 500000 zufällig gewählten Elementen in eine anfangs leere Hash-Tabelle der Größe 1000000 entstehen. Messen Sie einige Male die Kollisionszahlen und erklären Sie den gemessenen Unterschied. [1 Punkt]

Aufgabe 2 (*Selektion* [**10 Punkte**])

Gegeben sei eine Folge von n unsortierten Integer-Zahlen.

(a) Entwerfen Sie einen Algorithmus, welcher das kleinste Element der Folge mit $n-1$ Vergleichen findet. Vergleiche, welche nicht die Folgenelemente selbst betreffen, z. B. Vergleiche zwischen den Folgenindizes, werden nicht dazugezählt. Außerdem soll jedes einzelne Element maximal $\lceil \log_2 n \rceil$ mal verglichen werden. Zeigen Sie, dass Ihr Algorithmus höchstens die geforderte Anzahl von Vergleichen benötigt. [5 Punkte]

(b) Erweitern Sie das Verfahren so, dass es das zweitkleinste Element von n Elementen im Worst-Case mit $n + \lceil \log_2 n \rceil - 2$ Vergleichen bestimmt. Zeigen Sie, dass Ihr Algorithmus höchstens die geforderte Anzahl von Vergleichen benötigt. Wenden Sie Ihr Verfahren auf die Elemente

$$10, 5, 3, 2, 8, 7, 1, 6, 9$$

an. Stellen Sie dabei die notwendigen Schritte in geeigneter Form dar. [5 Punkte]

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 7

Abgabe: Montag, **17.06.2013**, 14:00 Uhr

- Die Übungen sollen in Gruppen von zwei bis drei Personen bearbeitet werden.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auf die abgegebenen Lösungen.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auch in die Quellcode-Dateien.
- Geben Sie Ihre Lösungen am **Anfang** der Globalübung, montags, 14:00 Uhr, ab.
- Schicken Sie den jeweiligen Quellcode bitte per **E-Mail** direkt an Ihre/n Tutor/in.
- Geben Sie außerdem den ausgedruckten Quellcode zusammen mit den schriftlichen Lösungen ab.
- Zu spät abgegebene Lösungen werden nicht bewertet.
- Sofern nicht anders gefordert, müssen alle Lösungen und Zwischenschritte kommentiert werden.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Aufgabe 1 (*Hashing* [**10 Punkte**])

(a) Güte von Hash-Funktionen
Betrachten Sie die folgenden Funktionen, die ein Wort $s = a_1 \dots a_n$ auf einen Hash-Wert zwischen 0 und m abbilden. Dabei sind die a_i als ASCII-Werte gegeben, also $0 \leq a_i \leq 127$ für alle i.

- $h_1(s) = n \mod m$
- $h_2(s) = (\sum_{k=1}^{n} a_k) \mod m$
- $h_3(s) = (\sum_{k=1}^{n} k \cdot a_k^2) \mod m$
- $h_4(s) = ((h_{good}(s)^{p-1} \mod p) \mod m)$

wobei $h_{good}(s)$ eine Hash-Funktion ist, die alle wichtigen an eine Hash-Funktion gestellten Eigenschaften (einfache Berechenbarkeit, Surjektivität und Streuung) erfüllt und p eine Primzahl ist. Erläutern Sie, inwiefern die Funktionen h_i ($1 \leq i \leq 4$) die drei genannten Eigenschaften einer guten Hash-Funktion erfüllen. [3 Punkte]

Hinweis: Wenn Sie sich nicht sicher sind, wie sich die Hash-Funktionen verhalten, ist es ratsam, sie auf ein paar Beispiel-Wörter anzuwenden. Für die vierte Hash-Funktion können Sie dabei beispielsweise die drei anderen Hash-Funktionen anstelle von h_{good} verwenden. Dies dient jedoch lediglich Ihrer Intuition und ist für die Bearbeitung dieser Aufgabe nicht zwingend erforderlich.

(b) Einfaches Hashing
Gegeben sei eine Hash-Tabelle der Größe 19 und die folgenden beiden Hash-Funktionen über der Universalmenge $U = \{0, \dots, 99\}$:

- $h_1(x) =$ Quersumme von x
- $h_2(x) = x \mod 19$

1. Fügen Sie die Werte $12, 99, 21, 76, 23, 30$ sowohl mit h_1 als auch h_2 mittels
   (i) Hashing mit Verkettung
   (ii) Hashing mit linearem Sondieren

   in jeweils eine Tabelle ein (es sind also 4 Tabellen zu erstellen). Geben Sie die nicht-leeren Teile der Tabellen nach jedem Einfügeschritt an. [3 Punkte]
2. Löschen Sie anschließend nacheinander die Werte 21, 12 und 76 aus allen Tabellen aus dem vorigen Aufgabenteil. Geben Sie die nicht-leeren Teile der Tabellen nach jedem Löschschritt an. Erläutern Sie, welches Vorgehen dafür jeweils nötig ist. [3 Punkte]
3. Suchen Sie in den Tabellen, die aus der letzten Teilaufgabe (nach dem Löschen) resultierten, nach dem Wert 12. Erläutern Sie, welches Vorgehen dafür jeweils nötig ist. [1 Punkt]

(c) Doppeltes Hashing
Diese Aufgabe ist aus der regulären Wertung genommen worden. Sie können jedoch durch die korrekte Lösung dieser Aufgabe Bonuspunkte erhalten. Die Bonuspunkte werden zu Ihren regulären Punkten addiert, als würden sie zu einer regulären Aufgabe gehören.

In der Vorlesung haben Sie unter anderem auch doppeltes Hashing kennen gelernt, bei dem eine zusätzliche Hash-Funktion verwendet wird, um Sondierungsschritte durchzuführen. In dieser Teilaufgabe sollen Sie dieses Verfahren und eine Variante davon implementieren. Dazu sollen Sie den von uns bereit gestellten Code vervollständigen. Letzterer enthält drei Klassen. Die Datei `MainClass.java` enthält die Hauptklasse, die zum Testen genutzt werden kann und nicht modifiziert werden soll. Ebenso soll die Datei `Hash.java` nicht modifiziert werden. Diese stellt eine einfache Datenstruktur für Hash-Werte mit einer zusätzlichen Markierung bereit, um angeben zu können, ob ein Hash-Eintrag frei (free), belegt (used) oder entfernt worden (deleted) ist. In der Datei `Hashing.java` soll von Ihnen die Methode `insert2` vervollständigt werden. Die Methode `insert1` implementiert bereits eine Einfüge-Operation nach doppeltem Hashing mit der Sondierungs-Funktion $h(k,i) = (h_1(k)+i\times h_2(k,m))$ mod m, wobei h_1 und h_2 Hash-Funktionen sind und h_2 sicherstellt, dass der resultierende Wert teilerfremd zu m ist. Die von Ihnen zu vervollständigende Methode soll im Gegensatz dazu bei einer Kollision die nächste Sondierungsposition sowohl für das neu einzufügende als auch für das bereits enthaltene Element an der Kollisionsposition berechnen. Sollte das alte Element an seiner neuen Position ohne weitere Sondierungen einzufügen sein, während das neue Element weitere Sondierungen benötigen würde, wird das alte Element an seine neue Position verschoben und das neue Element an der ursprünglichen Position des alten Elementes eingefügt. Ansonsten wird rekursiv versucht, das neue Element an der nächsten Sondierungsposition einzufügen.

Etwas formaler ausgedrückt funktioniert die von Ihnen zu implementierende Hashing-Variante für ein Element k_1 im i-ten Sondierungsschritt wie folgt:

- Ist $h(k_1, i)$ frei, so füge k_1 an dieser Position ein.
- Ist $h(k_1, i)$ belegt mit einem Element k_2, $h(k_1, i+1)$ ebenfalls belegt, k_2 wurde mit j Sondierungsschritten eingefügt (d. h. $h(k_1, i) = h(k_2, j)$) und $h(k_2, j+1)$ ist frei, so füge k_2 an der Position $h(k_2, j+1)$ ein und k_1 an der bisherigen Position von k_2.
- Ansonsten fahre mit der Sondierung für k_1 und $i+1$ fort.

Die von Ihnen zu ergänzenden Teile sind im Code durch Kommentare markiert. [2 Bonuspunkte]

(d) Experimenteller Vergleich
Diese Aufgabe ist aus der regulären Wertung genommen worden. Sie können jedoch durch die korrekte Lösung dieser Aufgabe Bonuspunkte erhalten. Die Bonuspunkte werden zu Ihren regulären Punkten addiert, als würden sie zu einer regulären Aufgabe gehören.

Die Testklasse gibt für das bereits vorgegebene und das von Ihnen ergänzte Hash-Verfahren aus, wie viele Kollisionen beim Einfügen von 500000 zufällig

gewählten Elementen in eine anfangs leere Hash-Tabelle der Größe 1000000 entstehen. Messen Sie einige Male die Kollisionszahlen und erklären Sie den gemessenen Unterschied. [1 Bonuspunkt]

Lösungsvorschlag

(a) Güte von Hash-Funktionen

h_1 ist zwar in konstanter Zeit pro Eingabewort (bei entsprechender Datenspeicherung) berechenbar, kann jedoch nur dann surjektiv sein, wenn die Eingabewörter sehr lang sind und m sehr klein ist. Außerdem gewährleistet sie eine Gleichverteilung der Indizes nur bei Gleichverteilung der Eingabelänge. Werden beispielsweise sehr unterschiedliche Wörter mit gleicher Länge eingefügt, so werden diese alle auf denselben Index abgebildet. Darüberhinaus werden ähnliche Schlüssel auf einen ziemlich engen Index-Bereich abgebildet.

h_2 ist in linearer Zeit aus der Eingabe berechenbar. Da der maximale Hash-Wert eines Wortes $n \cdot 127$ beträgt, darf auch hier m nicht viel größer als n sein, um Surjektivität zu erreichen. Sie ist auch nur dann gleichverteilend, wenn starke Bedingungen an die Eingabe geknüpft werden. Wie schon bei h_1 werden ähnliche Wörter auf einen engen Index-Bereich abgebildet. Beispielsweise werden alle Permutationen eines Wortes auf denselben Wert abgebildet.

h_3 behebt in gewisser Weise einige Schwachstellen von h_2 bei gleichbleibender asymptotischer Komplexität der Berechnung. Durch den zusätzlichen Vorfaktor und die Quadrierung sind größere Hash-Werte möglich als zuvor und auch Permutationen werden jetzt nicht mehr (per se) auf denselben Wert abgebildet. Ein Nachteil bleibt, dass Surjektivität nur dann erreicht wird, wenn m und n nicht zu weit auseinander liegen und tendenziell kürzere Wörter auf kleinere Hash-Werte abgebildet werden.

h_4 benutzt zwar eine eingebettete, gute Hash-Funktion, ist aber aus arithmetischen Gründen eine schlechte Wahl, da aus Fermats kleinem Satz folgt

$$h_4(x) = \begin{cases} 1 & h_{good}(x) \text{ ist nicht durch } p \text{ teilbar} \\ 0 & \text{sonst} \end{cases}$$

h_4 trifft also nur zwei mögliche Werte, unabhängig davon wie gut h_{good} oder p gewählt wurden.

(b) Einfaches Hashing

1. (i) 12 einfügen:

h_1		h_2	
3	12	12	12

99 einfügen:

3	12	4	99
18	99	12	12

21 einfügen:

		2	21
3	21 → 12	4	99
18	99	12	12

76 einfügen:

		0	76
3	21 → 12	2	21
13	76	4	99
18	99	12	12

23 einfügen:

3	21 → 12	0	76
5	23	2	21
13	76	4	23 → 99
18	99	12	12

30 einfügen:

		0	76
3	30 → 21 → 12	2	21
5	23	4	23 → 99
13	76	11	30
18	99	12	12

(ii) 12 einfügen:

h_1		h_2	
3	12	12	12

99 einfügen:

3	12	4	99
18	99	12	12

21 einfügen:

3	12	2	21
4	21	4	99
18	99	12	12

76 einfügen:

3	12	0	76
4	21	2	21
13	76	4	99
18	99	12	12

23 einfügen:

3	12	0	76
4	21	2	21
5	23	4	99
13	76	5	23
18	99	12	12

30 einfügen:

3	12	0	76
4	21	2	21
5	23	4	99
6	30	5	23
13	76	11	30
18	99	12	12

2. Beim Hashing mit Verkettung muss das Element in der jeweiligen Liste gesucht, dann gelöscht und anschließend noch gegebenenfalls Zeiger berichtigt werden.

   Beim Hashing mit linearem Sondieren wird jeweils zuerst in dem "Fach" gesucht, das mit dem Hashwert assoziiert ist. Enthält dieses das gesuchte Element, so kann es durch **del** ersetzt werden. Enthält es jedoch nicht das Element sondern **del** oder einen anderen Wert, so muss die Suche mit linearem Sondieren fortgesetzt werden.

   21 löschen:

(i)

	h_1	h_2
0		76
3	30 → 12	
4		23 → 99
5	23	
11		30
12		12
13	76	
18	99	

(ii)

	h_1	h_2
0		76
2		**del**
3	12	
4	**del**	
4		99
5	23	
5		23
6	30	
11		30
12		12
13	76	
18	99	

12 löschen:

(i)

	h_1	h_2
0		76
3	30	
4		23 → 99
5	23	
11		30
13	76	
18	99	

(ii)

	h_1	h_2
0		76
2		**del**
3	**del**	
4	**del**	
4		99
5	23	
5		23
6	30	
11		30
12		**del**
13	76	
18	99	

76 löschen:

(i)

	h_1	h_2
3	30	
4		23 → 99
5	23	
11		30
18	99	

(ii)

	h_1	h_2
0		**del**
2		**del**
3	**del**	
4	**del**	
4		99
5	23	
5		23
6	30	
11		30
12		**del**
13	**del**	
18	99	

3. Beim Hashing mit Verkettung muss die Liste der Werte, die mit dem Hashwert von 12 assoziiert ist, durchlaufen werden. In diesem Beispiel wird in

der ersten Tabelle die Liste mit dem Element 30 durchlaufen, bevor festgestellt wird, dass sich die 12 nicht in der Tabelle befindet. Bei der zweiten Tabelle wird dies sofort festgestellt, da die entsprechende Liste leer ist.

Beim Hashing mit linearem Sondieren wird jeweils zuerst in dem "Fach" gesucht, das mit dem Hashwert von 12 assoziiert ist. In diesem Beispiel enthalten jedoch beide nicht das Element 12 sondern **del**. Daher muss die Suche mit linearem Sondieren fortgesetzt werden. Dies gilt auch für den Fall, dass an einem Element nicht **del** sondern ein anderer Eintrag steht. Es wird dann nach 4 (für h_1) bzw. 1 (für h_2) Sondierungsschritt(en) festgestellt, dass sich die 12 nicht in der Tabelle befindet.

(c) Doppeltes Hashing
Siehe Quellcode.

(d) Experimenteller Vergleich
Das alternative Verfahren führt zu weniger Kollisionen. Dies liegt daran, dass lange Kollisionsketten manchmal dadurch vermieden werden können, dass bereits in der Tabelle vorhandene Elemente auf freie Positionen verschoben werden können. Kürzere Kollisionsketten verringern auch die Kollisionswahrscheinlichkeiten für spätere Einfüge-Operationen.

Aufgabe 2 (*Selektion* [**10 Punkte**])

Gegeben sei eine Folge von n unsortierten Integer-Zahlen.

(a) Entwerfen Sie einen Algorithmus, welcher das kleinste Element der Folge mit $n-1$ Vergleichen findet. Vergleiche, welche nicht die Folgenelemente selbst betreffen, z. B. Vergleiche zwischen den Folgenindizes, werden nicht dazugezählt. Außerdem soll jedes einzelne Element maximal $\lceil \log_2 n \rceil$ mal verglichen werden. Zeigen Sie, dass Ihr Algorithmus höchstens die geforderte Anzahl von Vergleichen benötigt. [6 Punkte]

(b) Erweitern Sie das Verfahren so, dass es das zweitkleinste Element von n Elementen im Worst-Case mit $n + \lceil \log_2 n \rceil - 2$ Vergleichen bestimmt. Zeigen Sie, dass Ihr Algorithmus höchstens die geforderte Anzahl von Vergleichen benötigt. Wenden Sie Ihr Verfahren auf die Elemente

$$10, 5, 3, 2, 8, 7, 1, 6, 9$$

an. Stellen Sie dabei die notwendigen Schritte in geeigneter Form dar. [2 Punkte]

(c) In der Vorlesung haben Sie den Algorithmus `PartitionSelect` kennengelernt mit welchem sich das k-kleinste Element einer Folge finden lässt. Während dieser Algorithmus im Average-case $O(n)$ Operationen benötigt, ist dessen Komplexität Im Worst-case $O(n^2)$. Beschreiben Sie eine Erweiterung des Algorithmus, welcher auch eine Worst-case Komplexität von $O(n)$ garantiert. Sie brauchen die Worst-case Komplexität von $O(n)$ nicht zu beweisen. [2 Punkte]

Lösungsvorschlag

(a) Finde kleinstes Element in Folge mit n Elementen:
Teile wie beim Merge Sort die Folge in zwei Partitionen uns bestimme rekursiv die Minima beider Partitionen. Das Minimum der Gesamtfolge ist dann das Minima der beiden Teilminima.

Algorithmus:

```
function PartitionMin(A[l..r])
    if r > l then
        m ← l + ⌊(r−l)/2⌋
        m_l ← PartitionMin(A[l..m])
        m_r ← PartitionMin(A[m + 1..r])
        if m_l ≤ m_r then
            return m_l
        else
            return m_r
        end if
    else
        return A[l]
    end if
end function
```

Um die maximale Anzahl von Vergleichen zu ermitteln stellen wir eine Rekursionsgleichung für die Anzahl der Vergleiche auf:

$$
\begin{aligned}
T(1) &= 0 \\
T(n) &= 2T(n/2) + 1 \\
&= 2\left[2T(n/4) + 1\right] + 1 \\
&= 4T(n/4) + 2 + 1 \\
&\quad\vdots \\
&= \sum_{i=0}^{\log_2 n - 1} 2^i \\
&= n - 1
\end{aligned}
$$

Für die Anzahl der Vergleiche eines einzelnen Elements lässt sich eine der beiden Partitionen vernachlässigen, da sich das Element nur in einer Partition befinden kann:

$$
\begin{aligned}
T(1) &= 0 \\
T(n) &= T(n/2) + 1 \\
&= \left[T(n/4) + 1\right] + 1 \\
&= 4T(n/4) + 1 + 1 \\
&\quad\vdots \\
&= \log_2 n
\end{aligned}
$$

Ist n keine Zweierpotenz, ist die Anzahl der Vergleiche eines einzelnen Elements also durch $\lceil \log_2 n \rceil$ nach oben beschränkt.

(b) Das zweitkleinste Element der Gesamtfolge muss sich unter den direkten Verlieren der Vergleiche mit dem kleinsten Element der Gesamtfolge befinden. Speiche also für das Minimum sämtliche direkten Verlierer. Dies sind nach Teilaufgabe (a) maximal $\lceil \log_2 n \rceil$ Elemente, unter welchen sich das Minimum mit maximal $\lceil \log_2 n \rceil - 1$ Vergleichen finden lässt. Damit ist also die Gesamtanzahl der benötigten Vergleiche $n - 1 + \lceil \log_2 n \rceil - 1 = n + \lceil \log_2 n \rceil - 2$.

TODO

(c) Der Worst-case tritt beim `PartitionSelect` Algorithmus dann auf, wenn als Pivotelement immer das kleinste oder größte Element der Folge gewählt wird. Um dies zu verhindern, muss also in Linearzeit sichergestellt werden, dass stets ein gewisser Prozentsatz des Array nach der Partitionierung abgespaltet werden kann. Wähle daher den Median der Mediane als Pivotelement.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 8

Abgabe: Montag, **24.06.2013**, 14:00 Uhr

- Die Übungen sollen in Gruppen von zwei bis drei Personen bearbeitet werden.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auf die abgegebenen Lösungen.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auch in die Quellcode-Dateien.
- Geben Sie Ihre Lösungen am **Anfang** der Globalübung, montags, 14:00 Uhr, ab.
- Schicken Sie den jeweiligen Quellcode bitte per **E-Mail** direkt an Ihre/n Tutor/in.
- Geben Sie außerdem den ausgedruckten Quellcode zusammen mit den schriftlichen Lösungen ab.
- Zu spät abgegebene Lösungen werden nicht bewertet.
- Sofern nicht anders gefordert, müssen alle Lösungen und Zwischenschritte kommentiert werden.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Aufgabe 1 (*AVL-Bäume* **[10 Punkte]**)

Der Balancegrad $balance(v)$ eines Knotens im Suchbaums v entspricht der Höhendifferenz seiner beiden Teilbäume w_1 und w_2:

$$balance(v) = height(subtree(w_1))–height(subtree(w_2)).$$

In einem AVL-Baum wird ein Knoten v als kritisch bezeichnet, wenn er unbalanciert ist, d.h. sobald $balance(v) \notin \{-1, 0, 1\}$. Immer wenn ein solcher Zustand erreicht worden ist, muss der kritische Knoten rotiert werden, um die Balanceeigenschaft wieder herzustellen.

(a) Geben sie alle möglichen unkritischen AVL-Bäume für folgende Knoten an [2 Punkte]:

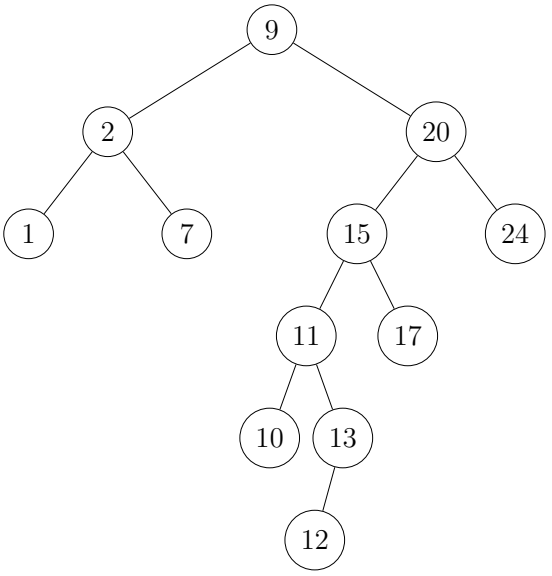
4, 5, 7, 9, 11

(b) Fügen Sie in einen leeren AVL-Baum die folgenden Zahlen in der gegebenen Reihenfolge ein:

5, 9, 23, 21, 10, 12, 19

Zeichnen Sie den AVL-Baum jeweils vor und nach jeder Rotation. Markieren Sie zusätzlich vor der Rotation den kritischen Knoten, sowie den zuletzt eingefügten Knoten. Geben Sie bei Doppelrotationen auch den Zwischenzustand des Baumes an. [4 Punkte]

(c) Geben Sie **maximal vier Rotationen** an, die den folgenden Baum der **Höhe fünf** in einen Baum der **Höhe drei** transformieren. Geben Sie auch den Zustand des Baumes nach jeder Rotation an. [4 Punkte]



Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Aufgabe 2 (*Rot-Schwarz-Bäume* [**10 Punkte**])

(a) Fügen Sie die folgenden Werte in der angegebenen Reihenfolge in einen anfangs leeren Rot-Schwarz-Baum ein. Geben Sie den entstandenen Rot-Schwarz-Baum nach jeder Knotenerzeugung, Umfärbung und Rotation an (mehrere Umfärbungen innerhalb eines Falles dürfen in einem Schritt vorgenommen werden). [3 Punkte]

$$42, 13, 17, 23, 31, 69, 77$$

(b) Löschen Sie nun aus dem in Aufgabenteil (a) entstandenen Rot-Schwarz-Baum die folgenden Werte in der angegebenen Reihenfolge. Geben Sie den entstandenen Rot-Schwarz-Baum nach jeder Knotenlöschung, Markierungsverschiebung, Umfärbung und Rotation an (mehrere Umfärbungen innerhalb eines Falles dürfen in einem Schritt vorgenommen werden). [3 Punkte]

$$13, 31, 42, 69$$

(c) Zeigen Sie, dass zu jeder Höhe h ein Rot-Schwarz-Baum $B(h)$ mit der folgenden Anzahl roter Knoten $r(B(h))$ existiert [4 Punkte]:

$$r(B(h)) = \sum_{i=2}^{h} \left((1 + (-1)^{i-h}) \cdot 2^{i-2} \right)$$

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 8

Abgabe: Montag, **24.06.2013**, 14:00 Uhr

- Die Übungen sollen in Gruppen von zwei bis drei Personen bearbeitet werden.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auf die abgegebenen Lösungen.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auch in die Quellcode-Dateien.
- Geben Sie Ihre Lösungen am **Anfang** der Globalübung, montags, 14:00 Uhr, ab.
- Schicken Sie den jeweiligen Quellcode bitte per **E-Mail** direkt an Ihre/n Tutor/in.
- Geben Sie außerdem den ausgedruckten Quellcode zusammen mit den schriftlichen Lösungen ab.
- Zu spät abgegebene Lösungen werden nicht bewertet.
- Sofern nicht anders gefordert, müssen alle Lösungen und Zwischenschritte kommentiert werden.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Aufgabe 1 (*AVL-Bäume* **[10 Punkte]**)

Der Balancegrad $balance(v)$ eines Knotens im Suchbaums v entspricht der Höhendifferenz seiner beiden Teilbäume w_1 und w_2:

$$balance(v) = height(subtree(w_1))–height(subtree(w_2)).$$

In einem AVL-Baum wird ein Knoten v als kritisch bezeichnet, wenn er unbalanciert ist, d.h. sobald $balance(v) \notin \{-1, 0, 1\}$. Immer wenn ein solcher Zustand erreicht worden ist, muss der kritische Knoten rotiert werden, um die Balanceeigenschaft wieder herzustellen.

(a) Geben sie alle möglichen unkritischen AVL-Bäume für folgende Knoten an [2 Punkte]:

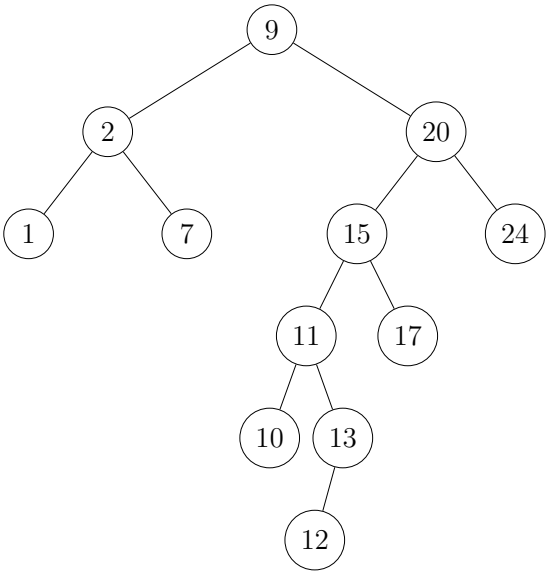
4, 5, 7, 9, 11

(b) Fügen Sie in einen leeren AVL-Baum die folgenden Zahlen in der gegebenen Reihenfolge ein:

5, 9, 23, 21, 10, 12, 19

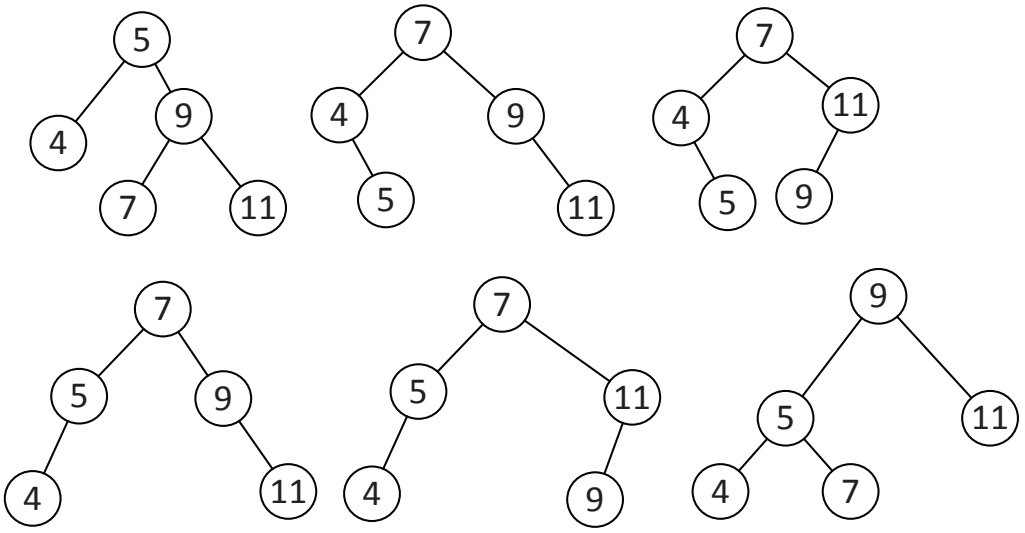
Zeichnen Sie den AVL-Baum jeweils vor und nach jeder Rotation. Markieren Sie zusätzlich vor der Rotation den kritischen Knoten, sowie den zuletzt eingefügten Knoten. Geben Sie bei Doppelrotationen auch den Zwischenzustand des Baumes an. [4 Punkte]

(c) Geben Sie **maximal vier Rotationen** an, die den folgenden Baum der **Höhe fünf** in einen Baum der **Höhe drei** transformieren. Geben Sie auch den Zustand des Baumes nach jeder Rotation an. [4 Punkte]

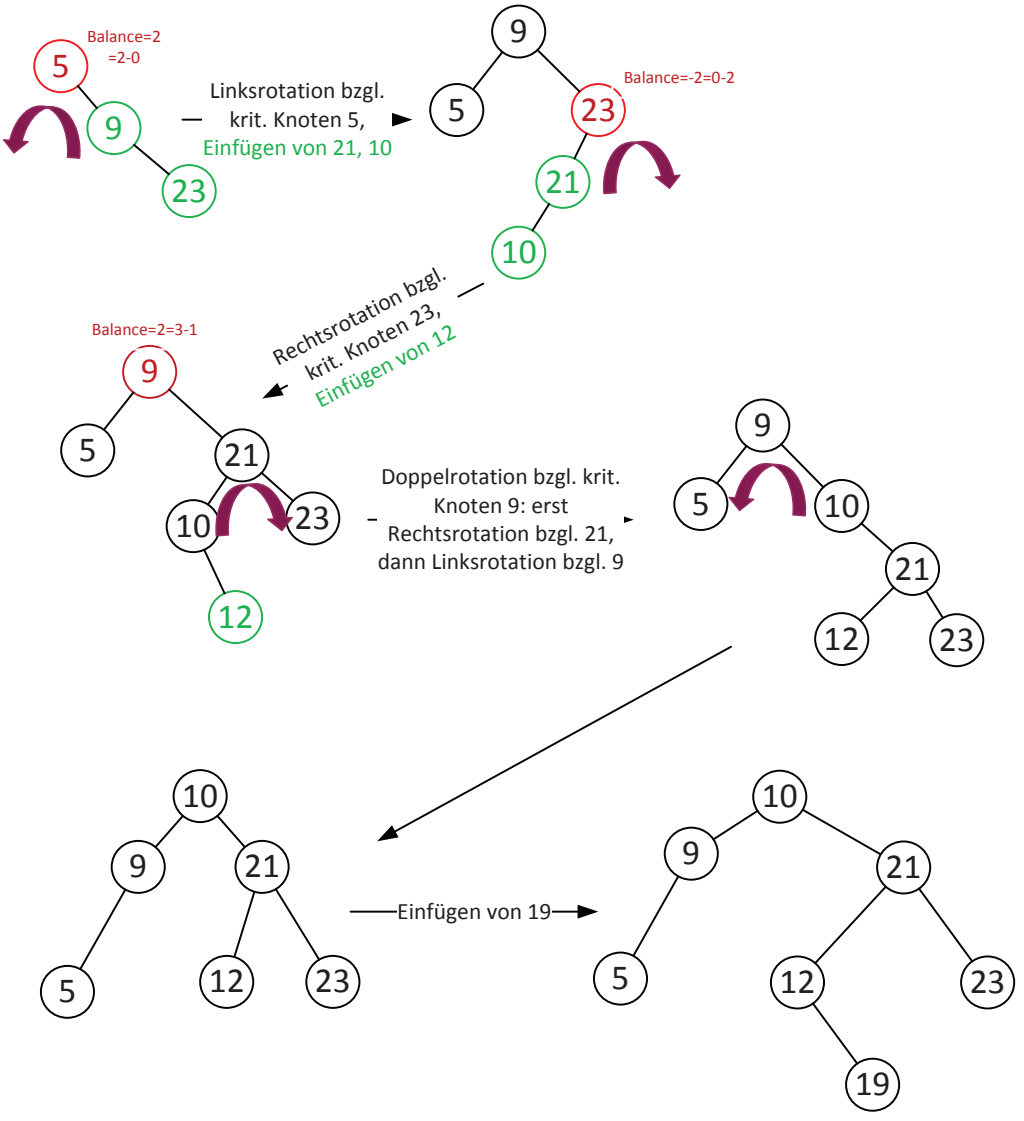


Lösungsvorschlag

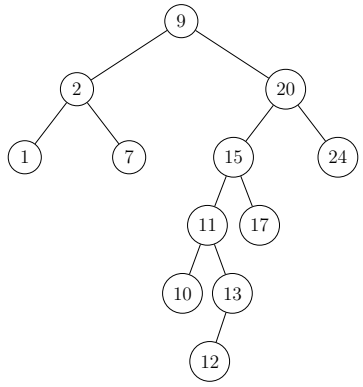
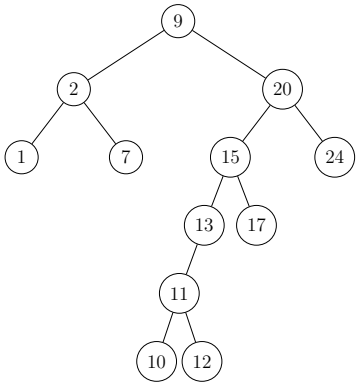
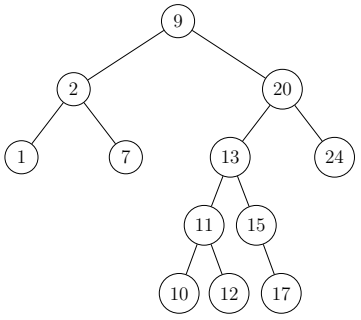
(a)



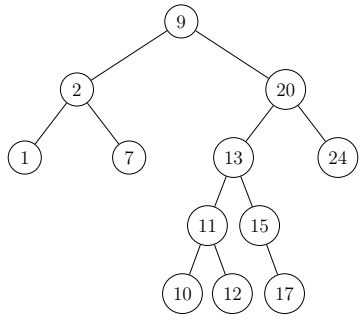
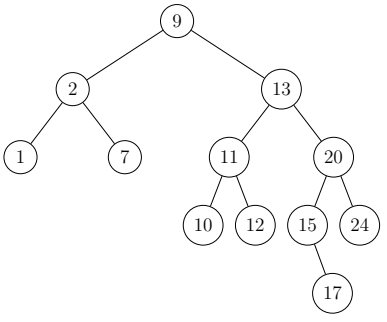
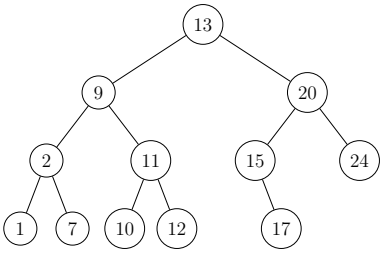
(b)



(c) Wir können den gegebenen Baum mit vier Rotationen in einen Baum der Höhe drei transformieren. Hierzu führen wir zuerst eine Linksrotation auf den Knoten 11 aus und anschließend eine Rechtsrotation auf den Knoten 15. Durch diese beiden Rotationen erhalten wir einen Baum der Höhe vier:

Ursprünglicher Baum:	Linksrotation auf Knoten 11:	Rechtsrotation auf Knoten 15
		

Auf diesen Baum führen wir nun noch eine Rechtsrotation auf den Knoten 20 durch, gefolgt von einer Linksrotation um die Wurzel und erhalten so einen Baum der Höhe drei:

Baum der Höhe vier:	Rechtsrotation auf Knoten 20:	Linksrotation auf Wurzel:
		

Aufgabe 2 (*Rot-Schwarz-Bäume* [**10 Punkte**])

(a) Fügen Sie die folgenden Werte in der angegebenen Reihenfolge in einen anfangs leeren Rot-Schwarz-Baum ein. Geben Sie den entstandenen Rot-Schwarz-Baum nach jeder Knotenerzeugung, Umfärbung und Rotation an (mehrere Umfärbungen innerhalb eines Falles dürfen in einem Schritt vorgenommen werden). [3 Punkte]

$$42, 13, 17, 23, 31, 69, 77$$

(b) Löschen Sie nun aus dem in Aufgabenteil a) entstandenen Rot-Schwarz-Baum die folgenden Werte in der angegebenen Reihenfolge. Geben Sie den entstandenen Rot-Schwarz-Baum nach jeder Knotenlöschung, Markierungsverschiebung, Umfärbung und Rotation an (mehrere Umfärbungen innerhalb eines Falles dürfen in einem Schritt vorgenommen werden). [3 Punkte]

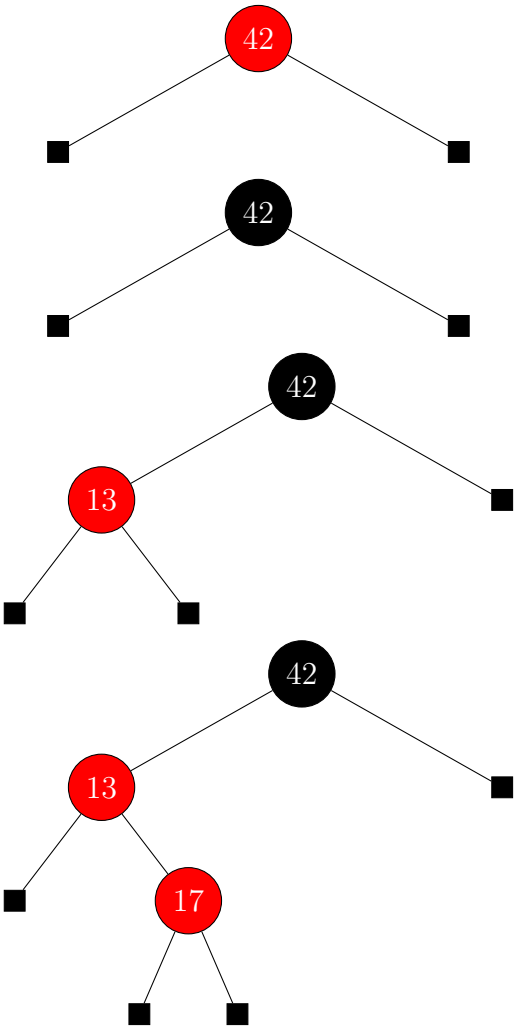
$$13, 31, 42, 69$$

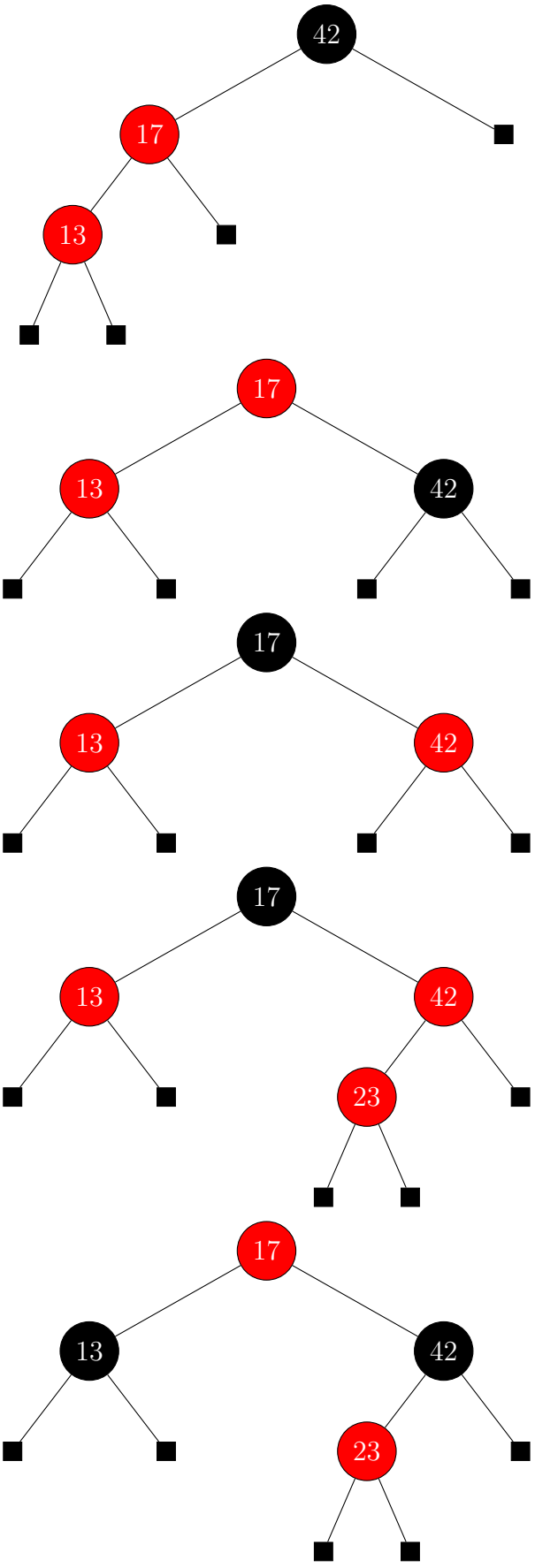
(c) Zeigen Sie, dass zu jeder Höhe h ein Rot-Schwarz-Baum $B(h)$ existiert mit der folgenden Anzahl roter Knoten $r(B(h))$ [4 Punkte]:

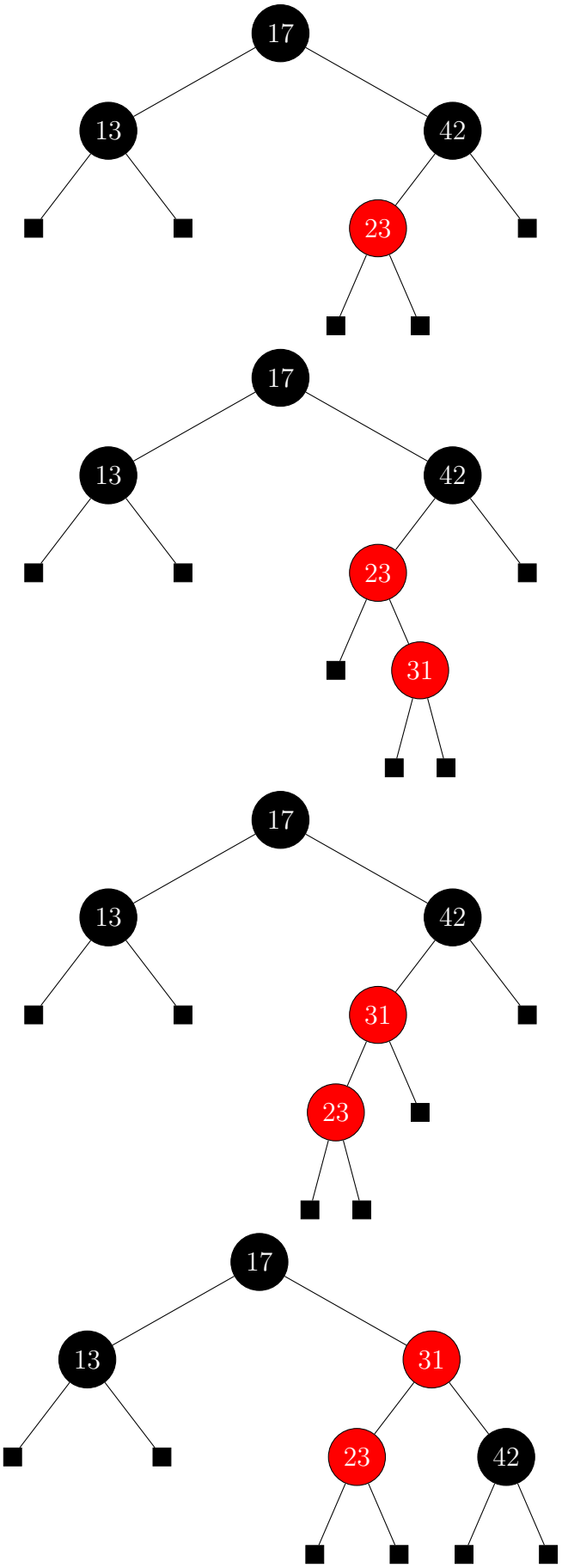
$$r(B(h)) = \sum_{i=2}^{h} \left((1 + (-1)^{i-h}) \cdot 2^{i-2} \right)$$

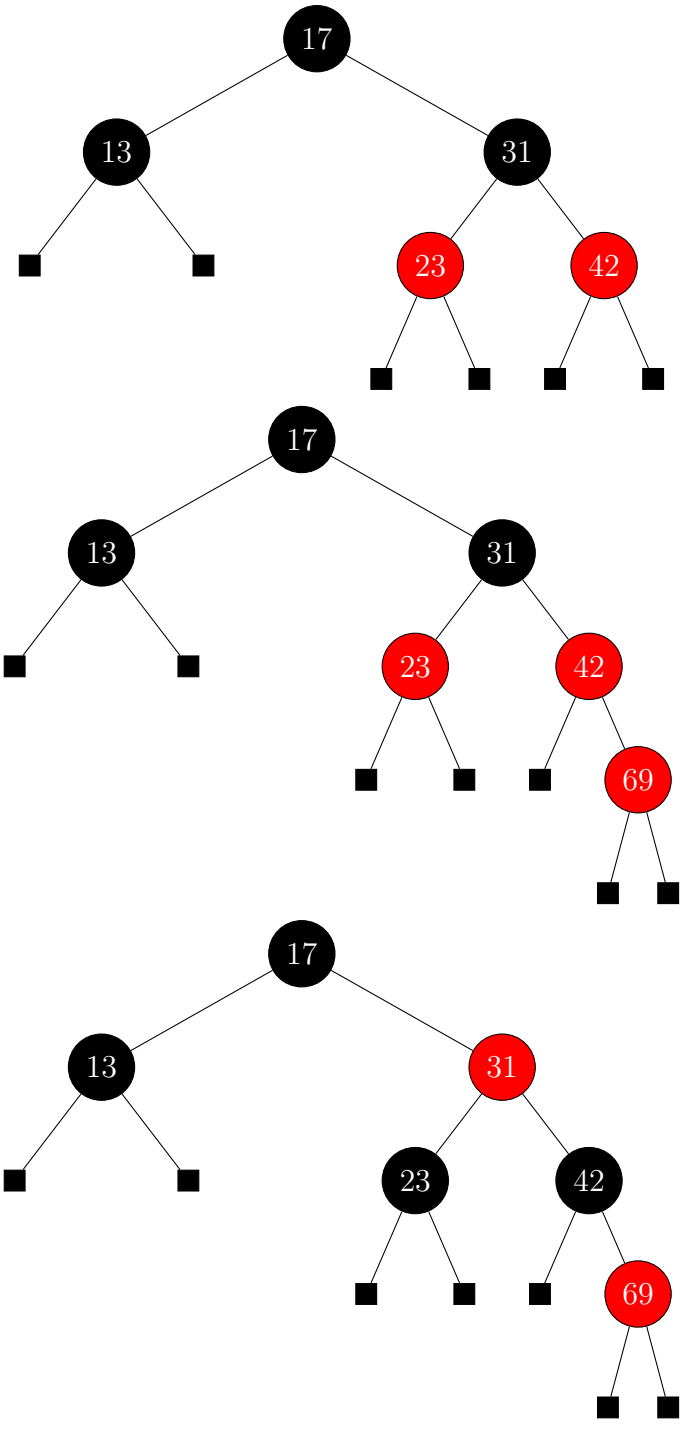
Lösungsvorschlag

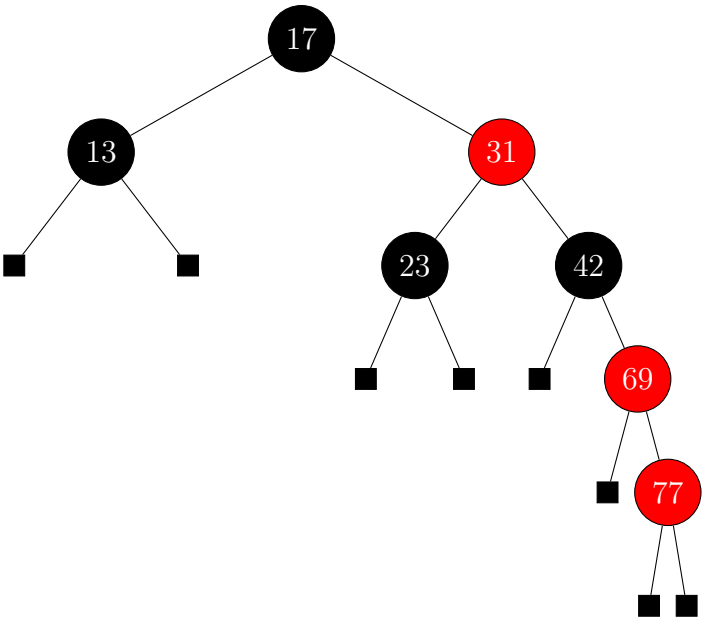
(a)

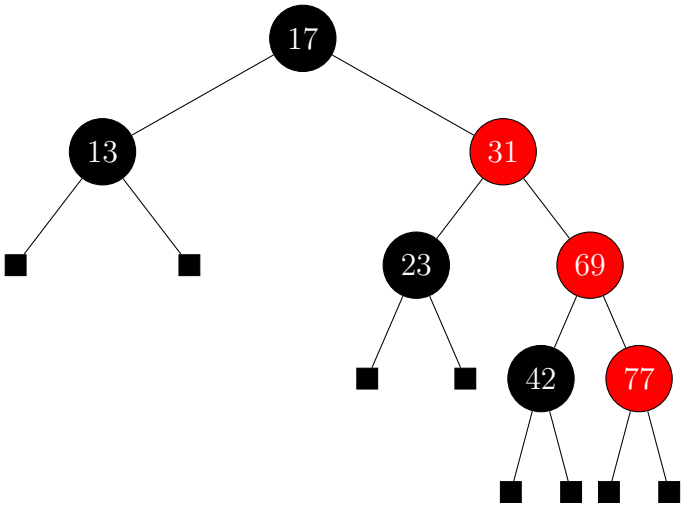


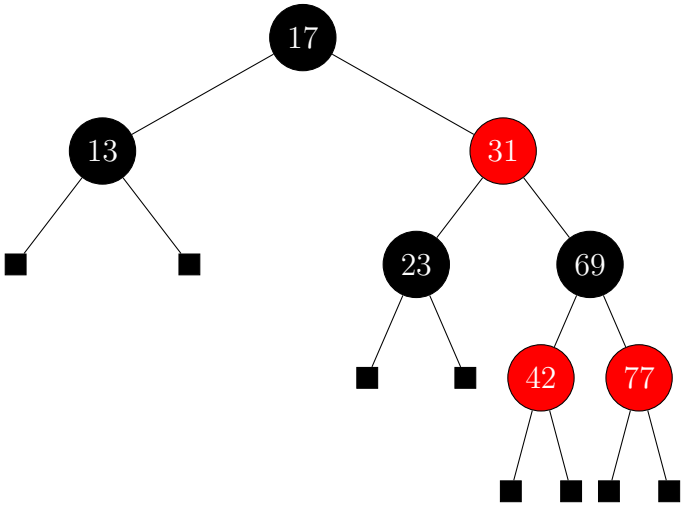

42
17
13
17
13
42
17
13
42
17
13
42
23
17
13
42
23


17
13
42
23
17
13
42
23
31
17
13
42
31
23
17
13
31
23
42

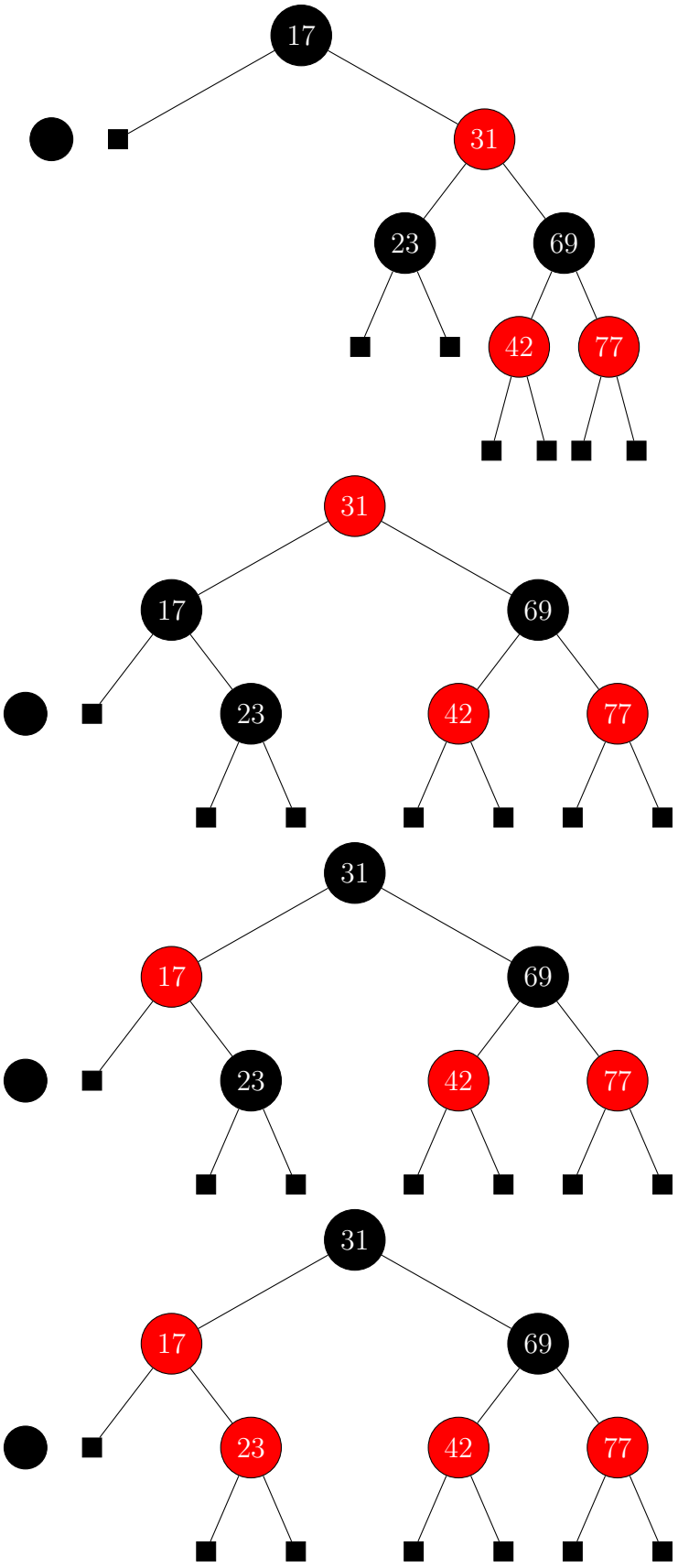

17
13
31
23
42
17
13
31
23
42
69
17
13
31
23
42
69

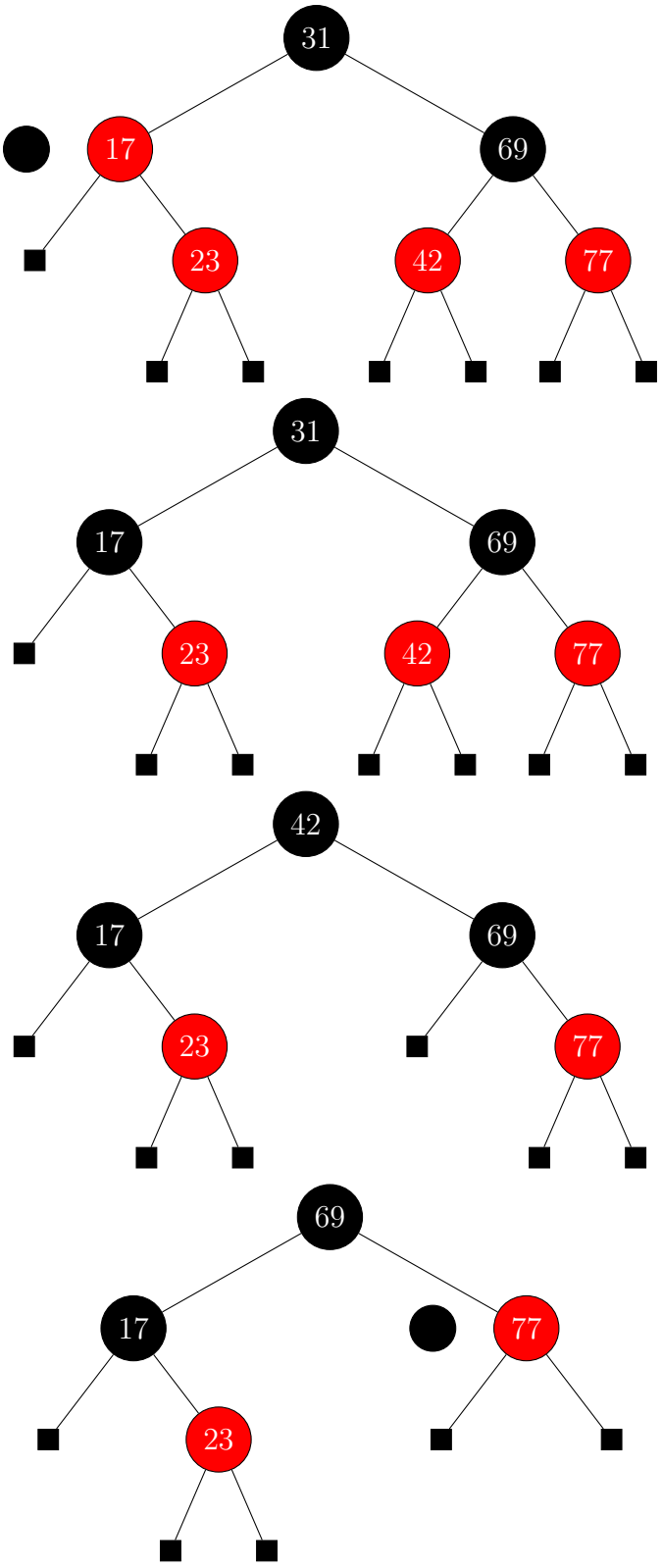


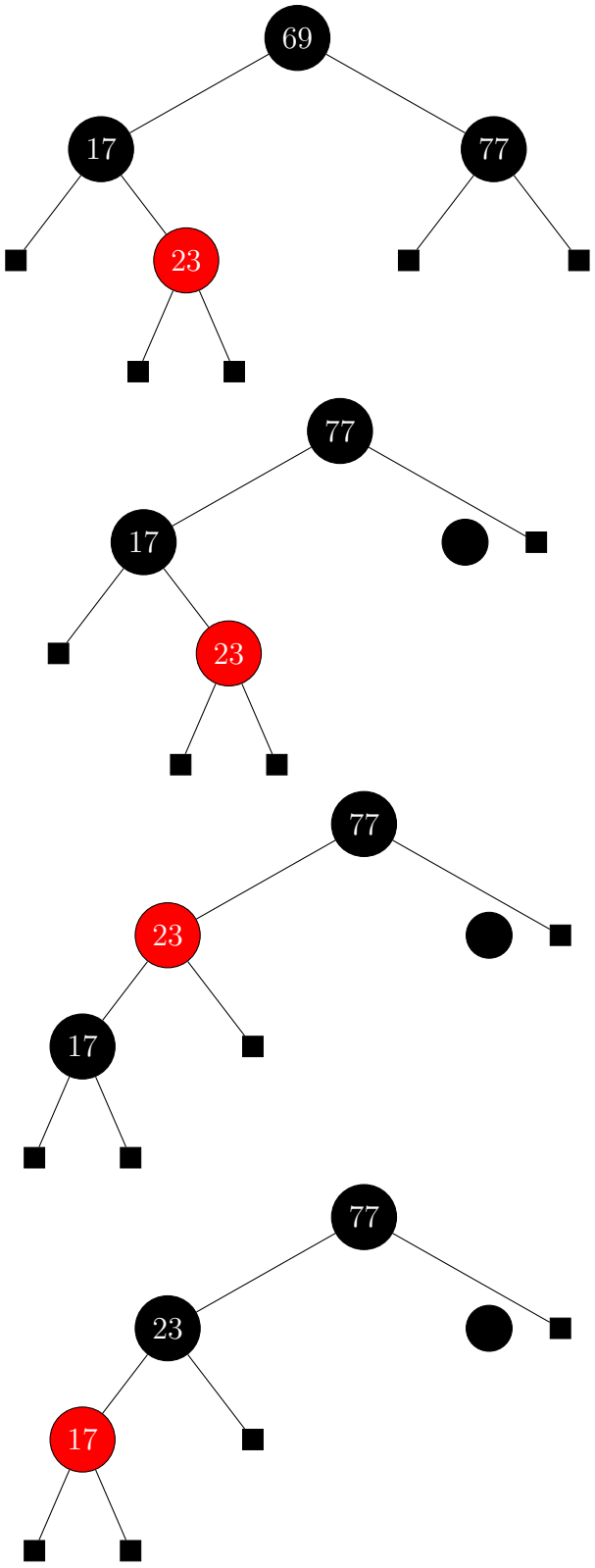


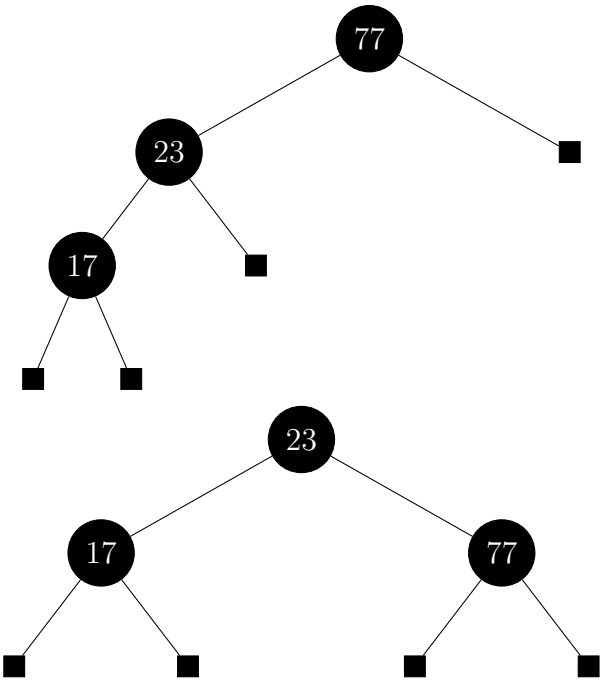


(b)


17
31
23
69
42
77
31
17
69
23
42
77
31
17
69
23
42
77
31
17
69
23
42
77


31
17
69
23
42
77
31
17
69
23
42
77
42
17
69
23
77
69
17
77
23


69
17
77
23
77
17
23
77
23
17
77
23
17



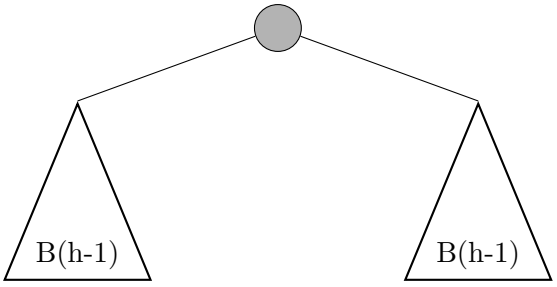
(c) Wir konstruieren einen entsprechenden Baum mit Höhe h induktiv und zeigen, dass die Eigenschaft für jede Höhe erfüllt ist:

Induktionsanfang:
Alle Bäume mit Höhe $h < 2$ bestehen nur aus Wurzel und Blättern. Nach Definition besitzen sie somit nur schwarze Knoten. Der Summenindex i startet bei 2 und läuft bis h. Somit ergibt sich bei allen Höhen $h < 2$ die Summe 0 und die Aussage ist für $h < 2$ gezeigt.

Induktionshypothese:
Die Aussage gelte für alle Höhen h' mit $0 \leq h' < h$, wobei h beliebig, aber fest ist.

Induktionsschritt:
Wir unterscheiden zwischen gerader und ungerader Höhe h:

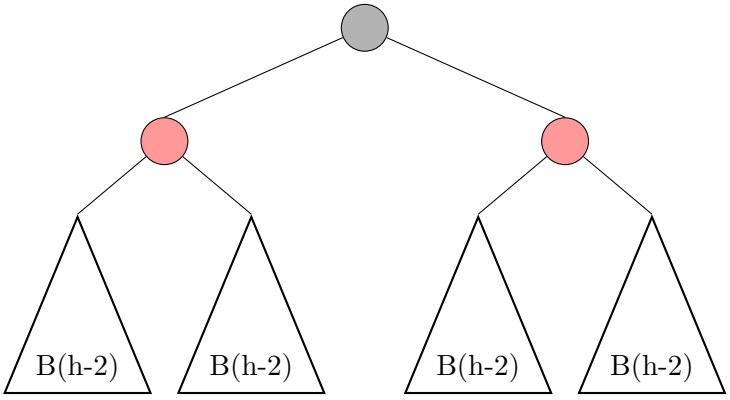
1. Fall: h ist ungerade.
Konstruiere $B(h)$ wie folgt:



Für den so konstruierten Baum $B(h)$ gilt für die Anzahl der roten Knoten $r(B(h))$:

$$
\begin{aligned}
r(B(h)) &= 2 \cdot r(B(h-1)) && \text{|Induktionshypothese} \\
&= 2 \cdot \sum_{i=2}^{h-1} \left((1 + (-1)^{i-(h-1)}) \cdot 2^{i-2} \right) \\
&= \sum_{i=2}^{h-1} \left((1 + (-1)^{i-h+1)}) \cdot 2^{i-1} \right) && \text{|Indexverschiebung} \\
&= \sum_{i=3}^{h} \left((1 + (-1)^{i-h}) \cdot 2^{i-2} \right) && \text{|}h \text{ ungerade } \Rightarrow (1 + (-1)^{2-h}) = 0 \\
&= \sum_{i=2}^{h} \left((1 + (-1)^{i-h}) \cdot 2^{i-2} \right)
\end{aligned}
$$

2. Fall: h ist gerade.
Konstruiere $B(h)$ wie folgt:



Für den so konstruierten Baum $B(h)$ gilt für die Anzahl der roten Knoten $r(B(h))$:

$$
\begin{aligned}
r(B(h)) &= 2 + 4 \cdot r(B(h-2)) && \text{|Induktionshypothese} \\
&= 2 + 4 \cdot \sum_{i=2}^{h-2} \left((1 + (-1)^{i-(h-2)}) \cdot 2^{i-2} \right) \\
&= 2 + \sum_{i=2}^{h-2} \left((1 + (-1)^{i-h+2}) \cdot 2^{i} \right) && \text{|Indexverschiebung} \\
&= 2 + \sum_{i=4}^{h} \left((1 + (-1)^{i-h}) \cdot 2^{i-2} \right) && \text{|}h \text{ gerade } \Rightarrow \\
& && (1 + (-1)^{3-h}) = 0 \wedge \\
& && (1 + (-1)^{2-h}) = 2 \\
&= 2 + \sum_{i=2}^{h} \left((1 + (-1)^{i-h}) \cdot 2^{i-2} \right) - 2 \cdot 2^{2-2} \\
&= \sum_{i=2}^{h} \left((1 + (-1)^{i-h}) \cdot 2^{i-2} \right)
\end{aligned}
$$

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 9

Abgabe: Montag, **01.07.2013**, 14:00 Uhr

- Die Übungen sollen in Gruppen von zwei bis drei Personen bearbeitet werden.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auf die abgegebenen Lösungen.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auch in die Quellcode-Dateien.
- Geben Sie Ihre Lösungen am **Anfang** der Globalübung, montags, 14:00 Uhr, ab.
- Schicken Sie den jeweiligen Quellcode bitte per **E-Mail** direkt an Ihre/n Tutor/in.
- Geben Sie außerdem den ausgedruckten Quellcode zusammen mit den schriftlichen Lösungen ab.
- Zu spät abgegebene Lösungen werden nicht bewertet.
- Sofern nicht anders gefordert, müssen alle Lösungen und Zwischenschritte kommentiert werden.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Aufgabe 1 (*B-Bäume* **[10 Punkte]**)

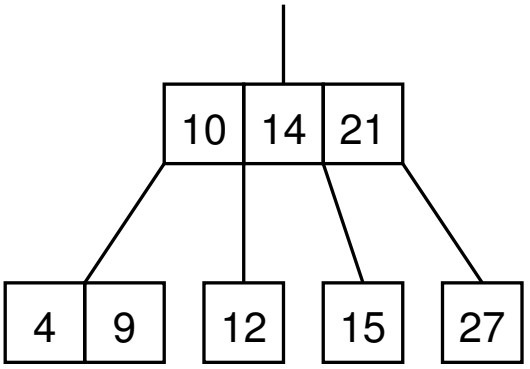
1. Fügen Sie folgende Schlüssel in dieser Reihenfolge in einen B-Baum mit minimalem Grad $t = 3$ ein. [4 Punkte]

   $$11, 19, 35, 41, 49, 25, 15, 17, 21, 7, 4, 3, 1$$

2. Löschen Sie in dieser Reihenfolge die Schlüssel

   $$14, 27, 10$$

   aus folgendem B-Baum mit minimalem Grad $t = 2$. [3 Punkte]

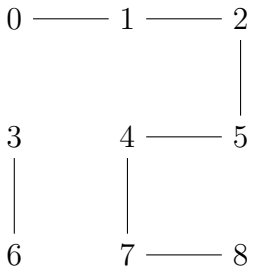


3. Zeigen oder widerlegen Sie: Bei gegebenen Schlüsselwerten ist die Zahl der Knoten in einem zugehörigen B-Baum eindeutig. [3 Punkte]

Zeichnen Sie für Aufgabenteil 1 bis 3 den Baum nach jedem Einfügen, Löschen und nach jeder Merge-, Split- oder Rotate-Operation.

Aufgabe 2 (*Union-Find* [**10 Punkte**])

In dieser Aufgabe wollen wir eine Erreichbarkeitsanalyse für Netzwerke implementieren. Das Netzwerk ist dabei gitterartig aufgebaut, d. h. jeder Knoten des Netzwerks hat höchstens vier direkte Verbindungen (nach links, unten, rechts und oben). Den Verbindungsstatus eines Knotens können wir daher als Bitfolge von vier Bits kodieren. Dabei entspricht das erste Bit dem Status der Verbindung nach links, das zweite der Verbindung nach unten, das dritte der Verbindung nach rechts und das vierte der Verbindung nach oben. Die jeweilige Verbindung ist vorhanden, wenn das entsprechende Bit gesetzt (also gleich 1) ist. Interpretieren wir diese Bitfolgen als ganze Zahlen, können wir mit den Zahlen von 0 bis 15 alle Verbindungszustände eines Knotens beschreiben. Zum Beispiel kodiert die Zahl 7 (Bitfolge 0111) einen Knoten, der Verbindungen nach unten, rechts und oben hat, aber nicht nach links.
Wir betrachten in dieser Aufgabe nur quadratische Gitternetzwerke. Daher hat jede Zeile eines solchen Netzwerks gleich viele Knoten und die Anzahl der Knoten in einer Zeile entspricht auch der Anzahl der Zeilen im gesamten Netzwerk. Ein solches quadratisches Netzwerk können wir also nun als Array von ganzzahligen Werten aus dem Wertebereich $\{0, \ldots, 15\}$ kodieren. Dabei stellt jede Zahl den Verbindungsstatus eines Knotens dar. Die Reihenfolge der Zahlen im Array entspricht dabei der Reihenfolge der Knoten im Netzwerk, wenn man es zeilenweise durchläuft. Das Array muss also eine quadratische Länge haben und man erhält zu einer Position die rechts oder links gelegene, indem man 1 zum entsprechenden Index im Array addiert oder subtrahiert (außer an den Randpositionen). Die unten oder oben gelegene Position erhält man, indem man n addiert oder subtrahiert, wobei das Array die Länge n^2 hat (wiederum mit Ausnahme von Randpositionen). Das folgende Beispiel illustriert diese Kodierung für ein Netzwerk der Größe 3×3:



Array:
$[2, 10, 12, 4, 6, 9, 1, 3, 8]$

Die von uns bereitgestellte Datei `MainClass.java` enthält eine Klasse, deren Hauptmethode drei Argumente erwartet: die Zahl n und zwei Positionen x und y als Zahlen zwischen 0 und $n^2 - 1$. Sie generiert dann zufällig eine Netzwerk-Kodierung wie oben beschrieben (also ein Array der Länge n^2 mit Werten aus dem Wertebereich $\{0, \ldots, 15\}$) und soll anschließend `true` oder `false` ausgeben, je nach dem, ob x und y (möglicherweise über mehrere Positionen) miteinander verbunden sind. Im Beispiel oben sind die Werte 0 und 8 miteinander verbunden, während es die Werte 3 und 4 nicht sind. Die Methode `connected`, welche den zuletzt genannten Test implementieren soll, ist in der bereitgestellten Klasse allerdings nicht fertig implementiert. Stattdessen sind die Methoden `union` und `find` aus der Vorlesung bereits implementiert und sollen nun von Ihnen dazu verwendet werden, die Implementierung der `connected` Methode fertig zu stellen.

Um beispielsweise ein Netzwerk der Größe 10×10 generieren zu lassen und zu prüfen, ob darin die Positionen 0 und 99 miteinander verbunden sind, muss das Programm folgendermaßen aufgerufen werden:

```
java MainClass 10 0 99
```

Hinweis: Um zu prüfen, ob das kleinste Bit in einer Zahl `x` *gesetzt ist, kann man den Test* `(x & 1) == 1` *verwenden. Für die drei nächsten Bits funktioniert der Test analog mit den Werten 2, 4 und 8 anstelle von 1 (an beiden Positionen).*

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 9

Abgabe: Montag, **01.07.2013**, 14:00 Uhr

- Die Übungen sollen in Gruppen von zwei bis drei Personen bearbeitet werden.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auf die abgegebenen Lösungen.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auch in die Quellcode-Dateien.
- Geben Sie Ihre Lösungen am **Anfang** der Globalübung, montags, 14:00 Uhr, ab.
- Schicken Sie den jeweiligen Quellcode bitte per **E-Mail** direkt an Ihre/n Tutor/in.
- Geben Sie außerdem den ausgedruckten Quellcode zusammen mit den schriftlichen Lösungen ab.
- Zu spät abgegebene Lösungen werden nicht bewertet.
- Sofern nicht anders gefordert, müssen alle Lösungen und Zwischenschritte kommentiert werden.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Aufgabe 1 (*B-Bäume* [**10 Punkte**])

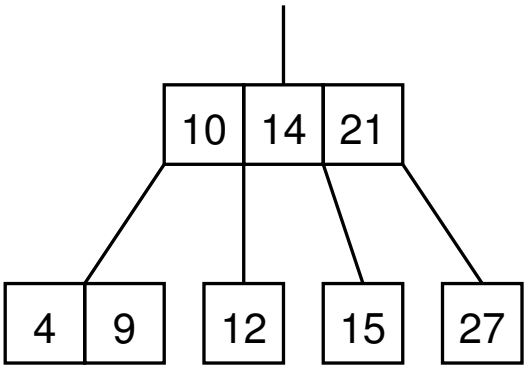
1. Fügen Sie folgende Schlüssel in dieser Reihenfolge in einen B-Baum mit minimalem Grad $t = 3$ ein.

   $$11, 19, 35, 41, 49, 25, 15, 17, 21, 7, 4, 3, 1$$

2. Löschen Sie in dieser Reihenfolge die Schlüssel

   $$14, 27, 10$$

   aus folgendem B-Baum mit minimalem Grad $t = 2$.



3. Zeigen oder widerlegen Sie: Bei gegebenen Schlüsselwerten ist die Zahl der Knoten in einem zugehörigen B-Baum eindeutig.

4. Gegeben sei eine aufsteigende, endliche Folge von ganzzahligen Schlüsselwerten mit n Elementen, also $[1, 2, \ldots, n]$. Diese Werte sollen nun in dieser Reihenfolge in einen B-Baum mit gegebenem minimalen Grad t eingefügt werden. Berechnen Sie möglichst genau, wie viele Split-Operationen beim Einfügen aller gegebenen Schlüsselwerte ausgeführt werden müssen. Für die Höhe des Baumes können Sie die Abschätzung $h \leq \log_t \frac{n+1}{2}$ benutzen.

Zeichnen Sie den Baum nach jedem Einfügen, Löschen und nach jeder Merge-, Split- oder Rotate-Operation.

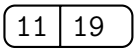
Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

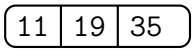
Informatik 8
RWTH Aachen

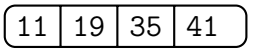
Lösungsvorschlag

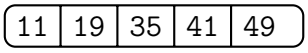
1. Der B-Baum nach jedem Einfügeschritt:

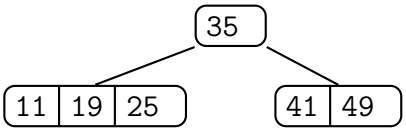
(1) 

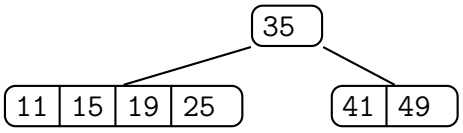
(2) 

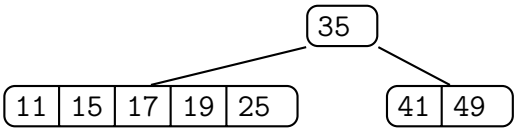
(3) 

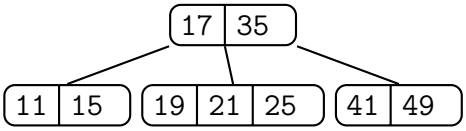
(4) 

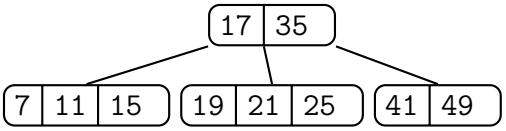
(5) 

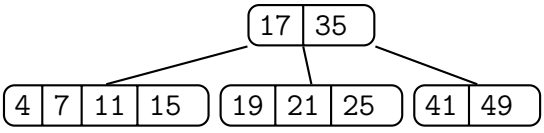
(6) 

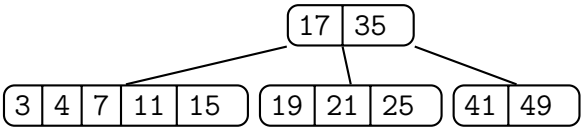
(7) 

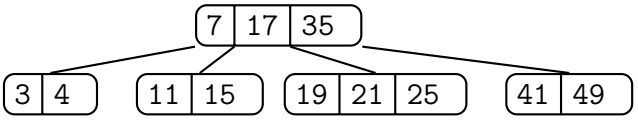
(8) 

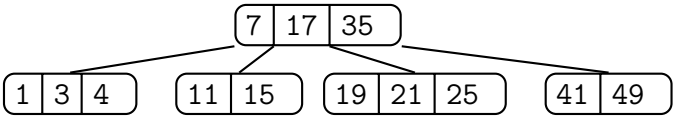
(9) 

(10) 

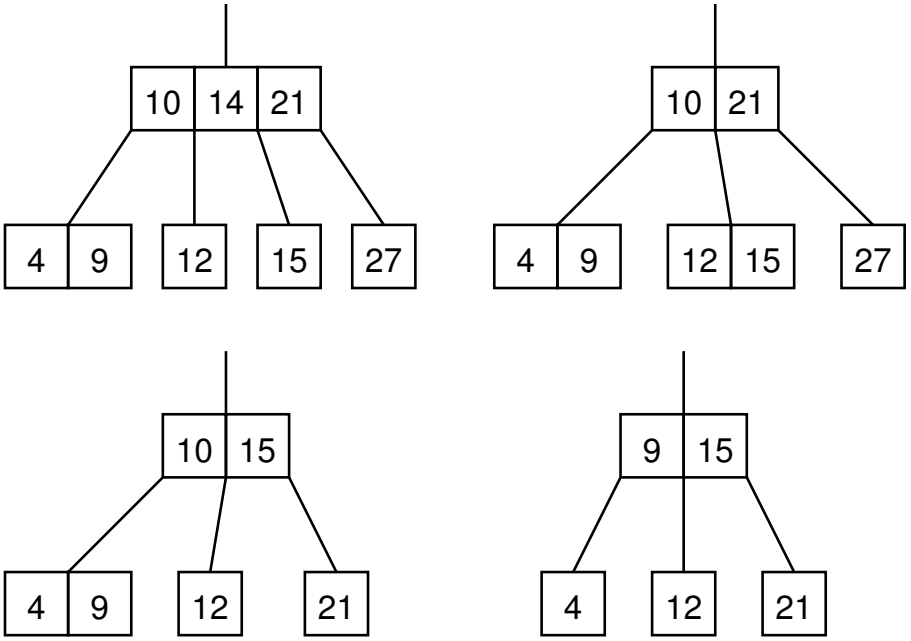
(11) 

(12) 

(13) 

(14) 

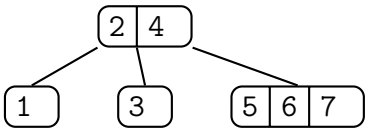
2. Der Baum jeweils nach den Löschoperationen:



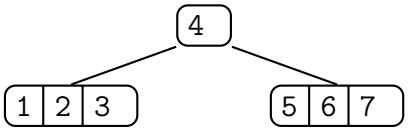
3. Die Zahl der Knoten ist bei gegebener Schlüsselmenge nicht eindeutig, da sie von der Reihenfolge, in der die Schlüssel in den B-Baum eingefügt werden, abhängig ist. Dies kann mit einem einfachen Gegenbeispiel gezeigt werden:

   Betrachte einen B-Baum mit $t = 2$ und der Schlüsselmenge $\{1, 2, 3, 4, 5, 6, 7\}$.

   - B-Baum nach Einfügen der Schlüssel in der Reihenfolge $[1, 2, 3, 4, 5, 6, 7]$.



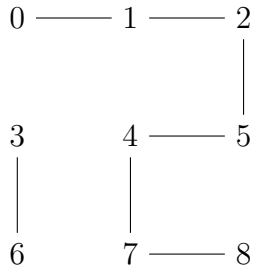
   - B-Baum nach Einfügen der Schlüssel in der Reihenfolge $[1, 4, 7, 2, 3, 5, 6]$.



Aufgabe 2 (*Union-Find* [**10 Punkte**])

In dieser Aufgabe wollen wir eine Erreichbarkeitsanalyse für Netzwerke implementieren. Das Netzwerk ist dabei gitterartig aufgebaut, d. h. jeder Knoten des Netzwerks hat höchstens vier direkte Verbindungen (nach links, unten, rechts und oben). Den Verbindungsstatus eines Knotens können wir daher als Bitfolge von vier Bits kodieren. Dabei entspricht das erste Bit dem Status der Verbindung nach links, das zweite der Verbindung nach unten, das dritte der Verbindung nach rechts und das vierte der Verbindung nach oben. Die jeweilige Verbindung ist vorhanden, wenn das

entsprechende Bit gesetzt (also gleich 1) ist. Interpretieren wir diese Bitfolgen als ganze Zahlen, können wir mit den Zahlen von 0 bis 15 alle Verbindungszustände eines Knotens beschreiben. Zum Beispiel kodiert die Zahl 7 (Bitfolge 0111) einen Knoten, der Verbindungen nach unten, rechts und oben hat, aber nicht nach links.
Wir betrachten in dieser Aufgabe nur quadratische Gitternetzwerke. Daher hat jede Zeile eines solchen Netzwerks gleich viele Knoten und die Anzahl der Knoten in einer Zeile entspricht auch der Anzahl der Zeilen im gesamten Netzwerk. Ein solches quadratisches Netzwerk können wir also nun als Array von ganzzahligen Werten aus dem Wertebereich $\{0, \ldots, 15\}$ kodieren. Dabei stellt jede Zahl den Verbindungsstatus eines Knotens dar. Die Reihenfolge der Zahlen im Array entspricht dabei der Reihenfolge der Knoten im Netzwerk, wenn man es zeilenweise durchläuft. Das Array muss also eine quadratische Länge haben und man erhält zu einer Position die rechts oder links gelegene, indem man 1 zum entsprechenden Index im Array addiert oder subtrahiert (außer an den Randpositionen). Die unten oder oben gelegene Position erhält man, indem man n addiert oder subtrahiert, wobei das Array die Länge n^2 hat (wiederum mit Ausnahme von Randpositionen). Das folgende Beispiel illustriert diese Kodierung für ein Netzwerk der Größe 3×3:



Array:
$[2, 10, 12, 4, 6, 9, 1, 3, 8]$

Die von uns bereitgestellte Datei `MainClass.java` enthält eine Klasse, deren Hauptmethode drei Argumente erwartet: die Zahl n und zwei Positionen x und y als Zahlen zwischen 0 und $n^2 - 1$. Sie generiert dann zufällig eine Netzwerk-Kodierung wie oben beschrieben (also ein Array der Länge n^2 mit Werten aus dem Wertebereich $\{0, \ldots, 15\}$) und soll anschließend `true` oder `false` ausgeben, je nach dem, ob x und y (möglicherweise über mehrere Positionen) miteinander verbunden sind. Im Beispiel oben sind die Werte 0 und 8 miteinander verbunden, während es die Werte 3 und 4 nicht sind. Die Methode `connected`, welche den zuletzt genannten Test implementieren soll, ist in der bereitgestellten Klasse allerdings nicht fertig implementiert. Stattdessen sind die Methoden `union` und `find` aus der Vorlesung bereits implementiert und sollen nun von Ihnen dazu verwendet werden, die Implementierung der `connected` Methode fertig zu stellen.
Um beispielsweise ein Netzwerk der Größe 10×10 generieren zu lassen und zu prüfen, ob darin die Positionen 0 und 99 miteinander verbunden sind, muss das Programm folgendermaßen aufgerufen werden:

```
java MainClass 10 0 99
```

Hinweis: Um zu prüfen, ob das kleinste Bit in einer Zahl `x` *gesetzt ist, kann man den Test* `(x & 1) == 1` *verwenden. Für die drei nächsten Bits funktioniert der Test analog mit den Werten 2, 4 und 8 anstelle von 1 (an beiden Positionen).*

Lösungsvorschlag

Siehe Quellcode.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 10

Abgabe: Montag, **08.07.2013**, 14:00 Uhr

- Die Übungen sollen in Gruppen von zwei bis drei Personen bearbeitet werden.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auf die abgegebenen Lösungen.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auch in die Quellcode-Dateien.
- Geben Sie Ihre Lösungen am **Anfang** der Globalübung, montags, 14:00 Uhr, ab.
- Schicken Sie den jeweiligen Quellcode bitte per **E-Mail** direkt an Ihre/n Tutor/in.
- Geben Sie außerdem den ausgedruckten Quellcode zusammen mit den schriftlichen Lösungen ab.
- Zu spät abgegebene Lösungen werden nicht bewertet.
- Sofern nicht anders gefordert, müssen alle Lösungen und Zwischenschritte kommentiert werden.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

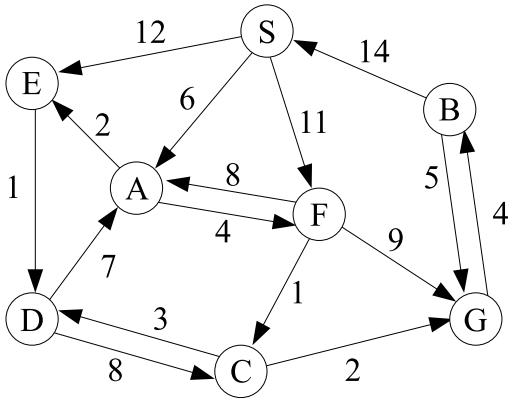
Informatik 8
RWTH Aachen

Aufgabe 1 (*Maximaler Abstand in Graphen* [**6 Punkte**])

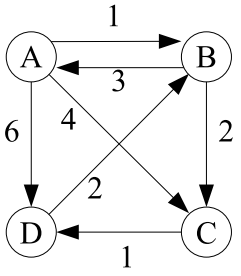
Sei $G = (V, E)$ ein Baum mit Gewichtsfunktion $w : E \to R^+$. Gesucht sind zwei Knoten $u, v \in V$, deren Abstand maximal ist (d.h. die Kosten des kürzesten Pfades von u nach v sollen maximal sein). Beschreiben und begründen Sie einen Algorithmus, der die beiden Knoten u und v und den Abstand von u und v in Laufzeit $O(|V|)$ bestimmt.

Aufgabe 2 (*Kürzeste Pfade* [**4 Punkte**])

1. Wenden Sie Dijkstras Algorithmus auf den folgenden Graphen G mit Startknoten S an. Geben Sie die Werte aller oberen Schranken nach jedem Durchlauf der Hauptschleife von Dijkstras Algorithmus in Form einer Tabelle an. Die Tabelle enthält also eine Spalte für jeden Knoten $v \in G$. Pro Iteration soll eine Zeile hinzugefügt werden, die die aktuellen oberen Schranken $\delta(S, v)$ enthält. Markieren Sie dabei in jeder Zeile den Knoten, den Sie auswählen. [2 Punkte]

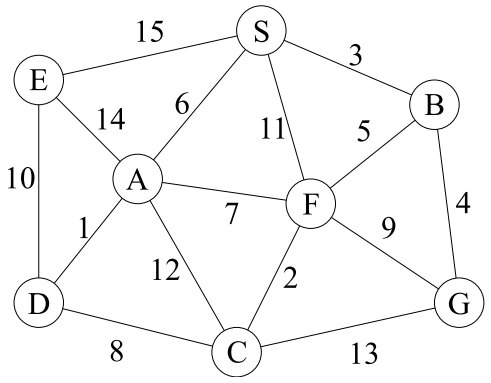


2. Wenden Sie den Algorithmus von Floyd-Warshall auf den unten angegebenen Graphen an, um die Längen aller kürzestens Pfade zu berechnen. Geben Sie die Distanzmatrix nach jedem Durchlauf der Hauptschleife an. [2 Punkte]



Aufgabe 3 (*Minimale Spannbäume* **[6 Punkte]**)

Gegeben sei folgender Graph:



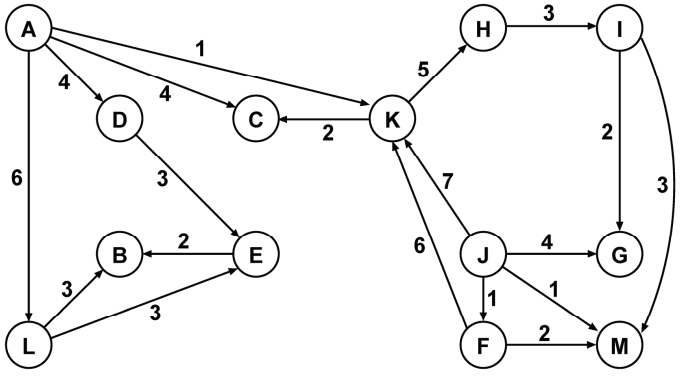
1. Wenden Sie den Algorithmus von Prim an, um einen minimal spannenden Baum des Graphen zu berechnen. Der Startknoten sei S. [3 Punkte]

2. Wenden Sie den Algorithmus von Kruskal an, um einen minimal spannenden Baum des Graphen zu berechnen. [3 Punkte]

Geben Sie in die Kanten in der Reihenfolge an, in der sie zur jeweiligen Lösungsmenge hinzugefügt werden. Da alle Kantengewichte verschieden sind, genügt es, jede Kante durch ihr Gewicht zu identifizieren.

Aufgabe 4 (*Topologisches Sortieren* [**4 Punkte**])

Gegeben sei der folgende gerichtete Graph G:



1. Bestimmen Sie für G eine topologische Sortierung und geben Sie die Knoten als sortierte Folge an. Die Kantengewichte können hierbei vernachlässigt werden. [3 Punkte]

2. Erweitern Sie G um eine beliebige Kante (ohne Gewicht), so dass keine topologische Sortierung für den modifizierten Graphen mehr existiert. [1 Punkt]

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 10

Abgabe: Montag, **08.07.2013**, 14:00 Uhr

- Die Übungen sollen in Gruppen von zwei bis drei Personen bearbeitet werden.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auf die abgegebenen Lösungen.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auch in die Quellcode-Dateien.
- Geben Sie Ihre Lösungen am **Anfang** der Globalübung, montags, 14:00 Uhr, ab.
- Schicken Sie den jeweiligen Quellcode bitte per **E-Mail** direkt an Ihre/n Tutor/in.
- Geben Sie außerdem den ausgedruckten Quellcode zusammen mit den schriftlichen Lösungen ab.
- Zu spät abgegebene Lösungen werden nicht bewertet.
- Sofern nicht anders gefordert, müssen alle Lösungen und Zwischenschritte kommentiert werden.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Aufgabe 1 (*Maximaler Abstand in Graphen* [**6 Punkte**])

Sei $G = (V, E)$ ein Baum mit Gewichtsfunktion $w : E \to R^+$. Gesucht sind zwei Knoten $u, v \in V$, deren Abstand maximal ist (d.h. die Kosten des kürzesten Pfades von u nach v sollen maximal sein). Beschreiben und begründen Sie einen Algorithmus, der die beiden Knoten u und v und den Abstand von u und v in Laufzeit $O(|V|)$ bestimmt.

Lösungsvorschlag

G ist ein Baum $\Rightarrow$ G ist ungerichtet, zusammenhängend und kreisfrei.
Idee: Dynamische Programmierung, bearbeite alle Teilbäume, ausgehend von den Blättern. Speichere pro Teilbaum mit Wurzel $v \in V$:

a) längsten Pfad p_v von v zu einem Blatt,

b) längsten Pfad P_v im Teilbaum mit Wurzel v.

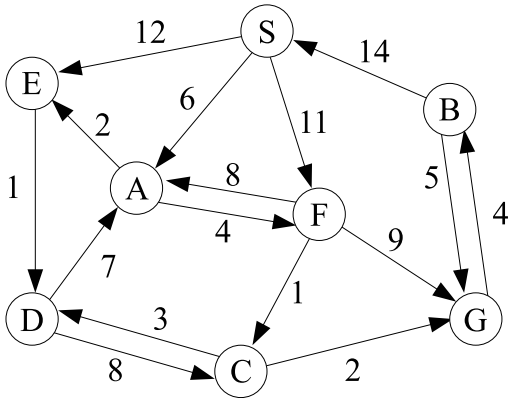
Initialisiere alle Blätter mit $P_v = p_v = \emptyset$.
Sei nun v ein innerer Knoten mit t_v bereits bearbeiteten Teilbäumen mit Wurzeln u_i, $1 \leq i \leq t_v$.

1. Berechne den Pfad p_v mit Länge
$$\max_i \{w(p_{u_i}) + w(v, u_i)\} \quad .$$
2. Berechne analog dazu den zweitlängsten Pfad p'_v von v zu einem Blatt.
3. Berechne längsten Pfad zwischen zwei Blättern, der v enthält. Dieser setzt sich aus den Pfaden p_v und p'_v zusammen. Vergleiche mit den längsten Pfaden der Teilbäume und finde
$$\max \left\{ \max_i \{w(P_{u_i})\}, w(p_v) + w(p'_v) \right\} \quad .$$
Der dazugehörige Pfad ist P_v.

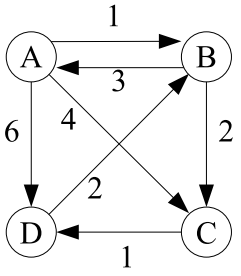
An der Wurzel des Baumes ist dann der unter 3. berechnete Pfad der insgesamt längste.
Laufzeit: Jeder Knoten $v \in V$ wird genau einmal besucht. Dabei müssen jeweils t_v Teilbäume betrachtet werden. Die Berechnung der Pfade p_v, p'_v und P_v benötigt einen Aufwand von jeweils $O(t_v)$. Insgesamt beträgt der Aufwand also
$$\sum_{v \in V} O(t_v) = O(|E|) = O(|V| - 1) = O(|V|) \quad .$$

Aufgabe 2 (*Kürzeste Pfade* [**4 Punkte**])

1. Wenden Sie Dijkstras Algorithmus auf den folgenden Graphen G mit Startknoten S an. Geben Sie die Werte aller oberen Schranken nach jedem Durchlauf der Hauptschleife von Dijkstras Algorithmus in Form einer Tabelle an. Die Tabelle enthält also eine Spalte für jeden Knoten $v \in G$. Pro Iteration soll eine Zeile hinzugefügt werden, die die aktuellen oberen Schranken $\delta(S, v)$ enthält. Markieren Sie dabei in jeder Zeile den Knoten, den Sie auswählen. [2 Punkte]



2. Wenden Sie den Algorithmus von Floyd-Warshall auf den unten angegebenen Graphen an, um die Längen aller kürzestens Pfade zu berechnen. Geben Sie die Distanzmatrix nach jedem Durchlauf der Hauptschleife an. [2 Punkte]



Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Lösungsvorschlag

1. Dijkstra:

	S	A	B	C	D	E	F	G	Vertex
0	0	6	∞	∞	∞	12	11	∞	S
1	0	6	∞	∞	∞	8	10	∞	A
2	0	6	∞	∞	9	8	10	∞	E
3	0	6	∞	17	9	8	10	∞	D
4	0	6	∞	11	9	8	10	19	F
5	0	6	∞	11	9	8	10	13	C
6	0	6	17	11	9	8	10	13	G
7	0	6	17	11	9	8	10	13	B

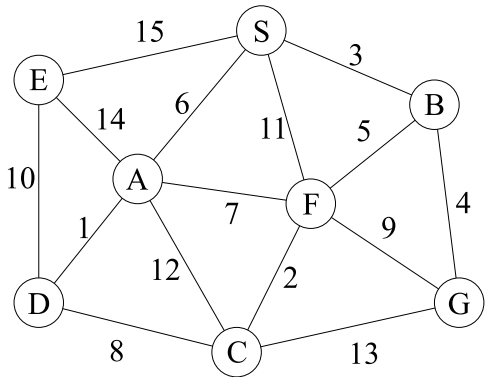
2. Floyd-Warshall:

$$D_0 = \begin{bmatrix} 0 & 1 & 4 & 6 \\ 3 & 0 & 2 & \infty \\ \infty & \infty & 0 & 1 \\ \infty & 2 & \infty & 0 \end{bmatrix}, D_1 = \begin{bmatrix} 0 & 1 & 4 & 6 \\ 3 & 0 & 2 & 9 \\ \infty & \infty & 0 & 1 \\ \infty & 2 & \infty & 0 \end{bmatrix}, D_2 = \begin{bmatrix} 0 & 1 & 3 & 6 \\ 3 & 0 & 2 & 9 \\ \infty & \infty & 0 & 1 \\ 5 & 2 & 4 & 0 \end{bmatrix},$$

$$D_3 = \begin{bmatrix} 0 & 1 & 3 & 4 \\ 3 & 0 & 2 & 3 \\ \infty & \infty & 0 & 1 \\ 5 & 2 & 4 & 0 \end{bmatrix}, D_4 = \begin{bmatrix} 0 & 1 & 3 & 4 \\ 3 & 0 & 2 & 3 \\ 6 & 3 & 0 & 1 \\ 5 & 2 & 4 & 0 \end{bmatrix}$$

Aufgabe 3 (*Minimale Spannbäume* **[6 Punkte]**)

Gegeben sei folgender Graph:



1. Wenden Sie den Algorithmus von Prim an, um einen minimal spannenden Baum des Graphen zu berechnen. Der Startknoten sei S. [3 Punkte]

2. Wenden Sie den Algorithmus von Kruskal an, um einen minimal spannenden Baum des Graphen zu berechnen. [3 Punkte]

Geben Sie die Kanten in der Reihenfolge an, in der sie zur jeweiligen Lösungsmenge hinzugefügt werden. Da alle Kantengewichte verschieden sind, genügt es, jede Kante durch ihr Gewicht zu identifizieren.

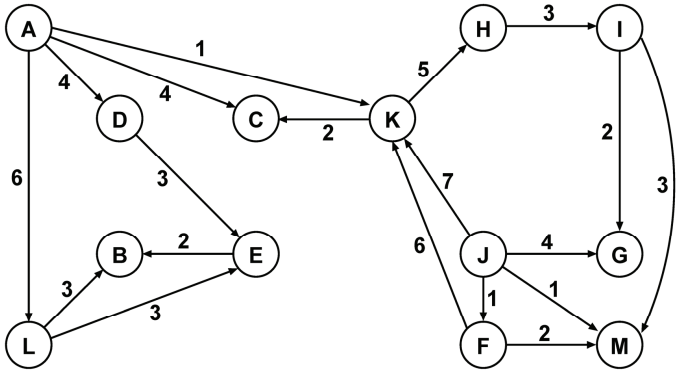
Lösungsvorschlag

1. Prim: $3, 4, 5, 2, 6, 1, 10$

2. Kruskal: $1, 2, 3, 4, 5, 6, 10$

Aufgabe 4 (*Topologisches Sortieren* [**4 Punkte**])

Gegeben sei der folgende gerichtete Graph G:



1. Bestimmen Sie für G eine topologische Sortierung und geben Sie die Knoten als sortierte Folge an. Die Kantengewichte können hierbei vernachlässigt werden. [3 Punkte]

2. Erweitern Sie G um eine beliebige Kante (ohne Gewicht), so dass keine topologische Sortierung für den modifizierten Graphen mehr existiert. [1 Punkt]

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Lösungsvorschlag

1. Da eine topologische Sortierung eines Graphen im Allgemeinen nicht eindeutig ist, sind mehrere Lösungen gültig. Hier ist eine mögliche Lösung angegeben:

$$J, F, A, K, H, I, M, G, C, L, D, E, B$$

2. Eine mögliche Kante, die einen Zyklus erzeugen würde, wäre (G, H). Auch hier gibt es viele Möglichkeiten. Jede Kante, die einen Zyklus in G erzeugt, ist eine gültige Lösung.