ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Institut für Verhaltenswissenschaften

inf | Informatik
Computer Science
Institut für Computersysteme

ETH-Leitprogramm Informatik:

Lineare Datentypen: Die Liste

Schultyp:	Gymnasium
Fachliche Vorkenntnisse:	Grundlegende Java-Kenntnisse (Arrays, Referenzen, Schleifen), Objekt-orientierte Programmierung gemäss Inverted Curriculum: Klassen, abstrakte Klassen und Polymorphie.
Bearbeitungsdauer:	8-10 Stunden
Autorin:	Susanne Cech
Betreuer:	Prof. Dr.-Ing. Ulrik Schroeder (RWTH Aachen) Prof. Dr. Juraj Hromkovič (ETH Zürich)
Fassung:	2005-09-30
Schulerprobung:	noch nicht erfolgt

Einführung

Worum geht es?

Seit einiger Zeit lernst du Programmieren mit Java. Du kennst alle Kontrollstrukturen, benutzt Klassen und Interfaces. Um Elemente in einem Programm zu speichern, benutzt du einfache Variablen oder Arrays.

Variablen und Arrays haben einige Nachteile: In einer Variable kannst du nur einen Wert speichern. Ein Array speichert zwar mehrere Elemente, aber man muss im vorhinein die Anzahl der Elemente festlegen. Was machst du, wenn du mehr als diese ursprünglich festgelegte Anzahl von Elementen speichern möchtest?

Wir möchten gerne beliebig viele Elemente speichern können. Das heisst auch, dass wir nicht beim Programmieren schon die Anzahl fixieren wollen. Eine Lösung dafür heisst Liste. Listen kennst du: Einkaufslisten, Hausaufgabenlisten, Checklisten, Wahllisten etc.

Weil die Liste so allgegenwärtig ist, benutzt man sie auch in der Informatik. Wir finden es sehr spannend, wie Bewährtes aus unserem Alltag in die Programmierung gefunden hat!

Lernziele

Du kannst abschätzen, für welche Anwendungen du einen Array oder eine Liste verwenden möchtest.

Du kannst die Operationen der Liste auf Papier darstellen.

Du kannst den Quelltext der Liste und die enthaltenen Methoden lesen, nachvollziehen und modifizieren.

Du kannst die Listen in einem Programm verwenden.

Ablauf

Im Alltag kennst und benutzt du viele unterschiedliche Behälter, um Gegenstände darin aufzubewahren. Wir untersuchen im ersten Kapitel die Eigenschaften normaler im Alltag benutzter Behälter.

Im zweiten Kapitel studieren wir Behälter in der Informatik. Wir zeigen dir, wie Behälter wie Arrays und Listen im Speicher dargestellt werden und wie man damit umgehen kann.

Listenobjekte haben einen besonderen Aufbau. Im Kapitel 3 lernst du die wichtigen Operationen für Listen auf Papier kennen.

Im Kapitel 4 programmieren wir endlich: Wir stellen dir den Programmtext für Listen vor. Um die Liste auch zu benutzen, erstellen wir ein kleines Programm für die Kontaktdatenverwaltung.

Eine Erweiterung der Liste stellen wir dir in Kapitel 5 vor. Auch hier kannst du programmieren.

INHALTSVERZEICHNIS

Arbeitsanweisung

Dieses Leitprogramm kannst du alleine durcharbeiten. Du brauchst Papier und Bleistift sowie einen Computer, auf dem eine Java-Entwicklungsumgebung installiert ist.

Die einzelnen Kapitel haben alle den gleichen Aufbau:

Zuerst erhältst du eine Einführung in das Kapitel. Es folgen die Lernziele. Damit weisst du genau, was du am Ende jedes Kapitels können musst.

Besonders wichtiger Stoff wie Erklärungen und Definitionen haben einen blauen Untergrund. Ein Beispiel:

Computer:

Ein Computer oder Rechner ist ein Apparat, der Informationen mit Hilfe einer programmierbaren Rechenvorschrift verarbeiten kann.

Der Lernstoff ist immer wieder unterbrochen von kleinen Fragen und Aufgaben. Diese sind rosa hinterlegt:

Aufgabe 0.0

Hast du alles vorbereitet: Papier, Bleistift, Computer?

Programmtexte haben einen gelben Hintergrund. Wir drucken immer nur Ausschnitte der Programmtexte. Die vollständigen Quelltexte mit Dokumentation findest du in digitaler Form. Das Verzeichnis wird im jeweiligen Kapitel angegeben.

```
1 /* Hello.java */
2
3 public class Hello {
4
5         public static void main (String[] args)
6         {
7                 System.out.println ("Viel Spass!");
8         }
9 }
```

Nach der Lernstoffvermittlung folgt eine kleine Zusammenfassung mit den wichtigsten Begriffen.

Die Lernkontrollen am Ende der Kapitel sind schriftlich. Die ersten drei Kapitel kannst du auf Papier lösen, die weiteren am Computer.

Kapitel 1

Alltagsgeschichten

1.1 Worum geht es?

Im ersten Kapitel besprechen wir einige Dinge, die du vielleicht jeden Tag benutzt: SMS-Speicher, Telefonbuch, Schublade für Socken. Dies sind Behälter, um Gegenstände aufzubewahren. Wir werden die Gemeinsamkeiten und Unterschiede dieser alltäglichen Behälter herausarbeiten.

1.2 Lernziele

Nach diesem Kapitel sollst du folgendes wissen:

- Welche Eigenschaften und Aufgaben haben alltägliche Behälter?
- Welche Gemeinsamkeiten und Unterschiede gibt es bei alltäglichen Behältern?

1.3 Der SMS-Speicher

Verschickst du gerne SMS? Hast du dir schon mal folgendes überlegt: Wie werden SMS in deinem Handy gespeichert, sodass du sie wieder lesen kannst? Was passiert, wenn du ein neues SMS bekommst? Wenn die Speicherkarte voll ist, welches SMS löscht du und wie funktioniert das?

Aufgabe 1.1

Hier darfst du ausnahmsweise dein Handy im Unterricht benutzen! Wenn du kein Handy besitzt, arbeite mit deiner Tischnachbarin oder deinem Tischnachbarn zusammen. Nimm dein Handy heraus und lass dir die Liste der SMS anzeigen. Wie sind die SMS angeordnet? Wo stehen die neueren SMS? Wo stehen die älteren SMS?

Aufgabe 1.2

Lass dir die folgenden SMS anzeigen: (1) Die neueste, (2) die vorletzte deiner besten Freundin, (3) die älteste. Was fällt dir auf?

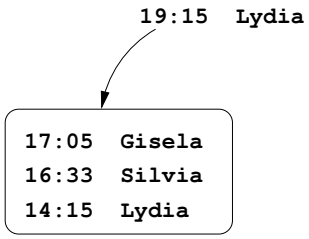


Abbildung 1.1: Ein neues SMS wird in den SMS-Speicher eingefügt

SMS werden in der Reihenfolge ihrer Ankunft gespeichert. Es gibt keine spezielle Ordnung, wie z.B. nach dem Sender sortiert. Was passiert nun, wenn du eine neue SMS bekommst? Es wird einfach als neueste SMS oben in den SMS-Speicher eingefügt (siehe Abbildung 1.3). Wenn du eine SMS löschen möchtest, musst du — ähnlich wie beim Lesen einer bestimmten SMS — die SMS-Liste durchklicken, bis du die richtige gefunden hast und löschen kannst.

1.4 Das Telefonbuch

Da wir gerade über Telefonieren sprechen: Sehen wir uns auch das Telefonbuch an! Die Einträge im Telefonbuch — Name und Anschrift, Telefon- und Faxnummern von Personen, Familien und Firmen — sind alphabetisch sortiert. Dadurch kannst du sehr einfach nach einem Eintrag suchen.

Aufgabe 1.3

Nimm das bereitgestellte Telefonbuch und suche nach “deinem” Eintrag, d.h. dem Eintrag deiner Familie! Beschreibe wie du vorgehst!

Das Telefonbuch dient der Aufbewahrung von Einträgen. Die Einträge sind alphabetisch sortiert, um den Zugriff auf die Einträge zu erleichtern. Dafür ist das Einfügen aufwändiger.

Wenn neue Einträge in das Telefonbuch aufgenommen werden, muss zuerst der richtige Platz gefunden werden. Es wird eine Suche wie in der Aufgabe durchgeführt. Danach kann der neue Eintrag gespeichert werden. Wird ein Eintrag aus dem Telefonbuch gelöscht, muss dieser auch zuerst gesucht werden.

1.5 Sockenschublade

Socken werden in einer Schublade aufbewahrt. Typischerweise werden die Socken einfach hineingeworfen. Das richtige Paar Socken zu finden kann ziemlich lange dauern…

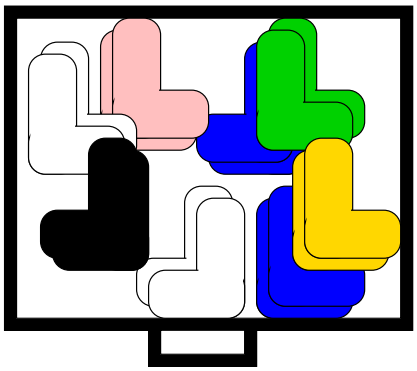


Abbildung 1.2: Die Unordnung der Sockenschublade

Eine Sockenschublade dient nur der Aufbewahrung, die Socken sind nicht speziell angeordnet oder sortiert (siehe Abbildung 1.2). Der Zugriff ist nicht organisiert — man greift blind hinein und wählt ein Paar aus.

Ordentliche Menschen bringen Ordnung auch in die Sockenschublade! Man kann die Socken sortiert in die Sockenschublade einräumen — zum Beispiel die weissen links und die schwarzen rechts. Dann kannst du mit einem Griff ein weisses oder schwarzes Paar auswählen, weil du die Anordnung der Socken kennst.

1.6 Tupperware?

Die besprochenen Beispiele wie der SMS-Speicher, das Telefonbuch oder die Sockenschublade haben einiges gemeinsam: Sie sind Behälter zur Aufbewahrung von Elementen, also von tatsächlichen oder virtuellen Gegenständen.

Das Telefonbuch ist ein Behälter für Adressen und Telefonnummern. Die Sockenschublade ist ein Behälter für Socken. Der SMS-Speicher im Handy bewahrt alle SMS auf. Die **Aufbewahrung** ist eine wichtige Eigenschaft der Behälter.

Dennoch unterscheiden sich die Behälter, wie Elemente aufgewahrt werden oder wie Elemente angeordnet sind. Gibt es eine bestimmte Anordnung der Elemente, kann auch **organisiert auf die Elemente zugegriffen** werden.

SMS werden in der Reihenfolge ihrer Ankunft angeordnet. So kann man leicht die neueste lesen. Um andere SMS zu lesen, muss man die Liste durchklicken. Das Telefonbuch ist alphabetisch sortiert, um uns die Suche nach den Einträgen und damit den Zugriff zu erleichtern. Die Sockenschublade hat keine bestimmte Anordnung der Socken — ausser die Socken sind nach Farben sortiert. Die Anordnung und der Zugriff sind nicht organisiert.

In diesem Leitprogramm besprechen wir Behälter, die Elemente in einer bestimmten Anordnung aufbewahren, damit organisiert auf diese Elemente zugegriffen werden kann. Der SMS-Speicher oder das Telefonbuch sind Beispiele für solche Behälter, die Sockenschublade allerdings nicht.

Aufgaben eines Behälters:

Elemente werden in einen Behälter *abgelegt* oder *eingefügt*. Auf die aufbewahrten Elemente kann man wieder *zugreifen* oder sie *ansehen*. Bei Bedarf können Elemente aus dem Behälter *entfernt* oder *gelöscht* werden.

1.7 Behälter für Menschen

Können wir unseren Behälterbegriff auch auf Menschen anwenden? Fällt dir eine Situation ein, wo Personen für einen organisierten Zugriff angeordnet werden? Bei der Supermarkt- oder Kinokasse musst du dich anstellen und warten, bis du dran kommst. Versuchen wir, uns diese Warteschlagen als Behälter vorzustellen! Das Ziel einer Warteschlange ist es, eine faire Reihenfolge der Personen sicherzustellen. Wer zuerst hinkommt, soll auch zuerst dran kommen. Die kurze Aufbewahrung dient dazu, eine Ordnung herzustellen.

In Bezug auf die Operationen bedeutet das: Man kommt hin, stellt sich an (das ist das “Einfügen”) und wartet. Wenn dann alle Personen vor einem bedient worden sind, kommt man dran (das “Zugreifen”) und wird bedient, dann geht man weg (wird “entfernt”).

1.8 Zusammenfassung

In diesem Kapitel haben wir alltägliche Behälter kennengelernt: der SMS-Speicher, das Telefonbuch, und die Sockenschublade. Diese Behälter bewahren Elemente auf. Auf die Elemente kann man zugreifen und sie wieder entfernen.

1.9 Lösungen zu den Aufgaben (Wissenssicherung)

Lösung 1.1:

Die SMS sind nach ihrer Ankunftszeit angeordnet. Im SMS-Speicher stehen die neuen SMS oben, die älteren unten in der Liste.

Lösung 1.2:

(1) Wenn du die neueste SMS lesen möchtest, reicht ein Knopfdruck, und sie wird dir am Display angezeigt. (2) Möchtest du die vorletzte SMS deiner Freundin lesen, musst du dich durch SMS durchgehen, bis du die richtige gefunden hast. (3) Um die älteste SMS anzuzeigen, musst du durch alle SMS durchgehen.

Lösung 1.3:

Wir suchen einen “Rolf Trachsler”. Der Familienname beginnt also mit ’T’. Deshalb schlagen wir das Telefonbuch ziemlich weit hinten auf und überprüfe den Buchstaben. Ist der Buchstabe kleiner als ’T’, schlage ich das Telefonbuch erneut weiter hinten auf. Ist der Buchstabe grösser als ’T’, schlage ich weiter vorne auf. Habe ich schliesslich den Buchstaben ’T’ gefunden, fange ich zu blättern an, bis ich bei der Seiten angekommen bin, die mit “Tr” beginnt. Hier lese ich nun Eintrag für Eintrag, bis ich den Eintrag für “Rolf Trachsler” gefunden habe.

1.10 Lernkontrolle

LK 1.1: Eigenschaften der Behälter

Was sind die Aufgaben eines Behälters? Beschreibe und erkläre die Aufgaben mit deinen eigenen Worten.

LK 1.2: Handy-Telefonbuch

Beschreibe die Eigenschaften des Telefonbuchs im Handy: Wie funktioniert das Hinzufügen eines neuen Eintrags? Wie sind die Einträge gespeichert? Wie kannst du auf einen Eintrag zugreifen?

1.11 Lösungen zur Lernkontrolle

LK Lösung 1.1: Eigenschaften der Behälter (K2)

Ein Behälter dient der Aufbewahrung von Gegenständen. Gegenstände werden abgelegt und somit gespeichert. Auf die Gegenstände kann man auch zugreifen (d.h. sie ansehen) oder wieder löschen.

LK Lösung 1.2: Handy-Telefonbuch (K3)

Die Einträge im Handy-Telefonbuch sind alphabetisch nach Namen sortiert. Ein neuer Eintrag wird eingetippt. Beim Speichern wird er in den bisherigen Bestand alphabetisch einsortiert. Nach den Einträgen kann man suchen: Tippt man den Anfangsbuchstaben ein, wird die Liste mit diesem Anfangsbuchstaben angezeigt, und man kann den richtigen durch Durchgehen ermitteln. Viele Handys bieten die Möglichkeit, die zuletzt gewählten Einträge anzuzeigen.

Kapitel 2

Der Container

2.1 Worum geht es?

In diesem Kapitel studieren wir, wie Behälter aufgebaut sind. Du wirst lernen, auf welche Art man “ablegen” und “zugreifen” kann. Das Art des Ablegens beeinflusst die Art des Zugreifens und umgekehrt. Die grundlegenden Datenstrukturen bestimmen, wie Elemente im Speicher dargestellt werden und wie auf die Elemente zugegriffen werden kann. Grundlegende Datenstrukturen bilden Bausteine für komplexe Datenstrukturen.

2.2 Lernziele

Nach diesem Kapitel sollst du folgendes wissen:

- Wie werden Elemente in Containern angeordnet?
- Wie unterscheiden sich die Containerarten bezüglich Eigenschaften und Operationen?
- Welche Eigenschaften hat ein sortierter Container?

2.3 Container in der Informatik

Behälter werden in der Informatik überall verwendet. In der Informatik werden Behälter als *Container* bezeichnet. Auf der Festplatte deines PC findest du viele Ordner (*Verzeichnisse, Directories*). Ein Ordner ist ein Container für Dateien und weitere Ordner. Der Internetbrowser speichert eine “History” mit den besuchten Webseiten der letzten Tage. Programme merken sich eine Reihe der letzten Befehle, um diese wieder rückgängig machen zu können (*Undo-Funktionalität*).

In der Informatik werden Container für bestimmte Zwecke entwickelt und für ihren Einsatzzweck angepasst. In diesem Kapitel lernst du, worauf du achten musst, um die optimale Datenstruktur zu wählen.

Aufgabe 2.1

Welche anderen Container bezogen auf die Informatik fallen dir ein? Nenne zwei verschiedene! Du kannst dazu den Computer benutzen.

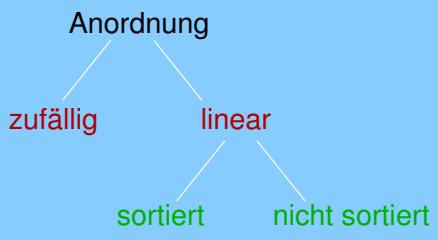
2.4 Grundlegende Organisationsformen

Elemente werden in einem Container gespeichert. Die Organisationsform legt fest, wie Elemente in einem Container organisiert bzw. angeordnet werden. Wir unterscheiden zwischen zufälliger und linearer Anordnung.

Organisationsformen:

Zufällig angeordnete Elemente: Sie haben keine Reihenfolge und keine Ordnung. Die Socken in der Sockenlade sind zufällig angeordnet.

Linear angeordnete Elemente: Die Elemente sind in einer Reihe angeordnet. Jedes Element hat mindestens einen Nachbar: Ein Element in der Mitte hat ein vorgehendes und ein nachfolgendes Element. Das erste Element hat nur ein nachfolgendes, aber kein vorgehendes Element. Das letzte Element hat nur ein vorgehendes, aber kein nachfolgendes Element.



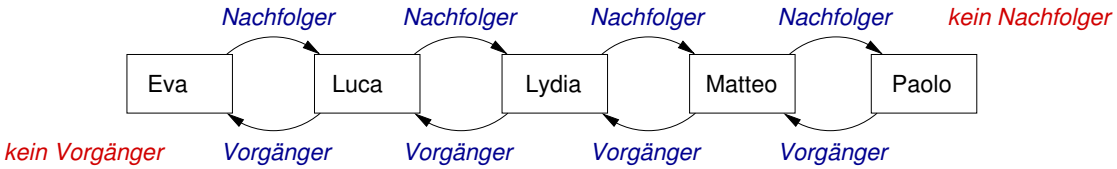
Diese *Nachbareigenschaft* gilt für linear angeordnete Elemente, d.h. wenn es eine Reihenfolge der Elemente gibt. Aufgrund dieser Nachbareigenschaft können wir weiter unterscheiden: Elemente können sortiert oder nicht sortiert angeordnet sein.

Abbildung 2.1 zeigt eine alphabetisch sortierte und eine nicht sortierte Liste. Beachte die Elemente mit bzw. ohne Vorgänger und Nachfolger!

Aufgabe 2.2

Überlege dir je einen Beispielscontainer für die drei möglichen Organisationsformen!

a) sortierte Liste



b) nicht sortierte Liste

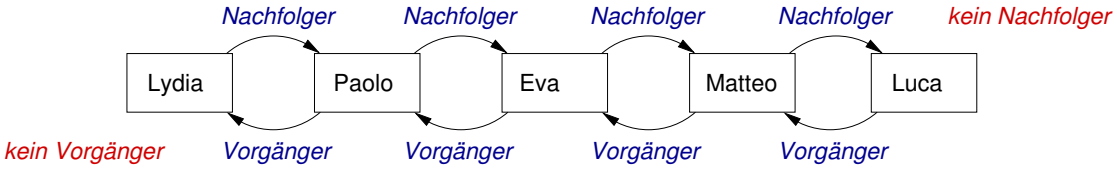


Abbildung 2.1: Lineare Anordnung

2.5 Grundlegende Daten- bzw. Speicherstrukturen

In der Informatik gibt es zwei grundlegende Strukturen, wie Elemente linear im Speicher verwaltet werden. Diese Strukturen werden als Bausteine für weitere Strukturen und Container verwendet.

Das Array:

Die Datenstruktur **Array** speichert eine fixe Anzahl von gleichartigen bzw. gleich grossen Elementen, die im Speicher Element nach Element aneinander gefügt werden. Hier besteht tatsächlich eine physische Nachbareigenschaft.

Die Grösse eines Arrays wird beim Erzeugen des Arrayobjektes festgelegt. Weil die Grösse des Arrays nicht mehr geändert werden kann, bezeichnet man den Array als eine **statische** Struktur.

Jedem Element ist eine Zahl (ein *Index*) zugewiesen. Mit einem Index kann man gezielt auf ein Element zugreifen. Jedes Objekt ist an einer bestimmten Adresse im Speicher. Referenzen verwenden diese Adressen, um auf die Elemente zu verweisen. Um auf ein Element gezielt zugreifen zu können, wird mittels des Index die notwendige Adresse berechnet.

Addressberechnung im Array:

Abbildung 2.2 zeigt dir einen Array mit 1024 Elementen. Wir schreiben `array[0]` für das erste Element und `array[1023]` für das letzte Element. Die *Grösse* eines Elements ist der benötigte Speicherplatz in Byte.

Die Adresse des Arrays ist gleichzeitig die Adresse des ersten Elements. Um die Adresse des zweiten Elements (`array[1]`) zu berechnen, addiert man zu der Adressse des Arrays die Grösse des ersten Elements. Für die Adresse von array[2] zählt man 2-mal die Grösse zu der Adresse des Arrays dazu. Allgemein gilt also diese Berechnungsformel:

```
Adresse (array[i]) := Adresse (array) + (i * Elementgrösse)
```

Man setzt den gewünschten Index ein und kann einfach die Adresse des Elements berechnen. Jetzt siehst du auch, warum man das erste Element mit `array[0]` anspricht — und warum InformatikerInnen mit 0 zu zählen beginnen!
Eigentlich werden in einem Array Referenzen gespeichert. Diese zeigen auf die eigentlichen Elemente. Eine Referenz braucht 4 Bytes Speicherplatz.

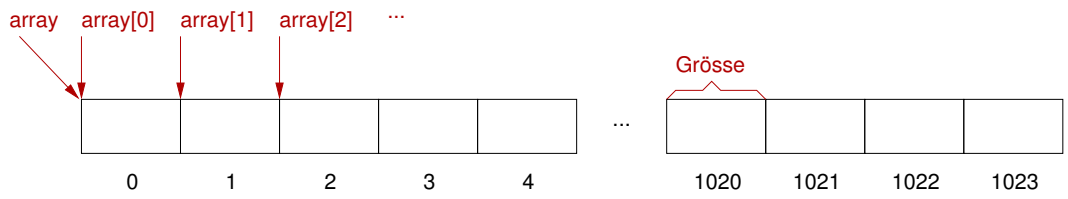


Abbildung 2.2: Addressberechnung im Array

Die Liste:

Die Datenstruktur **Liste** speichert eine variable Anzahl von Elementen. Im Speicher können die Elemente irgendwo stehen. Die einzelnen Elemente sind miteinander verkettet — die Nachbareigenschaft wird über Referenzen hergestellt.

Die Grösse einer Liste ist variabel. Zur Laufzeit kann eine Liste beliebig viel Elemente aufnehmen. Man bezeichnet deshalb eine Liste als eine **dynamische** Struktur.

Auf die einzelnen Elemente kann man nur in der gegebenen Reihenfolge zugreifen. Ausgehend vom ersten Element muss man allen Nachbar-Referenzen folgen, um das gewünschte Element zu erreichen.

Abbildung 2.3 zeigt einige alphabetisch sortierte Namen. In Abbildung 2.3(a) sind die Namen in einem Array gespeichert, in Abbildung 2.3(b) in einer Liste. Du kannst auch sehen, dass die Elemente in der Liste extra Speicherplatz für die Referenzen brauchen.

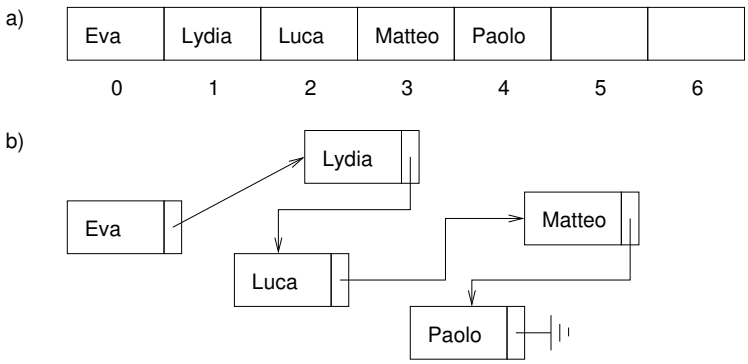


Abbildung 2.3: Datenstruktur Array und Liste

Aufgabe 2.3

In der Abbildung 2.3 wissen wir, dass Luca an der dritten Position in unserer Namensliste ist. Wie viele Schritte braucht man, um auf Luca zuzugreifen?

a) Array
b) Liste

2.6 Container-Operationen

Im ersten Kapitel haben wir typischen Aufgaben für Behälter kennen gelernt: **ein Element ablegen**, **auf ein Element zugreifen** und möglicherweise wieder **löschen**. Nun sehen wir uns die Eigenschaften dieser Operationen genau an.

Ablegen:

Elemente können sortiert oder nicht sortiert abgelegt werden. Arrays oder Listen können sowohl sortiert als auch nicht sortiert sein.

Zugreifen:

Zugriff auf Elemente kann direkt oder sequentiell erfolgen.

1. Direkter Zugriff: Ein Element kann an einer beliebigen Position eines Speichers gelesen oder geschrieben werden. Zum Beispiel gibst du der Stereoanlage den Befehl, das 5. Lied der CD zu spielen.

2. Sequentieller Zugriff: Auf die Elemente kann nicht beliebig zugegriffen werden, sondern der gesamte Bestand muss immer von Anfang an der Reihe nach durchsucht und abgearbeitet werden. Beispiele sind der SMS-Speicher oder die Warteschlange an der Kinokasse.

Die Art des Zugriffs hängt von der Struktur des Containers ab. Arrays bieten direkten Zugriff über den Index — wir haben bereits besprochen, wie die Adresse eines Elements berechnet wird. Auf Listenelemente hingegen kann nur sequentiell zugegriffen werden.

Aufgabe 2.4

Kann man auf Array-Elemente auch sequentiell zugreifen? Wenn ja, wie?

Löschen:

Zuerst muss die Position des zu löschenden Elements bekannt sein. Dann kann es entfernt werden.

Aufgabe 2.5

Vervollständige die folgende Tabelle:

Container	direkter Zugriff	sequentieller Zugriff
Videospiel mit verschiedenen Levels		
Videokassette		
DVD		

2.7 Aufwand der Operationen

Je nach Anforderung an die Verwaltung der Elemente sind die Operationen zum Ablegen und Zugreifen einfach oder schwierig. Man sagt, sie sind weniger oder mehr **aufwändig**. Die folgenden Überlegungen helfen den Aufwand der Operationen abzuschätzen.

Ordnung: Sollen die Elemente sortiert oder nicht sortiert sein? Wenn man die Elemente sortiert angeordnet haben möchte, werden typischerweise die Elemente sortiert eingefügt. Sortiert bedeutet immer, dass die Elemente nach einem bestimmten Kriterium, d.h. einem bestimmten Merkmal, sortiert sind. Bespiele sind eine alphabetische Sortierung nach Nachnamen, eine Sortierung nach Geburtsdatum. Das Ablegen wird also aufwändiger: Zuerst muss die geeignete Position gesucht werden. Danach kann das Element eingefügt werden.

Die Wahl der Datenstruktur entscheidet hier mit. Bei Listen können Referenzen umgebogen werden. So kann sehr einfach ein neues Element “zwischen” zwei bestehenden eingekettet werden. Bei Arrays müssen die grösseren Elemente um eine Position verschoben werden, damit das neue Element eingefügt werden kann.

Abbildung 2.4(a) zeigt, wie das neue Element “Luca” in eine sortierte Liste eingefügt wird: Die Referenzen werden entsprechend angepasst. In Abbildung 2.4(b) wird “Luca” in einen sortierten Array abgelegt. Hier müssen alle nachfolgenden Elemente um eine Position verschoben werden.

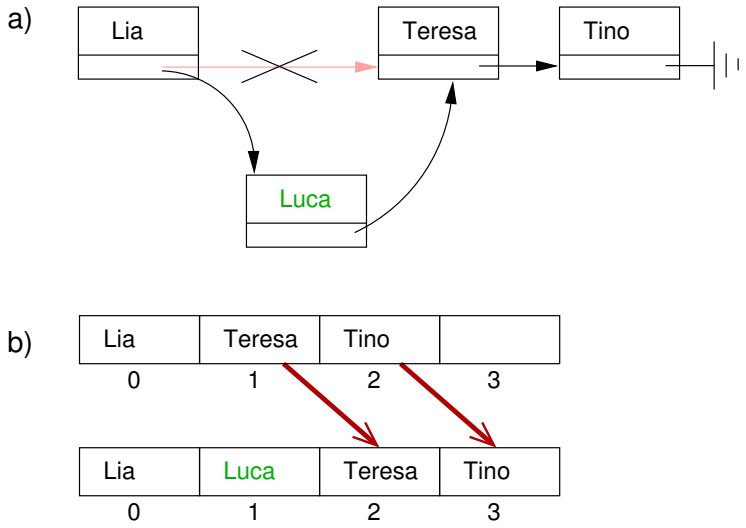


Abbildung 2.4: Einfügen des neuen Elements “Luca”

Zugriff: Soll der Zugriff direkt oder sequentiell sein? “Gib mir das 5. und das 12. Element!” sind Beispiele für direkten Zugriff. Für “Alle Buttons in der Symbolleiste sollen einen rosa Hintergrund bekommen” kann man gut sequentiellen Zugriff verwenden.

Häufigkeit der Operationen: Wie oft wird auf Elemente zugegriffen? Wie oft werden Elemente abgelegt? Werden öfter Elemente abgelegt als darauf zugegriffen (bzw. gelöscht) wird? Wieviele Elemente werden eingefügt?

Durch eine Abschätzung der Häufigkeiten der beiden Operationen kannst du sehen, welche der beiden wichtiger ist. Stelle dir die hunderttausenden Einträge im Telefonbuch vor! Viel öfter wird auf einen Eintrag zugegriffen als dass er verändert wird. So wird Wert darauf gelegt, dass das Suchen schnell ist und die Daten sortiert sind. Weniger wichtig ist der Aufwand des Einfügens.

Ein Array kann man verwenden, wenn man die Anzahl der Elemente kennt. Die Grösse des Arrays wird einmalig festgelegt und nicht mehr geändert werden. Wenn mehr Elemente im Array eingefügt werden sollen als Platz haben, dann muss ein neues, grösseres Array erstellt werden und alle Elemente müssen kopiert werden. Hier ist eine Liste vorteilhafter: Sie kann beliebig viele Elemente aufnehmen.

Aufwand für Arrayoperationen:

- **Einfügen eines Elements in ein Array:**

  Um ein Element in einen Array einfügen zu können, braucht es einen freien Platz im Array. An den Index des ersten freien Platzes kann man direkt das neue Element schreiben. Das ist eine einzige Operation.

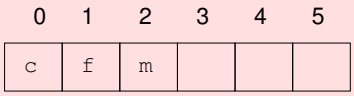
  Bei sortiertem Einfügen entscheidet die Position des neuen Elements über den Aufwand. Zum Beispiel hat das Array noch einen Platz frei — und wir möchten das bisher kleinste Element einfügen. Das ist der schlimmste Fall: Wir müssen fast alle Elemente um eine Position weiter verschieben, um Platz für dieses neue Element zu schaffen. Allerdings gehen wir nicht immer vom schlimmsten Fall aus. Deshalb können wir sagen, dass beim sortierten Einfügen im Durchschnitt die Hälfte der Elemente verschoben werden müssen.
- **Zugreifen auf ein Element in einem Array:**

  Der Zugriff erfolgt direkt mit dem Index auf ein Element. Die Adresse im Speicher wird mittels des Indexes berechnet.
- **Löschen eines Elements aus einem Array:**

  Beim Entfernen eines Elements wird ein Platz im Array frei. So entstehen “Löcher” im Array. Werden dann Elemente neu eingefügt, muss man den Array sequentiell durchgehen, um einen freien Platz zu finden. Alternativ kann man nach jedem Löschen alle nachfolgenden Elemente um eine Position nach vorne verschieben. Wie bei der vorigen Überlegung müssen durchschnittlich die Hälfte der Elemente verschoben werden.

Aufgabe 2.6

Gegeben ist ein Array aus Buchstaben. Füge diese Buchstaben sortiert ein: ’r’, ’e’, ’a’, ’z’. Zeichne jeden einzelnen Schritt! Was passiert, wenn das Array voll ist?



Aufwand für Listenoperationen:

- **Einfügen eines Elements in eine Liste:**

  Beim Einfügen eines Elements in eine Liste müssen immer einige Referenzen geändert werden, weil das Element eingehängt wird.

  Sortiertes Einfügen ist jedoch weniger aufwändig bei einer Liste als bei einem Array. Wie beim Array muss zuerst die richtige Position gefunden werden. Doch dann werden nur Referenzen geändert, weil das Element eingekettet wird. Kein einziges Element muss verschoben werden.
- **Zugreifen auf ein Element in einer Liste:**

  Zugreifen auf eine Liste erfolgt immer sequentiell. Das bedeutet, man muss die Liste von Anfang an, Element für Element, bis zum gesuchten Element durchgehen. Im Durchschnitt muss man die Hälfte der Elemente ansehen.
- **Löschen eines Elements aus einer Liste:**

  Zuerst muss das gewünschte Element gesucht werden. Auch hier werden durchschnittlich die Hälfte der Elemente angesehen. Das Löschen des Elements im Vergleich zum Array ist relativ einfach. Das zu löschende Element wird ausgekettet, und es werden nur Referenzen geändert. Kein Element muss verschoben werden.

Vergleich Aufwand Liste und Array:

Der sequentielle Zugriff einer Liste hat seinen Preis — durchschnittlich muss für das Zugreifen auf ein Element die halbe Liste angesehen werden. Das Array bietet hingegen direkten Zugriff.

Dafür ist eine Liste sehr flexibel, weil sie zur Laufzeit beliebig wachsen kann. Ist ein Array voll, muss ein neues Array erzeugt werden. Alle bisherigen Elemente müssen in das neue Array kopiert werden.

Einfügen bzw. Löschen an einer bestimmten Position ist bei einer Liste weniger aufwändig als bei einem Array.

Ein Array ist für eine sortierte Struktur nicht sehr geeignet. Beim Einfügen und Löschen müssen jeweils durchschnittlich die Hälfte der Elemente verschoben werden.

Aufgabe 2.7

Du möchtest ein CD-Abspielprogramm entwickeln. Wenn du eine CD einlegst, werden die Liedinformationen eingelesen und in einer Datenstruktur gespeichert. Die Lieder können in der gegebenen oder einer zufälligen Reihenfolge abgespielt werden.
Welche Datenstruktur verwendest du, um die Lieder der CD zu verwalten? Begründe deine Entscheidung!

2.8 Spezielle Listen

In diesem Leitprogramm behandeln wir die lineare Datenstruktur Liste. Es gibt Abwandlungen der Liste, bei denen das Ablegen oder der Zugriff eingeschränkt wird. Im folgenden besprechen wir einige spezielle Listen.

Die Warteschlange vor der Kinokasse oder im Supermarkt bezeichnet man als **Queue**, das englische Wort für Warteschlange. Das Prinzip heisst FIFO, First-In-First-Out (“zuerst hinein, zuerst heraus”). Hier ist das Ablegen sowie der Zugriff eingeschränkt: Man fügt an einem Ende ein und entnimmt am anderen Ende (siehe Abbildung 2.8(a)).

Auch der Teller- oder Bücherstapel existiert in der Informatik: der Stapel heisst auf englisch **Stack**. Hier wird das LIFO-Prinzip — Last-In-First-Out (“zuletzt hinein, zuerst heraus”) — verwendet. Wie auch bei der Queue sind das Ablegen und der Zugriff eingeschränkt. Allerdings wird am gleichen Ende eingefügt und auch wieder entnommen (siehe Abbildung 2.8(b)).

Eine spezielle Warteschlange ist die **Priority-Queue** — die Prioritätswarteschlange. Jedes Element hat ein Gewicht, d.h. eine Priorität, und wird gemäss dieses Gewichtes in der Warteschlange abgelegt. Jeweils das Element mit der höchsten Priorität wird behandelt. Das ist wie auf einer Notfallstation: Schwere Fälle werden vorne eingereiht.

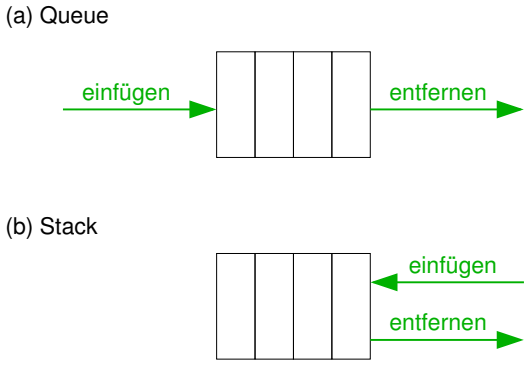


Abbildung 2.5: Queue und Stack

2.9 Komplexe Datenstrukturen

Die Liste und das Array sind besondere Datenstrukturen. Sie werden als Bausteine für komplexere Strukturen verwendet. Es gibt Listen oder Arrays, die als Elemente Listen speichern. Überlegen wir uns ein Beispiel!

Ein Rezept verlangt gewisse Zutaten und legt eine Reihenfolge fest, wie diese Zutaten verarbeitet werden. Wir können ein Rezept als eine Liste von Arbeitsschritten modellieren. Ein Kochbuch enthält eine Reihe von Rezepten, geordnet nach Vor-, Haupt- und Nachspeisen. Wir können also ein Array mit drei Elementen erzeugen. Zum Beispiel finden wir in `array[0]` eine Liste mit allen Rezepten für Vorspeisen.

Jedes Kochbuch hat aber auch hinten einen Index, indem die Rezepte alphabetisch sortiert aufgeführt sind. Im Kochbuch werden die entsprechenden Seitenzahlen angegeben. Zusätzlich verwalten wir also noch ein weiteres Array aus Buchstaben, der als Elemente alphabetisch sortierte Listen hat. Diese Einträge verweisen dann auf die Rezepte.

2.10 Zusammenfassung

In diesem Kapitel haben wir uns überlegt, wie Container aufgebaut und im Speicher organisiert sind. Elemente können zufällig oder linear in einem Container angeordnet sein. Die grundlegende Datenstrukturen Listen und Array organisieren Elemente linear, d.h. mit zwei Ausnahmen haben alle Elemente einen Vorgänger und einen Nachfolger. Auf die Elemente eines Containers kann man direkt oder sequentiell zugreifen. In einem Array kann man direkt auf Elemente zugreifen, in einer Liste sequentiell. In diesem Leitprogramm behandeln wir Listen. Spezielle Listen sind Queue, Priority-Queue und Stack. Listen und Array dienen als Bausteine für komplexere Datenstrukturen.

2.11 Lösungen zu den Aufgaben (Wissenssicherung)

Lösung 2.1:

- Eine Datei ist ein Container für Daten.
- Wenn du auf einem Windows-Rechner auf START drückst, erhältst du eine Liste der ausführbaren Programme.
- Die Systemsteuerung beinhaltet eine Liste der Konfigurationsmöglichkeiten.
- Jedes Programm besitzt eine Menuleiste mit einer Liste der möglichen Befehle.
- In der Systemsteuerung findest du ein Programm “Neue Programme hinzufügen oder löschen”. Wenn du es startest, wird eine Liste mit allen installierten Programmen erzeugt.

Lösung 2.2:

- Zufällig angeordnet: Briefe in einem Postkasten.
- Linear angeordnet und sortiert: Postkästen in einem Wohnblock (geordnet nach Wohnungen).
- Linear angeordnet und nicht sortiert: Stapel aus beliebigen Zeitschriften.

Lösung 2.3:

a) 1 Schritt

b) 3 Schritte

Lösung 2.4:

Mit einer Schleife kann man ein Array vom ersten bis zum letzten Element durchlaufen. Hier ist eine Möglichkeit mittels einer for-Schleife:

```
for (int i = 0; i < array.length; i++)
    System.out.println (array[i]);
```

Lösung 2.5:

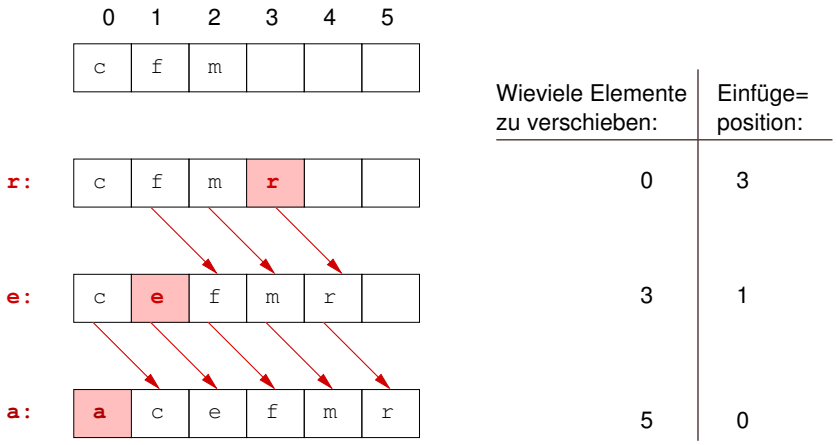
Container	direkter Zugriff	sequentieller Zugriff
Videospiel mit verschiedenen Levels		X
Videokassette		X
DVD	X	X

Bei einem Computerspiel muss man zuerst das erste Level fertig spielen. Danach kommt man erst in das zweite Level. Grundsätzlich muss man alle vorigen Levels durchgespielt haben, bis man in das nächsthöhere Level aufsteigen kann.

Eine Videokassette muss man von vorne bis hinten ansehen. Mit den Schnelllauftasten kann man die Suche etwas beschleunigen.

Eine DVD kann man sowohl von vorne bis hinten ansehen (sequentiell) als auch auf bestimmte Kapitel springen (direkt).

Lösung 2.6:



Wieviele Elemente zu verschieben:	Einfüge= position:
0	3
3	1
5	0

z: Das Array ist voll. Es kann kein weiteres Element eingefügt werden.

Lösung 2.7:

Die Anzahl der Elemente ist bekannt. Die Liedinformationen werden einmal zu Beginn eingelesen und eingefügt. Für die vorgegebene Reihenfolge muss sequentiell auf die Lieder zugegriffen werden. Für die zufällige Reihenfolge muss man direkt auf die Elemente zugreifen können, z.B. 5, 3, 12, 1, 4 usw. Deshalb verwendet man ein Array, um die Lieder zu verwalten.

2.12 Lernkontrolle

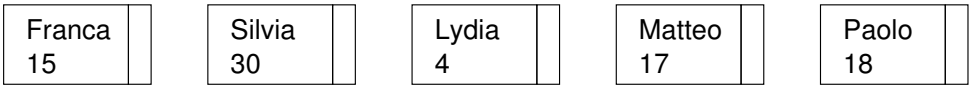
LK 2.1: Zugskompositionen

Du sollst für die SBB eine Software zur Verwaltung der Zugskompositionen schreiben. Eine Zugskomposition besteht aus verschiedenen Waggons, einer Lokomotive und manchmal aus einem Speisewagen. Je nach Auslastung werden Waggons abgehängt oder hinzugefügt. Am Endbahnhof wird die Lokomotive ans andere Ende gehängt.

Welche Datenstruktur verwendest du, um die Zugskompositionen zu verwalten? Begründe deine Entscheidung!

LK 2.2: Personen

Die Abbildung zeigt Elemente einer Liste. Die Elemente bestehen aus dem Namen und dem Alter einer bestimmten Person.



1. Verbinde die Elemente zu einer Liste: Sie soll nach dem Alter sortiert sein, die jüngste Person zuerst. Kennzeichne und beschrifte das erste bzw. das letzte Element.
2. Zeichne eine neues Element und füge es in die Liste ein: Tom, 13 Jahre.

LK 2.3: Fotoalbum

Du hast eine Menge von Fotos — ein richtiges Durcheinander. Du möchtest sie gerne in der zeitlichen Reihenfolge der Aufnahme in ein Album einkleben. Der Aufnahmezeitpunkt steht auf der Rückseite des Fotos.

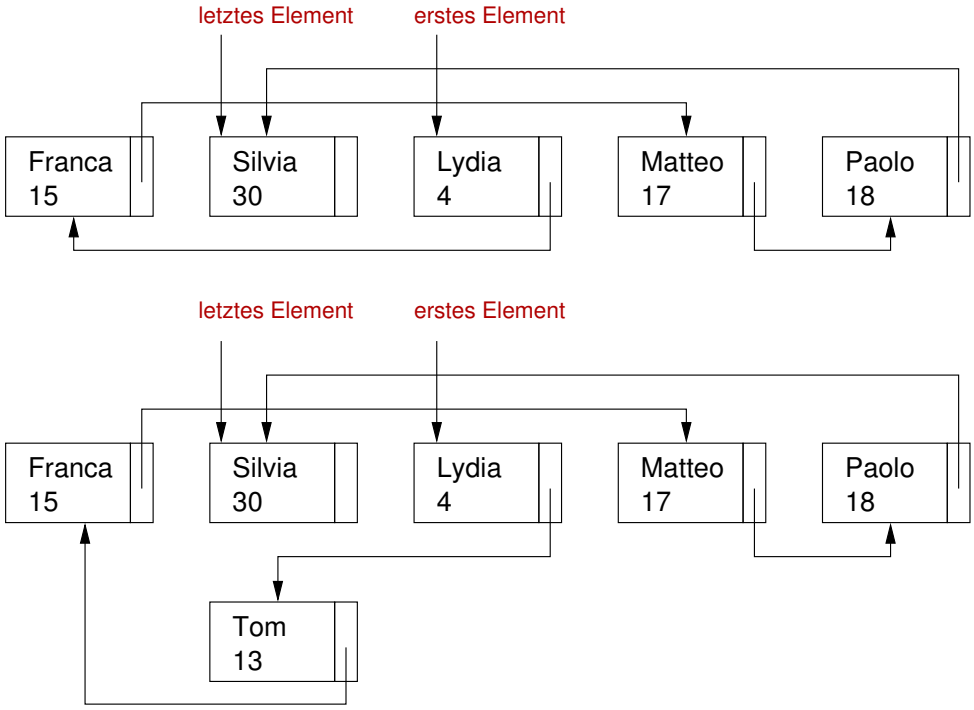
- Beschreibe wie und mit welchem Aufwand du auf die Menge der Fotos zugreifen kannst.
- Wie funktioniert das Sortieren und Einfügen der Fotos in das Fotoalbum? Berücksichtige dabei auch den Aufwand.

2.13 Lösungen zur Lernkontrolle

LK Lösung 2.1: Zugskompositionen (K3)

Die Zugskompositionen können sich häufig, auch mehrmals pro Tag, ändern. Zu Stosszeiten werden mehr, tagsüber weniger Waggons eingesetzt. Die Lokomotive wird bei einigen Bahnhöfen (Sackbahnhöfen) umgehängt. Die Anzahl der Waggons variiert also. Wir modellieren deshalb eine Zugskomposition als Liste von Waggons.

LK Lösung 2.2: Personen (K3)



LK Lösung 2.3: Fotoalbum (K3)

Für die unsortierten Fotos gibt es keine Ordnung. Man kann/muss auf jedes Foto zufällig zugreifen. Es existieren mehere Lösungen:

1. Man sucht in der unsortieren Fotomenge jeweils das älteste Foto heraus. Dieses wird dann in das Album eingeklebt. Das ist sehr aufwändig: In jedem Schritt muss man alle unsortierten Fotos ansehen und vergleichen, um jeweils das älteste zu finden.
2. Besser ist es, den Tisch als zusätzliche Datenstruktur zu verwenden. Im ersten Schritt werden die Fotos sortiert auf dem Tisch abgelegt. Man kann ein Foto nach dem anderen zufällig aus der nicht sortiereten Menge nehmen und in die bereits

sortierten auf den Tisch einfügen. Im zweiten Schritt werden die sortierten Fotos in das Album eingeklebt.

Wenn dann alle Fotos aus der unsortierten Menge genommen sind und neu sortiert sind, kann man sie in das Fotoalbum einkleben. Das ist sehr einfach. Man kann mit dem ältesten beginnen und sie der Reihe nach sequentiell einkleben.

3. Eine Abwandlung der zweiten Möglichkeit bezüglich der Sortierung: Für jedes Monat macht man eine neue Reihe. Der Zugriff auf den richtigen Monat ist wahlfrei (z.B. der März ist die 3. Zeile, der Juli die 7. Zeile). Innerhalb des Monats den richtigen Platz zu finden, ist relativ einfach. (es gibt offensichtlich weniger Fotos pro Monat als pro Jahr). Eventuell muss man die Fotos verschieben, um Platz für das neue Foto zu schaffen.

Kapitel 3

Verkettete Listen

3.1 Worum geht es?

Im Kapitel 2 haben wir die Eigenschaften von Listen besprochen, sozusagen die Listen von “aussen” betrachtet. Nun sehen wir uns die *innere* Struktur der Listen an, also die Realisierung einer Liste im Speicher. Auf Papier studieren wir anhand der Speicherstruktur die Operationen zum Ablegen, Zugreifen und Löschen eines Elements. Dazu dient auch die Operation zum “Durchlaufen” einer Liste. Wir unterscheiden zwischen nicht sortierten und sortierten Listen.

3.2 Lernziele

Nach diesem Kapitel sollst du folgendes wissen:

- Wie werden Listen im Speicher dargestellt? Welche Attribute kennzeichnen eine Liste?
- Wie funktionieren die Operationen zum Ablegen, Zugreifen und Löschen auf Papier? Welche Sonderfälle muss man beachten?
- Wie erhält man eine sortierte Liste?

3.3 Einführung

Eine verkettete Liste ist eine der einfachsten Containerstrukturen. Jedes Element, das in einer Liste gespeichert ist, ist mit seinem Nachfolger verkettet. So erhält man eine “Kette” von Elementen — deshalb der Name “verkettete Liste”. In der Abbildung 3.1 siehst du vier Elemente, die jeweils über eine Referenz `next` mit dem Nachfolger verbunden sind. Das letzte Element ist mit keinem weiteren Element verkettet, die Referenz hat den Wert `null`.

In der Abbildung 3.1 siehst du auch das eigentliche Listenobjekt. Um auf die Elemente zugreifen zu können, merkt sich die Liste das erste Element (`first`). Über die Referenz `first` kann man Elemente vorne einfügen. Um Elemente auch einfach hinten einfügen zu können, führt die Liste auch einen Verweis auf das letzte Element `last`.

Die Listenstruktur in Abbildung 3.1 besteht aus zwei Teilen, dem Listenobjekt und den verketteten Elementen. Ein roter Stift zeigt auf das eigentliche Listenobjekt mit den Feldern `first`, `last` und `count`. `first` und `last` sind Referenzen, die auf das erste Element bzw. auf das letzte Element zeigen. Der Integer `count` speichert die Anzahl der Elemente. Die einzelnen Elemente sind über `next` miteinander verbunden.

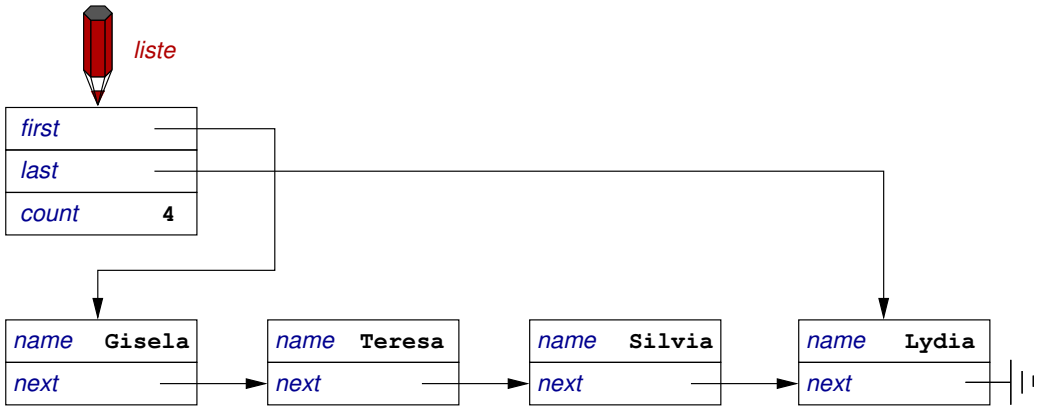


Abbildung 3.1: Verkettete Liste

Aufgabe 3.1

Zeichne eine Liste mit deinem Lieblingsmenü, bestehend aus mindestens drei Gängen. Die Liste kann (muss aber nicht) nach dem Gang sortiert sein.

Eine Liste, die keine Elemente beinhaltet, bezeichnet man als **leere Liste**. Abbildung 3.2 zeigt eine leere Liste. `first` und `last` sind `null`, und `count` ist auf `0` initialisiert.

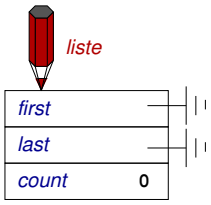


Abbildung 3.2: Eine leere Liste

3.4 Zugreifen und Iterieren

Schau dir die Abbildung 3.1 an. Nimm einen Stift und lege ihn auf das Listenobjekt — dorthin, wohin der rote Stift zeigt. Du kannst nun auf die Felder des Listenobjekts zugreifen, d.h. auf `first`, `last` und `count`.

Du kannst auch der Referenz des Feldes `first` folgen und befindest dich dann auf dem ersten Element. Du kannst nun also den Namen “Gisela” lesen. Oder du kannst über die Referenz `next` den Stift auf das nächste Element legen und so auf die Felder des nächsten Elements “Teresa” zugreifen. Du siehst, du kannst immer nur auf das Objekt zugreifen, auf das der Stift zeigt.

Das Stiftmodell:

Die folgenden Regeln gelten: (1) Du kannst den Stift immer auf den Ausgangspunkt zurücklegen — also auf das Listenobjekt. (2) Der Stift kann nur Referenzen folgen, um auf weitere Objekte zu gelangen.

Aufgabe 3.2

Setze den Stift auf das Listenobjekt (so wie in Abbildung 3.1). Wo landest du, wenn du der Referenz des Feldes `last` folgst? Wohin kannst du dich von hier aus bewegen?

Aufgabe 3.3

Setze den Stift wieder auf das Listenobjekt (siehe Abbildung 3.1). Navigiere über die Referenzen, bis du auf dem Element mit dem Namen “Silvia” stehst. Lege den Stift dafür jeweils ein Element weiter. Wie kannst du von hier aus “Teresa” erreichen?

Du siehst in der Abbildung 3.1 die ganze Listenstruktur — sozusagen alles auf einen Blick. Trotzdem kannst du nur auf ein Objekt zugreifen, auf das der Bleistift zeigt. Das ist der sequentielle Zugriff. Du kannst nicht einfach so z.B. auf das 3. Element zugreifen. Direkter Zugriff ist mit Listen nicht möglich.

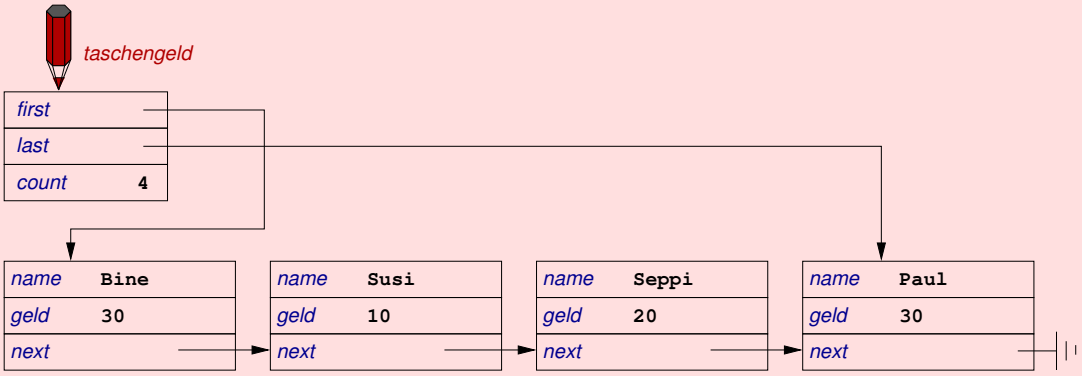
Iterieren:

Ausgehend vom Listenobjekt — der Ausgangsposition vom Bleistift — kannst du den Bleistift über `first` auf das erste Element zeigen lassen. Von dort aus kannst du über `next` das nächste Element erreichen usw. — bis du das letzte Element erreicht hast. So funktioniert der sequentielle Zugriff. Das Durchgehen einer Liste in der Reihenfolge der Elemente bezeichnet man als **Iterieren**.

Beim Iterieren greift man auf jedes Element zu. So kann man auch für jedes Element eine beliebige Operation durchführen: z.B. jedes Element am Bildschirm ausgeben, eine Berechnung durchführen oder überprüfen, ob es das gesuchte Element ist.

Aufgabe 3.4

Es ist Monatsbeginn, das Taschengeld der Kinder wird neu verteilt. Iteriere über die folgende Liste mit Hilfe des Stiftmodells. Erhöhe das Taschengeld um 40 Franken für jedes Kind in der Liste. Stelle die Zwischenschritte dar!



3.5 Zugreifen und Suchen

Es gibt verschiedene Arten, um auf Elemente zuzugreifen. Zwei davon haben wir bereits besprochen: (1) Vom Listenobjekt aus kannst du auf das erste bzw. das letzte Element zugreifen. (2) Wenn wir über eine Liste iterieren, greifen wir auf jedes Element zu.

Schliesslich möchten wir auch auf ein bestimmtes Element zugreifen, d.h. wir möchten nach einem bestimmten Element suchen. Zum Beispiel möchtest du im SMS-Speicher nach einem SMS von deiner Freundin suchen, das sie dir vor drei Tagen geschickt hat.

Man bestimmt ein Kriterium, also ein Merkmal, mit dem man zwischen Elementen unterscheiden kann. Für die Suche iteriert man über die Liste und überprüft jedes Element auf das gewählte Kriterium. Ist das Kriterium für ein Element erfüllt, ist die Suche erfolgreich und die Iteration kann beendet werden.

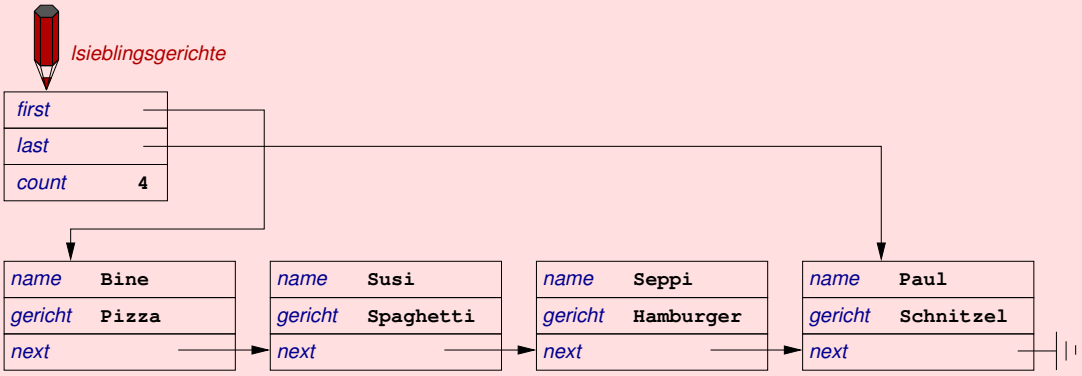
Zugreifen:

Auf Elemente kann man zugreifen, indem man

- eine bestimmte Regel hat und immer über `first` bzw. `last` auf das *erste* bzw. *letzte* zugreift,
- über die Liste iteriert und so auf *jedes* Element zugreift,
- ein Kriterium bestimmt und damit die Liste nach einem Element durchsucht. Ein **Kriterium** ist Merkmal, mit dem man zwischen Elementen unterscheiden kann.

Aufgabe 3.5

Arbeite mit dieser Liste: Welche Kriterien gibt es – womit kann man zwischen den Elementen unterscheiden und so eines aussuchen? Gib einige Beispiele an!



Aufgabe 3.6

Führe nun einige Suchvorgänge mit dem Stiftmodell durch! Iteriere über die Liste und vergleiche so jedes Element mit dem Suchkriterium! Was fällt dir auf?

- Suche nach einem Element mit dem Namen “Seppi”.
- Suche nach einem Element mit dem Lieblingsessen “Käse”.

Suchen:

Beim Suchen wird ein Kriterium bestimmt — das *Suchkriterium*. Mit diesem wird über die Liste iteriert und jedes Element überprüft. Entspricht ein Element dem Suchkriterium, wird die Suche und damit die Iteration beendet. Über die Liste wird komplett iteriert, wenn kein Element gefunden werden konnte, das dem Suchkriterium entspricht.

3.6 Ein Element ablegen

Im Kapitel 2 haben wir Listen als eine dynamische Struktur kennengelernt. Man kann beliebig viele Elemente in eine Liste einfügen und wieder löschen. Hier beschäftigen wir uns damit, wie man ein Element in eine verkettete Liste einfügt.

Einfügen in eine nicht sortierte Liste ist einfach, wenn wir ein neues Element entweder vorne oder hinten anfügen. Grundsätzlich müssen wir zwei Fälle unterscheiden: (1) die Liste ist leer oder (2) die Liste ist nicht leer.

In Abbildung 3.3 zeigen wir dir, wie man ein Element in eine leere Liste einfügt. Weil du nur auf ein Objekt zugreifen kannst, auf das auch ein Stift zeigt, brauchst du diesmal zwei Stifte: Der rote Stift zeigt auf das Listenobjekt, der blaue auf das neue Element. Um das neue Element mit der Liste zu verbinden, müssen wir `first` und `last` auf das neue Element setzen und die Anzahl der Elemente nachführen.

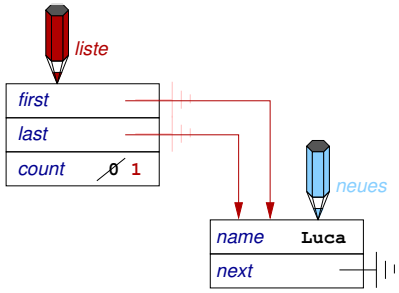
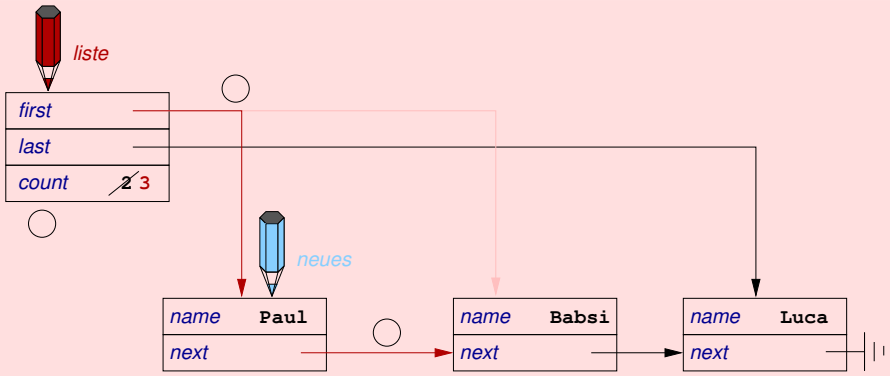


Abbildung 3.3: Einfügen des ersten Elements in eine verkettete Liste

Wir gehen davon aus, dass schon Elemente in der Liste gespeichert sind. Wir wollen das neue Element **vorne** einketten. Für das Einfügen eines neuen Elements in eine Liste mit mehreren Elementen muss man drei Schritte durchführen. Besonders wichtig ist die Reihenfolge dieser drei Schritte!

Aufgabe 3.7

In dieser Abbildung kannst du erkennen, welche drei Schritte dafür notwendig sind. Trage die richtige Reihenfolge in die vorgesehenen Kreise ein. Zur Hilfe verwende zwei Stifte: Der rote Stift zeigt auf das Listenobjekt, der blaue auf das neue Element. Bedenke, dass du nur mittels des Stiftes über die Referenzen auf ein anderes Element zugreifen kannst. Was passiert, wenn die Reihenfolge nicht richtig ist?



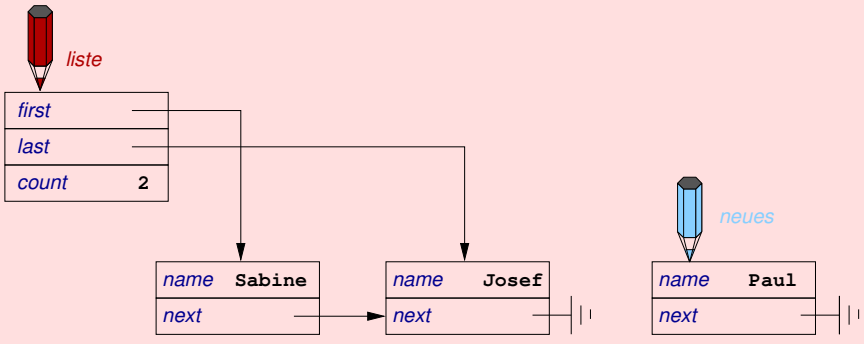
Aufgabe 3.8

Füge diese Elemente in eine Liste mit dem Namen `getraenke` ein: `Milch`, `Rivella`, `Kakao`, `Cola`. Wähle geeignete Namen für die Felder des Listenelements. Zeichne zu Beginn eine leere Liste. Stelle jeden Zwischenschritt dar.

Wir möchten nun die neuen Element **hinten** an die Liste einfügen. Das Listenobjekt speichert die Referenz `last` auf das letzte Element. Hinten Einfügen ist ganz einfach.

Aufgabe 3.9

Füge das neue Element in die Liste ein. Überlege und notiere die notwendigen Schritte in der richtigen Reihenfolge! Der rote Farbstift zeigt auf das Listenobjekt. Der blaue Farbstift zeigt auf das neue Element, das hinten eingefügt werden soll.



3.7 Ein Element löschen

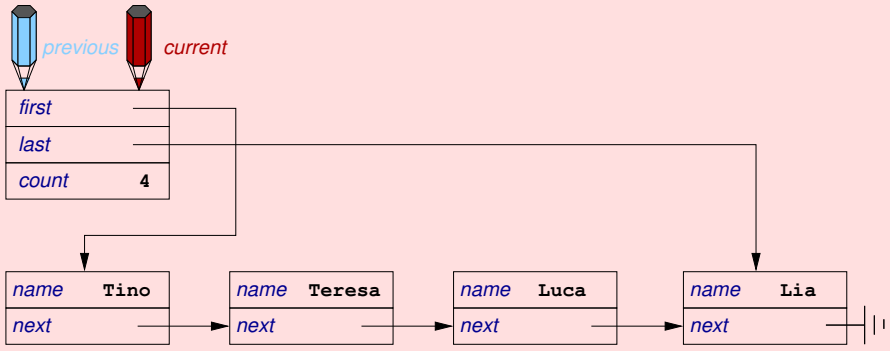
Es gibt viele Gründe, warum man ein Element löschen möchte. Der Speicher des Handys ist voll, der Kunde hat an der Kasse bezahlt, oder ein T-Shirt wird aus dem Kasten genommen. Auch beim Löschen kann es Regeln geben, um beispielsweise immer das erste bzw. das letzte Element zu entfernen.

Wir möchten allerdings ein bestimmtes Element löschen. Zuerst suchen wir gemäss einem Kriterium ein Element und iterieren dafür über die Liste. Haben wir ein passendes Element gefunden, wird dieses aus der Liste ausgekettet und entfernt. Beim Ausketten muss man gut aufpassen, denn einige Referenzen in der Liste ändern sich!

Aus unserer Beispielsliste werden wir nun nach und nach alle Elemente löschen und dabei alle möglichen Fälle studieren! Zuerst muss man das gewünschte Element suchen, d.h. über die Liste bis zum gesuchten Element iterieren. Wir brauchen zwei Stifte: Der rote Stift zeigt auf das aktuelle Element (*current*) und der blaue Stift auf dessen Vorgänger (*previous*). Als aktuelles Element bezeichnen wir das Element, das gerade "dran" ist, d.h. das Element, das gemäss dem Suchkriterium überprüft wird. Wir brauchen zwei Stifte, weil wir auf den Vorgänger des gefundenen Elements zugreifen müssen! Warum? Das überlegst du dir sofort bei der Aufgabe…

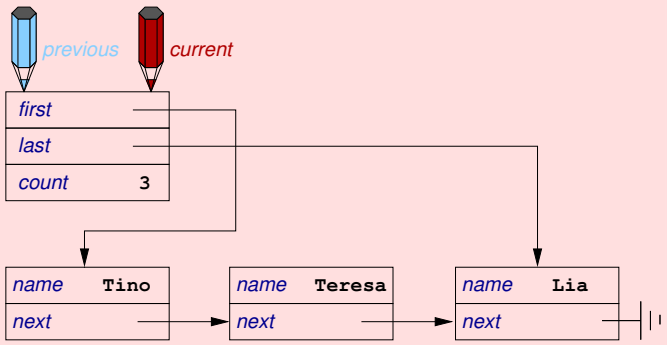
Aufgabe 3.10

Lösche "Luca" aus dieser Liste. Setze zunächst beide Stifte auf das erste Element: Der rote Stift soll auf das aktuelle Element zeigen, der blaue zeigt auf dessen Vorgänger. Wie iterierst du mit beiden Stiften, sodass der blaue Stift immer auf den Vorgänger des roten Stifts zeigt? Warum brauchst du den blauen Stift?
Wenn der rote Stift auf "Luca" zeigt, überlege dir: Wie kannst du "Luca" aus der Liste entfernen? Wie müssen dir Referenzen gesetzt werden? Welche Schritte sind notwendig? Zeichne und nummeriere die Schritte in der Abbildung!



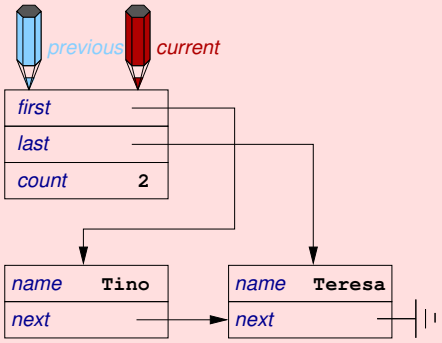
Aufgabe 3.11

Als nächstes sollst du “Lia” aus der Liste löschen. Verwende wieder den blauen und den roten Stift, um über die Liste zu iterieren. Zeichne und nummeriere die notwendigen Schritte.



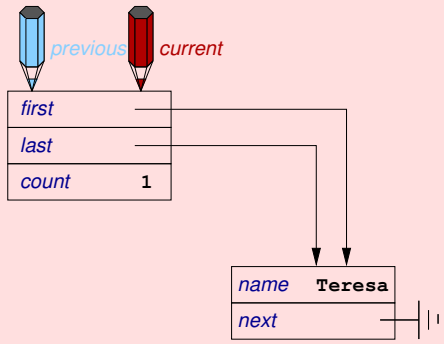
Aufgabe 3.12

Nun lösche “Tino” aus der Liste. Welche Schritte muss du durchführen? Trage die Reihenfolge in die Abbildung ein!



Aufgabe 3.13

Übrig bleibt nur noch das Element “Teresa”. Wie kannst du es entfernen?



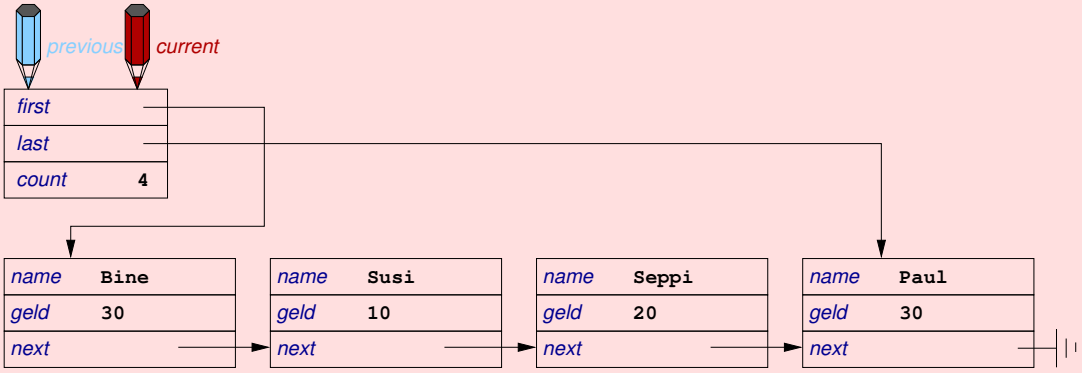
Löschen von Elementen:

Der Algorithmus funktioniert in vier Schritten: Zuerst muss über die Liste iteriert werden, um ein Element zu finden, das dem Suchkriterium entspricht. Danach wird das Element ausgekettet und die Anzahl der Elemente angepasst. Zuletzt kann das System das Element tatsächlich entfernen und dessen Speicherplatz freigeben.
Beim Ausketten muss man unterscheiden, ob das gesuchte Element (1) ein Element in der Mitte der Liste, (2) am Beginn oder (3) am Ende der Liste, oder (4) das einzige Element ist:

- **Löschen aus der Mitte:** Das Element wird ausgekettet, indem der Vorgänger auf den Nachfolger des zu löschenden Elements gesetzt wird.
- **Löschen am Anfang:** `first` wird auf das zweite Element gesetzt.
- **Löschen am Ende:** `last` wird auf das vorletzte Element gesetzt.
- **Löschen des einzigen Elements:** `first` und `last` werden auf `null` gesetzt.

Aufgabe 3.14

Langsam ziehen die Kinder aus und bekommen kein Taschengeld mehr. Benutze zwei verschiedenfarbige Stifte und lösche folgende Elemente in der gegebenen Reihenfolge aus der Liste: “Bine”, “Seppi”, “Paul”, “Susi”. Stelle jeden Schritt dar!



3.8 Eine sortierte Liste

Wir haben bis jetzt Listen ganz allgemein besprochen. Jetzt stellen wir dir die sortierten Listen vor. Es gibt zwei Möglichkeiten, eine sortierte Liste zu erhalten:

1. Immer wenn du neue Elemente einfügst, platzierst du sie gerade am richtigen Platz.
2. Du nimmst eine nicht sortierte Liste — und sortierst sie! Wir werden das Sortieren einer Liste nicht in diesem Leitprogramm besprechen.

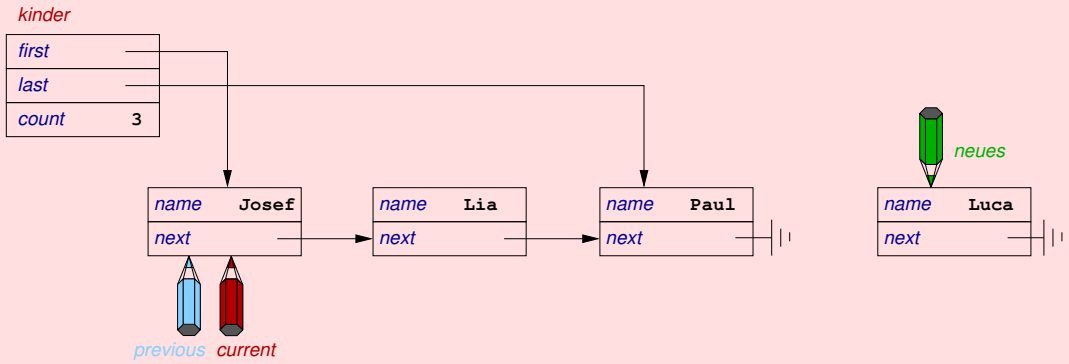
Möchtest du ein Element sortiert einfügen, muss zuerst der richtige Platz gefunden werden. Über die Liste wird also iteriert. Ähnlich wie beim Löschen gibt es vier mögliche Positionen, wo ein neues Element eingefügt werden kann:

- als erstes Element in eine leere Liste,
- als erstes Element (weil es das “kleinste” ist),
- als letztes Element (weil es das “grösste” ist), oder
- in die Mitte.

Wir sehen uns nur das Einfügen in die Mitte an, die anderen Fälle kennst du schon aus dem Kapitel 3.6. Wenn du dir nicht ganz sicher bist, lies sie noch einmal nach!

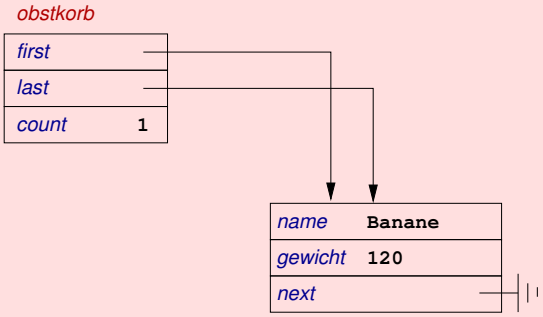
Aufgabe 3.15

Füge “Luca” sortiert in diese Liste ein. Die Liste ist alphabetisch nach Namen sortiert. Diesmal brauchst du sogar drei Stifte: der grüne zeigt auf das neue Element, der rote auf das aktuelle Element und der blaue auf dessen Vorgänger. Wenn du den richtigen Platz zum Einfügen gefunden hast, zeichne die Positionen der Stifte ein. Welche Operationen musst du durchführen? Trage sie in die Abbildung ein! Gib die richtige Reihenfolge an!



Aufgabe 3.16

Füge diese Elemente in der gegebenen Reihenfolge in die Liste absteigend nach dem Gewicht sortiert ein: Melone, 4000g; Kokosnuss, 1200g; Apfel, 80g. Benutze das Stiftmodell und zeichne jeden Zwischenschritt.



3.9 Zusammenfassung

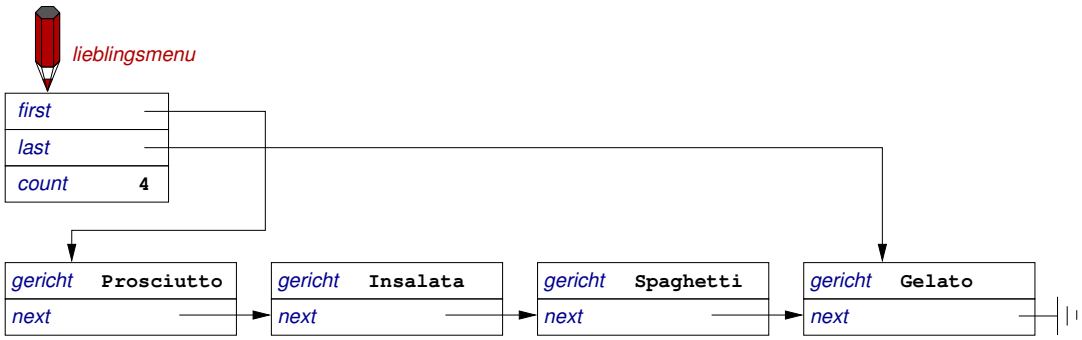
In diesem Kapitel haben wir studiert, welche Struktur eine Liste im Speicher hat: Eine Liste besteht aus zwei Teilen; dem Listenobjekt und den mit ihren Nachbarn verketteten Elementen. Eine Liste hat verschiedene Felder: `first`, `last` und `count`. Die Elemente besitzen das Feld `next`.

Wir haben das Ablegen, Zugreifen und Löschen besprochen. Elemente können vorne bzw. hinten eingefügt bzw. gelöscht werden. Auf Listen wird sequentiell zugegriffen. Oft benötigt man das vollständige oder teilweise Iterieren einer Liste. Listen können sortiert oder nicht sortiert sein. Sortierte Listen kann man erzeugen, indem man eine nicht sortierte Liste sortiert oder alle Elemente sortiert einfügt.

Im nächsten Kapitel werden wir einige Programmtexte für eine verkettete Liste studieren. Wir werden auch eine verkettete Liste verwenden, indem wir ein kleine Kontaktverwaltung erstellen. Endlich kannst du auch programmieren!

3.10 Lösungen zu den Aufgaben (Wissenssicherung)

Lösung 3.1:



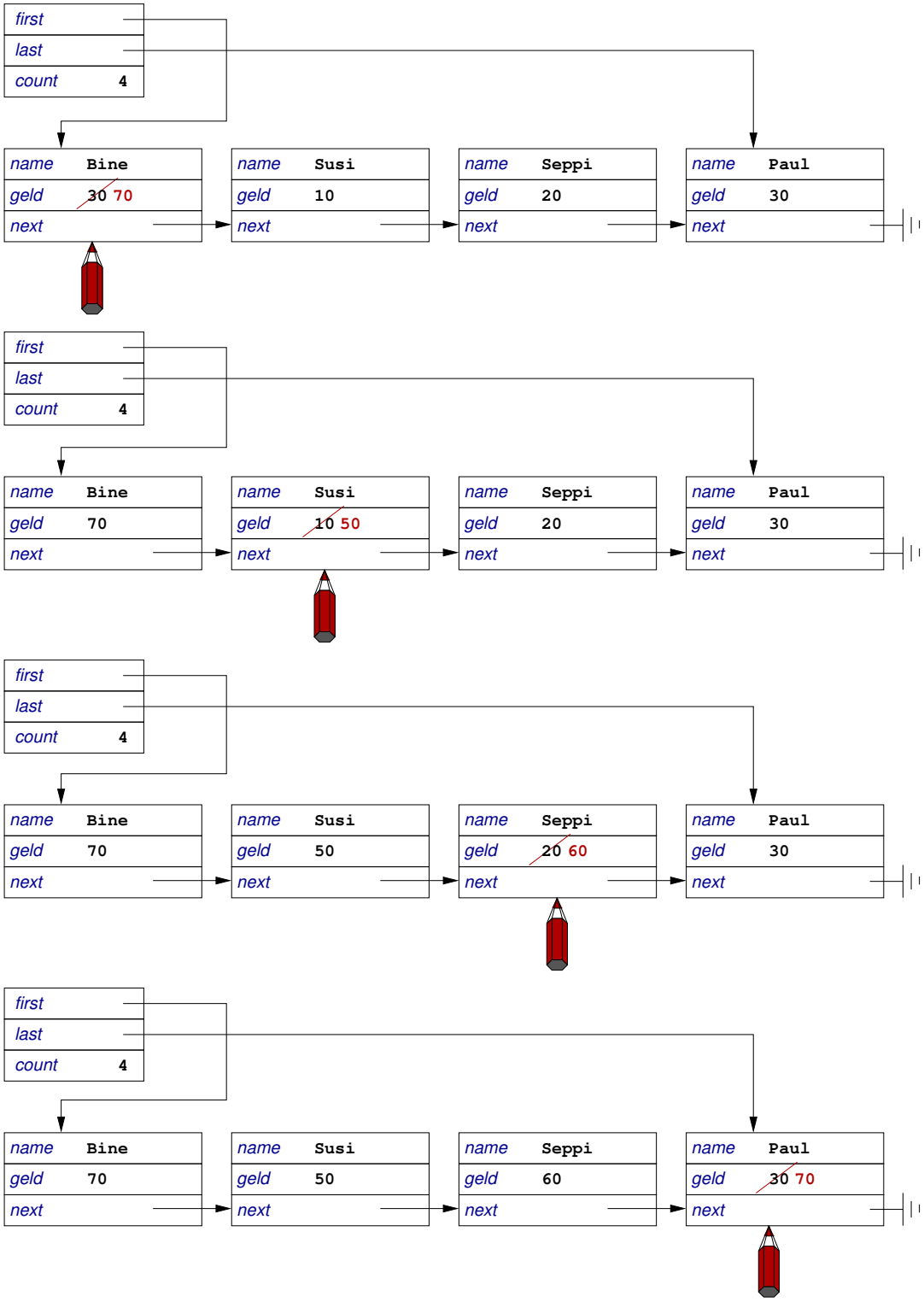
Lösung 3.2:

Du befindest dich auf dem letzten Element. Du kannst den Namen “Lydia” lesen. Du kannst der Referenz `next` nicht folgen: Sie ist mit keinem weiteren Element verbunden, d.h. sie ist auf `null` gesetzt.

Lösung 3.3:

Von “Silvia” aus kann man “Teresa” nur erreichen, indem man den Stift wieder auf das Listenobjekt setzt, von dort aus über `first` das erste Element (“Gisela”) anspricht, und von “Gisela” weiter auf “Teresa” setzt. Leider kann man nicht rückwärts gehen, denn es gibt keinen Verweis auf den Vorgänger.

Lösung 3.4:


first
last
count 4
name Bine
geld 30 70
next
name Susi
geld 10
next
name Seppi
geld 20
next
name Paul
geld 30
next
first
last
count 4
name Bine
geld 70
next
name Susi
geld 10 50
next
name Seppi
geld 20
next
name Paul
geld 30
next
first
last
count 4
name Bine
geld 70
next
name Susi
geld 50
next
name Seppi
geld 20 60
next
name Paul
geld 30
next
first
last
count 4
name Bine
geld 70
next
name Susi
geld 50
next
name Seppi
geld 60
next
name Paul
geld 30 70
next

Lösung 3.5:

Die Suchkriterien können der Name oder das Lieblingsgericht oder beides sein. Folgende Fragen kann man stellen:

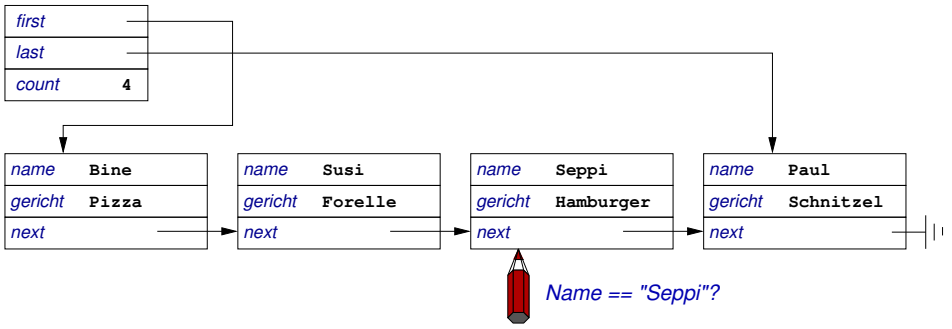
- Suche ein Objekt, das den Namen X hat!
- Suche ein Objekt, dessen Lieblingsgericht Y ist!

Natürlich können auch beide Suchkriterien kombiniert werden:

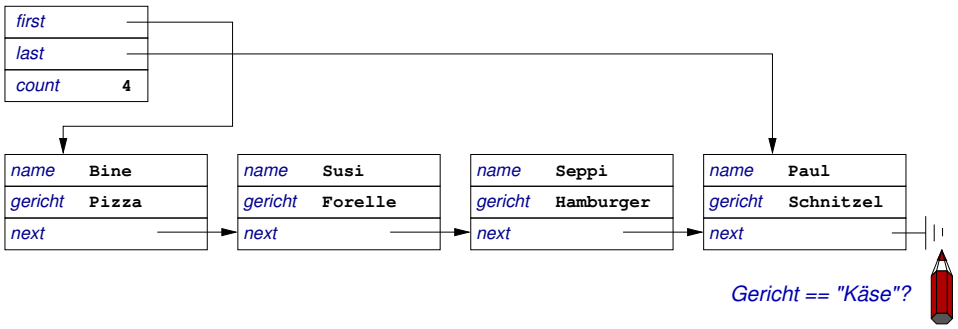
- Suche ein Objekt, das den Namen X und das Lieblingsgericht Y hat!
- Suche ein Objekt, das den Namen X oder das Lieblingsgericht Y hat!

Lösung 3.6:

- Es gibt eine Person “Seppi”. An dieser Stelle kann die Suche beendet werden.

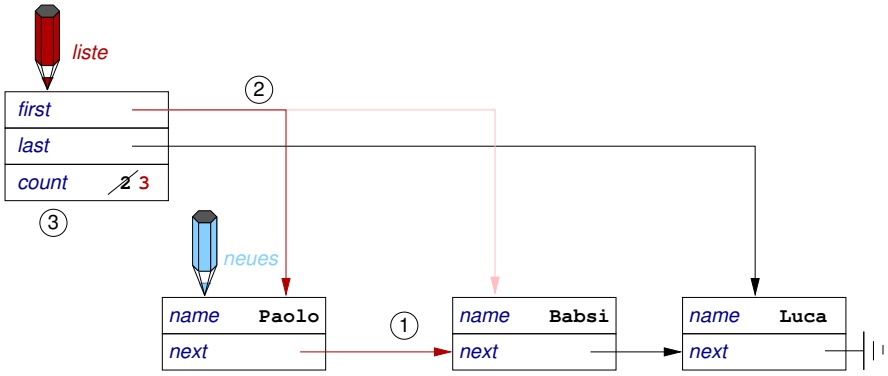


- Niemand in der Liste hat das Lieblingsessen “Käse”. Für die Suche muss also über die komplette Liste iteriert werden. Der Stift zeigt danach auf `null`.



Lösung 3.7:

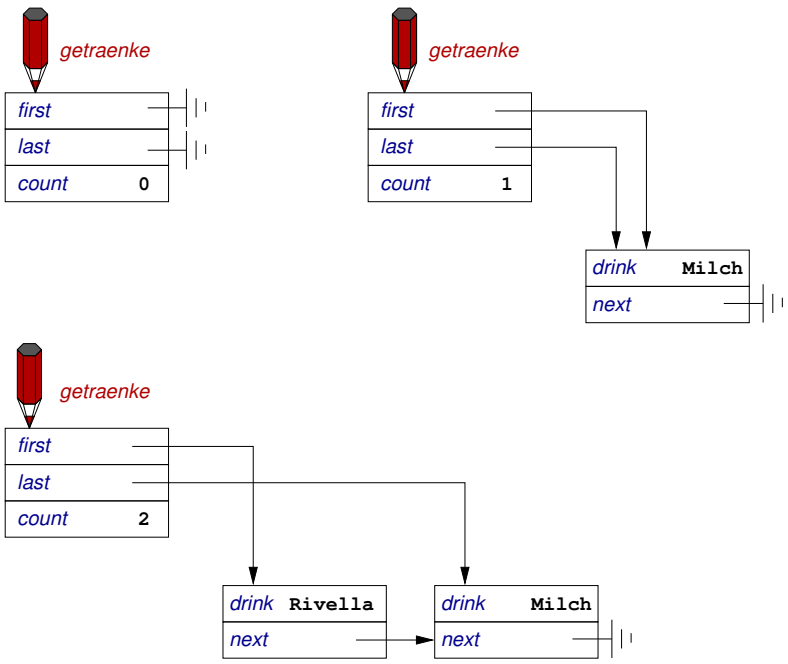
Um ein Element vorne in eine Liste mit mehreren Elementen einzufügen, müssen die folgenden drei Schritte durchgeführt werden, um ein Element vorne einzufügen:

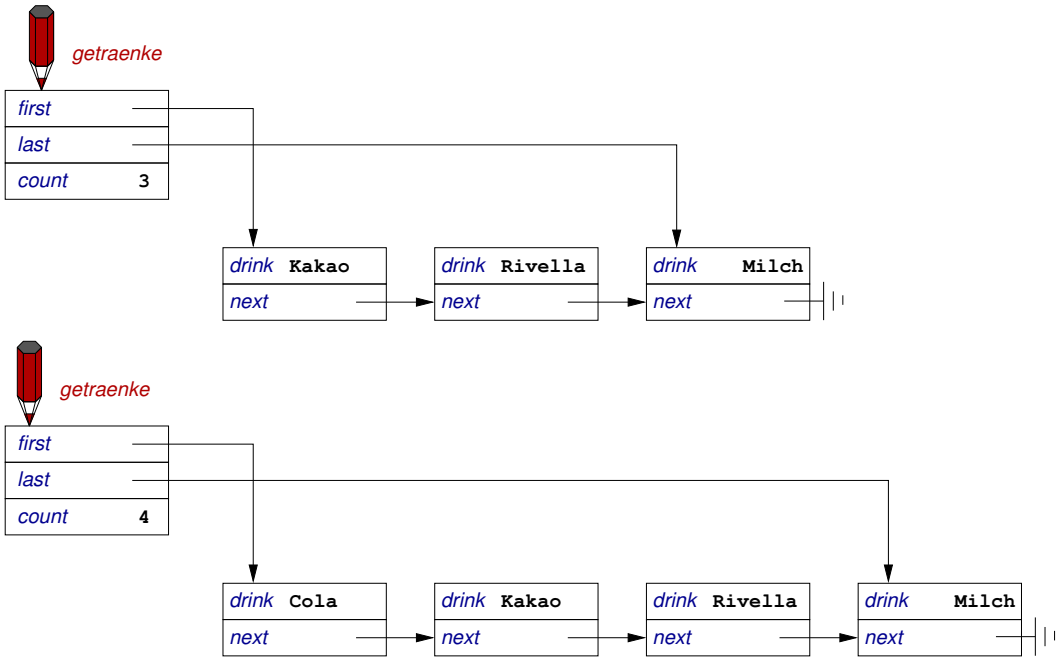


1. Verkette das neue Element mit dem ersten, indem du `next` vom neuen Element auf das erste Element setzt.
2. Setze `first` auf das neue Element.
3. Erhöhe die Anzahl der Elemente der Liste.

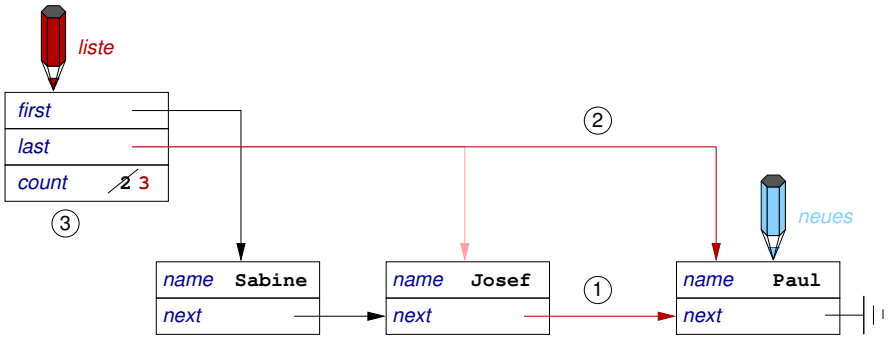
Die Reihenfolge der ersten beiden Operationen ist wichtig. Wenn du beispielsweise zuerst `first` auf das neue Element setzt, kannst du die ursprünglichen Listenelemente nicht mehr erreichen.

Lösung 3.8:





Lösung 3.9:



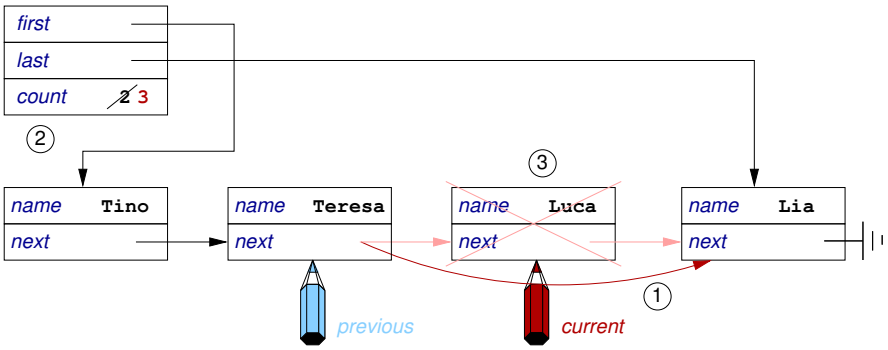
1. Verkette das letzte Element der Liste mit dem neuen Element, indem du `next` vom letzten Element auf das neue Element setzt.
2. Setze `last` auf das neue Element.
3. Erhöhe die Anzahl der Elemente um 1.

Auch hier ist die Reihenfolge der ersten beiden Operationen wichtig. Vertauscht du diese, kannst du das letzte Element nicht erreichen!

Lösung 3.10:

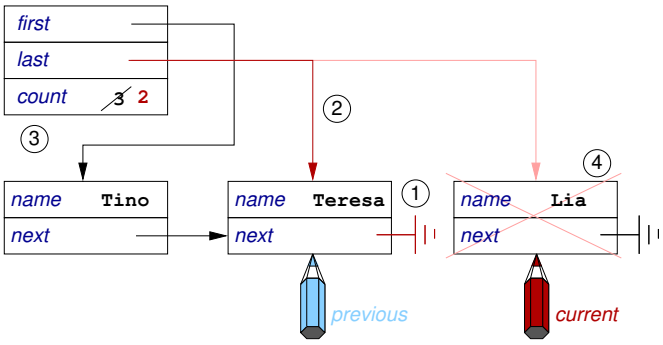
Zuerst werden beide Stifte auf das erste Element gelegt, d.h. man folgt `first`. Der rote Stift auf das Nachbarelement gesetzt. Danach wird der blaue Stift dorthin gelegt, wo der rote Stift hinzeigt, und der rote Stift kann ein Element nach hinten gesetzt werden.

Der rote Stift zeigt auf das zu löschende Element und der blaue auf den Vorgänger. Nun wird ein neuer Nachfolger des blauen Elements festgelegt: ① Der Nachfolger des roten Elements (“Lia”) wird zum Nachfolger des blauen Elements (“Teresa”). Wir brauchen den blauen Stift, um auf den Vorgänger zugreifen zu können. Nun ist das Element “Luca” aus der Liste ausgekettet. ② Die Anzahl der Elemente muss angepasst werden. ③ Das Element kann vom System entfernt werden.



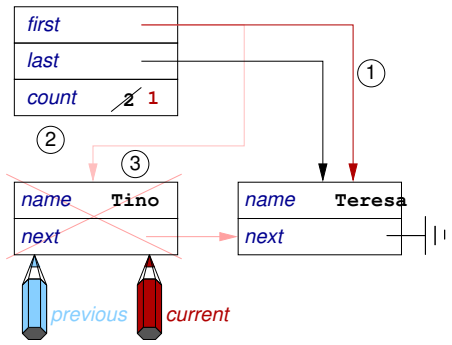
Lösung 3.11:

Der rote Stift zeigt auf das letzte Element “Lia”. “Teresa” ist der Vorgänger (*previous*) von “Lia”. Das vorletzte Element wird nun zum letzten Element: ① “Lia” wird ausgekettet, indem der Nachfolger von “Teresa” auf `null` gesetzt wird. ② `last` wird auf das vorletzte Element gesetzt. ③ Die Anzahl der Elemente angepasst, und ④ das System kann den Speicherplatz freigeben.



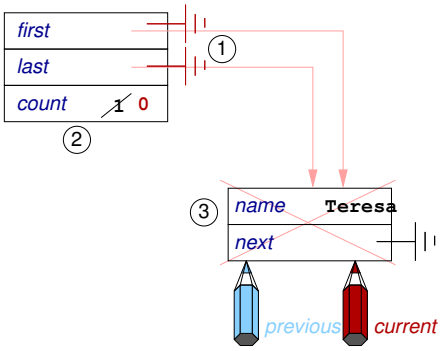
Lösung 3.12:

“Tino” ist das erste Element in der Liste. Das zweite Element wird das erste Element: ① `first` wird auf den Nachfolger des ersten Element gesetzt. ② Die Anzahl der Elemente wird erniedrigt und ③ das Element kann vom System gelöscht werden.

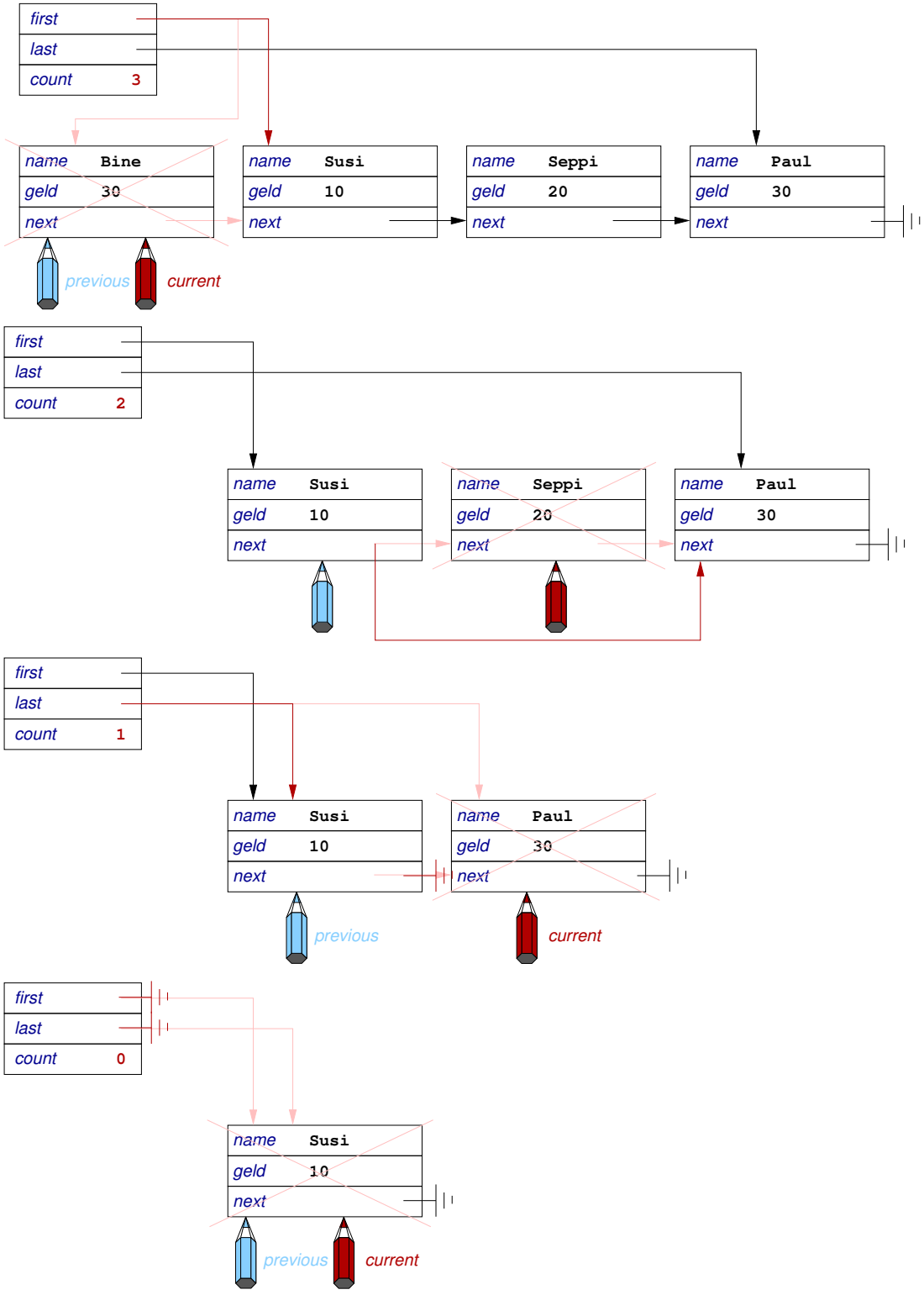


Lösung 3.13:

Das einzige Element einer Liste wird gelöscht, indem ① `first` und `last` auf `null` gesetzt werden. ② Die Anzahl der Elemente wird auf 0 gesetzt. ③ Das System gibt den Speicherplatz des Elements frei.

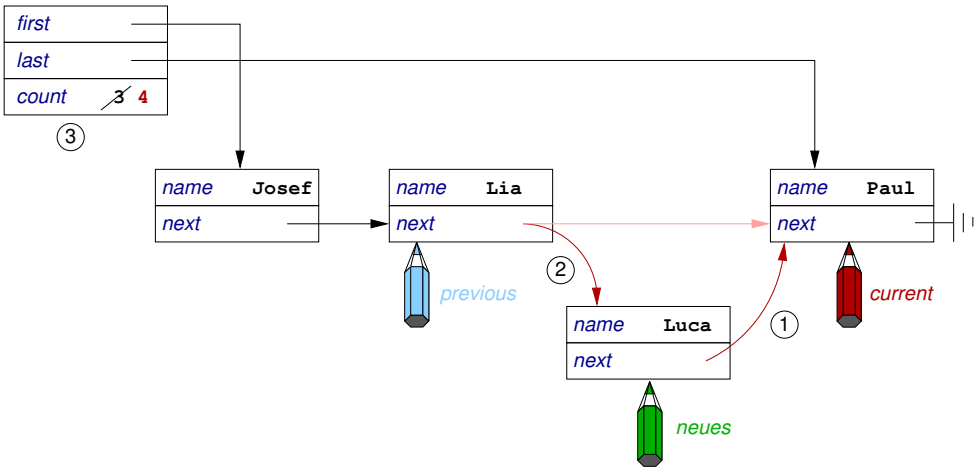


Lösung 3.14:


first
last
count 3
name Bine
geld 30
next
name Susi
geld 10
next
name Seppi
geld 20
next
name Paul
geld 30
next
previous
current
first
last
count 2
name Susi
geld 10
next
name Seppi
geld 20
next
name Paul
geld 30
next
first
last
count 1
name Susi
geld 10
next
name Paul
geld 30
next
previous
current
first
last
count 0
name Susi
geld 10
next
previous
current

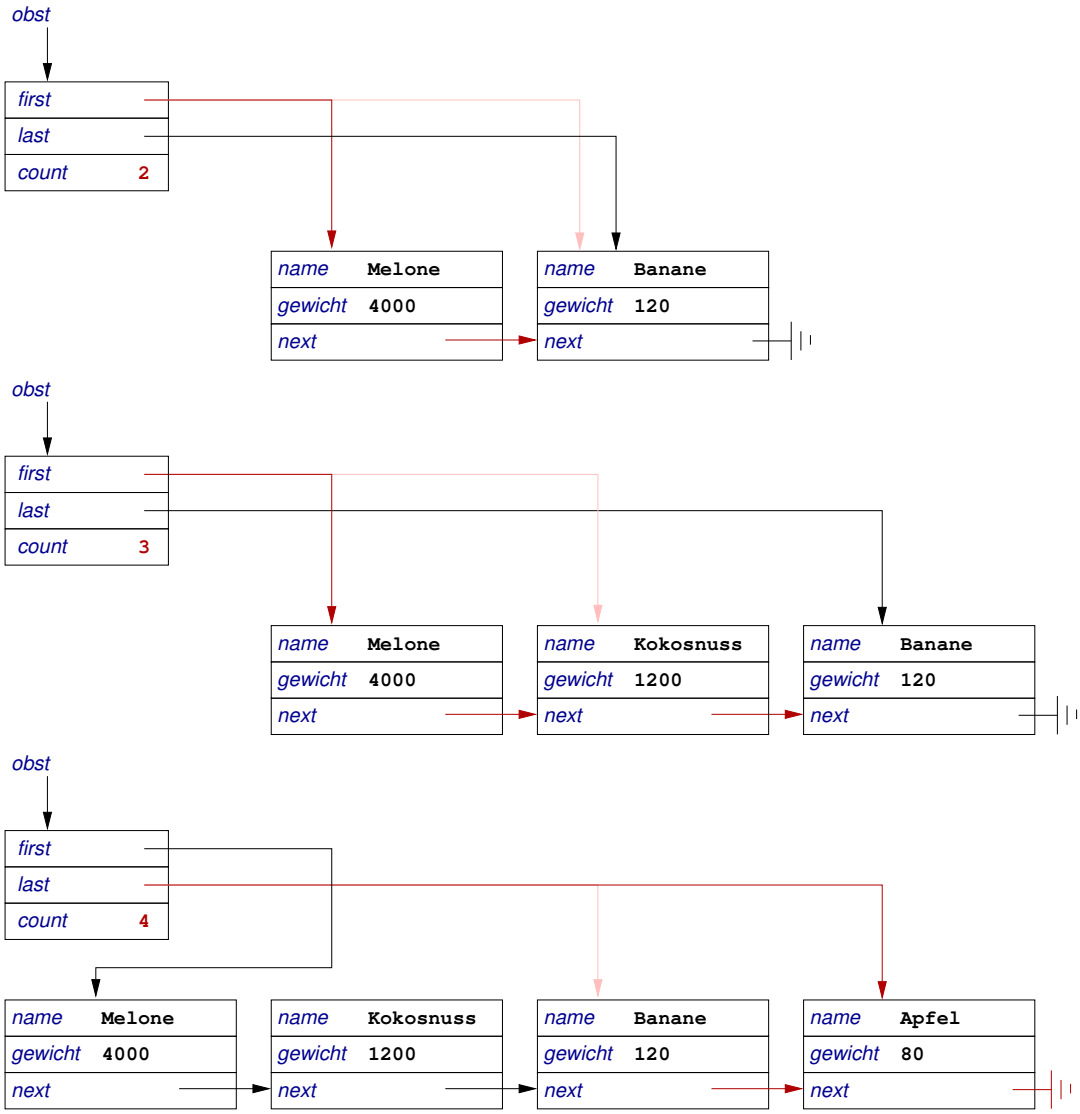
Lösung 3.15:

Der rote Stift zeigt auf das erste Element, das grösser als das neue Element ist (’P’ kommt im Alphabet nach ’L’). Der blaue Stift zeigt auf dessen Vorgänger (also auf “Lia”). Zwischen diesen beiden Elementen soll “Luca”eingefügt werden (auf “Luca”zeigt der grüne Stift”).



① Zuerst wird der Nachfolger des neuen Elements auf “Paul” gesetzt. ② Danach wird der Nachfolger des Vorgängers auf das neue Element gesetzt. ③ Die Anzahl der Elemente wird erhöht.

Lösung 3.16:


obst
first
last
count 2
name Melone
gewicht 4000
next
name Banane
gewicht 120
next
obst
first
last
count 3
name Melone
gewicht 4000
next
name Kokosnuss
gewicht 1200
next
name Banane
gewicht 120
next
obst
first
last
count 4
name Melone
gewicht 4000
next
name Kokosnuss
gewicht 1200
next
name Banane
gewicht 120
next
name Apfel
gewicht 80
next

3.11 Lernkontrolle

LK 3.1: Listenobjekt

Zeichne eine Liste im Hauptspeicher: Die Elemente speichern Buchstaben des Alphabets. Die Liste soll 3 beliebige Elemente haben. Benenne alle Felder des Listenobjekts sowie der Elementobjekte.

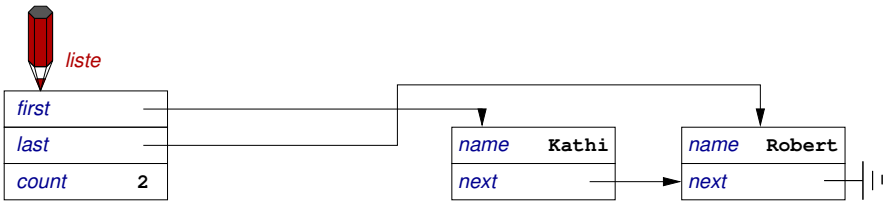
LK 3.2: Vorne Einfügen

a) Welchen Sonderfall muss man behandeln, wenn man ein Element in eine nicht sortierte Liste einfügen möchte? Zeichne eine Skizze.

b) Beschreibe die drei Schritte, um ein Element vorne an eine Liste einzufügen. Zeichne eine Skizze.

LK 3.3: Sortiertes Einfügen

Gegeben ist folgende Liste. Füge die 3 Namen in dieser Reihenfolge alphabetisch sortiert ein: Amelie, Lia, Tino. Benutze das Stiftmodell und stelle jeden Zwischenschritt dar.

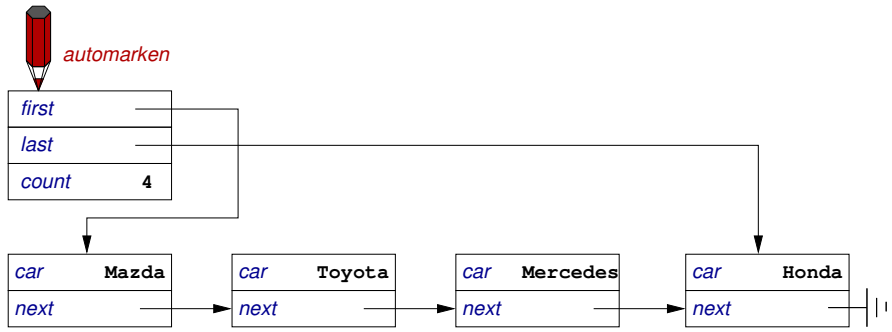


LK 3.4: Über eine Liste iterieren

Berechne das Gesamtgewicht dieses Obstkorbes. Mit Hilfe des Stiftmodells iteriere über diese Liste und berechne die Summe der Einzelgewichte. Stelle dir Zwischenschritte dar, indem du jede Position des Stiftes mit der Teilsumme des Gewichts einzeichnest.

LK 3.5: Löschen von Elementen

Gegeben ist eine Liste von Autos. Lösche diese Autos in der gegebenen Reihenfolge: Mercedes, Mazda, Toyota, Honda.

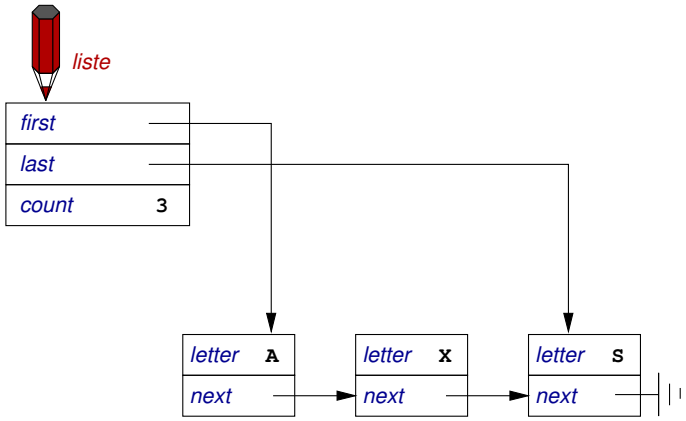


LK 3.6: Löschen

Beschreibe den Vorgang, wie man ein Element aus einer Liste löscht. Welches sind die vier Fälle? Wie müssen die Referenzen bei diesen vier Fällen gesetzt werden?

3.12 Lösungen zur Lernkontrolle

LK Lösung 3.1: Listenobjekt (K2)

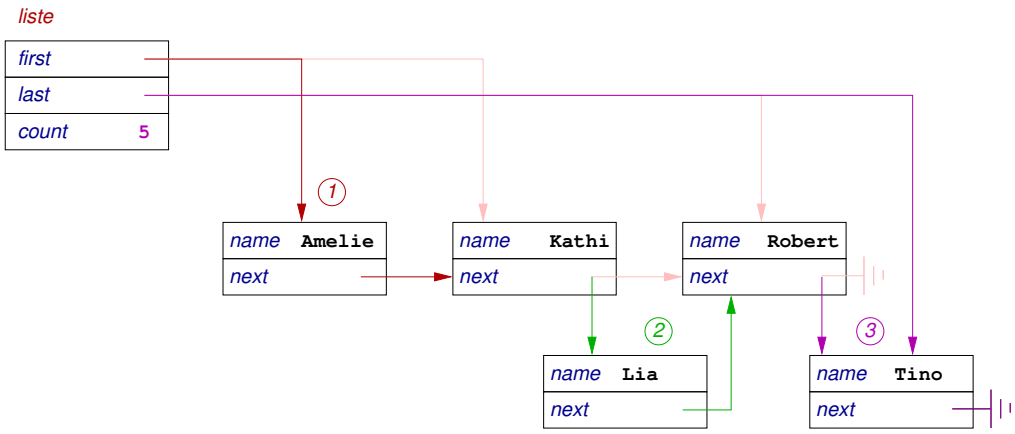


LK Lösung 3.2: Einfügen (K2)

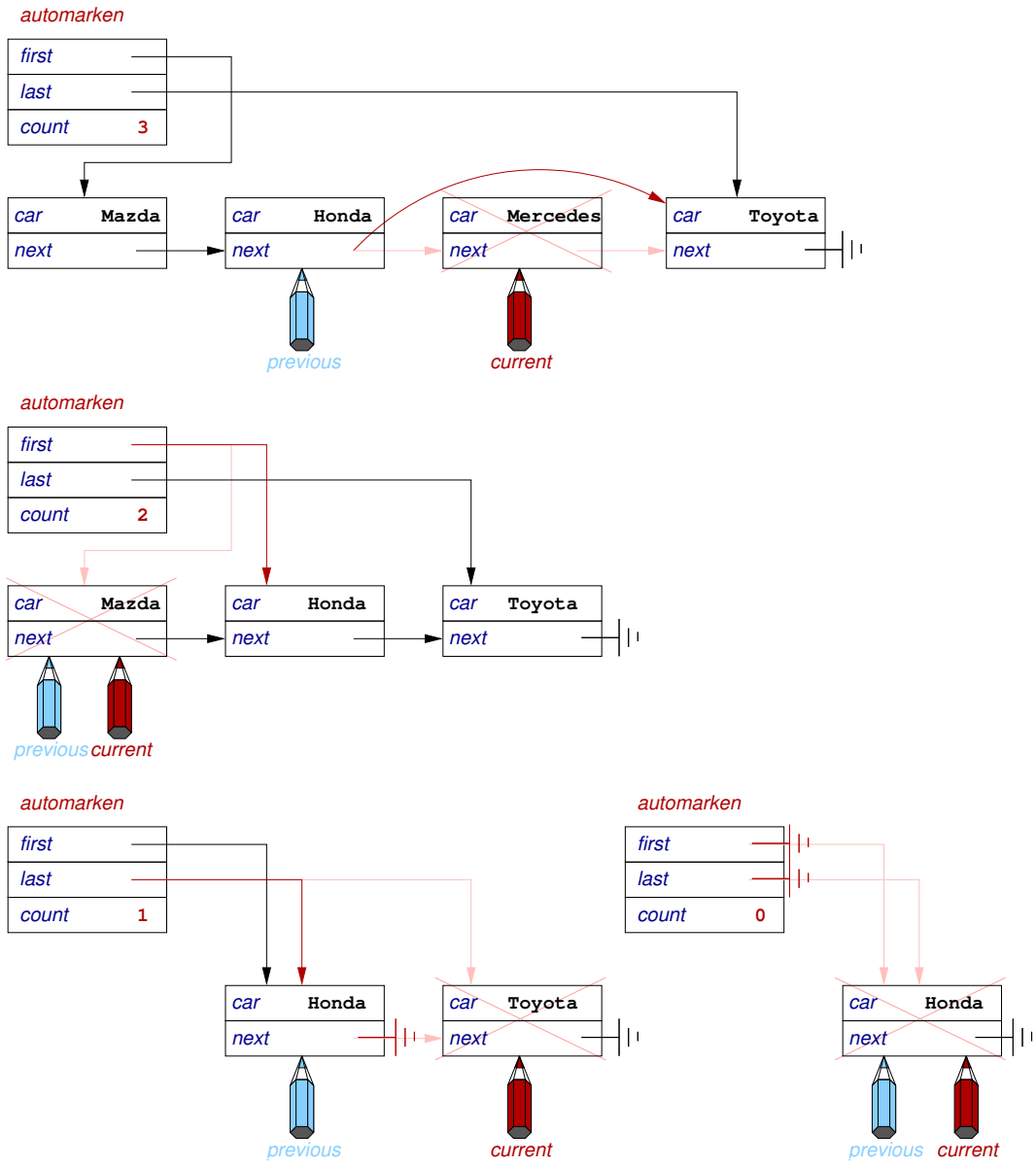
a) Der Sonderfall ist das Einfügen eines Elements in eine leere Liste. `first` und `last` werden auf das neue Element gesetzt. Die Anzahl der Elemente wird auf `1` geändert.

b) (a) Setze `next` vom neuen Element auf das erste Element.
(b) Setze `first` auf das neue Element.
(c) Erhöhe die Anzahl der Elemente der Liste.

LK Lösung 3.3: Sortiertes Einfügen (K3)



LK Lösung 3.4: Löschen von Elementen (K3)



LK Lösung 3.5: Löschen (K2)

Das Löschen besteht aus zwei Schritten. Zuerst muss das entsprechende Element in der Liste gesucht werden, d.h. über die Liste wird iteriert. Dabei zeigt `current` auf das gesuchte Element, `previous` zeigt auf den Vorgänger von `current`. Danach kann das Element entfernt werden. Folgende Fälle müssen unterschieden werden:

- Einziges Element: `first` und `last` werden `null`.
- Erstes Element: `first` wird auf `first.next` gesetzt.
- Letztes Element: `last` wird auf `previous` gesetzt.
- Element in der Mitte: `previous.next` wird auf `current.next` gesetzt.

Bei allen Fällen wird `count` wird um `1` erniedrigt.

Kapitel 4

Listen programmiert

4.1 Worum geht es?

In diesem Kapitel besprechen wir alle Klassen, die notwendig sind, um eine verkettete Liste zu entwickeln und verwenden. Dazu gehört eine Klasse, die Listenelemente modelliert. Eine Listenklasse beschreibt die Listenoperationen. Um die Liste auch zu verwenden, erstellen wir eine einfache Klasse zur Verwaltung einer Kontaktliste. Als Listenelement schreiben wir daher eine Klasse Kontakt. Wir werden gemeinsam Programmtext ansehen. Du wirst auch Programmtext ändern bzw. selbst Operationen schreiben.

4.2 Lernziele

Nach diesem Kapitel sollst du folgendes können:

- Den Programmtext, im speziellen die einzelnen Methoden, Zeile für Zeile nachvollziehen.
- Einzelne Methoden verändern.
- Neue Methoden zu den bestehenden Klassen hinzufügen.

4.3 Das Programm ausführen

Wir wollen ein kleine Kontaktliste schreiben. Sie soll Kontaktdaten von Personen mit ihren Telefonnummern und E-Mail-Adressen verwalten. Im Verzeichnis `linkedlist/-single/` findest du die folgenden Dateien:

`LinkedElement.java`	Basis für alle Elemente einer Liste
`LinkedList.java`	Eine einfach verketteten Liste
`Contact.java`	Ein Kontakt für die Kontaktliste
`ContactList.java`	Die Kontaktliste
`Main.java`	Die Main-Datei
`Makefile`	Datei zum Kompilieren und Ausführen des Programms

Kompiliere und starte das Programm im Verzeichnis `linkedlist/single/`. In der Textdatei `linkedlist/README` findest du genaue Anweisungen dafür. Sieh dir den Output an! Was fällt dir auf?

In diesem Kapitel zeigen wir den Programmtext der einzelnen Klassen immer ausschnittsweise. Du kannst den Text hier im Kapitel mitlesen oder dir die Quelltexte ausdrucken. Diese enthalten auch die Dokumentation, die wir der Einfachheit hier weglassen.

4.4 Die abstrakte Klasse LinkedElement

Um die Elemente zu repräsentieren, erstellen wir eine Klasse `LinkedElement`. Im Kapitel 3 hast du gelernt, dass jedes Listenelement mit seinem Nachbar verkettet ist. Deshalb hat die Klasse ein Attribut `next` vom Typ `LinkedElement`. In Zeile 13 kannst du lesen, dass diese Klasse abstrakt ist. Von einer abstrakten Klasse kann man kein Objekt erzeugen. Eine Subklasse muss zuerst eine abstrakte Klasse erweitern — dann kann man von dieser Subklasse Objekte erzeugen.

Warum ist nun die Klasse `LinkedElement` abstrakt? In Kapitel 3 speichern die Listenelemente Namen und Taschengeld, Autos, Getränke etc. Hätten wir reine `LinkedElement`-Objekte, würden diese nur Referenzen auf den Nachfolger verwalten — und keine weiteren Daten speichern. So haben wir uns für diese abstrakte Klasse entschieden.

```
13 public abstract class LinkedElement
14 {
15    private LinkedElement next;  /* Nachfolger */
16
17    /* Nachfolger  */
18    public final LinkedElement getNext ()
19    {
20       return next;
21    }
22
23    /* Setze den Nachfolger auf 'next'.  */
24    public final void setNext (LinkedElement next)
25    {
26       this.next = next;
27    }
28
29    /* Ist 'other' gleich? */
30    public abstract boolean isEqualTo (LinkedElement other);
31
32 }
```

Das Attribut `next` verkettet die Listenelemente. Die beiden Operationen `getNext()` und `setNext()` haben schon eine Implementation: `getNext()` ermöglicht den Zugriff auf das Attribut `next`. Mit `setNext()` kann man einen neuen Nachfolger für

dieses Element festlegen.

Die Methode `isEqualTo` vergleicht zwei Listenelemente auf Gleichheit. Im Vergleich zu `getNext()` und `setNext()` ist sie abstrakt. Wir wissen noch nicht, was genau in der Liste gespeichert sein wird und was überhaupt verglichen werden soll. Deshalb ist diese Funktion abstrakt — und eine Subklasse von `LinkedElement` muss diese Funktion implementieren.

4.5 Die Klasse LinkedList

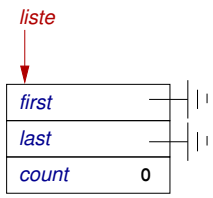
Aus dem Kapitel 3 erinnerst du dich, wie ein Listenobjekt aussieht: Es verweist auf das erste und das letzte Element und speichert die Anzahl der Elemente. Wir haben Elemente in eine Liste eingefügt und gelöscht, über die Liste iteriert.

Die Felder des Listenobjekts stellen die Attribute dieser Klasse dar. Du findest die Attribute in den Zeilen 14–16. `first` bezeichnet das erste Element, `last` das letzte Element. Beide sind vom Typ `LinkedElement`. `count` speichert die Anzahl der Listenelemente und ist daher ein Integer. Alle Attribute sind privat. Andere Klassen können nur über die entsprechenden Get-Methoden (`getFirst()`, `getLast()`, `getCount()`) darauf zugreifen. Die Attribute können nicht von ausserhalb der Klasse verändert werden.

```
12 public class LinkedList {
13
14    private LinkedElement first;   /* Erstes Element */
15    private LinkedElement last;    /* Letztes Element */
16    private int count;             /* Anzahl der Elemente */
```

Wenn du ein neues Listenobjekt erzeugst, wird der Konstruktor aufgerufen. Die Liste ist leer und wir müssen für die Attribute entsprechende Anfangswerte bestimmen: `first` und `last` werden auf `null` initialisiert, `count` auf `0` gesetzt.

```
19    public LinkedList ()
20    {
21       first = null;
22       last = null;
23       count = 0;
24    }
```



Im Programmtext (Zeilen 27–30) folgt nun eine sehr hilfreiche Funktion. Wir haben sie im Kapitel 3 oft verwendet: Mit `isEmpty()` kannst du überprüfen, ob die Liste “leer” ist, d.h. keine Elemente enthält. Diese Funktion hat einen Boolean als Rückgabewert. Die Liste ist leer, wenn `first` auf `null` zeigt. Wie du sehen wirst, verwenden wir die Funktion in den weiteren Operationen!

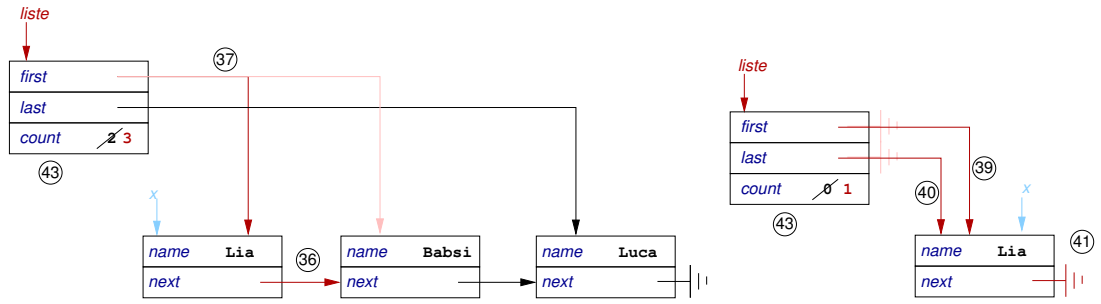
```
27    public boolean isEmpty ()
28    {
29        return (first == null);
30    }
```

Aufgabe 4.1

Die Funktion `isEmpty()` gibt wahr (`true`) zurück, wenn die Liste leer ist, d.h. wenn `first` `null` ist. Welche anderen Möglichkeiten gibt es festzustellen, ob eine Liste leer ist? Beschreibe mindestens zwei andere Möglichkeiten!

Der Befehl `add()` (im Programmtext in den Zeilen 33–43) fügt ein neues Element an den Anfang der Liste ein. Schau dir sowohl den Programmtext als auch die Abbildung 4.1 an. Die Kreise in den Abbildungen geben die entsprechende Zeile im Programmtext an.

Zuerst wird geprüft, ob schon Elemente in der Liste gespeichert sind (Zeile 35). Wenn ja, dann wird in Abbildung 4.1(a) der Nachfolger vom neuen Element auf das erste Element gesetzt. Danach wird das erste Element `first` auf das neue Element gesetzt (Zeilen 36–37). Gibt es allerdings noch kein Element in der Liste (siehe Abbildung 4.1(b), Zeile 28), dann wird das neue Element als erstes und als letztes bestimmt (Zeilen 39–40). Der Nachfolger wird auf `null` gesetzt (Zeile 41). In beiden Fällen wird die Anzahl der Elemente um eines erhöht (Zeile 43).



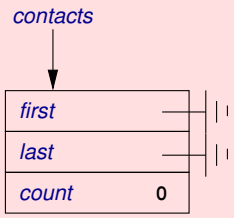
(a) Einfügen in Liste mehreren Elementen (b) Einfügen in leere Liste

Abbildung 4.1: Ein Element einfügen

```
33    public void add (LinkedElement x)
34    {
35       if (!isEmpty ()) {
36          x.setNext (first);
37          first = x;
38       } else {
39          first = x;
40          last = x;
41          x.setNext (null);
42       }
43       count ++;
44    }
```

Aufgabe 4.2

Spiele den Programmtext Zeile für Zeile durch! Gegeben hast du diese leere Liste! Füge die folgenden Elemente ein und stelle jeden Zwischenschritt dar:



Name	Telefon
Babsi	55238
Eva	38712
Bine	14320

Die nächste Operation heisst `remove()` (Zeilen 48–82). Das Suchkriterium wird als Argument verpackt und übergeben. Ist ein passendes Element in der Liste gespeichert, wird es aus der Liste ausgekettet und als Rückgabewert geliefert. Wenn das Element nicht in der Liste gefunden werden konnte, wird `null` zurück gegeben.

Im Kapitel 3 haben wir diese Schritte identifiziert, um ein Element zu löschen:

1. Ein Element wird gesucht.
2. Wenn das Element gefunden wurde:
   (a) Das Element wird ausgekettet.
   (b) Die Anzahl der Elemente wird erniedrigt.
   (c) Das Element wird gelöscht.

In Java wird das Element nicht von der Programmiererin oder vom Programmierer selbst gelöscht. Der Garbage Collector entfernt alle Objekte, die nicht mehr referenziert werden. Den letzten Schritt müssen (und können) wir selbst nicht durchführen.

Sehen wir uns den Programmtext an! Wir können nur ein Element löschen, wenn es überhaupt Elemente in der Liste gibt (Zeile 53). Der Algorithmus besteht aus zwei Teilen:

Zuerst wird das Element gesucht (Zeilen 55–59) und danach ausgekettet (Zeilen 61–79).

Wir benötigen zwei lokale Variablen (Zeilen 50–51): `current` und `previous` sind beide vom Typ `LinkedElement`. Anfangs sind sie auf `first` initialisiert.

Für das Suchen (Zeilen 56–59) wird über die Liste iteriert. `current` wird auf das erste Element gesetzt. Dann folgt eine Schleife: Solange das aktuelle Element einen Nachfolger hat und das aktuelle Element nicht das gesuchte ist, rückt `previous` um eines nach (Zeile 57), und `current` wird ein Element weiter gesetzt (Zeile 58). `previous` und `current` werden also solange um eines weiter gerückt, bis wir entweder das Element gefunden haben oder die Liste zu Ende ist. Abbildung 4.2 veranschaulicht die Schritte.

```
48    public LinkedElement remove (LinkedElement x)
49    {
50       LinkedElement current = first;
51       LinkedElement previous = first;
52
53       if (!isEmpty ()) {
54
55          /* (1) Suche Element x */
56          while ((current != null) && (!current.isEqualTo (x))) {
57              previous = current;
58              current = current.getNext ();
59          }
```

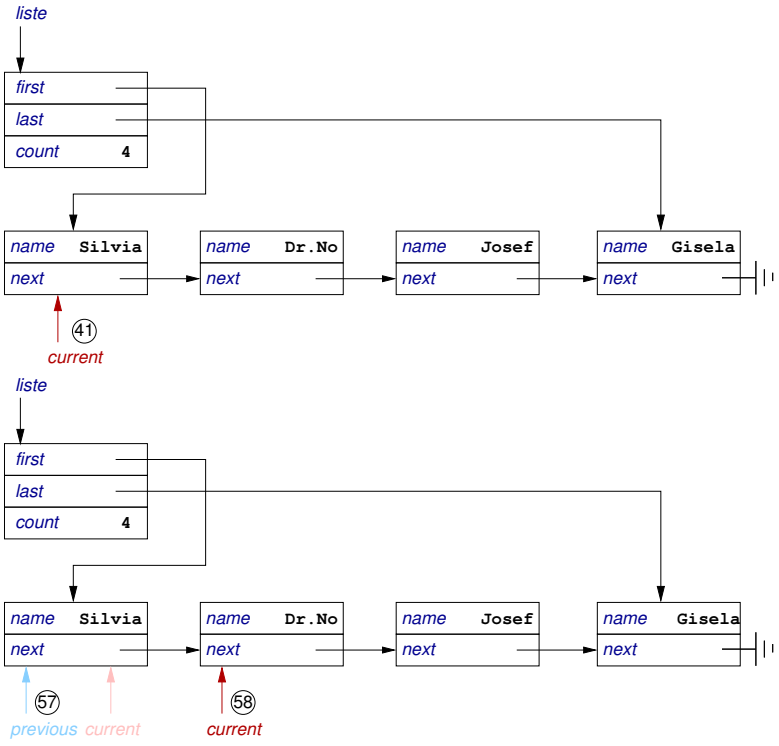


Abbildung 4.2: Suche nach dem zu löschendem Element

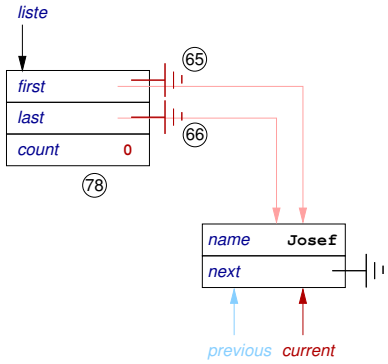
Nun folgt der zweite Teil des Algorithmus — das Ausketten des Elements aus der Liste (Zeilen 61–79)! Vom ersten Teil wissen wir, dass `current` entweder auf das gesuchte Element zeigt oder `null` ist, wenn kein Element gefunden werden konnte.

Wir müssen abfragen, ob `current` auf ein Element zeigt und nicht `null` ist (Zeile 62). Dann können wir das Element ausketten. Wie im Kapitel 3.7 besprochen, müssen wir vier verschiedene Fälle behandeln. Zusammengefasst, wir können entweder

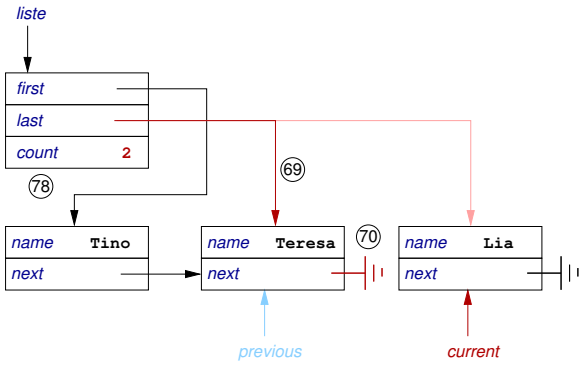
- das einzige,
- das letzte,
- das erste Element oder
- ein Element in der Mitte ausketten.

```
61          /* (2) Kette Element x aus */
62          if (current != null) {
63             if ((current == first) && (current == last)) {
64                /* Kette einziges Element aus: */
65                first = null;
66                last = null;
67             } else if (current == last) {
68                /* Kette letztes Element aus: */
69                last = previous;
70                previous.setNext (null);
71             } else if (current == first) {
72                /* Kette erstes Element aus: */
73                first = current.getNext ();
74             } else {
75                /* Kette Element in der Mitte aus: */
76                previous.setNext (current.getNext ());
77             }
78             count--;
79          }
80       }
81       return current;
82    }
```

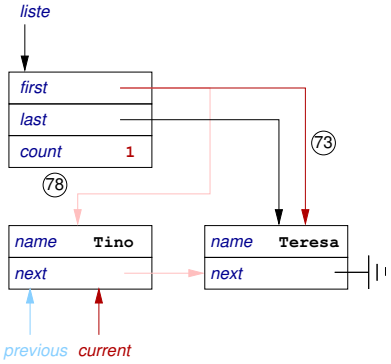
1. Zeile 63: Das einzige Element wird entfernt: Wenn `first`, `last` und `current` auf dasselbe Element zeigen, dann setzen wir `first` und `last` auf `null` — und die Liste ist leer!



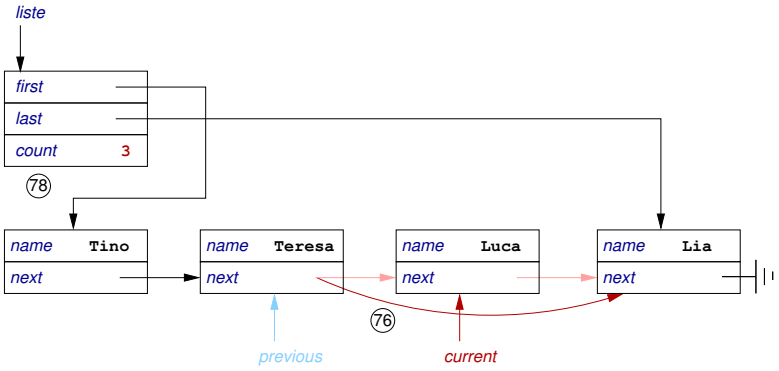
2. Zeile 67: Wir löschen das letzte Element in der Liste, wenn `current` und `last` auf dasselbe Element zeigen. `last` wird auf den Vorgänger des zu löschenden Elements gesetzt. Wir setzen noch den Nachfolger von `last` auf `null`. Dadurch wird `previous` das letzte Element.



3. Zeile 71: Zeigen `current` und `first` auf dasselbe Element, dann löschen wir das erste Element. Dazu setzen wir `first` auf den Nachfolger des zu löschenden Elements.



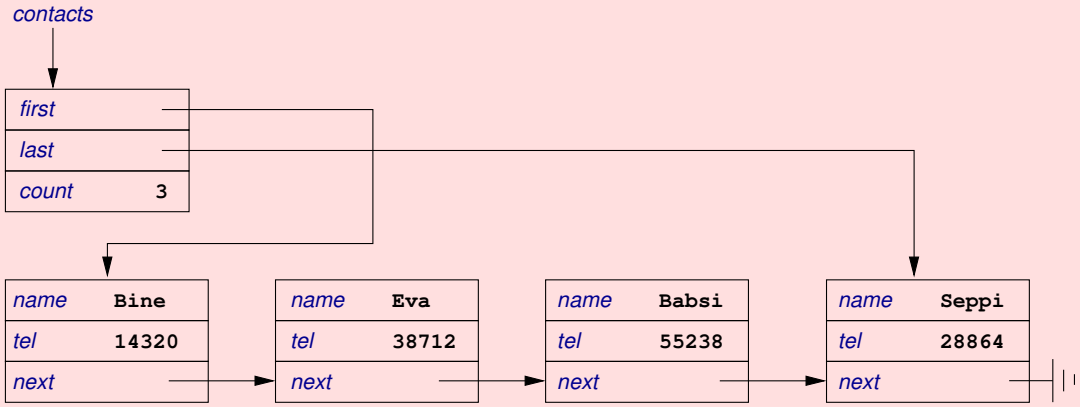
4. Zeile 74: Übrig bleibt der Fall, wenn sich das Element in der Mitte der Liste befindet. Wir löschen es, indem wir den Nachfolger des Vorgängers auf den Nachfolger des zu entfernenden Elements setzen. Anders gesagt: Der Nachfolger von `previous` wird der Nachfolger von `current`!



Was dürfen wir nicht vergessen? Wir müssen noch die Anzahl der Elemente aktualisieren (Zeile 78)!

Aufgabe 4.3

Spiele den Programmtext durch! Aus der gegebenen Liste lösche folgende Elemente in der gegebenen Reihenfolge: Eva, Bine, Seppi, Babsi. Zeichne die Zwischenschritte!



Die `toString()`-Funktion wandelt die Liste in einen String um. Diese Funktion ist in der Klasse `Object` definiert und ist daher für alle Klassen verfügbar. Jede Klasse sollte diese Funktion überschreiben, um selbst bestimmen zu können, wie die String-Darstellung aussieht.

Wie könnte eine String-Darstellung einer Liste aussehen? Jedes Element der Liste sollte als String dargestellt werden. Wir wollen also über die Liste iterieren und dabei jedes Element in die String-Darstellung der Liste einfügen. Wie du gleich sehen wirst, verwenden wir dabei die `toString()` Methode der Klasse `Object`! Das bedeutet, dass die Klasse `Object`, `LinkedElement` oder eine ihrer Subklassen entscheidet, wie ein Element als String dargestellt wird.

```
85    public String toString ()
86    {
87       StringBuffer result = new StringBuffer ();
88
89       for (LinkedElement current = first; current != null;
90             current = current.getNext ()) {
91          result.append (current.toString());
92          result.append ("\n");
93       }
94       return result.toString ();
95    }
```

Sieh dir den Programmtext an (Zeilen 85–95)! Wir benutzen eine Variable vom Typ `StringBuffer` (Zeile 87). Ein StringBuffer erlaubt dir, Strings mit der Operation `append()` an diesen StringBuffer zu hängen.

Wir verwenden für diese Funktion eine `for`-Schleife (Zeile 89), um über die Liste zu iterieren. Zur Initialisierung setzen wir `current` auf das erste Element. Solange `current` nicht `null` ist, wird der Rumpf (Zeilen 91–92) ausgeführt: Das aktuelle Element wird in einen String umgewandelt und an den `StringBuffer` angehängt. Danach folgt ein Zeilenumbruch, der durch das spezielle Zeichen `'\n'` dargestellt wird. Am Ende wird der StringBuffer in einen String umgewandelt und zurück gegeben.

4.6 Eine Liste mit Kontakten

In der Klasse `LinkedList` finden wir alle Operationen einer verketteten Liste. Die abstrakte Klasse `LinkedElement` gibt Operationen vor, die alle Listenelemente kennen müssen. Nun geht es darum, die Klassen `LinkedList` und `LinkedElement` zu verwenden.

Wir möchten eine Kontaktliste programmieren, um unsere Kontakte zu verwalten. Zur Kontaktliste möchten wir Kontakte hinzufügen und auch löschen können. Wir möchten die Kontaktliste ausgeben können. Von einem Kontakt interessiert uns der Name der Person, ihre Telefonnummer und E-Mail-Adresse.

Daraus können wir erkennen, dass wir eine Klasse `ContactList` und eine Klasse `Contact` entwerfen werden. In der Abbildung 4.3 siehst du, wie die beiden neuen Klassen mit den Klassen `LinkedElement` und `LinkedList` zusammen hängen:

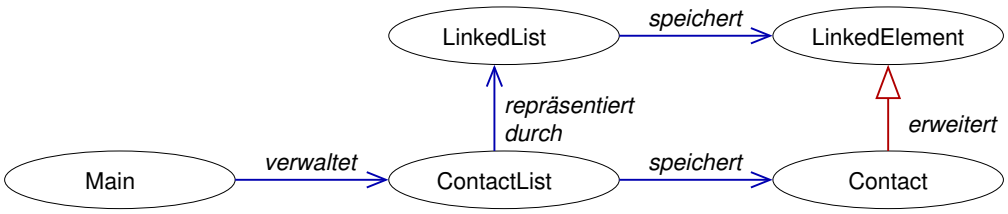


Abbildung 4.3: Klassendiagram

4.7 Die Klasse Contact

Kontaktdaten bestehen aus Name, Telefonnummer und E-Mail-Adresse. Der Name, die Telefonnummer und die E-Mail-Adresse sind alle vom Typ `String` (Zeilen 19–21).

```
17 public class Contact extends LinkedElement
18 {
19    private String name;    /* Name */
20    private String phone;   /* Telefonnummer */
21    private String email;   /* Email-Adresse */
```

Wir haben uns entschieden, dass man den Namen nicht ändern können sollte — Telefonnummer und E-Mail-Adresse schon. Der Name wird also gleich im Konstruktor übergeben und initialisiert (Zeilen 24–27).

```
24    public Contact (String name)
25    {
26       this.name = name;
27    }
```

Nachfolgend in den Zeilen 29–45 findest du die “Getters”. Diese Funktionen erlauben dir, die Attribute zu lesen. Mit den “Setters” (Zeile 47–57) kannst du die Werte der Attribute ändern.

```
29    /* Name */
30    public final String getName ()
31    {
32       return name;
33    }
34
35    /* Email-Adresse */
36    public final String getEmail ()
37    {
38       return email;
39    }
40
```

```
41      /* Telefonnummer */
42      public final String getPhone ()
43      {
44          return phone;
45      }
46
47      /* Setze die Telefonnummer of 'no'. */
48      public void setPhone (String no)
49      {
50          this.phone = no;
51      }
52
53      /* Setze die Email-Adresse auf 'addr'. */
54      public void setEmail (String addr)
55      {
56          this.email = addr;
57      }
```

Wir implementieren auch eine `toString()`-Funktion, um einen Kontakt als String darstellen zu können (Zeilen 70–79). Wir verwenden wieder einen `StringBuffer` als lokale Variable. Wir fügen den Namen, die Telefonnummer und die E-Mail-Adresse hinzu. Zur Rückgabe wandeln wir den `StringBuffer` in einen `String` um.

```
70      public String toString ()
71      {
72          StringBuffer result = new StringBuffer ();
73          result.append (name);
74          result.append (": ");
75          result.append (phone);
76          result.append (" ");
77          result.append (email);
78          return result.toString ();
79      }
```

Die Kontakte sollen in einer Kontaktliste abgelegt werden können. Die Kontaktliste benutzt eine verkettete Liste. Ein Kontakt ist also das Element, das in einer Liste gespeichert wird. Das bedeutet, dass die Kontakte verkettbar sein müssen. Deshalb erweitert die Klasse `Contact` die Klasse `LinkedElement`. Im Programmtext kannst du das in der Zeile 17 lesen.

Von der Klasse `LinkedElement` erben wir das Attribut `next`, sowie die Methoden `setNext()` und `getNext()` für den Zugriff auf dieses Attribut. Jedoch lässt die Klasse `LinkedElement` eine Methoden abstrakt. Für diese müssen wir eine Implementierung zur Verfügung stellen. `isEqualTo()` ist eine Vergleichsoperationen, die wir z.B. bei `remove` verwenden. Als Kriterium für den Vergleich verwenden wir den Namen des Kontakts.

Die Klasse `String` stellt eine Operation `compareTo()` zur Verfügung. Diese Operation vergleicht zwei Strings. Das Resultat ist `0`, wenn beide Strings gleich sind, d.h. die gleiche Sequenz von Zeichen besitzen. Ist der Argument-String alphabetisch grösser, ist das Ergebnis negativ. Ist der Argument-String kleiner, ist das Ergebnis positiv. Das nutzen wir für die Vergleichsoperation.

```
60    public boolean isEqualTo (LinkedElement other)
61    {
62       if (other instanceof Contact) {
63          String name2 = ((Contact)other).getName ();
64          return (name.compareTo (name2) == 0);
65       } else
66          return false;
67    }
```

Die Operation arbeitet auf Objekten vom Typ `LinkedElement`. Die Superklasse `LinkedElement` kennt die Erben nicht. Deswegen müssen wir beim Vergleich achten, dass beide Elemente vom gleichen Typ sind (Zeile 62). In diesem Fall speichern wir in einer lokalen Variable den Namen der Vergleichskontakts. Dafür müssen wir das Vergleichselement `other` in den Typ `Contact` umwandeln (Zeile 63). Danach vergleichen wir die beiden Namen mit Hilfe der oben beschriebenen Funktion `compareTo()`.

4.8 Die Klasse ContactList

Die Klasse `ContactList` implementiert eine sehr einfache Kontaktliste. Als interne Datenstruktur verwendet sie eine verkettete Liste, um die Elemente der Kontaktliste (d.h. die Kontakte) abzuspeichern. Sie verwendet ein privates Attribut `representation`, das vom Typ `LinkedList` ist (siehe Zeile 13). Der Konstruktor (Zeilen 16–19) erzeugt ein neues Objekt vom Typ `LinkedList`.

```
10 public class ContactList {
11
12    /* Interne Repräsentation dieser Kontaktliste */
13    private LinkedList representation;
14
15    /* Konstruktor */
16    public ContactList ()
17    {
18       representation = new LinkedList ();
19    }
```

Warum erweitert die Klasse `ContactList` nicht die Klasse `LinkedList`? Die Klasse `ContactList` ist doch auch eine Liste, die Kontakte speichert und aufbewahrt? Die Klasse `ContactList` bietet eine andere Schnittstelle als die Klasse `LinkedList`. Wir haben entschieden, nicht von der Klasse `LinkedList` zu erben, sondern sie zu verwenden.

Der Befehl `addContact()` (Zeilen 23–30) dient dazu, einen neuen Kontakt in die Kontaktliste einzufügen. Als Argumente finden wir einen Namen, eine Telefonnummer und eine E-Mail-Adresse vom Typ `String`. Es wird ein neuer Kontakt erzeugt (Zeile 25). Danach wird die Telefonnummer und die E-Mail-Adresse festgelegt (Zeilen 26–27). Der Kontakt wird in die Kontaktliste eingetragen (Zeile 28) und eine Bestätigung am Bildschirm ausgegeben (Zeile 29).

```
23    public void addContact (String name, String tel, String mail)
24    {
25       Contact c = new Contact (name);
26       c.setPhone (tel);
27       c.setEmail (mail);
28       representation.add (c);
29       System.out.println ("Added " +c+ " to the contact list");
30    }
```

Die Operation `delContact()` (vgl. Zeilen 33–45) verwendet als Parameter einen String, denn als Vergleichskriterium gilt der Name. Mit diesem Namen wird eine lokale Variable vom Typ `Contact` erzeugt (Zeile 35). Wir geben eine Information aus, dass nun versucht wird, einen Kontakt mit diesem Namen zu löschen (Zeile 37). Wir rufen die entsprechende Funktion `remove()` der `LinkedList` auf. Du erinnerst dich bestimmt: Wenn das gewünschte Element gelöscht werden konnte, bildet es auch den Rückgabewert der Lösch-Funktion. War das Element nicht in der Liste, wird `null` zurück gegeben. Wir fragen den Ausgang der Lösch-Operation ab (Zeile 40–44) und geben entsprechend eine Meldung am Bildschirm aus.

```
33    public void delContact (String name)
34    {
35       Contact c = new Contact (name);
36
37       System.out.print ("Try to delete "+ name + ": ");
38       LinkedElement removed = representation.remove (c);
39
40       if (removed != null)
41          System.out.println ("succeeded");
42       else
43          System.out.println ("not in list");
44       System.out.println (this);
45    }
```

Auch diese Klasse hat eine `toString()`-Funktion. Wir benutzen wieder einen `StringBuffer`: Zuerst schreiben wir eine Titelzeile, fügen die String-Repräsentation der Liste ein und hängen dann eine Abschlusszeile an. Du siehst, wir benutzen die String-Repräsentation der Liste. Die Liste verwendet die String-Repräsentation der Elemente, d.h. der Kontakte!

```
48    public String toString ()
49    {
50       StringBuffer result = new StringBuffer ();
51
52       result.append ("--- My Contact List ------\n");
53       result.append (representation.toString ());
54       result.append ("-------------------------\n");
55
56       return result.toString ();
57    }
```

4.9 Die Klasse Main

Die Klasse `Main` ist die Applikation für die Kontaktliste. Sehen wir uns zunächst die `main()`-Funktion an (Zeilen 28–35). Wir sehen eine lokale Variable `contacts` vom Typ `ContactList`. Ein neues Objekt wird initialisiert, dann fügen wir einige Kontakte in die Kontaktliste ein (Zeile 31). Wir geben die Liste auf dem Bildschirm aus (Zeile 32), löschen einige Kontakte (Zeile 33) und geben wieder die Liste aus (Zeile 34). Wir schreiben nur `contacts` als Argument für `System.out.println`. Hier wird automatisch die Funktion `toString()` für `contacts` aufgerufen. Du musst es also nicht extra hinschreiben.

```
28    public static void main(String[] args)
29    {
30       ContactList contacts = new ContactList ();
31       addSomeContacts (contacts);
32       System.out.println (contacts);
33       deleteSomeContacts (contacts);
34       System.out.println (contacts);
35    }
```

Die zweite Methode, die von `main()` aufgerufen wird, heisst `addSomeContacts()`. Du kannst sie in den Zeilen 11–17 finden. In dieser Methode rufen wir einige Male eine Methode `addContact()` auf. An den Parametern kannst du erkennen, dass wir jeweils einen Namen, eine Telefonnummer und eine E-Mail-Adresse übergeben.

```
11    private static void addSomeContacts (ContactList contacts)
12    {
13       contacts.addContact ("Luca", "19 373", "luca@mail.com");
14       contacts.addContact ("Lydia", "23 101", "lydia@mail.com");
15       contacts.addContact ("Lia", "33 098", "lia@mail.com");
16       contacts.addContact ("Tino", "99 762", "tino@mail.com");
17    }
```

In der `main`-Funktion werden nun einige Kontakte aus der Kontaktliste gelöscht. Den dazugehörigen Befehl `deleteSomeContacts()` findest du in den Zeilen 19–26. Hier wird einige Male eine Operation `delContact()` der Kontaktliste aufgerufen. Als Parameter steht jeweils ein Name, d.h. ein `String`.

```
19    private static void deleteSomeContacts (ContactList contacts)
20    {
21       contacts.delContact ("Tino");
22       contacts.delContact ("Lydia");
23       contacts.delContact ("Matteo"); /* nicht gespeichert */
24       contacts.delContact ("Susi");   /* nicht gespeichert */
25       contacts.delContact ("Lia");
26    }
```

4.10 Sortierte Listen

In diesem Kapitel haben wir verkettete Listen allgemein besprochen. In Kapitel 3 haben wir zwei Möglichkeiten entdeckt, wie man zu einer sortierten Liste kommt. Die erste ist, dass man eine nicht sortierte Liste sortiert. Die zweite ist, dass man Elemente sortiert in die Liste einfügt. Diese zweite Methode wollen wir jetzt erarbeiten. Den entsprechenden Code findest du im Verzeichnis `linkedlist/single-sorted`.

Du kannst dich erinnern: Um ein Element sortiert in eine Liste einzufügen, muss zuerst die richtige Position für dieses Element gefunden werden. Es werden alle zu kleinen Elemente übergangen, bis man ein Element erreicht, das grösser als das neue Element ist. Hier ist der richtige Platz zum Einfügen: An dieser Stelle kann das neue Element eingekettet werden. Vier Fälle müssen beim Einketten beachtet werden, denn das neue Element ist entweder

- das allererste Element in einer leeren Liste,
- als kleinste Element,
- als grösste Element oder
- irgendwo in der Mitte.

Welche Änderungen müssen wir an den bestehenden Klassen vornehmen? Es reicht nicht mehr, in der Klasse `LinkedElement` eine `isEqualTo()`-Funktion anzubieten. Wir müssen auch vergleichen können, ob ein Element kleiner als ein anderes ist. Deshalb erweitern wir die Klasse `LinkedElement` mit einer abstrakten Methode `isLessThan()`. Diese Methode ist aus dem gleichen Grund abstrakt wie die Methode `isEqualTo()`: Diese Klasse weiss noch nicht, wie die tatsächlichen Elemente aussehen werden und überlässt deshalb die Implementierung ihren Subklassen.

```
32    /* Ist 'other' kleiner? */
33    public abstract boolean isLessThan (LinkedElement other);
```

Nachdem wir die Klasse `LinkedElement` verändert haben, müssen wir auch die Klasse `Contact` anpassen und die abstrakte Methode `isLessThan` implementieren. Wie bei der Funktion `isEqualTo()` müssen die beiden Objekte zumindest den gleichen Typ haben, um sie vergleichen zu können. Laut der Signatur kann `other` eine beliebige Subklasse von `LinkedElement` sein. Deshalb fragen wir, ob `other` vom Typ `Contact` ist (Zeile 72).

```
69    /* Ist dieser Kontakt alphabetisch kleiner als 'other'?  */
70    public boolean isLessThan (LinkedElement other)
71    {
72       if (other instanceof Contact) {
73          String name2 = ((Contact)other).getName ();
74          return (name.compareTo (name2) < 0);
75       } else
76          return false;
77    }
```

Hat `other` einen anderen Typ, geben wir `false` zurück (Zeile 76). Bei gleichem Typ können wir `other` in den Typ `Contact` umwandeln (Zeile 73) und so die Methoden von `Contact` benutzen. Wir speichern den Namen des anderen Kontakts in einer lokalen Variable. Wie schon bei `isEqualTo()` verwenden wir die Funktion `compareTo` der Klasse `String`, um diese beiden Strings zu vergleichen (Zeile 74). Zu Erinnerung: Ist der Argument-String alphabetisch grösser, ist das Ergebnis dieses Vergleichs negativ.

Die Klasse `LinkedList` müssen wir auch verändern: Der Befehl `add` darf nicht wie bisher die Elemente vorne einketten, sondern muss sie an der richtigen Position sortiert einfügen. Die Klasse `ContactList` sowie die Klasse `Main` müssen nicht verändert werden.

4.11 Zusammenfassung

In diesem Kapitel haben wir die Klassen besprochen, die notwendig sind, um eine verkettete Liste zu entwickeln und verwenden. Die abstrakte Klasse `LinkedElement` legt Operationen der Listenelemente fest. Die Klasse `LinkedList` implementiert alle Listenoperationen. Um die Liste auch zu verwenden, haben wir eine Klasse `ContactList` geschrieben, die Kontakte verwaltet. Die Klasse `Contact` modelliert das Listenelement. Die Klasse `Main` ist die Applikation, die die Kontaktliste verwendet. Wir haben gelernt, wie man eine sortierte Liste programmieren kann.

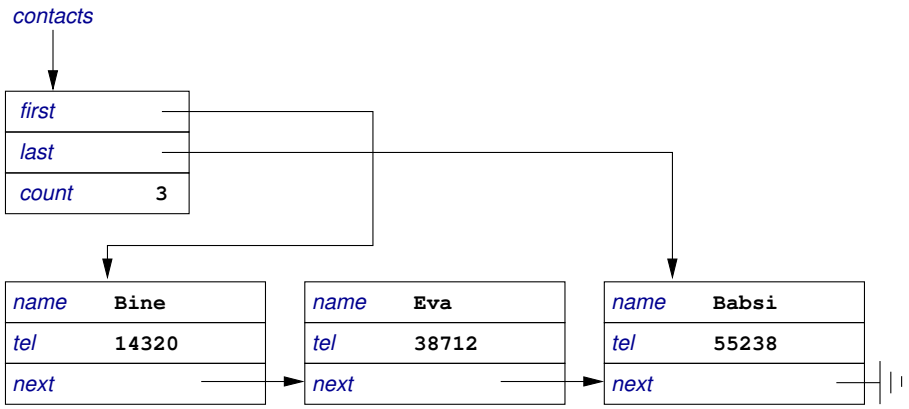
4.12 Lösungen zu den Aufgaben (Wissenssicherung)

Lösung 4.1:

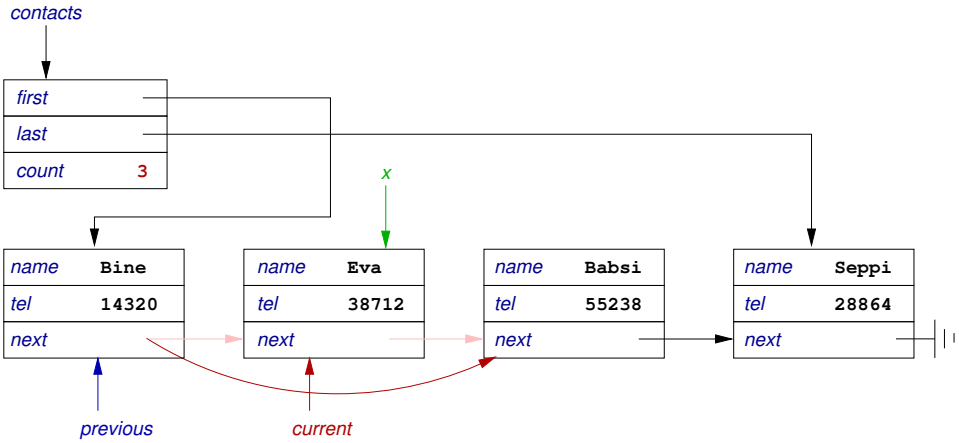
Die Liste ist leer, wenn

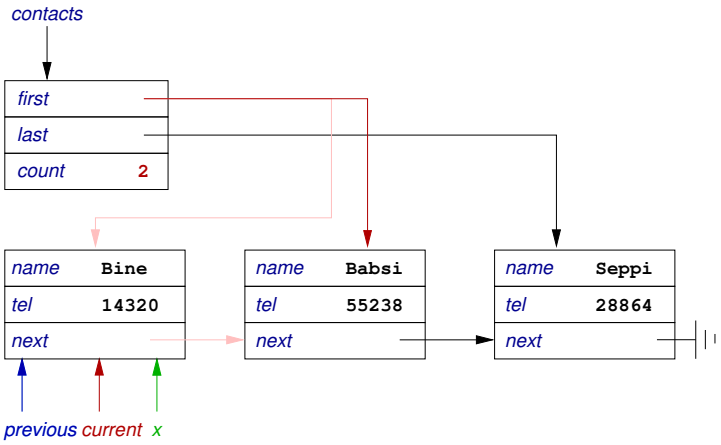
- `last` auf `null` zeigt oder
- `count` gleich `0` ist.

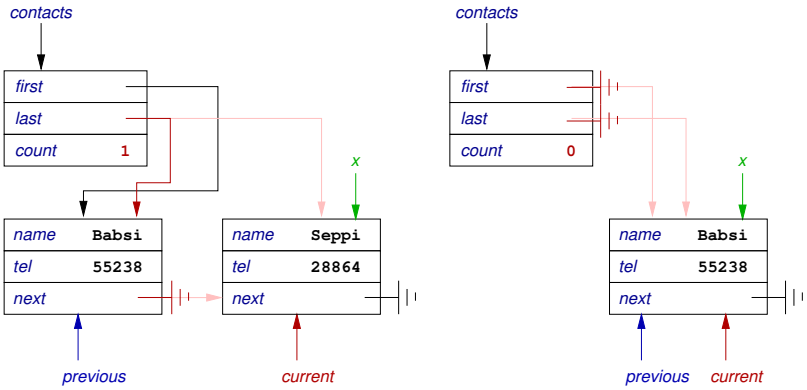
Lösung 4.2:



Lösung 4.3:


contacts
first
last
count 3
x
name Bine
tel 14320
next
name Eva
tel 38712
next
name Babsi
tel 55238
next
name Seppi
tel 28864
next
previous
current


contacts
first
last
count 2
name Bine
tel 14320
next
name Babsi
tel 55238
next
name Seppi
tel 28864
next
previous current x


contacts
first
last
count 1
x
name Babsi
tel 55238
next
name Seppi
tel 28864
next
previous
current
contacts
first
last
count 0
x
name Babsi
tel 55238
next
previous current

4.13 Lernkontrolle

LK 4.1: append

Schreibe in der Klasse `LinkedList` einen Befehl `void append(ListElement x)`, der neue Elemente hinten anfügt. Modifiziere das Programm, damit diese Methode verwendet wird!

LK 4.2: Mehr Kontaktdaten

Wir möchten die Kontaktliste erweitern. Es soll nun auch die Adresse und das Alter für jeden Kontakt gespeichert werden. Ändere entsprechend die Klasse `Contact`! Welche anderen Klassen musst du auch anpassen?

LK 4.3: Sortierte Liste

Programmiere die Methode `add (LinkedElement x)` für die Klasse `LinkedList` im Verzeichnis `single-sorted`: Die Elemente sollen sortiert eingefügt werden.

Für unsere Kontaktverwaltung ist es praktisch, wenn die Kontakte dem Namen nach alphabetisch sortiert sind — besonders wenn wir viele Kontakte eingetragen haben!

4.14 Lösungen zur Lernkontrolle

LK Lösung 4.1: append (K3)

```
/* Erweiterung zu LinkedList.java */

    /* Fuege Element 'x' hinten an. */
    public void append (LinkedElement x)
    {
        if (!isEmpty ()) {
            last.setNext (x);
            last = x;
        } else {
            first = x;
            last = x;
        }
        x.setNext (null);
        count ++;
    }
```

Damit `append()` verwendet wird, muss die Klasse `ContactList` angepasst werden.

LK Lösung 4.2: Mehr Kontaktdaten (K3)

```
/* Erweiterung zu Contact.java */

public class Contact extends LinkedElement
{
/* ..... */
    private String address;  /* Adresse */
    private String age;      /* Alter */

/* ..... */

    /* Adresse */
    public final String getAddress ()
    {
        return address;
    }

    /* Alter */
    public final String getAge ()
    {
        return age;
    }

```

```
23 /* ..... */
24
25     /* Setze die Adresse auf 'addr'. */
26     public void setAddress (String addr)
27     {
28         this.address = addr;
29     }
30
31     /* Setze das Alter auf 'alter'. */
32     public void setAge (String alter)
33     {
34         this.age = alter;
35     }
36
37 /* ..... */
38
39     /* String-Repräsentation */
40     public String toString ()
41     {
42         StringBuffer result = new StringBuffer ();
43         result.append (name);
44         result.append (": ");
45         result.append (phone);
46         result.append (" ");
47         result.append (email);
48         result.append (" ");
49         result.append (address);
50         result.append (" ");
51         result.append (age);
52         return result.toString ();
53     }
54
55 }
```

Die Klasse `ContactList` muss angepasst werden, damit die zusätzlichen Daten erfasst werden.

LK Lösung 4.3: Sortierte Liste (K3)

```
/* single-sorted/LinkedList.java */

    /* Füge Element 'x' sortiert in diese Liste ein. */
    public void add (LinkedElement x)
    {
        LinkedElement current = null;
        LinkedElement previous = null;

        if (isEmpty ()) {
            first = x;
            last = x;
         x.setNext (null);
        } else {

            /* (1) Finde den richtigen Platz */
             current = first;
             previous = first;

             while ((current != null) && (current.lessThan (x)))
                 {
                 previous = current;
                 current = current.getNext ();
             }

             /* (2) Füge das Element ein */
             if (first == current) {
             /* x ist kleinstes Element */
                 x.setNext (current);
                 first = x;
             } else if (previous == last) {
             /* x ist grösstes Element */
                 previous.setNext (x);
                 last = x;
             x.setNext (null);
             } else {
             /* x ist in der Mitte */
                 previous.setNext (x);
                 x.setNext (current);
             }
        }
        count ++;
    }
```

Die Klasse `ContactList` muss angepasst werden.

Kapitel 5

Doppelt verkettete Listen

5.1 Worum geht es?

In den vorigen zwei Kapiteln haben wir verkettete Listen studiert, deren Elemente jeweils mit dem nachfolgenden Element verkettet sind. Dadurch ist es unmöglich, beide Nachbarn anzusehen und auf den Vorgänger direkt zuzugreifen. Wir haben uns beim Iterieren Abhilfe geschafft, indem wir eine Referenz auf den Vorgänger `previous` mitgezogen haben.

In diesem Kapitel erweitern wir die verketteten Listen. Die Listenelemente speichern neu auch eine Referenz auf den Vorgänger. Listen, deren Elemente nur mit dem Nachfolger verkettet sind, nennt man auch *einfach verkettete Listen*. Listen, deren Elemente mit dem Vorgänger und dem Nachfolger verkettet sind, bezeichnet man als *doppelt verkettete Listen*. Wir werden die Unterschiede dieser zwei Strukturen herausarbeiten und eine doppelt verkettete Liste implementieren.

5.2 Lernziele

Nach diesem Kapitel sollst du folgendes können und wissen:

- Wie unterschieden sich einfach und doppelt verkettete Listen?
- Wie werden die Einfüge- und Löschmethoden programmiert?
- Lesen und Verändern von Programmtext für doppelt verkettete Listen.

5.3 Das Programm

Du findest alle notwendigen Dateien im Verzeichnis `linkedlist/double/`. In der Textdatei `linkedlist/README` findest du genaue Anweisungen für das Kompilieren und Ausführen des Programms.

5.4 Der Unterschied liegt im Listenelement

In Kapitel 2 haben wir schon von vorgehenden und nachfolgenden Elementen gesprochen. Bisher waren die Listenelemente nur mit dem Nachfolger über die Referenz `next` verbunden. Die Listenelemente der doppelt verketteten Liste sind zusätzlich mit dem Vorgänger verkettet.

In Abbildung 5.1(a) siehst du, wie eine doppelt verkettete Liste im Speicher dargestellt ist. Die Listenobjekte sind nicht nur über `next` mit dem Nachfolger verbunden, sondern auch über `previous` mit dem Vorgänger.

Zum Vergleich findest du in Abbildung 5.1(b) eine einfach verkettete Liste. Das Listenobjekt der einfach verketteten Liste hat die gleichen Felder wie das Listenobjekt der doppelt verketteten Liste. In der einfach verketteten Liste zeigt die Referenz `next` auf den Nachfolger. Die Listenelemente der doppelt verketteten Liste besitzen die Referenzen `next` und `previous`.

Einfach und doppelt verkettete Listen:

Einfach und doppelt verkettete Listen unterscheiden sich nur bei den Listenelementen:

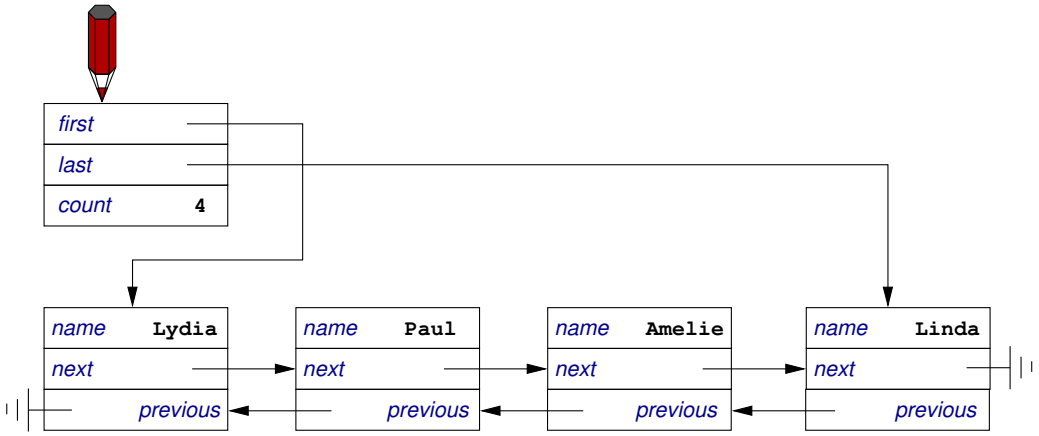
- Listenelemente der **einfach verketteten Liste** sind über die Referenz `next` mit dem Nachfolger verbunden.
- Listenelemente der **doppelt verketteten Liste** sind *zusätzlich* über die Referenz `previous` mit dem Vorgänger verkettet.

Bei der einfach verketteten Liste kann man in eine Richtung über die Liste iterieren: Ausgehend von `first` kann man Element für Element bis zu `last` gehen. Bei der doppelt verketteten Liste kann man beliebig vorgehen: von vorne nach hinten, von hinten nach vorne, einige Elemente vorwärts, dann zurück. Das Iterieren ist viel flexibler.

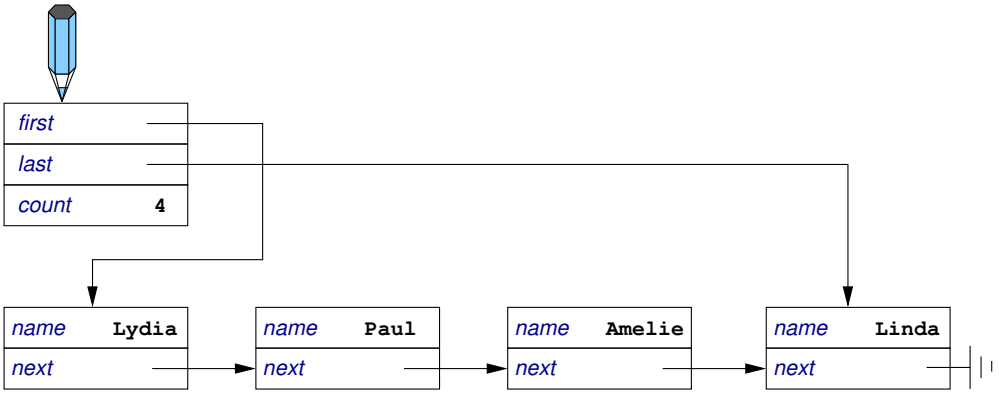
Aufgabe 5.1

Verwende wieder das Stiftmodell aus dem Kapitel 3. Führe die folgenden Anweisungen zuerst mit der doppelt verketteten Listen, danach mit der einfach verketteten Liste durch. Arbeite zuerst mit mit der Liste in Abbildung 5.1(a), danach mit der Liste in Abbildung 5.1(b). Welche Anweisung ist jeweils aufwändiger — zähle dafür die Operationen!

- Navigiere den Stift bis zu “Linda”!
- Navigiere von “Linda” bis zu “Amelie”!



(a) Doppelt verkettete Liste



(b) Einfach verkettete Liste

Abbildung 5.1: Doppelt und einfach verkettete Listen

Diese Flexibilität hat einen kleinen Preis. Jedes Element hat ein Attribut mehr. Das braucht mehr Speicherplatz. Die Einfüge-Operation ist ein wenig aufwändiger, denn wir müssen auch die Referenz `previous` beachten. Auch beim Löschen muss `previous` berücksichtigt werden. Allerdings wird das Iterieren der Liste einfacher und flexibler.

Aufgabe 5.2

Schreibe eine neue Methode `toReversedString()`, die über Liste von hinten nach vorne iteriert und dabei eine String-Repräsentation erzeugt.

5.5 Die abstrakte Klasse LinkedElement

Die abstrakte Klasse `LinkedElement` erhält noch ein Attribut für den Vorgänger. Das Attribut `previous` ist vom Typ `LinkedElement`. Zwei Operationen ermöglichen den Zugriff auf dieses Attribut: `getPrevious()` liefert den Vorgänger, `setPrevious()` legt einen neuen Vorgänger fest.

```
13 public abstract class LinkedElement
14 {
15    private LinkedElement next;      /* Nachfolger */
16    private LinkedElement previous;  /* Vorgänger */
17
18    /* Nachfolger  */
19    public final LinkedElement getNext ()
20    {
21       return next;
22    }
23
24    /* Vorgänger */
25    public final LinkedElement getPrevious ()
26    {
27       return previous;
28    }
29
30    /* Setze den Nachfolger auf 'next'.  */
31    public final void setNext (LinkedElement next)
32    {
33       this.next = next;
34    }
35
36    /* Setze den Vorgänger auf 'prev'.  */
37    public final void setPrevious (LinkedElement prev)
38    {
39       this.previous = prev;
40    }
41
42    /* Ist 'other' gleich? */
43    public abstract boolean isEqualTo (LinkedElement other);
44
45 }
```

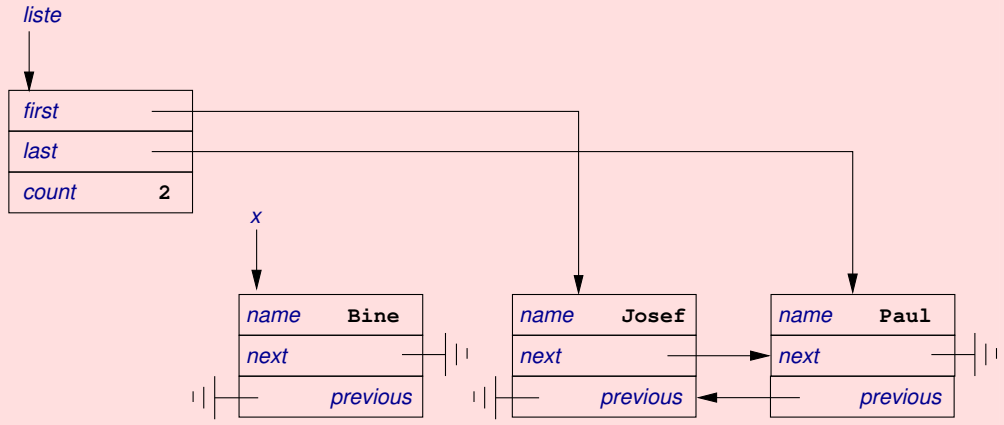
Die Klasse `Contact` muss nicht verändert werden. Sie erbt das neue Attribut sowie die neuen Methoden.

5.6 Die Klasse LinkedList

Wir müssen nur zwei Befehle etwas abändern, um die zusätzliche Referenz `previous` nachzuführen. Beim Einfügen eines Elements müssen wir auch die Vorgänger verbinden. Beim Entfernen müssen die Vorgänger auch richtig geändert werden.

Aufgabe 5.3

Betrachten wir zuerst das Einfügen. Füge das neue Element in die vorhandene Liste vorne ein. Nummeriere die Operationen, sollte eine Reihenfolge notwendig sein.



Aufgabe 5.4

Passe nun selbständig den Programmtext der Methode `add` in der Klasse `LinkedList` an!

Aufgabe 5.5

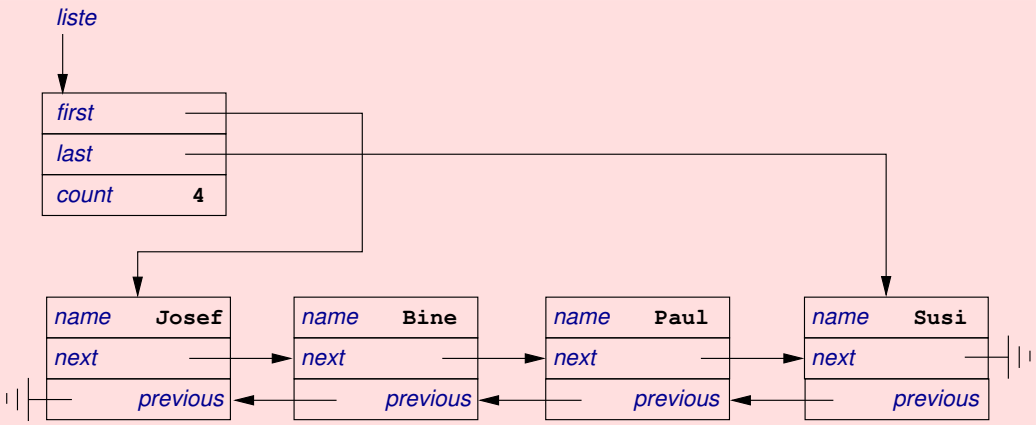
Schreibe eine Methode `append(LinkedElement x)`, die ein neues Element hinten anfügt!

Nun studieren wir das Entfernen eines Elements. Du erinnerst dich: Zuerst muss ein Element in der Liste gesucht werden. Danach kann es ausgekettet werden. Beim Suchen des Elements wird über die Liste iteriert. Bei der einfach verketteten Liste haben wir eine zusätzliche Referenz mitgeführt: `previous` hat auf den Vorgänger des aktuellen Elements `current` gezeigt. Das brauchen wir nun nicht mehr, denn wir können in der Liste vorwärts oder rückwärts navigieren, und wir können direkt auf den Vorgänger zugreifen. Beim Entfernen gibt es — wie auch bei der einfach verketteten Liste — vier Fälle zu beachten. Man kann entweder das einzige, das letzte, das erste oder ein Element in der Mitte ausketten.

```
50    /* Lösche Element 'x'.
51     * Wenn 'x' ausgekettet werden konnte, gib 'x' zurück, sonst
          'null'. */
52    public LinkedElement remove (LinkedElement x)
53    {
54       LinkedElement current = null;
55
56       if (!isEmpty ()) {
57          /* (1) Suche Element x */
58          current = first;
59          while ((current != null) && (!current.isEqualTo (x))) {
60              current = current.getNext ();
61          }
62
63          /* (2) Kette Element x aus */
64          if (current != null) {
65             if ((current == first) && (current == last)) {
66                /* Kette einziges Element aus: */
67                first = null;
68                last = null;
69             } else if (current == last) {
70                /* Kette letztes Element aus: */
71                last = last.getPrevious ();
72                last.setNext (null);
73             } else if (current == first) {
74                /* Kette erstes Element aus: */
75                first = first.getNext ();
76                first.setPrevious (null);
77             } else {
78                /* Kette Element in der Mitte aus: */
79                LinkedElement previous = current.getPrevious ();
80                LinkedElement next = current.getNext ();
81                previous.setNext (next);
82                next.setPrevious (previous);
83             }
84             count--;
85          }
86       }
87       return current;
88    }
```

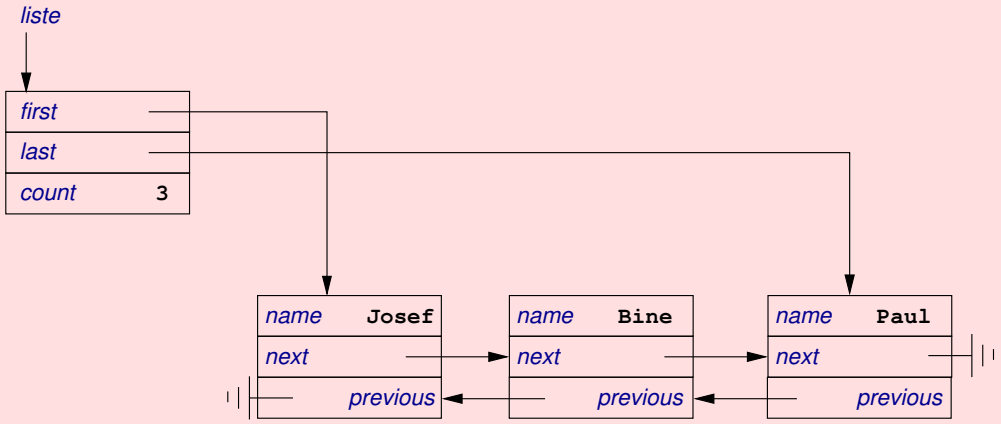
Aufgabe 5.6

Lösche die Elemente in der gegebenen Reihenfolge, indem du den Code von `remove` Zeile für Zeile durchspielst. Stelle jeden Zwischenschritt dar. Wenn du Referenzen veränderst, schreibe die entsprechende Zeilennummer hin. Lösche zunächst “Susi”!



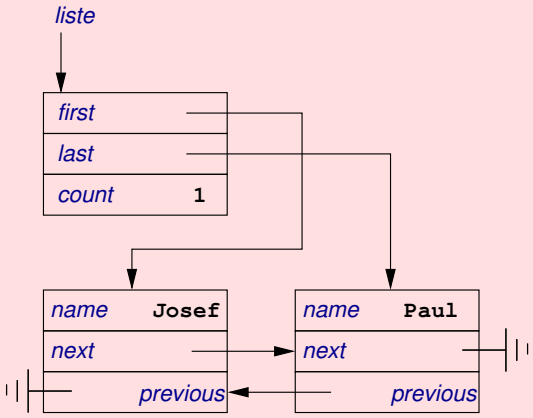
Aufgabe 5.7

Entferne “Bine” aus der Liste!



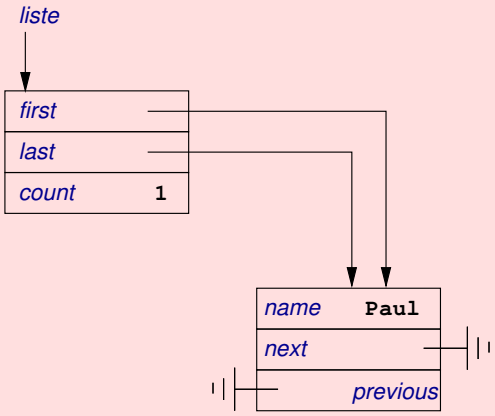
Aufgabe 5.8

Als nächstes lösche “Josef”:



Aufgabe 5.9

Zum Schluss lösche noch das letzte Element “Paul”!



5.7 Zusammenfassung

Wir haben den Unterschied zwischen einfach und doppelt verketteter Liste erarbeitet. Das Element hat nun einen Vorgänger und einen Nachfolger. Dadurch wird das Iterieren sehr flexibel. Allerdings müssen beim Einfügen und Löschen die Referenzen für den Vorgänger und Nachfolger nachgeführt werden.

5.8 Lösungen zu den Aufgaben (Wissenssicherung)

Lösung 5.1:

- Bis zu “Linda”: Bei beiden ist es 1 Schritt, wenn der Referenz `last` gefolgt wird. Bei beiden sind es 4 Schritte, wenn über die Liste komplett iteriert wird.
- Von “Linda” zu “Amelie”: 1 Schritt rückwärts bei der doppelt verketteten Liste, 4 Schritte bei der einfach verketteten Liste (der Stift muss zuerst zurück auf das Listenobjekt gesetzt werden, dann muss über die Liste neu iteriert werden.

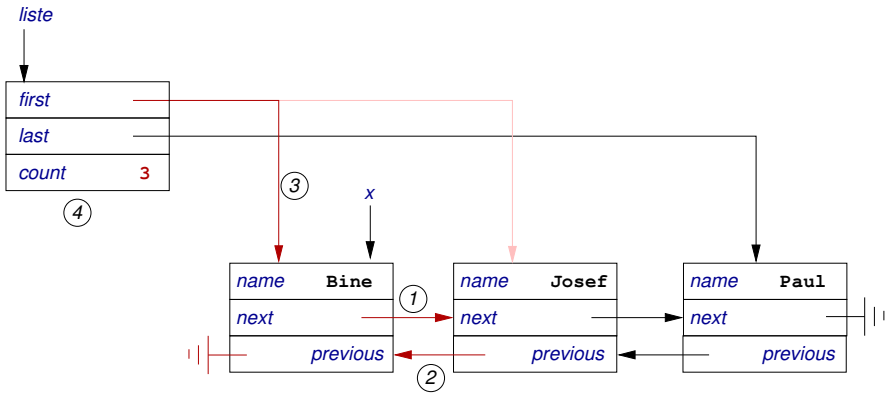
Lösung 5.2:

```
/* LinkedList.java */

   /* Umgekehrte String-Repräsentation */
   public String toReversedString ()
   {
      StringBuffer result = new StringBuffer ();

      for (LinkedElement current = last; current != null;
            current = current.getPrevious ()) {
         result.append (current.toString());
         result.append ("\n");
      }
      return result.toString ();
   }
```

Lösung 5.3:

Im Vergleich zur einfach verketten Liste gibt es eine Operation mehr: ① Zuerst setzen wir den Nachfolger des neuen Elements auf das erste Element. ② Danach wird der Vorgänger des ersten Elements auf das neue Element gesetzt. ③ `first` wird auf das neue Element gesetzt, und ④ die Anzahl der Elemente wird um eins erhöht.

Für die richtige Reihenfolge ist es möglich, die Operationen ① und ② zu tauschen. Jedoch muss das neue Element an die bisherigen Elemente angekettet sein, bevor `first` neu gesetzt wird.



Lösung 5.4:

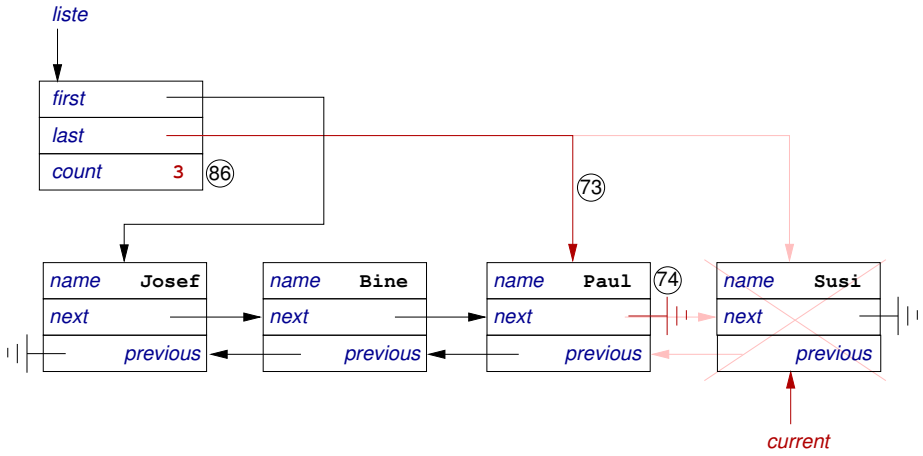
```
1  /* LinkedList.java */
2
3     /* Füge Element 'x' vorne hinzu. */
4     public void add (LinkedElement x)
5     {
6        if (!isEmpty ()) {
7           x.setNext (first);
8           x.setPrevious (null);
9           first.setPrevious (x);
10          first = x;
11       } else {
12          x.setNext (null);
13          x.setPrevious (null);
14          first = x;
15          last = x;
16       }
17       count ++;
18    }
```

Lösung 5.5:

```
1  /* LinkedList.java */
2
3     /* Füge Element 'x' hinten an. */
4     public void append (LinkedElement x)
5     {
6        if (!isEmpty ()) {
7           last.setNext (x);
8           x.setPrevious (last);
9           x.setNext (null);
10          last = x;
11       } else {
12          x.setPrevious (null);
13          x.setNext (null);
14          first = x;
15          last = x;
16       }
17       count ++;
18    }
```

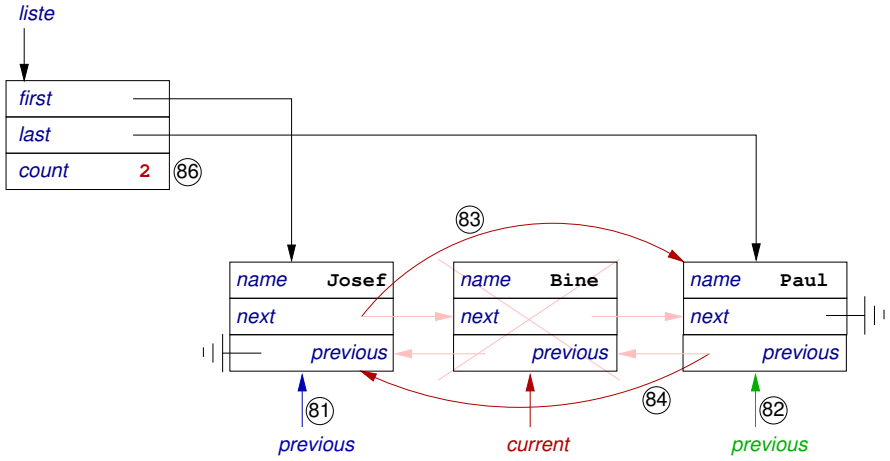
Lösung 5.6:

Das Löschen des *letzten* Elements funktioniert gleich wie bisher: `last` wird auf seinen Vorgänger `last.getPrevious()` gesetzt.



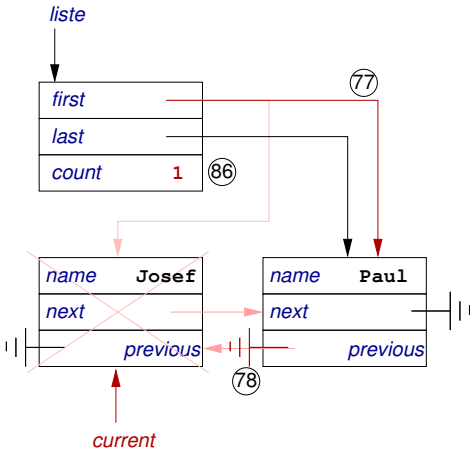
Lösung 5.7:

Befindet sich das zu löschende Element in der *Mitte* der Liste: Sowohl der Nachfolger des Vorgängers als auch der Vorgänger des Nachfolgers von `current` muss neu gesetzt werden.



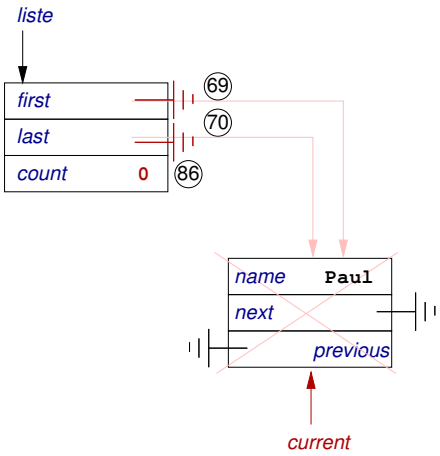
Lösung 5.8:

Wenn wir das *erste* Element löschen, muss `first` auf seinen Nachfolger `first.getNext()` rücken. Neu ist, dass wir den Vorgänger des neuen ersten Elements auf `null` schreiben müssen.



Lösung 5.9:

Löschen wir das *einzige* Element aus der Liste, ändert sich bei einer doppelt verketteten Liste nichts. Wie bisher werden `first` und `last` auf `null` gesetzt.



5.9 Lernkontrolle

LK 5.1: Vergleich

Vergleiche einfach und doppelt verkettete Listen bezüglich:

- Listenelement
- Aufwand beim Einfügen
- Aufwand beim Entfernen
- Iterieren

LK 5.2: Sortiertes Einfügen

Schreibe eine Methode `addSorted(LinkedElement x)` für die Klasse `LinkedList`: Die Elemente sollen sortiert einfügt werden. Welche Klassen musst du anpassen, damit diese Methode zum Einfügen verwendet wird?

5.10 Lösungen zur Lernkontrolle

LK Lösung 5.1: Vergleich (K4)

Das Listenelement bei der einfach verketteten Liste hat nur einen Nachfolger. Bei der doppelt verketteten Liste jedoch einen Vorgänger und einen Nachfolger.

Der Aufwand beim Einfügen vorne bzw. hinten in die Liste ist bei einfach und doppelt verketteter Liste ungefähr gleich. Fügt man das Element vorne ein, muss zusätzlich der Vorgänger des vormals ersten Elements auf das neue Element gesetzt werden. Fügt man hinten ein, muss der Vorgänger des neuen Elements auf das ursprünglich letzte Element gesetzt werden. Beim sortierten Einfügen muss zuerst über die Liste iteriert werden, um die richtige Position zu finden. Bei der einfach verketteten Liste muss eine Hilfsreferenz `previous` mitgeführt werden. Bei der doppelt verketteten Liste müssen Vorgängerreferenzen nachgeführt werden.

Der Aufwand beim Entfernen ist ähnlich. Bei der einfach verketteten Liste muss für die Suche des Elements noch eine zusätzliche Referenz als Vorgänger des aktuellen Elements `current` mitgezogen werden. Dieser Aufwand entfällt bei der doppelt verketteten Liste, denn man kann auch rückwärts gehen. Allerdings müssen bei der doppelt verketteten Liste auch die Vorgängerreferenzen nachgeführt werden.

Das Iterieren ist bei der einfach verketteten Liste nur in eine Richtung möglich — von vorne nach hinten. Hingegen bei der doppelt verketteten Liste kann man in beide Richtungen gehen.

LK Lösung 5.2: Sortiertes Einfügen (K3)

```
/* LinkedList.java */

   /* Füge Element 'x' sortiert hinzu. */
   public void addSorted (LinkedElement x)
   {
      LinkedElement current;

      if (isEmpty ()) {
         first = x;
         last = x;
         x.setNext (null);
         x.setPrevious (null);
      } else {
         /* (1) Finde den richtigen Platz: */
         current = first;
         while ((current != null) && (current.isLessThan (x))) {
            current = current.getNext ();
         }

         /* (2) Kette das Element ein: */
         if (first == current) {
            /* x ist das kleinste Element */
            x.setNext (first);
            first.setPrevious (x);
            x.setPrevious (null);
            first = x;
         } else if (current == null) {
            /* x ist das grösste Element */
            last.setNext (x);
            x.setPrevious (last);
            x.setNext (null);
            last = x;
         } else {
            /* x ist in der Mitte */
            LinkedElement previous = current.getPrevious ();
            previous.setNext (x);
            x.setPrevious (previous);
            x.setNext (current);
            current.setPrevious (x);
         }
      }
      count ++;
   }
```

REFERENZEN

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley, 1983.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 1990.

[3] Niklaus Wirth. *Algorithmen und Datenstrukturen*. Teubner, 5th edition, 1999.

[4] Mark Allen Weiss. *Data Structures and Problem Solving Using Java*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[5] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[6] László Böszörményi, Carsten Weich. *Programmieren mit Modula-3, Eine Einführung in stilvolle Programmierung*. Springer, 1995.

Kapiteltests

Kapiteltest 1

Aufgabe 1: Eigenschaften eines Kühlregals

Erkläre die Eigenschaften dieses Behälters: ein Kühlregal mit Milchprodukten (Milch, Joghurt, Rahm etc.) im Supermarkt. Berücksichtige die unterschiedlichen Produktgruppen sowie Ablaufdaten!

Aufgabe 2: Aufgaben eines Kühlregals

Erkläre die Aufgaben des obigen Behälters!

Aufgabe 3: Sortierte Behälter

Welcher der folgenden Behälter sind sortiert bzw. nicht sortiert? Gib gegebenenfalls ein mögliches Sortierkriterium an!

Behälter	nicht sortiert	sortiert	Sortierkriterium
Parkplatz			
Karteikasten			
Einkaufstasche			
Bücherbrett			

Aufgabe 4: Unterschiede zwischen Behältern

Vergleiche ein Buch (ein Roman) mit dem Teletext. Was sind die Gemeinsamkeiten? Was sind die Unterschiede?

	Buch (Roman)	Teletext	gemeinsam
Aufbau			
Lesen			
Grösse			

Kapiteltest 2

Aufgabe 1: Organisationsform

Rechnungen werden laufend nummeriert und in einem Ordner abgeheftet. Sind die Rechnungen linear oder zufällig organisiert? Welche Eigenschaft hat diese Organisationsform?

Aufgabe 2: Array

Kreuze die richtige(n) Aussagen an!

- ☐ Das Array bietet direkten und sequentiellen Zugriff.
- ☐ Ein Array kann beliebig viele Elemente speichern.
- ☐ Nur auf das erste Element im Array kann man direkt zugreifen.
- ☐ Im Speicher ist ein Array ein zusammenhängender Block von Elementen.

Aufgabe 3: Liste

Kreuze die richtige(n) Aussagen an!

- ☐ Die Grösse der Liste wird schon beim Programmieren festgelegt.
- ☐ Auf die Elemente einer Liste wird sequentiell zugegriffen.
- ☐ Das letzte Listenelement hat nur einen Nachfolger; das erste Listenelement hat nur einen Vorgänger.
- ☐ Listenelemente werden über Referenzen im Speicher verbunden.

Aufgabe 4: Operationen und Aufwand

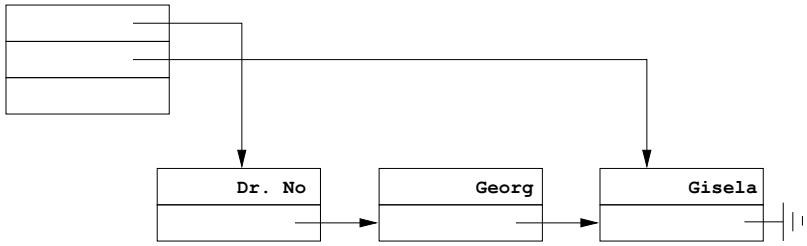
Kreuze die richtige(n) Aussagen an!

- ☐ Elemente in Arrays und Listen sind zufällig angeordnet. Deswegen können sie sortiert oder nicht sortiert sein.
- ☐ Soll ein Element aus der Liste gelöscht werden, reicht eine einzige Operation.
- ☐ Kennt man den Index eines Elements, kann man in einer einzigen Operation auf ein Element im Array zugreifen.
- ☐ Beim sortierten Einfügen in eine Liste müssen keine Elemente verschoben werden.

Kapiteltest 3

Aufgabe 1: Listenattribute

Diese Liste zeigt eine Liste von Personen. Ergänze:



Aufgabe 2: Einfügen

Füge “Elmar” in die nach Namen sortierte Liste von Aufgabe 1 ein. Zeichne zunächst das Element. Benutze das Stiftmodell und gib die richtige Reihenfolge der Operationen an!

Aufgabe 3: Löschen

In einem Stack wird jeweils das erste Element gelöscht — über die Liste muss also nicht iteriert werden. Matteo nennt die Schritte für diesen Algorithmus. Leider hat er einen kleinen Fehler gemacht. Korrigiere den Algorithmus!

1. Zuerst muss überprüft werden, ob die Liste leer ist. Nur wenn Elemente in der Liste gespeichert sind, werden die folgenden Schritte ausgeführt:
2. `first` wird auf den Nachfolger von `first` gesetzt.
3. Die Anzahl der Elemente `count` wird um `1` erniedrigt.

Aufgabe 4: Sortierte Listen

Welche der folgenden Aussagen ist/sind richtig?

Eine sortierte Liste kann man erhalten, indem man:

- ☐ Elemente nur sortiert einfügt.
- ☐ die Elemente in ein Array kopiert.
- ☐ eine nicht sortierte Liste sortiert.
- ☐ `first` und `last` vertauscht.

Kapiteltest 4

Aufgabe 1: Leere Liste

Welche der folgenden Zeilen überprüft, ob eine Liste leer ist?

- ☐ `(first == last)`
- ☐ `(first == null)`
- ☐ `(count == 0)`
- ☐ `(last == null)`

Aufgabe 2: Programmtext verstehen

Was macht der folgende Programmtext?

```
1    public void tuetwas()
2    {
3        LinkedElement previous, current, next;
4
5        if (!isEmpty ()) {
6            previous = first;
7            current = first.getNext ();
8            first.setNext (null);
9
10           while (current != null) {
11
12               next = current.getNext ();
13               current.setNext (previous);
14               previous = current;
15               current = next;
16           }
17
18           last = first;
19           first = previous;
20       }
21   }
```

Aufgabe 3: Contact

Wir möchten bei jedem Kontakt das Geburtsdatum und die URL der Homepage speichern.

- Welche Attribute bzw. Methoden musst du zur Klasse `Contact` hinzufügen? Gib die Signaturen an — die Methoden selbst musst du nicht schreiben.
- Welche Methoden der Klasse `Contact` musst du ändern?
- Welche anderen Klasse musst du ändern, damit du diese Information nutzen kannst?

Aufgabe 4: Löschen

Wie kann das Löschen des letzten Elements aus einer mehrelementigen Liste erfolgen?

- ☐ Man setzt `count` auf `count-1`.
- ☐ Man setzt `last` auf `null`.
- ☐ Man setzt `last` auf das vorletzte Element. Dieses muss vom Anfang der Liste her gefunden werden.
- ☐ Man dreht alle Nachfolger auf den Vorgänger zeigen und löscht das erste Element.

Kapiteltest 5

Aufgabe 1: Einfach und doppelt verkettete Listen

Welche der folgenden Aussagen ist/sind korrekt?

- ☐ Die Elemente einer doppelt verketteten Liste haben ein zusätzliches Attribut `previous`.
- ☐ Über eine doppelt verketteten Liste kann nur rückwärts iteriert werden.
- ☐ Das Löschen des letzten Elements ist weniger aufwändig in einer einfach verketteten Liste als in einer doppelt verketteten Liste.
- ☐ Eine doppelt verkettete Liste benötigt doppelt so viel Speicherplatz wie eine einfach verkettete Liste.

Aufgabe 2: Programmtext verstehen und anpassen

Die Methode `delLast()` löscht das letzte Element einer Liste. Was ist falsch? Korrigiere die Methode!

```
1    public void delLast ()
2    {
3        last = last.getPrevious ();
4        if (count == 1)
5            first = null;
6        else
7            last.setNext (null);
8        count = count – 1;
9    }
```

Aufgabe 3: Programmtext verstehen

Was macht die folgende Methode? Zeichne eine Skizze!

```
1   public void doSomething (LinkedList other}
2   {
3       LinkedElement p1 = this.getFirst ();
4       LinkedElement p2 = other.getFirst ();
5       LinkedElement current = null;
6
7       this.count = 0;
8       if ( p1 != null ) {
9           current = p1;
10          p1 = p1.getNext ();
11          this.count++;
12      } else if ( p2 != null) {
13          current = p2;
14          p2 = p2.getNext ();
15          this.count++;
16      }
17      this.first = current;
18
19      while ((p1 != null) || (p2 != null)) {
20
21          if (p2 != null) {
22              current.setNext (p2);
23              p2.setPrevious (current);
24              this.count++;
25              current = p2;
26              p2 = p2.getNext ();
27          }
28          if (p1 != null) {
29              current.setNext (p1);
30              p1.setPrevious (current);
31              this.count++;
32              current = p1;
33              p1 = p1.getNext ();
34          }
35      }
36      this.last = current;
37  }
```

Aufgabe 4: Programmtext verstehen und anpassen

Dieser Befehl dreht eine Liste um. Funktioniert dies auch für doppelt verkettete Listen? Wenn nein, füge die fehlenden Instruktionen hinzu!

```
1    public void reverse ()
2    {
3        LinkedElement previous, current, next;
4
5        if (!isEmpty ()) {
6            previous = first;
7            current = first.getNext ();
8            first.setNext (null);
9
10           while (current != null) {
11
12               next = current.getNext ();
13               current.setNext (previous);
14               previous = current;
15               current = next;
16           }
17
18           last = first;
19           first = previous;
20       }
21   }
```

Lösungen der Kapiteltests

Lösungen zum Kapiteltest 1

Aufgabe 1: Eigenschaften der Behälter (K3)

Das Kühlregal bewahrt die verschiedenen Milchprodukte auf. Beispielsweise stehen alle Joghurtarten zusammen, sodass man als Kunde mit einem Blick die Joghurts findet, um danach seine Geschmacksrichtungen aussuchen kann. Ebenso sind die Milcharten wie z.B. Bio- und Nicht-Bio-Milch, entrahmte Milch angeordnet. Die Produktgruppen sind so angeordnet, sodass man als Kunde einfach zugreifen und aussuchen kann. Bei einem einzelnem Produkt (z.B. Bio-Vollmilch) findet man oft die Packungen mit kürzerer Ablaufzeit vorne, die mit längerer Ablaufzeit weiter hinten.

Aufgabe 2: Operationen der Container (K3)

Die Produkte werden vom Regalbetreuter in das Regal einsortiert: Die neue Lieferung mit dem längeren Ablaufdatum wird weiter hinten einsortiert. Dies ist umständlich und aufwändig, denn die älteren Produkte müssen nach vorne genommen werden. (Aber die Verkäufer möchten natürlich zuerst die älteren Produkte verkaufen.) Der Kunde wählt ein Produkt aus (sieht es an) und entfernt dieses aus dem Regal. Abgelaufene Produkte werden in der Regel vom Regalbetreuer entfernt.

Aufgabe 3: Sortierte Behälter (K3)

Behälter	nicht sortiert	sortiert	Sortierkriterium
Parkplatz	X		
Karteikasten		X	z.B. alphabetisch oder nach Datum
Einkaufstasche	X		
Bücherbrett		X	z.B. nach Autor oder Genre

Aufgabe 4: Unterschiede zwischen Behältern (K3)

	Buch (Roman)	Teletext	gemeinsam
Aufbau			Seiten
Lesen	von vorne bis hinten (sequentiell)	Lesen von bestimmten Seiten (dynamisch)	—
Grösse	fest Inhalt festgelegt	dynamisch Inhalt ändert sich	— —

Lösungen zum Kapiteltest 2

Aufgabe 1: Organisationsform (K2)

Die Rechnungen sind linear organisiert (= Organisationsform), denn jede Rechnung hat einen Nachbarn (= Eigenschaft).

Aufgabe 2: Array (K1)

- ☒ Das Array bietet direkten und sequentiellen Zugriff.
- ☐ Ein Array kann beliebig viele Elemente speichern.
- ☐ Nur auf das erste Element im Array kann man direkt zugreifen.
- ☒ Im Speicher ist ein Array ein zusammenhängender Block von Elementen.

Aufgabe 3: Liste (K1)

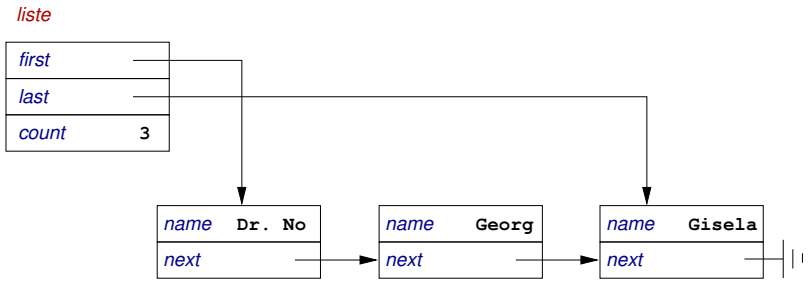
- ☐ Die Grösse der Liste wird schon beim Programmieren festgelegt.
- ☒ Auf die Elemente einer Liste wird sequentiell zugegriffen.
- ☐ Das letzte Listenelement hat nur einen Nachfolger; das erste Listenelement hat nur einen Vorgänger.
- ☒ Listenelemente werden über Referenzen im Speicher verbunden.

Aufgabe 4: Operationen und Aufwand (K2)

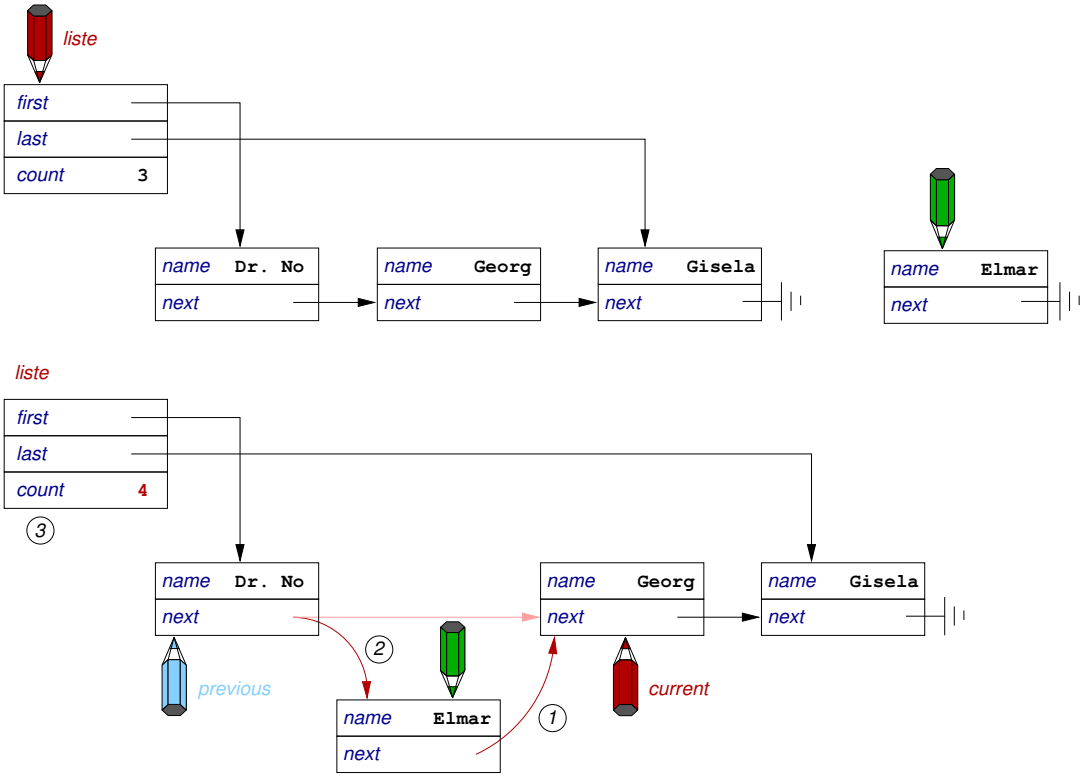
- ☐ Elemente in Arrays und Listen sind zufällig angeordnet. Deswegen können sie sortiert oder nicht sortiert sein.
- ☐ Soll ein Element aus der Liste gelöscht werden, reicht eine einzige Operation.
- ☒ Kennt man den Index eines Elements, kann man in einer einzigen Operation auf ein Element im Array zugreifen.
- ☒ Beim sortierten Einfügen in eine Liste müssen keine Elemente verschoben werden.

Lösungen zum Kapiteltest 3

Aufgabe 1: Listenattribute (K1)



Aufgabe 2: Einfügen (K3)



Aufgabe 3: Löschen (K4)

Matteo hat einen Spezialfall vergessen: Was passiert, wenn das einzige Element gelöscht wird? Das sind die richtigen Schritte.

1. Zuerst muss überprüft werden, ob die Liste leer ist. Nur wenn Elemente in der Liste gespeichert sind, werden die folgenden Schritte ausgeführt:
2. `first` wird auf den Nachfolger von `first` gesetzt.
3. Wird das einzige Element der Liste gelöscht (`count == 1`), muss auch `last` auf `null` gesetzt werden. `first` wird schon durch die vorige Anweisung auf `null` gesetzt.
4. Die Anzahl der Elemente `count` wird um `1` erniedrigt.

Aufgabe 4: Sortierte Listen (K2)

Eine sortierte Liste kann man erhalten, indem man:

☒ Elemente nur sortiert einfügt.
☐ die Elemente in ein Array kopiert.
☒ eine nicht sortierte Liste sortiert.
☐ `first` und `last` vertauscht.

Lösungen zum Kapiteltest 4

Aufgabe 1: Leere Liste (K2)

Welche der folgenden Zeilen überprüft, ob eine Liste leer ist?

☐ `(first == last)`
☒ `(first == null)`
☒ `(count == 0)`
☒ `(last == null)`

Aufgabe 2: Programmtext verstehen (K4)

Diese Methode könnte `reverse` oder `mirror` heissen: sie dreht die Liste um.

Aufgabe 3: Contact (K3)

- Attribute:

```
1    private String birthday;    /* Date birthday; */
2    private String homepage;    /* URL homepage; */
```

- Methoden zum Lesen und Schreiben der Attribute:

```
1    public String getBirthday () { /* ... */ }
2    public String getURL () { /* ... */ }
3
4    public void setBirthday (date: String)  { /* ... */ }
5    public void setURL (url: String) { /* ... */ }
```

- Die Methode `toString()` sollte geändert werden, damit auch die neuen Attribute in der String-Darstellung dabei sind.
- Die Klasse `ContactList` verwendet die Klasse `Contact`. Hier müssen die Befehle zum Speichern einer Person abgeändert werden.

```
1     private static void addContact (String name, String tel,
2          String mail, String date, String url)
3    {
4         Contact c = new Contact (name);
5         c.setPhone (tel);
6         c.setEmail (mail);
7         c.setBirthday (date);
8         c.setURL (url);
9         contactList.add (c);
10        System.out.println ("added " + c);
11   }
```

- Die Klasse `Main` muss auch angepasst werden:

```
1     private static void addSomeContacts ()
2     {
3         addContact ("Luca", "19 373", "luca@mail.com",
4             "20.8.1975",  "http://www.luca.com");
5         /* ... */
6     }
```

Aufgabe 4: Löschen (K2)

- ☐ Man setzt `count` auf `count-1`.
- ☐ Man setzt `last` auf `null`.
- ☒ Man setzt `last` auf das vorletzte Element. Dieses muss vom Anfang der Liste her gefunden werden.
- ☐ Man dreht alle Nachfolger auf den Vorgänger zeigen und löscht das erste Element.

Lösungen zum Kapiteltest 5

Aufgabe 1: Einfach und doppelt verkettete Listen (K2)

- ☒ Die Elemente einer doppelt verketteten Liste haben ein zusätzliches Attribut `previous`.
- ☐ Über eine doppelt verketteten Liste kann nur rückwärts iteriert werden.
- ☐ Das Löschen des letzten Elements ist weniger aufwändig in einer einfach verketteten Liste als in einer doppelt verketteten Liste.
- ☐ Eine doppelt verkettete Liste benötigt doppelt so viel Speicherplatz wie eine einfach verkettete Liste.

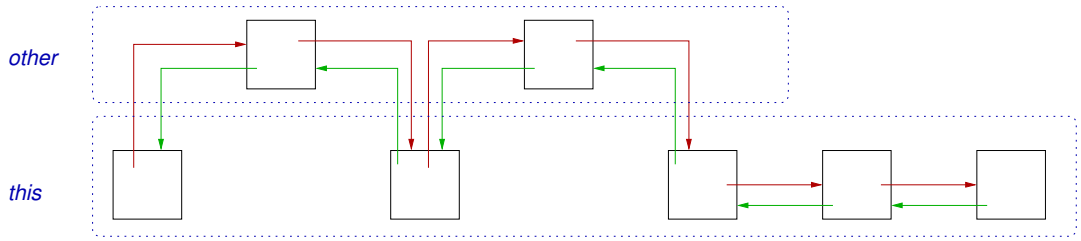
Aufgabe 2: Programmtext verstehen und anpassen (K4)

Die Methode funktioniert nicht, wenn die Liste leer ist!

```
1   public void delLast ()
2   {
3       if (!isEmpty ()) {
4           last = last.getPrevious ();
5           if (count == 1)
6               first = null;
7           else
8               last.setNext (null);
9           count = count - 1;
10      }
11  }
```

Aufgabe 3: Programmtext verstehen (K4)

Die Methode könnte `merge` heissen. Sie nimmt eine zweite Liste und fügt beide reissverschlussartig zusammen.



Aufgabe 4: Programmtext verstehen und anpassen (K4)

```
1   /* Drehe die Liste um. */
2   public void reverse()
3   {
4       LinkedElement previous;
5       LinkedElement current;
6       LinkedElement next;
7
8       if (!isEmpty ()) {
9           previous = first;
10          current = first.getNext ();
11          first.setNext (null);
12
13          while (current != null) {
14              next = current.getNext ();
15              current.setNext (previous);
16              /* Neu: Verkette auch previous */
17              previous.setPrevious (current);
18              previous = current;
19              current = next;
20          }
21
22          last = first;
23          first = previous;
24          /* Neu: Vorgänger von first muss null sein */
25          first.setPrevious (null);
26      }
27  }
```