1 Ordnung und Suche (Lösungen)

› 1.1 In beiden Fällen gibt es viele unterschiedliche Kriterien, die für Zahlen oder für Texte lineare Ordnungen bestimmen. Wir zeigen hier nur einige Möglichkeiten.

Lösung a: Betrachten wir zuerst den klassischen Vergleich von Zahlen nach ihrer Grösse. Dadurch entsteht die nachstehende Folge:

-6, -3, -1, 0, 2, 3, 5, 7

Ein anderes, ebenfalls natürliches Kriterium ist, jeder Zahl ihren absoluten Wert zuzuordnen und die Zahlen nach ihren absoluten Werten zu vergleichen. Auf diese Art und Weise erhalten wir die nachstehenden Folgen:

0, -1, 2, 3, -3, 5, -6, 7 oder 0, -1, 2, -3, 3, 5, -6, 7.

Wir haben zwei mögliche Folgen, weil nach dem Kriterium des absoluten Wertes -3 und 3 gleich gross sind. Wir würden diese zwei Folgen auch erhalten, wenn wir als Kriterienwert jeder Zahl a ihr Quadrat a^2 zuordnen und die Grösse der Quadrate vergleichen würden.

Nehmen wir jetzt ein weiteres Kriterium, das jeder Zahl a den Wert K(a) = (10 + a) mod 3 zuordnet.

Ordnen wir dem Kriterium entsprechend die Werte für unsere Zahlen:

K(5) = (10 + 5) mod 3 = 0
K(3) = (10 + 3) mod 3 = 1
K(0) = (10 + 0) mod 3 = 1
K(-1) = (10 + (-1)) mod 3 = 0
K(-6) = (10 + (-6)) mod 3 = 1
K(-3) = (10 + (-3)) mod 3 = 1
K(2) = (10 + 2) mod 3 = 0
K(7) = (10 + 7) mod 3 = 2

Wir sehen, dass bezüglich dieses Kriteriums vier Elemente (-6, -3, 3, 0) gleich sind und die drei Elemente -1, 2 und 5 ebenfalls gleich bewertet sind. Somit kann man viele nicht absteigende Folgen nach diesem Kriterium generieren, zum Beispiel:

2, -1, 5, -6, -3, 0, 3, 7
-1, 5, 2, 0, 3, -3, -6, 7
2, 5, -1, 0, -3, -6, 3, 7

Lösung b: Die Grundlage für die lexikographische Ordnung ist die Ordnung der Buchstaben des lateinischen Alphabets a, b, c, ..., x, y, z. Wenn man kleine sowie grosse Buchstaben verwendet, kann man z. B. jeweils den kleineren Buchstaben vor den grösseren setzen:

a, A, b, B, c, C, ..., y, Y, z, Z.

In der gegebenen Menge haben wir in Dateinamen auch Leerzeichen. Leerzeichen können wir auch als Symbole des Alphabets betrachten und sie unter den Symbolen ganz am Ende einordnen. Wir können beliebige Symbole der Rechnertastatur einbeziehen, wenn wir ihre Ordnung innerhalb des Alphabets festlegen. Nach der lexikographischen Ordnung erhalten wir die folgende Folge der Dateinamen:

Bergrestaurant, freie Wölfe und Bären, Naturmuseum, Spielplatz, Zoo

Wir können im Alphabet alle kleinen Buchstaben allen Grossbuchstaben vorziehen, indem wir die Ordnung des Alphabets a, b, c, ..., x, y, z, A, B, C, ..., X, Y, Z wählen. In diesem Fall sieht die resultierende Folge der Dateinamen anders aus:

freie Wölfe und Bären, Bergrestaurant, Naturmuseum, Spielplatz, Zoo

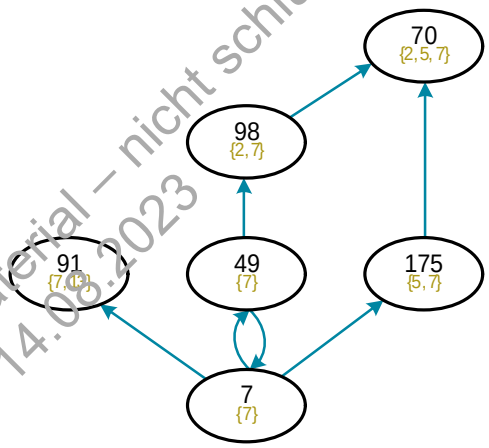
Als Letztes betrachten wir die sogenannte **kanonische Ordnung**. Hier werden die Texte zunächst nach ihrer Länge (Anzahl der Zeichen) von kürzeren zu längeren aufsteigend geordnet. Für Texte gleicher Länge bestimmt man dann die genaue Reihenfolge lexikographisch. Für die betrachteten Dateinamen entsteht somit die folgende Folge:

Zoo, Spielplatz, Naturmuseum, Bergrestaurant, freie Wölfe und Bären.

W 1.2 Die Zahlen 1 und 2310 sind vergleichbar mit allen anderen Zahlen in der Menge und man kann 1 als das Minimum und 2310 als das Maximum dieser Menge bezeichnen. Die folgenden Paare von Zahlen sind nicht vergleichbar bezüglich der Relation «kleiner gleich bezüglich der Faktorisierung»:

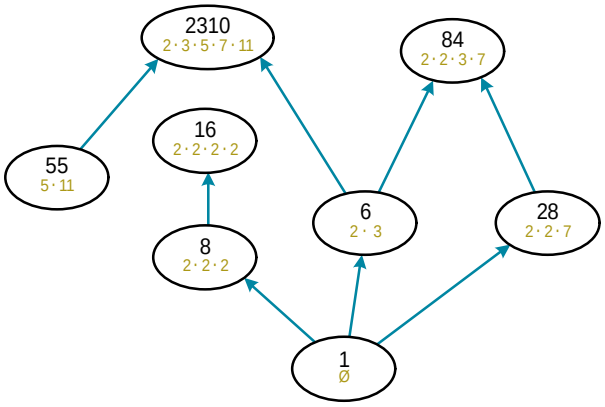
(55, 8), (55, 6), (55, 16), (55, 28), (55, 84), (6, 28).

W 1.3 Lösung: Wir erhalten beispielsweise den folgenden Graphen:



Hier sind die Zahlen 7 und 49 kleiner gleich alle Zahlen in dieser Menge, also kann man 7 und 49 als Minima dieser Menge betrachten. Es gibt keine Zahl, die grösser gleich alle anderen Zahlen dieser Menge ist. Also hat diese Menge kein Maximum bezüglich der betrachteten Relation «kleiner gleich bezüglich der Faktorisierung».

W 1.4 Lösung: In der neuen Definition von «kleiner gleich bezüglich der Faktorisierung» zählt das Vielfache der einzelnen Primfaktoren. Der resultierende Graph sieht wie folgt aus:

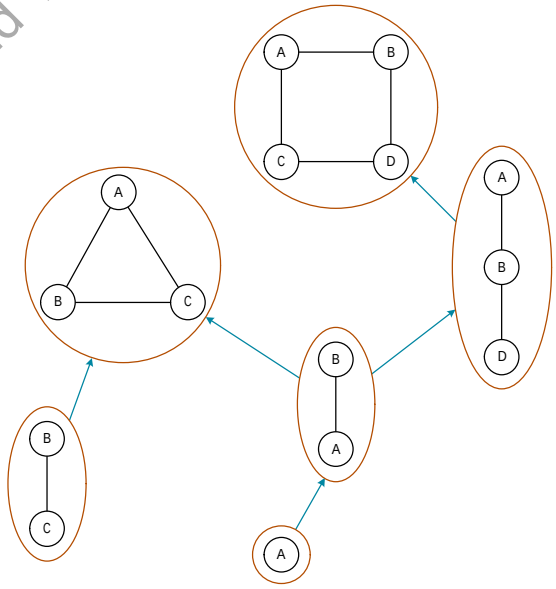


Wir sehen, dass bezüglich dieser Relation 1 das Minimum ist. Ein Maximum existiert nicht, weil kein Element grösser gleich 16, 84 und 2310 ist. Welche ist die kleinste Zahl x, die ein Maximum der Menge {1, 6, 8, 16, 28, 55, 84, 2310, x} wäre?

W 1.5 Lösung: Dieses Kriterium ordnet jeder Zahl eine Bewertung zu, die wiederum eine Zahl ist. Somit erhalten wir eine lineare Ordnung, weil beliebige zwei Zahlen ihren Werten nach vergleichbar sind. Wir sehen dies auch an dem gerichteten Graphen für die Menge {1, 6, 8, 16, 28, 55, 84, 2310}:

Wir sehen hier, dass 8 «gleich» 16 ist, und dass 6, 28 und 55 ebenfalls paarweise «gleich» sind. In gerichteten Graphen erkennt man, dass a und b «gleich» sind, daran, dass ein gerichteter Weg von a nach b sowie von b nach a existiert.

W 1.6 Lösung:



Finden Sie den Graphen mit der kleinsten Anzahl an Kanten, der das Maximum bilden wird, wenn man ihn als zusätzlichen Graphen zu der Graphenmenge hinzufügt.

› 1.7

Lösung a: Ein Maximum kann man immer mit neun derartigen Operationen finden, allgemein mit «der Anzahl der Karten minus 1» vielen Operationen. Die Strategie ist einfach. Man verwendet die Operation, um zuerst die erste und zweite Karte zu «vergleichen», danach die zweite und dritte, die dritte mit der vierten und am Ende die neunte mit der zehnten. Egal, wo sich die Karte mit dem maximalen Wert befindet, nachdem man diese Karte aufgedeckt hat, «wandert» sie in jeder nachfolgenden Operation eine Position nach rechts. Somit steht nach dem Vergleich der neunten und der zehnten Karte das Maximum an der zehnten Position ganz rechts.

Lösung b: Zum Entwurf eines Sortieralgorithmus für die verdeckten Karten nutzen wir die Modularität, wie wir sie beim Programmieren (Band 1, «Programmieren und Robotik») gelernt haben. Mit neun Operationen, wie in Teilaufgabe a beschrieben, platzieren wir die Karte mit dem grössten Wert ganz rechts. Diese Karte muss man nicht mehr umplatzieren und auch nicht mit anderen Karten zu vergleichen. Für die restlichen neun Karten gehen wir die Suche nach dem Maximum wie in Teilaufgabe a an. Somit erhalten wir mit acht Operationen den zweitgrössten Wert auf der neunten Position, auf die diese Karte auch definitiv hingehört. Danach betrachten wir die 8 verbliebenen Karten und so weiter. Somit ist die Anzahl der Vergleiche insgesamt:

$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45$$

Probieren Sie es selbst mit unsortierten Karten aus, bei denen die lineare Ordnung definiert ist, oder spielen Sie es in der interaktiven Lernumgebung auf https://einfachinformatik.inf.ethz.ch/kindergarten/#/sort/bubbleSort/2 durch.

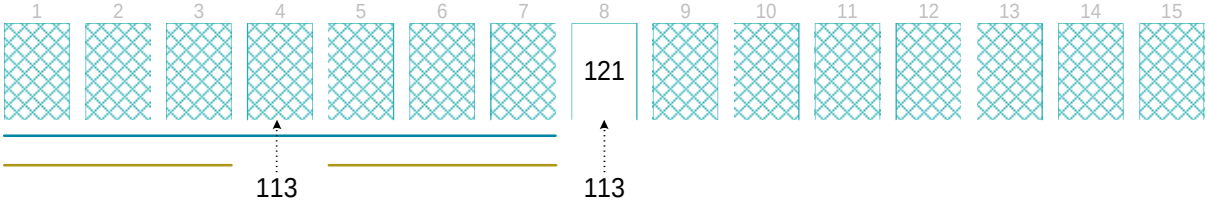
‹ 1.8

Lösung a: Wenn man einmal die Vergleiche der Nachbarn von links nach rechts durchgeführt und dabei keine zwei Elemente vertauscht hat, dann ist man sicher, dass die Folge schon sortiert ist – das erste Element ist kleiner als das zweite, das zweite Element ist kleiner als das dritte usw. Deswegen kann man Bubblesort anhalten, wenn man in einem Durchlauf von links nach rechts keine Änderung der Reihenfolge vorgenommen hat. Beachten Sie hierzu die Lösung von Aufgabe 5.12 im Band «Programmieren und Robotik».

Lösung b: Die herausforderndste Eingabe ist die absteigend sortierte Folge n, n-1, …, 3, 2, 1. Man braucht n-1 Vergleiche mit Vertauschen, um das grösste Element von links nach rechts aufsteigen zu lassen. Für das zweitgrösste Element braucht man n-2 «Schritte» (Verschiebungen) nach rechts usw.

› 1.9

Lösung: Die 15 sortierten Karten liegen auf den Positionen 1, 2, …, 15. Die mittlere Position ist hier die Position 8. Sieben Karten liegen rechts von dieser Position und sieben Karten liegen links von dieser Position.



Wir drehen die Karte auf Position 8 um und vergleichen ihre Zahl mit 113. Falls auf der Karte auf Position 8 die Zahl 113 steht, sind wir fertig. Falls die Zahl auf Position 8 kleiner als 113 ist, befindet sich die gesuchte Karte auf einer der Positionen 9 bis 15. Falls 113 kleiner als die Zahl auf Position 8 ist, wissen wir, dass die gesuchte Karte links, also auf einer der Positionen 1 bis 7 liegt. Somit reduziert sich der Suchraum mit dem Aufdecken einer Karte auf weniger als die Hälfte. Im Beispiel in der Abbildung zeigt die aufgedeckte Karte die Zahl 121, weshalb sich der verbleibende Suchraum (türkis markiert) links von der aufgedeckten Karte auf die Hälfte reduziert.

Jetzt verwendet man die gleiche Strategie, um den restlichen Suchraum wieder zu halbieren. Bei sieben Karten ist 4 die mittlere Position und somit decken wir diese Karte auf. Danach haben wir entweder die Karte mit 113 gefunden oder den Suchraum (olivgrün markiert) auf 3 Karten links oder rechts der aufgedeckten Karte reduziert.

Beim Suchraum bestehend aus drei Karten drehen wir wieder die Karte in der Mitte um. Wenn wir im Vorhinein wissen, dass sich die Karte 113 in der sortierten Folge befindet, sind wir fertig. Entweder ist es die aufgedeckte Karte oder die benachbarte Karte links von der letzten aufgedeckten Karte oder die benachbarte Karte rechts. Somit reichen drei Umdrehoperationen, um eine gesuchte Karte unter den 15 sortierten Karten zu finden. Wenn man nicht im Vorhinein weiss, ob die Karte mit 113 in der Folge vorhanden ist, muss man im schlimmsten Fall eine weitere Karte umdrehen.

W 1.10 Lösung: Die interaktive Lernumgebung gibt automatisch die Rückmeldung zu Ihrer Aktivität.

W 1.11 Lösung a: Wenn man häufig mit binären Zahlen arbeitet, weiss man auswendig, dass $2^{10} = 1024$ und $2^9 = 512$ ist. Somit ist $\text{dislog}_2(1000) = 10$. 10 Vergleiche reichen, um ein Element unter 1000 zu finden, wenn sicher ist, dass es vorkommt.

Lösung b: Man kann experimentell den diskreten Logarithmus mit Basis 2 von einer Million bestimmen. Einfacher ist es zu beobachten:

$$10^6 = 10^3 \cdot 10^3 < 1024 \cdot 1024$$

Das bedeutet, dass man mit zehn Vergleichen einen Suchraum mit einer Million Elemente auf eine Grösse kleiner als 1000 reduziert. Weitere zehn Vergleiche reichen, um das gesuchte Element im Suchraum kleiner als 1000 zu finden. Tatsächlich ist es so, dass $\text{dislog}_2(1000000) = 20$ ist. Beeindruckend ist, dass 20 Vergleiche reichen, um das gesuchte Element in einer sortierten Folge von einer Million Elementen zu finden.

Lösung c: Hier setzen wir die Strategie aus Teilaufgabe b fort. Es gilt:

$$10^9 = 10^3 \cdot 10^3 \cdot 10^3 < 1024 \cdot 1024 \cdot 1024$$

Somit reichen 10 Vergleiche, um einen Suchraum von einer Milliarde Elemente auf einen Suchraum mit weniger als einer Million Elemente zu reduzieren. Wenn wir jetzt die Lösung aus Teilaufgabe b anwenden, sehen wir, dass 30 Vergleiche reichen, um ein gesuchtes Element in einer sortierten Folge von einer Milliarde Elementen zu finden. Dies dokumentiert ausdrücklich die Effizienz der binären Suche. Tatsächlich gilt:

$$\text{dislog}_2(1\,000\,000\,000) = 30$$

W 1.12 Lösung: Die notwendige Speichergrösse für x ist 700 · 64 = 44800 Bits (oder 1400 = 700 · 2 adressierbare Speichereinheiten). Wenn x[0] an den Adressen 313 und 314 liegt, steht x[1] an den Adressen 315 und 316, x[2] an den Adressen 317 und 318 usw. Allgemein steht der Wert von x[i] für i = 0, 1, ..., 700 an den Adressen:

$$313 + 2i \text{ und } 313 + 2i + 1$$

‹ 1.13 Lösung: Hier gibt es eine Vielzahl möglicher Lösungsstrategien. Eine natürliche Strategie ist es, die Matrix m[i, j] zeilenweise zu speichern; das bedeutet, die Folge

m[0, 0], m[0, 1], m[0, 2], ..., m[0, 9], m[1, 0], m[1, 1], m[1, 2], ...,
m[1,9], m[2,0], m[2, 1], ..., m[9, 0], m[9, 1], ..., m[9,8], m[9, 9]

im Speicher abzuspeichern.

Was bedeutet es dann, den Wert m[i, j] zu finden? Der Wert m[i, j] ist in diesem Fall an der Adresse

$$i \cdot 10 + j$$

gespeichert. Wenn die Matrix m[i, j] auf diese Art und Weise in einem eindimensionalen Array x gespeichert wird, würde dies bedeuten: m[i, j] = x[i · 10 + j].

Mit einem solchen Vorgehen kann man zweidimensionale Arrays im Speicher implementieren und der Computer kann für jede Variable m[i, j] die Adresse der Speichereinheit für m[i, j] selbstständig ausrechnen und somit auf den Wert von m[i, j] direkt zugreifen.

W 1.14 Lösung: Wir zeichnen eine Tabelle. Die Zeilen der entstandenen Tabelle entsprechen den Registern, für die wir die dynamische Änderung ihrer Inhalte beobachten. Die Spalten entsprechen den Zeitpunkten der Beobachtung, die nach der Ausführung der angegebenen Zeile festgehalten sind.

	4	7	4	7	4	7	4	7	4	7	4
REG(0)	5	10	5	10	5	10	5	10	5	10	5
Register(1000)	222	222	222	222	222	222	222	222	222	222	222
Register(1001)	0	0	0	0	0	0	0	0	62	62	62
Register(1002)	999	999	498	498	248	248	123	123	123	123	123
Register(1003)	0	0	0	0	0	0	0	0	62	62	93
Register(1004)	-	499	499	249	249	124	124	61	61	92	92
Register(1005)	2	2	2	2	2	2	2	2	2	2	2

Man kann die Simulation leicht verfolgen, wenn man wahrnimmt, dass sich in jedem Register(i) die Zahl 2i für i = 0, 1, ..., 999 befindet. Wir sehen, dass sich nach fünf Durchläufen der Schleife (die in Zeile 4 anfängt) der Suchraum auf den Bereich von Register(93) bis zu Register(123) reduziert hat. Das entspricht einer Grösse von 31 Registern. Simulieren Sie das Programm weiter, bis es den Index 111 des gespeicherten Wertes 222 findet.

W 1.15

Lösung a: Inhalt*(Register(1)) = Inhalt(Register(Inhalt(Register(1)))) = Inhalt(Register(4)) = 1

Lösung b: Inhalt*(Register(2)) = Inhalt(Register(Inhalt(Register(2)))) = Inhalt(Register(72)) = 13

Lösung c: Inhalt*(Register(3)) = Inhalt(Register(Inhalt(Register(3)))) = Inhalt(Register(1)) = 4

Lösung d: Inhalt*(Register(5)) = Inhalt(Register(Inhalt(Register(5)))) = Inhalt(Register(5)) = 5

‹ 1.16

Lösung: Zuerst vereinbaren wir, dass der Wert von Variable i, die in der Schleife von i = 51 bis i = 61 läuft, in Register(1) gespeichert wird. In Register(2) halten wir den Wert i+1. Register(3) verwenden wir als eine Hilfsvariable für das Vertauschen von Inhalten auf die Adressen i und i+1.

```
0   Register(1) ← 51
1   Register(2) ← 52
2   if Register(1) = 62 GOTO 10
3   if Inhalt*(Register(1)) ≤ Inhalt*(Register(2)) GOTO 7
4   Register(3) ← Inhalt*(Register(1))
5   Register(Inhalt(Register(1))) ← Inhalt*(Register(2))
6   Register(Inhalt(Register(2))) ← Inhalt*(Register(3))
7   Register(1) ← Inhalt(Register(1)) + 1
8   Register(2) ← Inhalt(Register(2)) + 1
9   GOTO 2
10  HALT
```

Simulieren Sie die Arbeit des Programms mit einer Tabelle mit 15 Registern und REG(0), indem Sie die Inhalte aller Register jeweils nach der Ausführung der Programmzeile 2 dokumentieren.

› 1.17

Lösung: Eine allgemeine Strategie kann wie folgt aussehen:

Mit Vorteil beginnt man hinten und schreibt den Inhalt des vorletzten Felds ins letzte Feld, danach ins vorletzte Feld den Inhalt eines Feldes davor, und immer so weiter, bis der Inhalt vom dritten Feld ins vierte Feld kopiert wurde. Anschliessend kann der neue Name ins dritte Feld geschrieben werden.

Beachten Sie dabei, dass der vormalige Inhalt des letzten Feldes so verloren gehen würde, falls das Feld nicht leer wäre.

Formal kann man die entsprechenden Zuweisungsoperationen für unser Beispiel wie folgt beschreiben:

```
baum[6] ← baum[5]
baum[5] ← baum[4]
baum[4] ← baum[3]
baum[3] ← baum[2]
baum[2] ← "Eiche"
```

Wir sehen, dass wir fünf Zuweisungen gebraucht haben. Mit weniger als 5 kann es nicht gelingen, weil am Ende die fünf Felder baum[2], baum[3], baum[4], baum[5] und baum[6] neue Inhalte haben müssen. Wenn das Array jetzt n Felder baum[0], …, baum[n-1] gehabt hätte und wir die Inhalte ab dem zweiten Feld verschieben müssten, hätte dies n-2 Zuweisungsoperationen bedeutet (siehe auch Beispiel 5.5. in «Programmieren und Robotik»).

Natürlich könnte man bei den Verschiebungen auch anders vorgehen, indem man zwei Hilfsvariablen x und y verwendet, um die Einträge bei den Übertragungen zwischenspeichern zu können. Sehen Sie sich auch Beispiel 5.5 in dem Band «Programmieren und Robotik» an.

```
X ← baum[2]
baum[2] ← „Eiche“
y ← baum[3]
baum[3] ← x
x ← baum[4]
baum[4] ← y
y ← baum[5]
baum[5] ← x
baum[6] ← y
```

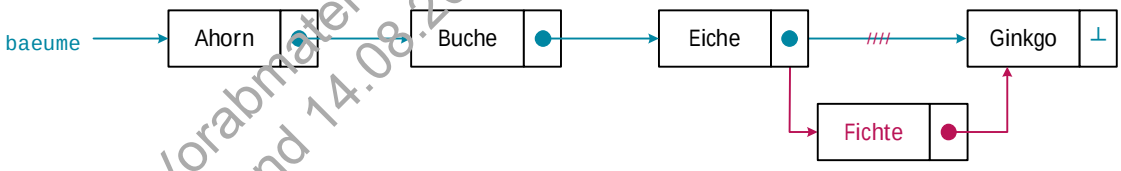
Wir sehen, dass wir bei dieser Vorgehensweise sogar neun Zuweisungsoperationen benötigt hätten.

W 1.18 Lösung: Ein Element zu entfernen, scheint einfacher zu sein, als ein Element hinzuzufügen. Wir brauchen definitiv keine Zwischenspeicherungen, weil das zur Entfernung bestimmte Element nicht aufbewahrt werden muss. Somit sieht die Lösung ganz einfach aus:

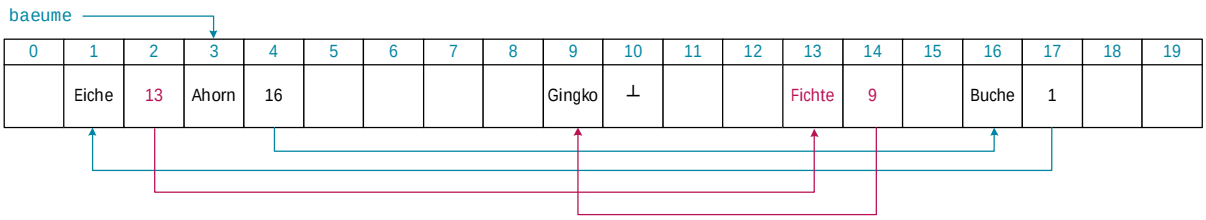
```
baum[3] ← baum[4]
baum[4] ← baum[5]
baum[5] ← ⊥
```

Die letzte Zuweisung ist nicht notwendig. Sie leert einfach das Feld baum[5]. Das Zeichen ⊥, genannt Nil, verwenden wir für einen leeren Inhalt.

W 1.19 Lösung a: Das neue Element «Fichte» sollte direkt nach «Eiche» und direkt vor «Ginkgo» eingeordnet werden. Abstrakt, ohne Adressenangabe, soll die Änderung wie in der folgenden Abbildung umgesetzt werden.



In der vollständigen Umsetzung mit Speicherzellenadressen sieht die Speicherung der Liste `baeume` wie in der Abbildung unten aus. Die Fichte liegt an den Adressen 13 und 14 und deswegen setzen wir an die Adresse 14 den Zeiger 9 auf «Ginkgo». Die Adresse (der Zeiger) der zweiten Komponente des Elements «Eiche» (in Speicherzelle 2) ändert sich auf 13 und somit wird «Fichte» der Nachfolger von «Eiche».

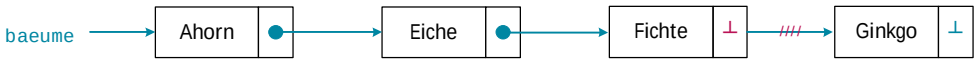


Ein Programm in Makro-Assembler zur Umsetzung der Einfügung von «Fichte» könnte wie folgt aussehen:

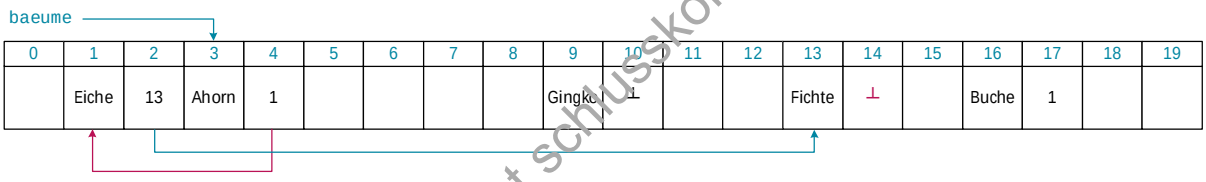
```
Register(13) ← „Fichte“
Register(14) ← Inhalt(Register(2))
Register(2) ← 13
```

Lösung b: Die Umsetzung der Entfernung des Elements „Buche“ sieht auf abstrakter Ebene wie folgt aus:

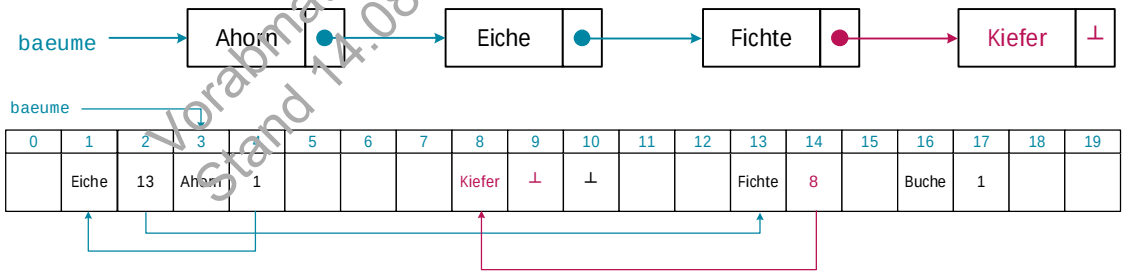
Es ist interessant zu beobachten, dass der Eintrag «Buche» dabei nicht gelöscht wird. Es führt allerdings kein Zeiger mehr zu dieser Adresse und somit wird der Speicherinhalt nicht verwendet. Dies bedeutet auch, dass die entsprechenden zwei Adressen 16 und 17 für neue Belegungen zur Verfügung stehen. Deswegen ist es auch nicht wichtig, ob man die alten Inhalte hier sofort löscht oder später einfach überschreibt. Wenn man das letzte Element «Ginkgo» entfernen will, muss man den Zeiger von «Fichte» auf ⊥ setzen und somit wird «Fichte» das letzte Element der Liste `baeume`.



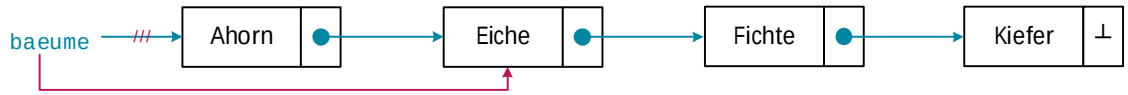
Der tatsächliche Inhalt im Speicher nach der Entfernung von «Buche» und «Ginkgo» aus der Liste sieht wie folgt aus:



Lösung c: Wenn man «Kiefer» in die lexikographische Ordnung einfügt, wird «Kiefer» das letzte Element der Liste. In abstrakter Darstellung sieht dies so aus, dass ⊥ in der zweiten Komponente von «Fichte» an Adresse 14 (das alte Ende der Liste) durch den Zeiger 8 ersetzt wird. An die Adresse 8 kommt «Kiefer» und an Adresse 9 wird «Ginkgo» mit ⊥ überschrieben.

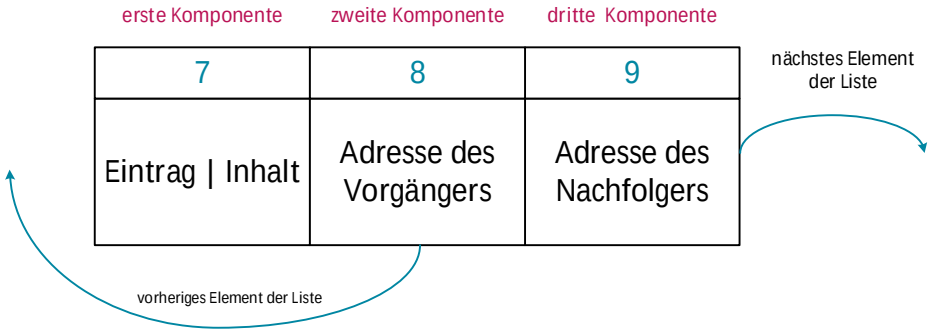


Lösung d: Das nullte Element und somit den Anfang der Liste kann man nur so entfernen, dass der Zugang zur Liste aus der Tabelle der Variablen nicht verloren geht. Die einzige Möglichkeit ist, den Zeiger aus dem Tabelleneintrag `baeume` auf „Eiche“ umzustellen. Zeichnen Sie den tatsächlichen Inhalt im Speicher nach der Entfernung von «Ahorn» aus der Liste.

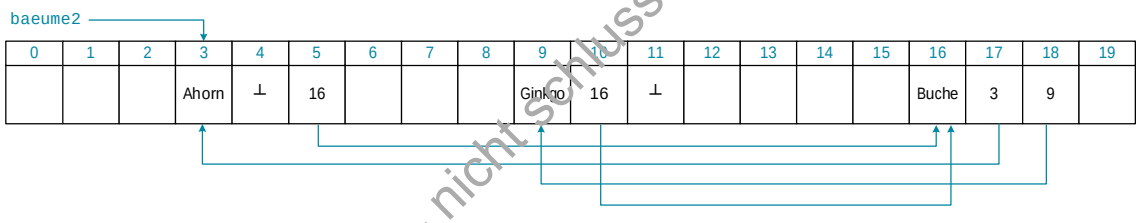


‹ 1.20

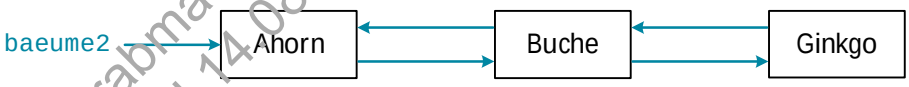
Lösung a: In einer einfach verketteten Liste hat jedes Element einen Zeiger auf das nachfolgende Element. Um sich in beide Richtungen in der Liste bewegen zu können, brauchen wir für jedes Element zusätzlich einen Zeiger auf den Vorgänger in der Liste. Deswegen entwerfen wir die Liste so, dass die Elemente drei Komponenten haben – Inhalt (gespeicherte Daten), Adresse des Vorgängers und Adresse des Nachfolgers.



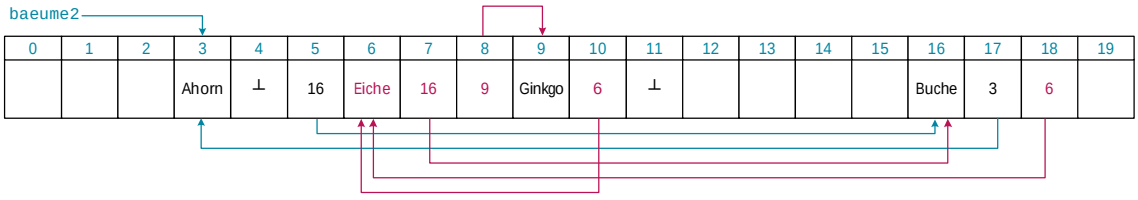
Bei der Implementierung der Liste aus Beispiel 1.5 müssen wir nur darauf achten, dass wir für jedes Element der Liste drei adressierbare, aufeinanderfolgende Speichereinheiten zur Verfügung haben.



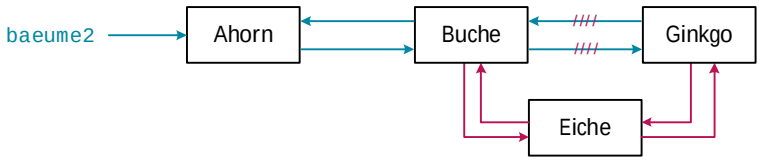
Eine vereinfachte Darstellung der Liste baeume2 ohne konkrete Adressen könnte wie folgt gezeichnet werden:



Lösung b: Wenn wir zu der Liste baeume2 das Element «Eiche» hinzufügen und die lexikographische Ordnung halten wollen, muss «Eiche» zwischen «Buche» und «Ginkgo» platziert werden. Die drei Speichereinheiten 6, 7 und 8 sind frei (kein Zeiger verweist auf sie, siehe Abbildung oben) und somit kann das Element «Eiche» in diesen drei Speichereinheiten platziert werden:



Das Einfügen von „Eiche“ in baeume2 kann in der abstrakten Darstellung wie folgt veranschaulicht werden:



In Makro-Assembler kann man das Einfügen von „Eiche“ mit folgenden Instruktionen bewirken:

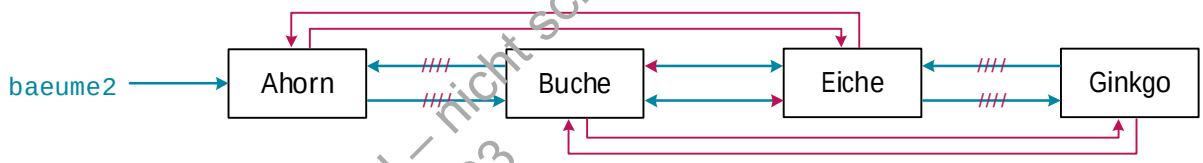
```
Register(6) ← "Eiche"
Register(7) ← Inhalt(Register(10))
Register(8) ← Inhalt(Register(18))
Register(10) ← 6
Register(18) ← 6
```

Lösung c: Die erste Idee könnte sein, die Inhalte „Eiche“ und „Buche“ auszutauschen und alle Zeiger stehen zu lassen. In Makro-Assembler würde dies wie folgt aussehen:

```
Register(1) ← Inhalt(Register(16))
Register(16) ← Inhalt(Register(6))
Register(6) ← Inhalt(Register(1))
```

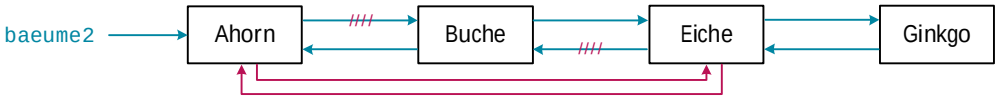
Register(1) ist frei und wir haben es nur verwendet, um den Austausch der Inhalte der Speichereinheiten 6 und 16 umzusetzen. Diese Umsetzung der Idee hat aber zwei Schwächen. Man braucht dazu die globale Sicht und muss wissen, auf welchen Adressen sich Eiche und Buche im Speicher befinden. Zweitens können hinter "Eiche" und "Buche" als Schlüssel grosse Dateien sein, die man nicht umplatzieren will.

Eine andere Möglichkeit wäre, die Inhalte der Elemente stehen zu lassen und die Zeiger anzupassen. Abstrakt sieht dies wie folgt aus:

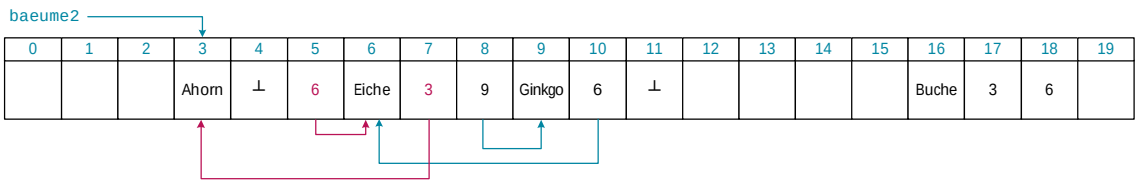


Wir müssen hier sechs (in unserem Fall alle) Zeiger ändern. Zwischen «Buche» und «Eiche» wechseln die Rollen und die Richtungen der Zeiger. Das Element «Buche» ist jetzt Nachfolger von «Eiche» und «Eiche» ist somit der Vorgänger von «Buche». Setzen Sie diese Implementierung in der Abbildung mit physischen Speicherzellen um.

Lösung d: In der abstrakten Darstellung überspringt man einfach das Element «Buche» mittels direkter Zeigerverbindung zwischen «Ahorn» und «Eiche»:



In der physischen Umsetzung im Speicher sehen wir, dass es reicht, tatsächlich nur zwei Zeiger (Adressen) zu ändern – dass Zeiger aus dem Element „Buche“ herausgehen, stört dabei nicht. Diese Zeiger muss man nicht löschen, weil zum Element „Buche“ kein Zeiger führt, und somit sind die entsprechenden drei Speichereinheiten für eine neue Belegung frei verfügbar:



W 1.21

Lösung a: Es handelt sich eindeutig um die einfach verkettete Liste. Carla kann den Weg nur in eine Richtung gehen und sieht das Datum nur, wenn sie auf der Lichtung (auf dem Speicherplatz) steht. Sie muss den Weg von Lichtung zu Lichtung gehen, bis sie die gesuchte 7 findet.

Lösung b: Wenn Carla Glück hat, ist die 7 schon auf der ersten Lichtung. Wenn Carla Pech hat, ist die 7 auf der letzten Lichtung, und sie muss alle 82 Lichtungen besuchen.

Lösung c: Solange man beliebige ganze Zahlen auf der Lichtung hat und somit 7 die sowohl kleinste als auch grösste Zahl sein kann, hilft die Sortierung der Elemente der Liste für die Suche nach 7 nicht. Wenn aber nicht im Voraus klar wäre, dass die 7 sicher kommen wird, dann könnte sie in der ersten Lichtung zu suchen aufhören, in der eine Zahl erscheint, die grösser als 7 ist.

Lösung d: Wenn die 7 in der Suche gleich häufig an jeder Position von 1 bis 82 vorkommt, ist die durchschnittliche Anzahl der besuchten Lichtungen:

$$\frac{1 + 2 + 3 + 4 + \ldots + 82}{82} = 41.5$$

W 1.22

Lösung: Nach der Ausführung der angegebenen Anweisungen entsteht die Liste mit den fünf Elementen `X0, X1, X2, X3` und `X4` aus der Abbildung:

Obwohl wir `X4 = Knoten(77)` ohne die Eingabe des zweiten Parameters `naechste` erzeugt haben, wird `X4` generiert und `self.naechste` wird auf `None` gesetzt. Das passiert nur deswegen, weil wir in

```
def __init__(self, inhalt=0, naechste=None)
```

den Standardwert von `naechste` auf `None` gesetzt haben.

Mit `print(X4)` erhalten wir den Inhalt 77.

Mit `print(X0.naechste)` erhalten wir den Inhalt 22 von `X1`, weil `X0.naechste` der Knoten `X1` ist.

Mit `print(X1.naechste.naechste)` erhalten wir den Inhalt 10 von `X3`, weil `X1.naechste` der Knoten `X2` ist, und `X1.naechste.naechste = X2.naechste` und somit `X3` ist.

Mit `print(neueListe.anker)` erhalten wir den Inhalt 77 von `X0`, weil `neueListe.anker` der Knoten `X0` ist.

Mit `y = X0.naechste.naechste` erhält die Variable y den Knoten `X2`. Somit ist das Resultat von `print(y)` der Inhalt 5 von `X2`.

W 1.23 Lösung: Wir erzeugen zunächst die Knoten der Liste von rechts nach links und dann verankern wir die Liste am Knoten `Y0`, der am weitesten links steht.

```
1 Y2 = Knoten(-50, None)
2 Y1 = Knoten(100, Y2)
3 Y0 = Knoten(-7, Y1)
4 drei = Liste(Y0)
```

‹ 1.24 Lösung: Auch wenn die in der Abbildung gezeichnete Gesamtstruktur keine einfach verkettete Liste als Datenstruktur ist, sehen wir, dass man sie mit unseren Klassen `Knoten` und `Liste` leicht erzeugen kann. Wir erzeugen zunächst die letzten zwei Knoten `Z0` und `Z1` mit den Inhalten 0 und 7 und danach die zwei Knoten `X1` und `Y1` mit den Inhalten 5 und 6, wobei diese beiden Knoten denselben Nachfolger `Z0` besitzen (es gilt `X1.naechster = Y1.naechster = Z0`).

```
1 Z1 = Knoten(0, None)
2 Z0 = Knoten(7, Z1)
3 X1 = Knoten(6, Z0)
4 Y1 = Knoten(5, Z0)
5 X0 = Knoten(4, X1)
6 Y0 = Knoten(3, Y1)
7 liste1 = Liste(X0)
8 liste2 = Liste(Y0)
```

Somit erhalten wir zwei Listen `X0, X1, Z0, Z1` und `Y0, Y1, Z0, Z1`, die beide dieselben beiden letzten Knoten (Elemente) besitzen.

‹ 1.25 Lösung: Wir erzeugen hier eine neue Klasse `Konto` mit drei Komponenten `knummer`, `inhaber` und `kstand`.

```
1  class Konto:
2      def __init__(self, knummer, inhaber, kstand = 0)
3          self.knummer = knummer
4          self.inhaber = inhaber
5          self.kstand = kstand
6
7      def __str__(self):
8          return str(self.kstand)
9
10 konto0 = Konto(77777, "Lucky_Luke", 1000000)
11 print(konto0)
```

In Zeile 2 steht beim dritten Parameter `kstand = 0` (der Standardwert wird auf 0 gesetzt). Das bedeutet, dass, wenn man hier keinen Parameterwert angibt, der Kontostand `kstand` automatisch auf null gesetzt wird. Somit erzeugt die Anweisung

```
konto007 = Konto(007, "Agent")
```

das Konto

`007, "Agent", 0.`

W 1.26 Lösung:

```
1 liste26 = Liste(None)
2 liste26.einfuegen(-4)
3 liste26.einfuegen(3)
4 liste26.einfuegen(-7)
5 liste26.einfuegen(6)
6 liste26.einfuegen(12)
7 print(liste26)
```

W 1.27 Lösung: Die folgende Methode `suchen()` liefert beim Aufruf `drei.suchen(100)` den ersten Knoten von links in der Liste drei aus Beispiel 1.7, der den Inhalt `100` hat. Falls die `drei` leer ist oder kein Knoten den Inhalt `100` hat, gibt die Methode `None` zurück.

```
def suchen(self, inhalt):
    knoten = self.anker
    while knoten != None and knoten.inhalt != inhalt:
        knoten = knoten.naechster
    return knoten
```

Beachten Sie bitte folgendes: Die Methode `suchen` muss unter `class Liste:` eingerückt werden, weil sie ein Teil der Definition der Klasse `Liste` ist. Auf die Objekte einer Klasse kann man nur solche Methoden anwenden, die innerhalb der Klasse definiert worden sind.

‹ 1.28 Lösung: Die Strategie bei der Entwicklung der Methode sieht wie folgt aus. In zwei Variablen `vorgaenger` und `knoten` speichert man zwei Knoten – den zuletzt besuchten und den aktuell besuchten. Somit hat man Zugriff auf die Änderung der Variablen `naechster` des letztbesuchten Knotens, wenn man im aktuellen Knoten den gesuchten Wert findet. Dies ermöglicht es auch zu erkennen, wenn der erste Knoten (Spezialfall) der Liste entfernt werden soll.

```
def loeschen(self, inhalt):
    vorgaenger = None
    knoten = self.anker
    while knoten != None and knoten.inhalt != inhalt:
        vorgaenger = knoten
        knoten = knoten.naechster
    if knoten != None:
        if vorgaenger == None):
            self.anker = knoten.naechster
        else:
            vorgaenger.naechster = knoten.naechster
```


W 1.29 Lösung: Hier präsentieren wir das gesamte Modul für die Klassen `Knoten` und `Liste`, das man auch so kompakt schreiben muss, wenn man die Methode auf konkrete Objekte anwenden will. Dieses Modul unterscheidet sich zu dem bisher betrachteten Modul in der Methode `einfuegen()`, die die Anforderungen dieser Aufgabe erfüllt. In der Variablen `neuerKnoten` erzeugt man den neuen Knoten mit dem mittels Parameter gegebenen Wert. Ähnlich wie vorher läuft man dann durch die Liste von links nach rechts und verwendet zwei Variablen `vorgaenger` und `knoten`, um Zugang zum vorher besuchten und aktuell besuchten Knoten zu haben. Wenn man die richtige Stelle zum Einfügen zwischen `vorgaenger` und `knoten` gefunden hat, müssen zwei Zeiger umdefiniert werden, um das Einsetzen umzusetzen.

```
class Knoten:
    def __init__(self, inhalt=0, naechster=None):
        self.inhalt = inhalt
        self.naechster = naechster

    def __str__(self):
        return str(self.inhalt)

class Liste:
    def __init__(self, anker=None):
        self.anker = anker
```

```
12
13      def __str__(self):
14          anzeige = " – "
15          knoten = self.anker
16          while knoten != None:
17              anzeige = anzeige + str(knoten.inhalt) + " – "
18              knoten = knoten.naechster
19          return anzeige
20
21      def findeLetzten(self):
22          if self.anker == None:
23              return None
24          knoten = self.anker
25          while knoten.naechster != None:
26              knoten = knoten.naechster
27          return knoten
28
29      def einfuegen(self, inhalt):
30          vorgaenger = None
31          knoten = self.anker
32          neuerKnoten = Knoten(inhalt)
33          while knoten != None and knoten.inhalt < inhalt:
34              vorgaenger = knoten
35              knoten = knoten.naechster
36          if vorgaenger == None:
37              neuerKnoten.naechster = self.anker
38              self.anker = neuerKnoten
39          else:
40              neuerKnoten.naechster = vorgaenger.naechster
41              vorgaenger.naechster = neuerKnoten
42
43      def suchen(self, inhalt):
44          knoten = self.anker
45          while knoten != None and knoten.inhalt < inhalt:
46              knoten = knoten.naechster
47              if knoten.inhalt == inhalt:
48                  return knoten
49              else:
50                  return None
51
52      def loeschen(self, inhalt):
53          vorgaenger = None
54          knoten = self.anker
55          while knoten != None and knoten.inhalt != inhalt:
56              vorgaenger = knoten
57              knoten = knoten.naechster
58          if knoten != None:
59              if vorgaenger == None:
60                  self.anker = knoten.naechster
61              else:
62                  vorgaenger.naechster = knoten.naechster
63
64  liste = Liste()
65  liste.einfuegen(7)
66  liste.einfuegen(2)
67  liste.einfuegen(8)
68  print(liste)
69  liste.loeschen(8)
70  print(liste)
```


In den Zeilen 64 bis 69 erzeugt das Programm eine Liste mit den aufsteigenden Werten 2, 7 und

8, und danach löscht es das Element mit Wert 8. Überprüfen Sie die Funktionalität des Programms in «TigerJython».

‹ 1.30

Lösung:

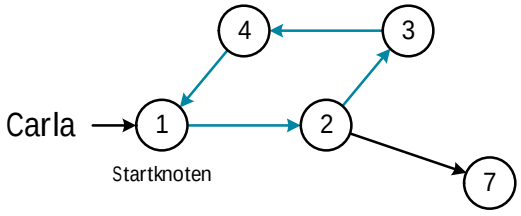
```
class Knoten:
    def __init__(self, inhalt=0, vorgaenger=None, naechster=None):
        self.inhalt = inhalt
        self.vorgaenger = vorgaenger
        self.naechster = naechster

    def __str__(self):
        return str(self.inhalt)

class Liste:
    def __init__(self, anker=None):
        self.anker = anker

    def __str__(self):
        anzeige = " - "
        knoten = self.anker
        while knoten != None:
            anzeige = anzeige + str(knoten.inhalt) + " - "
            knoten = knoten.naechster
        return anzeige

    def einfuegen(self, inhalt):
        knoten = self.anker
        neuerKnoten = Knoten(inhalt)
        # Wenn die Liste leer ist oder der Wert des einzufügenden
        # Knotens kleiner ist als der des Ankers, muss der Knoten vorne eingefügt werden:
        if self.anker == None or self.anker.inhalt > inhalt:
            neuerKnoten.naechster = self.anker
            if self.anker != None:
                self.anker.vorgaenger = neuerKnoten
            neuerKnoten.vorgaenger = None
            self.anker = neuerKnoten
        else:
            while knoten.naechster != None and knoten.naechster.inhalt < inhalt:
                # Anders als bei einfach verketteten Listen
                # müssen wir uns den Vorgänger nicht merken, da wir direkt auf ihn zugreifen können
                knoten = knoten.naechster
            neuerKnoten.vorgaenger = knoten
            neuerKnoten.naechster = knoten.naechster
            if knoten.naechster != None:
                knoten.naechster.vorgaenger = neuerKnoten
            knoten.naechster = neuerKnoten

    def suchen(self, inhalt):
        knoten = self.anker
        # Wir könnten auch != statt < verwenden, jedoch würden
        # wir damit ggf. unnötige Werte anschauen, weil die Liste sortiert ist
        while knoten != None and knoten.inhalt < inhalt:
            knoten = knoten.naechster
        if knoten.inhalt == inhalt:
            return knoten
        else:
            return None

```

```
55      def loeschen(self, inhalt):
56          knoten = self.anker
57          while knoten != None and knoten.inhalt < inhalt:
58              knoten = knoten.naechster
59          if knoten != None and knoten.inhalt == inhalt:
60              if knoten.vorgaenger == None:
61                  self.anker = knoten.naechster
62              else:
63                  knoten.vorgaenger.naechster = knoten.naechster
64              if knoten.naechster != None:
65                  knoten.naechster.vorgaenger = knoten.vorgaenger
66                  # Knoten muss nicht explizit gelöscht werden
67
68 liste = Liste()
69 liste.einfuegen(7)
70 print(liste)
71 liste.einfuegen(8)
72 print(liste)
73 liste.einfuegen(2)
74 print(liste)
75 liste.einfuegen(3)
76 print(liste)
77 liste.loeschen(3)
78 print(liste)
```

Bei der Entwicklung des Moduls beobachten wir, dass gewisse Methoden gleichgeblieben sind und andere Modifikationen erfordert haben. Die Generierung der Klasse `Knoten` hat sich geändert, weil `Knoten` jetzt drei Komponenten besitzt. Die zusätzliche Komponente beinhaltet den vorherigen Knoten (falls vorhanden). Die Generierung der Liste ist unverändert, weil es sich wieder nur um einen Anker an einem Knoten handelt. Die Methode `__str__()` bleibt ebenfalls für beide Klassen gleich. Die Methode `suchen()` läuft durch die Liste von links nach rechts und deswegen würde sie genauso gut auch bei einfach verketteten Listen funktionieren.

Was sich ändert, sind die Methoden zur Änderung der Liste durch Einfügen und Löschen. Auf einer Seite braucht man bei den doppelt verketteten Listen die Hilfsvariable `vorgaenger` nicht mehr, weil man aus dem aktuellen Knoten direkten Zugang zu den vorherigen Knoten hat. Andererseits erfordert die Änderung der doppelt verketteten Liste, doppelt so viele Zeiger umzustellen wie bei der einfach verketteten Liste.

› 1.31 Lösung a: Es ist tatsächlich so: Wenn Carla alles vergisst, was sie bisher gesehen hat, kann sie nicht einmal bemerken, dass sie einen Knoten (eine Lichtung) betreten hat, bei dem (der) sie schon einmal war. Deswegen ist es möglich, dass Carla ewig in einem Kreis laufen wird, ohne die 7 zu finden. Mit der Strategie, immer links bei Vorhandensein mehrerer Möglichkeiten zu gehen, läuft Carla zum Beispiel im folgenden Graphen unendlich lange.



Wenn Carla in Knoten 1 startet, geht sie zu Knoten 2, und danach der Strategie folgend bei Verzweigungen nach links von Knoten 2 zu Knoten 3. So wird sie ewig den blauen Pfeilen folgend im Kreis 1 2 3 4 1 laufen, ohne je zu Knoten 7 zu gelangen.