

Kapitel 4

Implementierung von Listen



Übersicht

In diesem Kapitel wollen wir beginnen, mit dem Computer zu arbeiten. Wir wollen uns mit der Implementierung von Listen beschäftigen. Wir wollen also nun die Theorie aus Kapitel 2 in Java „übersetzen“.

Damit die Implementierung nicht zu abstrakt wird, wollen wir uns als konkretes Beispiel eine E-Mailliste vornehmen. Dieses Beispiel kannst Du später, mit nur ein wenig Aufwand auf eine Liste mit anderem Inhalt verallgemeinern.



Lernziele

- Nach der Bearbeitung dieses Kapitels bist Du in der Lage, allgemeine Listen zu implementieren.
- Siehst und erkennst Du die Sonderfälle, die bei der Implementierung von Listen zu beachten sind.
- Du wirst in der Lage sein, Quellcode in Zusammenhang mit Listen zu deuten und zu verstehen, sowie auf Korrektheit zu überprüfen.

Für dieses Kapitel haben wir den Javaeditor BlueJ benutzt.



Aufgabe 4.1:

Starte Deinen Java-Editor und erstelle ein neues Projekt für die Bearbeitung der Aufgaben in diesem Kapitel. In der Mediothek findest Du weitere Hinweise dazu.

Gib dem Projekt einen aussagekräftigen Namen, so dass Du das Projekt später wieder findest, für den Fall, dass Du einmal auf eine implementierte Liste zurückgreifen willst.



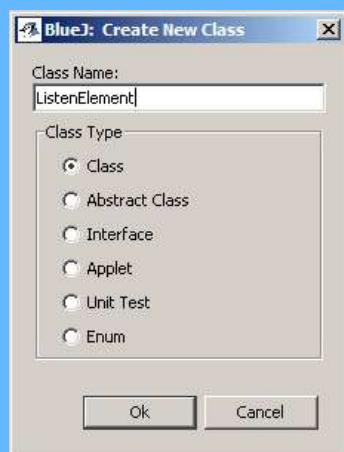
4.1 Die Klasse *ListenElement*- Attribute

Bevor wir mit der Implementierung der eigentlich Liste beginnen, wollen wir zuerst die einzelnen Listenelemente implementieren. Wie Du aus Kapitel 2 noch weißt, besteht ein Listenelement aus einem Datenobjekt und einem Zeiger auf das nächste Listenelement.



Aufgabe 4.2:

Erstelle eine neue Klasse *ListenElement*.



Wie wir in der Übersicht schon erwähnt haben, wollen wir in diesem Kapitel eine mögliche E-Mailliste implementieren. Damit wäre also der Dateninhalt eines Listenelements, wie Du sicherlich schon erraten hast, eine E-Mail. Damit das Beispiel allerdings nicht zu sehr ausfertig, wollen wir eine E-Mail auf die folgenden drei Eckpunkte reduzieren.

- Absender
- Betreff

- Inhalt (nur als reiner Text betrachtet, keine möglichen Dateianhänge)



Aufgabe 4.3:

- Überlege Dir geeignete Datentypen für gerade aufgelisteten drei Eckpunkte einer E-Mail.
- Danach erstellst Du bitte eine Klasse *EMail* mit diesen drei Attributen und einer Methode *drucke()*, die alle relevanten Informationen einer E-Mail auf der Konsole bzw. dem Terminal Window ausgibt. Weiter sollst Du für die Klasse einen Konstruktor implementieren, der die folgenden Werte *String Absender*, *String Betreff*, *String Text* übergeben bekommt und anhand dieser die Attribute der Klasse *EEmail* gesetzt werden.
- Schreibe noch für die drei Attribute eine Methode *getAttributeName()*, die den jeweiligen Wert des Attributs zurückgibt, wobei Attributname eines der drei Attribute bezeichnet.

Wenn Du die letzte Aufgabe erfolgreich gemeistert hast, dann können wir uns wieder der Klasse *ListenElement* zuwenden und ihre Attribute definieren. Aus Kapitel 2 wissen wir, dass ein Listenelement aus einem Objekt und einem Zeiger, der auf das nächste Element zeigen soll, besteht. Als Objekt können wir in unserem Falle ein Objekt der Klasse *EEmail* verwenden. Der Zeiger auf das nächste Element kann durch ein Objekt vom Typ *ListenElement* realisiert werden. Wir stellen den Zeiger auf das nächste Element also durch eine Variable dar, in der die Adresse des nächsten Listenelements gespeichert ist, da eine Variable (Objektvariable) in Java nicht das Objekt selbst, sondern die Adresse speichert, wo das Objekt im Speicher abgelegt ist.

Den Zeiger realisieren wir durch ein *ListenElement* namens *next*, welches auf das nächste Listenelement zeigen soll, und das Objekt durch eine *EEmail* namens *mail*, das eine E-Mail darstellt.



Aufgabe 4.4:

- Füge diese beiden Attribute in der Klasse *ListenElement* ein. Ebenso sollst Du in dieser Aufgabe einen Konstruktor, der die beiden Attribute auf dem Konstruktur übergeben Werte initialisiert, implementieren.

- (b) Recherchiere im Internet über die Bedeutung der Konstanten `null` in Java und überlege Dir, wie Du mittels dieser Konstanten den bereits implementierten Konstruktor dazu benutzen kannst, einen Konstruktur zu haben, der nur den Inhalt eines Listenelements setzt.

4.2 Die Klasse *Listenelement* - Methoden

Nach wir im letzten Abschnitt den groben Rahmen, also die Attribute und Konstruktoren der Klasse *Listenelement* implementiert haben, wollen wir uns jetzt mit den Methoden dieser Klasse beschäftigen.

Als erstes brauchen wir zwei Methoden, um den Zeiger auf das andere Listenelement manipulieren, also verändern und auslesen, zu können. Wir wollen diese mit `setNext(...)` bzw. `getNext(...)` bezeichnen.



Aufgabe 4.5:

Überlege Dir,

- welche der beiden Methoden ein Argument braucht und welches dieses ist.
- welchen Rückgabewert die andere Methode hat.

Damit bist Du nun in der Lage dies zu implementieren, was Du nun auch tun sollst.

Um weiter sinnvoll auf den eigentlichen Inhalt eines jeden Listenelementes zu greifen zu können, brauchen wir noch eine Methode `getContent(...)`, die uns die in dem Listenelement gespeicherte E-Mail zurückliefert.



Aufgabe 4.6:

Überlege Dir auch hier wieder zuerst,

- ob die Methode einen Parameter braucht.
- welchen Rückgabetyp die Methode hat.

Mit diesen Informationen kannst Du nun auch diese Methode zu implementieren.

4.3 Die Klasse *Liste* - Attribute

Nachdem wir nun eine ganze Reihe an Vorarbeiten erfolgreich abgeschlossen haben, beschäftigen wir uns nun mit der Klasse *Liste*.



Aufgabe 4.7:

Aus Kapitel 2 weißt Du, dass eine Liste aus verketteten Listenelementen und aus einem Zeiger auf das erste und das letzte Element besteht. Überlege Dir geeignete Attribute für diese Eigenschaften sowie einen Konstruktor, der eine leere Liste anlegt (Hinweis: mittels der Konstanten *null*).

Für die Implementierung wollen wir einen zusätzlichen Zeiger, welchen wir mit *currentElement* bezeichnen wollen, einfügen, den wir vorwärts über die Liste bewegen können, also mit einem bestimmten Methodenaufruf ein Listenelement weitersetzen können. Ferner können wir auf dieses Listenelement zugreifen - doch dazu gibt es im nächsten Abschnitt mehr Informationen.



Aufgabe 4.8:

- (a) Erstelle eine neue Klasse *Liste*.
- (b) Implementiere Deine Überlegungen (d.h. Konstruktor und Attribute) der letzten Aufgabe.
- (c) Erstelle ein weiteres Attribut, dass es ermöglicht, später einen solchen zusätzlichen Zeiger aus der letzten Bemerkung zu implementieren. Überlege Dir auch hier wieder zuerst, welchen Datentyp dieser besitzen muss, bevor Du diesen implementierst. Wir wollen diesen zusätzlichen Zeiger wie oben erwähnt mit *currentElement* bezeichnen. Das *currentElement* soll vom Konstruktor auf *null* gesetzt werden.

4.4 Die Klasse *Liste* - Methoden

So, nachdem wir auch wieder für die Klasse *Liste* den groben Rahmen gelegt haben, wollen wir uns jetzt mit den Methoden dieser Klasse auseinandersetzen. Jetzt wird es richtig spannend. In diesem Abschnitt werden wir unsere Liste „zum Leben erwecken“, in dem Sinne, dass wir Methoden implementieren, mit denen wir mit der Liste arbeiten können und diese zur Speicherung von Daten einsetzen können.

Als erstes wollen wir eine Methode schreiben, die feststellt, ob unsere Liste leer ist. Wann ist unsere Liste leer? Genau dann, wenn es keinen Kopf gibt. Programmiertechnisch gibt es dann keinen Kopf, wenn das entsprechende Attribut auf eine leere Referenz, also *null* zeigt.



Aufgabe 4.9:

- (a) Schreibe eine Methode *isEmpty()*, die den einen boolschen Wert als Rückgabe hat und überprüft, ob die Liste leer ist. Wenn die Liste leer ist, dann soll *true* zurückgegeben werden.
- (b) In der Vorbemerkung wurde für eine entsprechende Abfrage auf das Kopfelement zurückgegriffen. Gibt es einen Grund, der dagegen sprechen würde, eine analoge Abfrage mit dem Fuß zu implementieren?

Als nächsten wollen wir uns dem *currentElement* zuwenden. Wie bereits erwähnt zeigt er immer auf ein Element der Liste und wir können ihn immer mittels der Methode *next()* auf das Nachfolgeelement des jeweiligen Elementes setzen. Dabei ist ein Sonderfall zu beachten: Wenn das *currentElement* auf das letzte Element zeigt und *next()* wird aufgerufen, dann soll er auf das erste Element der Liste gesetzt werden. Um auch wieder sofort zum Anfang der Liste zu gelangen, wollen wir eine weitere Methode *toFirst()* schreiben, die das *currentElement* auf das erste Element setzt. Wenn die Liste leer ist, soll nichts verändert werden.



Aufgabe 4.10:

- (a) Schreibe eine Methode *toFirst()*, die die geforderten Eigenschaften hat.

- (b) Überlege Dir genau die Bedingungen für die Methode `next()` bevor Du diese implementierst. Danach kannst Du sie implementieren.



Aufgabe 4.11:

Ein Freund zeigt Dir das folgenden Stück Quelltext:

```
1 // Kommentar 1
2     public EMail getItem(){
3         if (!this.isEmpty()){
4             return currentElement.getContent();
5         } else {
6             return null;
7         }
8     }
```

- Beschreibe ausführlich, was diese Methode macht. Beachte dabei besonders auf die Bedingungen, d.h. wann sie was macht.
- Schreibe aus Deinen Überlegungen einen knappen Kommentar für diese Methode, den man statt Kommentar 1 in den Quelltext einfügen kann.
- Übernehme diese paar Zeilen Quelltext in Deine Implementierung.

Nachdem wir nun die Operationen mit dem zusätzlichen Zeiger implementiert haben, wollen wir uns einer der wichtigsten Methoden zuwenden, dem Einfügen. Wenn Du nicht mehr genau weißt, wie das Einfügen vonstatten geht, dann nutze jetzt die Gelegenheit und ließ es noch einmal im Kapitel 2 nach.

Wir wollen hier eine besondere Methode zum Einfügen nutzen: Die Methode `insertBehind(...)` soll, wie der Name schon vermuten lässt, das neue Element immer hinter dem Element einfügen, auf das das `currentElement` zeigt. Dabei gibt es einige Sonderfälle zu beachten. Mehr dazu erfährst Du in der folgenden Aufgabe:



Aufgabe 4.12:

Ein Freund schickt Dir auch noch eine weitere Methode. Leider sind bei der Übertragung ein paar Zeilen bzw. Anweisungen verloren gegangen.

```

1  /**
2   * This method allows us, to insert E-Mails into the list.
3   * The mail will be inserted behind the currentElement.
4   */
5  public void insertBehind(EMail mail){
6      ListenElement insert = /* Übertragungsfehler 1*/
7      if(this.isEmpty()){
8          /* Übertragungsfehler 2*/
9          this.currentElement = insert;
10     } else {
11         if(currentElement == this.Fuss){
12             /* Übertragungsfehler 3*/
13         } else {
14             insert.setNext((this.currentElement).getNext());
15             /* Übertragungsfehler 4*/
16         }
17     }
18 }

```

- (a) Versuche aus dem Quelltext die drei möglichen Fälle herauszufinden, die auftreten können. Beschreibe dabei jeweils, wie die Liste für einen solchen Fall aussehen muss.
- (b) Beschreibe, wie man bei jedem Fall vorgehen muss, damit die Anforderungen an die Methode erfüllt sind.
- (c) Schreibe die Zeilen bzw. Anweisungen neu, bei denen es bei der Übertragung Fehler gegeben hat und übernehme die gesamte Methode in Deine Implementierung.
- (d) Welchen Nachteil birgt diese Implementierung hinsichtlich der Positionen, an denen wir einfügen können?

Bevor wir das Kapitel erfolgreich mit der fertigen Implementierung der Liste abschließen und zur Lernkontrolle kommen, müssen wir noch die Methode `delete()` implementieren.

Was macht die Methode `delete()`? Die Methode soll das Element löschen, auf das das `currentElement` zeigt. Wie Du sicherlich noch aus Kapitel 2 weißt, sind beim Löschen einige Sonderfälle zu beachten, die wir jetzt noch einmal aufzählen wollen:

- (1) Die Liste ist leer, es kann also nicht gelöscht werden.
- (2) Die Liste beinhaltet nur ein Element. Die Liste ist also nach dem Löschen leer.
- (3) Wenn die Kopf- oder Fußelemente gelöscht werden, dann müssen die ent-

sprechenden Zeiger richtig gesetzt werden.

Wenn keiner der Sonderfälle auftritt und nicht entsprechend des Sonderfalles anders reagiert werden muss, dann kann man einfach den Zeiger `next()` des Elementes vor dem `currentElement` auf das nächste Element, also den Nachfolger von `currentElement` setzen. Jetzt können wir nicht mehr beim Durchlaufen der Liste auf diesen Eintrag zugreifen, da er in der Verkettung nicht mehr vorkommt. Jedoch zeigt `currentElement` noch auf diesen Eintrag der Liste. Um den Eintrag nun endgültig zu löschen, müssen wir noch den Zeiger `currentElement` ändern. Danach zeigt kein Zeiger mehr auf den Listeneintrag und dieser wird dann folglich bei der weiteren Ausführung eines Programms aus dem Speicher gelöscht. Wie wollen wir jetzt den Zeiger anpassen? Wenn das zu löschen Element der Fuß der Liste war, dann soll das neue `currentElement` der neue Fuß sein. Sonst vereinbaren wir: Das `currentElement` soll dann auf das Nachfolgeelement des zu löschen zeigen.



Aufgabe 4.13:

- (a) Überlege Dir eine Struktur von Abfragen für die Sonderfälle.
- (b) Schreibe eine Methode `getElementBefore(ListenElement elem)`, die nur innerhalb der Klasse Liste zur Verfügung steht und den Vorgänger des Listenelementes `elem` zurückliefert. Wenn die Liste leer ist oder `elem` der Kopf der Liste ist, soll `null` zurückgeben werden.
- (c) Implementiere die Methode `delete()`.

Wenn Du nun willst, kannst Du Deine Implementierung testen, beispielsweise, in dem Du ein paar Mails einfügst, alle wieder löscht und überprüfst, ob die Liste wieder leer ist. Oder Du kannst ein paar E-Mails an unterschiedlichen Stellen einfügen und gucken, ob die Liste intern die gewünschte Reihenfolge hat. Sicherlich fallen Dir noch weitere mögliche „Tests“ ein, um Deine Implementierung auf Herz und Nieren zu testen. In der Mediothek haben wir Dir ein Beispiel für einen solchen Test abgedruckt. Wenn Du dich bei den Methodenbezeichnungen an unsere Vorgaben gehalten hast, dann sollte dieser Test ohne Probleme laufen.

4.5 Lösungen zu den Aufgaben

Bevor wir die Lösungen zu den Aufgaben angeben, möchten wir dich auf folgendes aufmerksam machen:

Der hier vorgestellte Quellcode ist nicht als die Musterlösung zu verstehen, sondern als eine mögliche Lösung. Wahrscheinlich wird Deine Implementierung an vielen Stellen abweichen. Sollte dies der Fall sein, dann versuche unsere Lösung nachzuvollziehen und überlege, ob Deine Lösung das gleiche Ergebnis liefert und ob Du die gleichen Sonderfälle beachtet hast und entsprechend reagiert hast. Wenn dies der Fall ist, dann ist Deine Lösung mit hoher Wahrscheinlichkeit richtig. Wenn Du Dir nicht sicher bist, dann frage Deinen Lehrer um Rat.

Lösung zu 4.1

Um ein neues Projekt zu erstellen klickt man auf „Projekt“ → „New Projekt“, wählt einen geeigneten Speicherort und vergibt einen treffenden Namen. Als ein solcher Name bietet sich beispielsweise „Die Liste“ oder einfach „Liste“ an.

Nothelfer Wenn Du noch nicht mit BlueJ umgehen kannst, dann schaue am besten einmal in die Mediothek, dort findest Du ein Tutorial zu diesem Programm. Wenn Du kein BlueJ hast oder einen anderen Java-Editor verwenden möchtest, kannst Du dies natürlich auch tun.

Lösung zu 4.2

Um eine neue Klasse zu erstellen klickt man auf „Edit“ → „New Class“ und vergibt der Klasse den entsprechenden Namen.

Lösung zu 4.3

- Der Datentyp String bietet sich an für alle drei Attribute an.
- Lösung zu Teil (c) befindet sich am Ende des Quelltextes.

```
1 /**
2 * The class EMail represents a simple EMail, using the attributes "Absender",
3 * "Betreff" and "Text".
4 *
5 *
6 * @author Chr. Graf, T. Reimes, V. Venediktov
7 * @version 1.0
```

```

8  /*
9  public class EMail{
10    /**
11     * This are the attributes of the class. They are marked as private, so that
12     * nobody can access them. In this example an email ist reduced to the following
13     * attributes:
14     */
15    private String Absender;
16    private String Betreff;
17    private String Text;
18
19    /**
20     * Constructor for objects of class EMail.
21     */
22    public EMail (String Absender, String Betreff, String Text){
23        this.Absender = Absender;
24        this.Betreff = Betreff;
25        this.Text = Text;
26    }
27
28    /**
29     * This method prints the attributes of the class on the screen.
30     * With this method you will be able to test your implementation
31     * later on.
32     */
33    public void drucke () {
34        System.out.println("E-Mail von " + this.Absender);
35        System.out.println("Betreff " + this.Betreff);
36        System.out.println("Text " + this.Text);
37        System.out.println("#-----#");
38    }
39
40    /**
41     * We also need some getters for the attributes, because they are marked
42     * as private. We will renounce on setters, because there is no need to
43     * change the attributes after creating an email.
44     */
45    public String getAbsender(){
46        return this.Absender;
47    }
48
49    public String getBetreff(){
50        return this.Betreff;
51    }
52    public String getText(){
53        return this.Text;
54    }
55 }
```

Lösung zu 4.4

- (a) Die Lösung findest Du unter „Quelltext zur Klasse *ListenElement*“.
- (b) *null* ist eine vordefinierte Konstante, die als leere Referenz bezeichnet wird.
Den neuen Konstruktor erhälst Du, in dem Du den vorhandenen Konstruktor wie folgt aufrufst: *ListenElement(EMailmail, null)*.

Lösung zu 4.5

- (a) Die Methode `setNext(...)` braucht ein Argument. Dieses Argument ist genau das neue Listenelement, auf den das Listenelement zeigen soll.
- (b) Die Methode liefert ein Listenelement zurück, der Rückgabetyp ist also `ListenElement`.

Die Implementierung der Methode findest Du unter „Quelltext zur Klasse `ListenElement`“.

Lösung zu 4.6

- (a) Nein, die Methode braucht keinen Parameter, da sie keine Werte annimmt sondern nur einen Wert zurückliefert.
- (b) Die Methode hat den Rückgabetyp `EMail`, da sie genau die in dem Listenelement gespeicherte E-Mail zurückliefern soll.

Die Implementierung der Methode findest Du unter „Quelltext zur Klasse `ListenElement`“.

Quelltext zur Klasse `ListenElement`

```
1 /**
2 * This class represents an entry of a list.
3 *
4 * @author Chr. Graf, T. Reimes, V. Venedikov
5 * @version 1.0
6 */
7 public class ListenElement
8 {
9     /**
10    * This are the attributes of the class:
11    * next is a pointer to the next entry of the list
12    * mail is a pointer to an email, which is
13    * stored in this listentry
14    */
15    private ListenElement next;
16    private EMail mail;
17
18    /**
19    * This is the constructor of the class
20    */
21    ListenElement(EMail mail, ListenElement next){
22        this.mail = mail;
23        this.next = next;
24    }
25
26    /**
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
159
160
161
162
163
164
165
166
167
168
169
169
170
171
172
173
174
175
176
177
178
179
179
180
181
182
183
184
185
186
187
187
188
189
189
190
191
192
193
194
195
196
197
197
198
199
199
200
201
202
203
204
205
206
206
207
208
208
209
209
210
211
212
213
213
214
214
215
215
216
216
217
217
218
218
219
219
220
220
221
221
222
222
223
223
224
224
225
225
226
226
227
227
228
228
229
229
230
230
231
231
232
232
233
233
234
234
235
235
236
236
237
237
238
238
239
239
240
240
241
241
242
242
243
243
244
244
245
245
246
246
247
247
248
248
249
249
250
250
251
251
252
252
253
253
254
254
255
255
256
256
257
257
258
258
259
259
260
260
261
261
262
262
263
263
264
264
265
265
266
266
267
267
268
268
269
269
270
270
271
271
272
272
273
273
274
274
275
275
276
276
277
277
278
278
279
279
280
280
281
281
282
282
283
283
284
284
285
285
286
286
287
287
288
288
289
289
290
290
291
291
292
292
293
293
294
294
295
295
296
296
297
297
298
298
299
299
300
300
301
301
302
302
303
303
304
304
305
305
306
306
307
307
308
308
309
309
310
310
311
311
312
312
313
313
314
314
315
315
316
316
317
317
318
318
319
319
320
320
321
321
322
322
323
323
324
324
325
325
326
326
327
327
328
328
329
329
330
330
331
331
332
332
333
333
334
334
335
335
336
336
337
337
338
338
339
339
340
340
341
341
342
342
343
343
344
344
345
345
346
346
347
347
348
348
349
349
350
350
351
351
352
352
353
353
354
354
355
355
356
356
357
357
358
358
359
359
360
360
361
361
362
362
363
363
364
364
365
365
366
366
367
367
368
368
369
369
370
370
371
371
372
372
373
373
374
374
375
375
376
376
377
377
378
378
379
379
380
380
381
381
382
382
383
383
384
384
385
385
386
386
387
387
388
388
389
389
390
390
391
391
392
392
393
393
394
394
395
395
396
396
397
397
398
398
399
399
400
400
401
401
402
402
403
403
404
404
405
405
406
406
407
407
408
408
409
409
410
410
411
411
412
412
413
413
414
414
415
415
416
416
417
417
418
418
419
419
420
420
421
421
422
422
423
423
424
424
425
425
426
426
427
427
428
428
429
429
430
430
431
431
432
432
433
433
434
434
435
435
436
436
437
437
438
438
439
439
440
440
441
441
442
442
443
443
444
444
445
445
446
446
447
447
448
448
449
449
450
450
451
451
452
452
453
453
454
454
455
455
456
456
457
457
458
458
459
459
460
460
461
461
462
462
463
463
464
464
465
465
466
466
467
467
468
468
469
469
470
470
471
471
472
472
473
473
474
474
475
475
476
476
477
477
478
478
479
479
480
480
481
481
482
482
483
483
484
484
485
485
486
486
487
487
488
488
489
489
490
490
491
491
492
492
493
493
494
494
495
495
496
496
497
497
498
498
499
499
500
500
501
501
502
502
503
503
504
504
505
505
506
506
507
507
508
508
509
509
510
510
511
511
512
512
513
513
514
514
515
515
516
516
517
517
518
518
519
519
520
520
521
521
522
522
523
523
524
524
525
525
526
526
527
527
528
528
529
529
530
530
531
531
532
532
533
533
534
534
535
535
536
536
537
537
538
538
539
539
540
540
541
541
542
542
543
543
544
544
545
545
546
546
547
547
548
548
549
549
550
550
551
551
552
552
553
553
554
554
555
555
556
556
557
557
558
558
559
559
560
560
561
561
562
562
563
563
564
564
565
565
566
566
567
567
568
568
569
569
570
570
571
571
572
572
573
573
574
574
575
575
576
576
577
577
578
578
579
579
580
580
581
581
582
582
583
583
584
584
585
585
586
586
587
587
588
588
589
589
590
590
591
591
592
592
593
593
594
594
595
595
596
596
597
597
598
598
599
599
600
600
601
601
602
602
603
603
604
604
605
605
606
606
607
607
608
608
609
609
610
610
611
611
612
612
613
613
614
614
615
615
616
616
617
617
618
618
619
619
620
620
621
621
622
622
623
623
624
624
625
625
626
626
627
627
628
628
629
629
630
630
631
631
632
632
633
633
634
634
635
635
636
636
637
637
638
638
639
639
640
640
641
641
642
642
643
643
644
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346

```

```

27 * This are the getters and setters for the attributes.
28 * There is no setter for the mail-attribute, because
29 * there ist no need to chance the email stored in this
30 * entry.
31 */
32 public ListenElement getNext(){
33     return next;
34 }
35
36 public void setNext(ListenElement next) {
37     this.next = next;
38 }
39
40 public EMail getContent() {
41     return mail;
42 }
43 }
```

Lösung zu 4.7

Kopf und Fuß zeigen jeweils auf Listenelemente, so müssen also vom Typ ListenElement sein. Da wir vom Kopf aus jedes Listenelement erreichen können (Folge entsprechend lange den Zeigern auf das Nachfolgerelement) braucht die Klasse *Liste* nur die beiden Attribute Kopf und Fuß. Der Konstruktor, der eine leere Liste anlegt, muss Kopf und Fuß auf *null* setzen.

Lösung zu 4.8

- (a) Um eine neue Klasse zu erstellen klickt man auf „Edit“ → „New Class“ und vergibt der Klasse den entsprechenden Namen.
- (b) Die Implementierung der Attribute und des Konstruktors findest Du unter „Quelltext zur Klasse *Liste*“.
- (c) Auch hier muss der Datentyp wieder „ListenElement“ sein, da es ein Zeiger auf ein ListenElement ist. Die Implementierung des zusätzlichen Attributes findest Du unter „Quelltext zur Klasse *Liste*“.

Nothelfer zu Teil (c) Auch wenn der Zeiger *currentElement* eine gewisse Funktionalität hat, auf die wir später noch eingehen werden, ist er erstmal vom Prinzip her genau ein Zeiger auf einen beliebigen Eintrag in der Liste, vgl. dazu auch Fuß bzw. Kopf der Liste.

Lösung zu 4.9

- (a) Die Implementierung der Methode `isEmpty()` findest Du unter „Quelltext zur Klasse *Liste*“.
- (b) Wenn die Liste richtig implementiert ist, spricht nichts dagegen, für die Abfrage den Fuß der Liste zu benutzen.

Nothelfer Solltest Du Probleme damit haben, nachzuvollziehen, warum die Liste genau in diesem Fall leer ist, dann schaue Dir noch einmal die Definition der Liste aus Kapitel 2 an und mache Dir klar, wovon es in der Definition „genau einen gibt“ bzw. wovon es „beliebig viele“, also insbesondere auch keinen, gibt: „ein Zeiger auf das erste Element der Liste“ - diesen gibt es immer. Wenn es kein erstes Element, also überhaupt kein Element, in der Liste gibt, kann der Zeiger auf „nichts“ zeigen. Für die programmietechnische Überprüfung braucht man nun nur noch eine simple Abfrage, wohin der Kopf bzw. Fuß zeigt.

Lösung zu 4.10

Die Implementierung den beiden Methoden findest Du unter „Quelltext zur Klasse *Liste*“. Die Bedingungen für die Methode `next()` kannst Du aus den *if*-Abfragen ablesen.

Nothelfer zu Teil (b) Die beiden Bedingungen lauten wie folgt:

- Wenn das `currentElement` auf das letzte Element zeigt, dann ...
- Sonst ...

Lösung zu 4.11

Wenn die Liste leer ist, gibt die Methode `null`, also eine leere Referenz, zurück. Wenn sie nicht leer ist, liefert die Methode die E-Mail zurück, die in dem Listen-element gespeichert ist, auf das das `currentElement` zeigt.

Ein möglicher Kommentar wäre: „Wenn die Liste nicht leer ist, dann wird die E-Mail zurückgeliefert, die in dem Eintrag gespeichert ist, auf den das `currentElement` zeigt.“ bzw. „If the list isn't empty, the returnvalue will be the E-Mail which is stored in the entry pointed by `currentElement`.“

Nothelfer Wenn Du nicht darauf kommst, was diese Methode macht, dann gebe Dir eine Liste mit zwei oder drei Elementen in unserer Schreibweise vor und „führe“ die Anweisungen darauf aus. Beachte, dass wir bei unserer Schreibweise den Zeiger `currentElement` noch nicht hatten.

Lösung zu 4.12

- (a) Die Fälle sind in der Reihenfolge ihrer Abfrage aufgelistet:
- (1) Die Liste ist leer.
 - (2) Das `currentElement` zeigt auf den Fuß.
 - (3) Das `currentElement` zeigt auf ein beliebiges Element (außer dem Fuß) der Liste.
- (b) Die Reaktionen sind in der Reihenfolge ihres Auftretens aufgelistet:
- (1) Lasse Kopf, Fuß und `currentElement` auf das neue Element zeigen.
 - (2) Lasse den Zeiger des Fußelementes für das nächste Element auf das neue Element zeigen und setze den Fuß einen Eintrag weiter.
 - (3) Setze den Zeiger `next` des neuen Elementes auf den Nachfolger von `currentElement` und setze den Zeiger `next` des `currentElement`'s auf das neue Element.
- (c) Die Implementierung der Methode bzw. die fehlenden Stücke findest Du unter „Quelltext zur Klasse `Liste`“
- (d) Das Problem ist, dass wir nicht vorne an der Liste neue Einträge einfügen können.

Nothelfer

- (a) Überlege genau, welche Abfragen gemacht werden und wie die Liste aussehen muss, damit die entsprechende Abfrage ein positives Ergebnis liefert.
- (b) Mit den Überlegungen aus der letzten Aufgabe kannst Du nun gucken, wie Du vorgehen musst, um Dein Ziel, das Einfügen hinter einer bestimmten Position, zu erreichen. Überlege wieder genau, welche Veränderungen die Liste machen muss.
- (c) „Übersetze“ Deine Überlegungen und Skizzen zu dem letzten Aufgabenteil in entsprechende Anweisungen für den Computer.
- (d) Zeichne Dir eine Liste mit zwei Elementen auf und versuche an den drei möglichen Positionen einzufügen. Benutze dabei allerdings nur Methoden, die Du bereits implementiert hast. Gerne kannst Du dieses Experiment auch

am Computermachen, in dem Du Dir ein kleines Testprogramm schreibst, was genau diese drei Fälle untersucht.

Lösung zu 4.13

- (a) Eine mögliche Struktur für Abfragen erkennst Du innerhalb des folgenden Quelltextes. Wenn innerhalb einer Methode ohne Rückgabewert das Schlüsselwort `return;` steht, wird der Aufruf der Methode ohne Ausführung der folgenden Zeilen beendet und an den nächsten Befehl nach dem Aufruf der Methode gesprungen.

```
1     public void delete(){
2         if(this.isEmpty()){
3             return;
4         }
5         if(Kopf == Fuss){
6             /* ... */
7             return;
8         }
9         if(currentElement == Fuss){
10            /* ... */
11            return;
12        }
13        if(currentElement == Kopf){
14            /* ... */
15            return;
16        }
17        /* ... */
18    }
```

- (b) Den Quelltext zu der Methode findest Du unter „Quelltext zur Klasse *Liste*“.
(c) Siehe „Quelltext zur Klasse *Liste*“.

Nothelfer

- (a) Schreibe Dir die Sonderfälle kurz auf und überlege Dir, welche Bedingungen beispielsweise an Kopf oder Fuß gestellt sind, damit dieser Fall eintritt. Erarbeite daraus Deine Abfragen, anhand deren, Du die einzelnen Sonderfälle unterscheiden kannst.
- (b) Prinzip: Fange am Kopf der Liste an, und überprüfe Eintrag für Eintrag, ob dieser der Vorgänger ist. Wenn ja, dann gebe den entsprechenden Eintrag zurück. Beachte, dass die Suche nur ausgeführt wird, wenn die Liste nicht leer ist. Überlege Dir weiter, wann Du `null` zurückgeben musst, weil es keinen Vorgänger gibt.

Quelltext zur Klasse *Liste*

```
1  /**
2  * This class is the E-Mail-List.
3  *
4  * @author Chr. Graf, T. Reimes, V. Venediktor
5  * @version 1.0
6  */
7  public class Liste
8  {
9      /**
10     * These are the attributes of the class.
11     * Kopf - pointer for the first element of the list
12     * Fuss - pointer for the last element of the list
13     * currentElement - additional pointer, which can be
14     * moved forward over the list
15     */
16    private ListenElement Kopf;
17    private ListenElement Fuss;
18    private ListenElement currentElement;
19
20    /**
21     * Constructor for objects of class Liste
22     */
23    public Liste(){
24        this.Kopf = null;
25        this.Fuss = null;
26        this.currentElement = null;
27    }
28
29    /**
30     * This method checks, if the list is empty. When it's
31     * empty, it will return "true".
32     */
33    public boolean isEmpty(){
34        // A list is empty when it doesn't have an headelement.
35        return (this.Kopf == null);
36    }
37
38    /**
39     * These two methods are for the
40     * operations using the currentElement.
41     */
42    public void toFirst(){
43        if (!this.isEmpty()){
44            // the list isn't empty
45            // so move the currentElement
46            this.currentElement = Kopf;
47        }
48    }
49
50    public void next(){
51        if (!this.isEmpty()){
52            // the list isn't empty
53            // so move the currentElement
54            if (this.currentElement != Fuss){
55                // case one: the currentElement doesn't
56                // point the end of the list
57                this.currentElement = (this.currentElement).getNext();
58            } else {

```

```

59         // case two: it points to the end of the list
60         // so we'll move it to the first element
61         this.toFirst();
62     }
63 }
64 }
65
66 /**
67 * This method returns the E-Mail stored in this entry.
68 */
69 public EMail getItem(){
70     // If the list isn't empty,
71     if(!this.isEmpty()){
72         // get the pointed mail and return it.
73         return currentElement.getContent();
74     } else {
75         return null;
76     }
77 }
78
79 /**
80 * This method allows us, to insert E-Mails into the list.
81 * The mail will be inserted behind the currentElement.
82 */
83 public void insertBehind(EMail mail){
84     // Construct a new listentry, storing the mail.
85     ListenElement insert = new ListenElement(mail, null);
86     // case one: The List is empty.
87     if(this.isEmpty()){
88         // point all pointers at the new element
89         this.Kopf = insert;
90         this.Fuss = insert;
91         this.currentElement = insert;
92     } else {
93         // case two: the end is pointed
94         if(currentElement == this.Fuss){
95             // add the element at the end of the list
96             (this.Fuss).setNext(insert);
97             // correct the pointer to the end of the list
98             Fuss = Fuss.getNext();
99         } else {
100             // case three: any element in the list is pointed
101             insert.setNext((this.currentElement).getNext());
102             (this.currentElement).setNext(insert);
103         }
104     }
105 }
106
107 /**
108 * This method computes the predecessor of the entry elem.
109 * If the list is empty or the elem is the head, it will return
110 * null. Note: If the elem isn't contained, this method will
111 * trigger an error when the end is reached. So it's marked as
112 * private.
113 */
114 private ListenElement getElementBefore(ListenElement elem){
115     ListenElement ret = null;
116     // see description of the method
117     if ((!this.isEmpty())&&(elem != Kopf)){
118         ret = Kopf;

```

```

119     while (ret.getNext() != elem){
120         ret = ret.getNext();
121     }
122 }
123 return ret;
124 }
125
126 /**
127 * This method deletes the element, which is pointed by currentElement.
128 */
129 public void delete(){
130     // case one: the list is empty, just leave
131     if(this.isEmpty()){
132         return;
133     }
134     // case two: only one element, delete it and leave
135     if(Kopf == Fuss){
136         Kopf = null;
137         Fuss = null;
138         currentElement = null;
139         return;
140     }
141     // case three: the end is pointed
142     if(currentElement == Fuss){
143         // get the predecessor and let the foot point on it
144         Fuss = this.getElementBefore(Fuss);
145         // delete the predecessor of the end
146         Fuss.setNext(null);
147         // correct the currentElement
148         currentElement = Fuss;
149         return;
150     }
151     // case four: the head must be deleted
152     if(currentElement == Kopf){
153         // set the head as the Element after the head
154         Kopf = Kopf.getNext();
155         // korrekt currentElement
156         currentElement = Kopf;
157         return;
158     }
159     // all other cases:
160     // get predecessor, save it in a temporarily variable
161     ListenElement temp = this.getElementBefore(currentElement);
162     // skip one entry, this is even the entry, which has been to be
163     // deleted
164     temp.setNext(currentElement.getNext());
165     // correct the currentElement
166     currentElement = temp.getNext();
167 }
168
169 /**
170 * This method was written for testing the implementation.
171 * It prints the list from the head to the end.
172 * The currentElement will be marked, before it is printed.
173 */
174 public void test() {
175     // the iterator - starting by printing the head
176     ListenElement now = Kopf;
177     // as long as the end isn't reached
178     while(now != null) {

```

```

179     if (now == currentElement){
180         System.out.println("Jetzt kommt das currentElement:");
181     }
182     // print the mail
183     (now.getContent()).drucke();
184     // an go on with the next one
185     now = now.getNext();
186 }
187 }
188
189 /**
190 * This two methods are for the
191 * operations using the currentElement.
192 */
193 public void toLast(){
194     if (!this.isEmpty()){
195         // the list isn't empty
196         // so move the currentElement
197         this.currentElement = Fuss;
198     }
199 }
200 public void previous(){
201     if (!this.isEmpty()){
202         // the list isn't empty
203         // so move the currentElement
204         if (this.currentElement != Kopf){
205             // case one: the currentElement doesn't
206             // point the beginning of the list
207             this.currentElement = getElementBefore(this.currentElement);
208         } else {
209             // case two: it points to the beginng of the list
210             // so we'll move it to the last element
211             this.toLast();
212         }
213     }
214 }
215
216 /**
217 * This method allows us, to insert E-Mails into the list.
218 * The mail will be inserted before the currentElement.
219 */
220 public void insertBefore(EMail mail){
221     // create a new entry for the list
222     ListenElement insert = new ListenElement(mail, null);
223     if (this.isEmpty()){
224         // case one: the list is empty
225         this.Kopf = insert;
226         this.Fuss = insert;
227         this.currentElement = insert;
228     } else {
229         // case two: the currentElement points at the head
230         if (currentElement == this.Kopf){
231             // point the next element of the new one to the head
232             insert.setNext(Kopf);
233             // declare the new head
234             Kopf = insert;
235         } else {
236             // the new element must point at the currentElement
237             insert.setNext(currentElement);
238             // include the new element in the list

```

```
239             (getElementBefore(currentElement)).setNext(insert);
240         }
241     }
242 }
```

4.6 Lernfortschrittskontrolle



Lernfortschrittskontrolle 4.1:

Die Klasse „Liste“ wurde um folgendes Attribut erweitert:

```
private int NumberOfElements;
```

In diesem Attribut soll gespeichert werden, wieviele Elemente in der Liste vorhanden sind.

- (a) Diskutiere, in wie weit ein solches Attribut sinnvoll ist, und was man ohne dieses Attribut machen müsste, um festzustellen, wieviele Elemente in der Liste vorhanden sind.
- (b) Erweitere Deine Implementierung so, dass das Attribut bei allen Operationen geändert wird, bei denn die Länge der Liste verändert wird.
- (c) Schreibe eine Methode `getNumberOfElements()`, die die Länge der Liste als Rückgabewert hat. (Getter für das Attribut `NumberOfElements`)
- (d) Wie kannst Du nun auch noch ganz schnell herausfinden, ob die Liste leer ist oder nicht?



Lernfortschrittskontrolle 4.2:

Bis jetzt konnte sich das `currentElement` nur vorwärts über die Liste bewegen. Dies wollen wir jetzt ändern, in dem wir ermöglichen, dass er auch rückwärts gehen kann.

- (a) Schreibe eine Methode `toLast()`, die das Attribut `currentElement` auf das letzte Element setzt.
- (b) Schreibe eine Methode `previous()`, die das Attribut `currentElement` auf das Vorgängerelement des `currentElement's` setzt. Welche Methode kann die dabei behilflich sein? Welche Sonderfälle treten auf? Wie kann man darauf reagieren?
- (c) Schreibe eine Methode `insertBefore(EMail mail)`, die analog zu `insertBehind(...)` die E-Mails vor dem `currentElement` einfügt. Überlege Dir auch hier zuerst die Sonderfälle und notiere diese. Wie muss man bei den Sonderfällen reagieren?

4.7 Lösungen der Lernfortschrittskontrolle

Aufgabe 1

- (a) Mit dem Attribut hat man die Möglichkeit, ganz schnell festzustellen, wie viele Elemente in der Liste vorhanden sind. Ist dieses vorhanden, so kann man in konstanter Zeit feststellen, wie viele Einträge die Liste hat.

Ist ein solches Attribut nicht vorhanden, so muss man die Liste linear durchlaufen und für jedes Element einen dafür angelegten Zähler erhöhen. Da dabei jeder Eintrag durchlaufen werden muss, dauert es je nach Länge der Liste relativ lange, bis man ihre Länge ermittelt hat.

- (b) Wir geben im folgenden nur die zusätzlichen Zeilen in den einzelnen Methoden an, die bei einer erfolgreichen Ausführung der Operation ausgeführt werden müssen.

- *insertBehind(...)*

```
1     NumberOfElements++;
```

- *delete(...)*

```
1     NumberOfElements--;
```

- *Liste()*

```
1     NumberOfElements = 0;
```

- (c) Hier die gesuchte Methode. Man kann auf das Attribut nicht direkt zugreifen, da es als *private* markiert ist.

```
1 public int getNumberOfElements() {
2     return NumberOfElements;
3 }
```

- (d) Man vergleicht einfach den Rückgabewert von *getNumberOfElements()* mit der Konstanten 0. Wenn nämlich die Liste leer ist, dann sind auch keine Elemente in ihr vorhanden. Damit hat das Attribut *NumberOfElements* den Wert 0.

Aufgabe 2

- (a) Die Implementierung der Methode findest Du unter „Quelltext zur Klasse *ListenElement*“.
- (b) Die Implementierung der Methode findest Du unter „Quelltext zur Klasse *ListenElement*“.

(c) Die Implementierung der Methode findest Du unter „Quelltext zur Klasse *ListenElement*“.

Die Sonderfälle sind:

- (1) Die Liste ist leer.
- (2) Das *currentElement* zeigt auf den Kopf.
- (3) Das *currentElement* zeigt auf ein beliebiges Element (außer dem Kopf) der Liste.

Wie reagiert man nun auf diese Sonderfälle?

- (1) Lasse Kopf, Fuß und *currentElement* auf das neue Element zeigen.
- (2) Lasse den Zeiger des neuen Elementes auf den Kopf zeigen und setze anschließend den Kopf auf das neue Element.
- (3) Setze den *next* Zeiger des *currentElement* auf das neue Element. Setze den Zeiger des Vorgängers vom *currentElement* auf das neue Element.

Kapitel 2

Die Liste



Übersicht

Im ersten Teil dieses Kapitels wollen wir eine graphische, formale Darstellung für Listen entwickeln, mit der wir auch in den folgenden Kapiteln noch arbeiten werden und die grundlegend für die informatische Sichtweise der Liste ist.

Im zweiten Teil des Kapitels wollen wir zusammen mit der neuen Sichtweise die auf Listen definierten Operationen (einfügen, suchen, löschen usw.) erarbeiten.



Lernziele

Nachdem Du dieses Kapitel bearbeitet hast,

- kannst Du erklären, wie eine dynamische Liste auf einem Computer verwirklicht werden kann.
- kennst Du eine graphische Notation zur Veranschaulichung von Listen im Speicher eines Computers.
- kannst Du mit Hilfe der vorgestellten Notation die Operationen auf einer Liste visualisieren.

2.1 Aufbau einer einfach verketteten Liste

Bevor wir uns mit den genauen Funktionen einer Liste beschäftigen, müssen wir erst den Aufbau einer Liste betrachten. Wir müssen uns also überlegen, wie wir eine Liste formal beschreiben können, da man eine Liste auf verschiedenste Arten darstellen kann.

Berechtigterweise stellt sich jetzt die Frage, warum machen wir das? Wir führen

eine graphische Notation ein, um zu verstehen, wie der Computer intern mit einer Liste arbeitet bzw. wie er eine Liste in seinem Speicher ablegt. Weiter ist es mit der neuen Schreibweise, die wir gleich einführen werden, recht einfach, die Operationen, wie beispielsweise Einfügen oder Suchen, zu erklären und zu verstehen. Später, wenn wir Listen implementieren wollen, hilft uns diese Darstellung ebenfalls weiter: Wir müssen die Grafiken „nur“ in Quellcode übersetzen und haben dann eine fertige Liste, in der wir beliebige Daten abspeichern können.

Normalerweise stehen die Listenelemente, wenn wir beispielsweise den Posteingang aus Kapitel 1 betrachten, untereinander. Dies bedeutet aus Sicht eines einzelnen Elementes, dass immer ein Element unter ihm steht, es sei denn es ist das letzte Element der Liste. Dadurch entsteht eine Art Verkettung zwischen den Elementen, jedes Element hat ein nachfolgendes Element, dieses hat ebenfalls ein Nachfolgeelement, usw. .

Bevor wir jetzt Listenelemente formal definieren, wollen wir uns erst einmal ansehen, wie man eine gewöhnliche Einkaufsliste, notiert auf einem Blatt Papier, formaler aufschreiben kann:

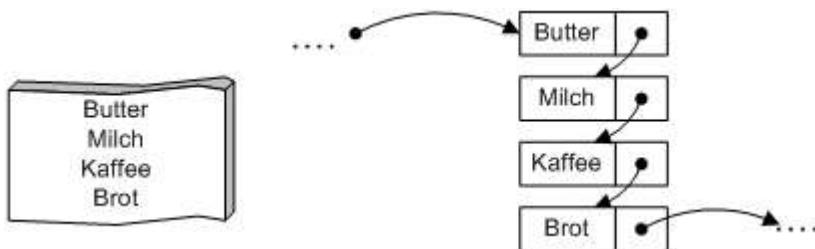


Abbildung 2.1: Von der Einkaufsliste zur formalen Darstellung

Wie wir sehen, ist jedem einzelnen Eintrag der ursprünglichen Liste ein einzelner „Rahmen“ zugeordnet worden. Neben dem Feld in dem der Gegenstand eingetragen ist, den wir einkaufen wollen, gibt es noch ein zweites Feld. In eben diesem zweiten Feld beginnt ein Pfeil, der auf den nächsten Eintrag zeigt.

Du wirst Dich jetzt sicherlich fragen, warum wir das machen. Die Antwort ist recht einfach, aber von zentraler Bedeutung: Wir Menschen sehen sofort, welcher Eintrag in einer Liste der nächste ist, weil wir intuitiv sehen, wie die Liste angeordnet ist und einfach feststellen können, wo sich das nächste Element befindet, da wir gelernt haben, einen Text von links nach rechts und von oben nach unten zu lesen. Der Computer hingegen weiß das nicht. Da wir dem Computer aber beibringen wollen, mit Listen zu arbeiten, müssen wir ihm sagen, wo der nächste Eintrag steht bzw. wo er ihn finden kann. Wir wollen dies im Folgenden tun, in dem wir zu jedem Eintrag einen Verweis hinzufügen, der auf den nächsten Eintrag zeigt.

Nach der Verdeutlichung durch ein simples Beispiel anhand einer Einkaufsliste können wir jetzt genau formalisieren, was wir im Folgenden unter einem Listen-element verstehen werden.



Definition (Listenelement)

Einen Eintrag in einer Liste wollen wir ab jetzt mit Listenelement bezeichnen.

Ein solches Listenelement besteht aus einem Objekt (z.B. einem Text) und einem Zeiger auf ein anderes Listenelement.

Die Objekte unserer Einkaufsliste sind Lebensmittel, wobei diese weitere Eigenschaften neben den Namen besitzen, beispielsweise Mengenangaben. Das Objekt „Lebensmittel“ besteht also aus einem Namen und einer Mengenangabe.

Um die recht abstrakte Definition des Listenelements noch besser zu verdeutlichen, führen wir folgende graphische Notation ein:

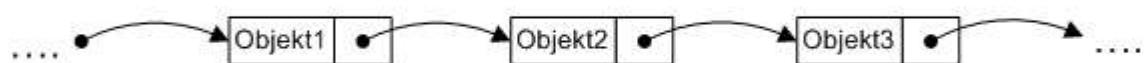


Abbildung 2.2: Listenelemente

Hier sieht man nun, dass die einzelnen Listenelemente immer mit genau einem nachfolgenden Element verbunden sind. Der Zeiger wird hier durch einen Pfeil zwischen den beiden Elementen dargestellt. Dies bedeutet im Sinne der obigen Definition, dass man von einem Element aus seinen Nachfolger „erreichen“ kann. Dadurch entsteht zwangsläufig die Verkettung der Elemente, da jedes Element auf genau ein weiteres zeigt. Im Unterschied zu den „normalen“ Darstellungen schreiben wir die Elemente nebeneinander und nicht mehr untereinander auf. Dies geschieht aber eher aus Platzgründen, vom Prinzip her könnte man die Elemente immer noch untereinander oder beliebig verteilt aufschreiben, weil wir durch die Einführung des Zeigers jederzeit wissen, wo sich das Nachfolgerelement befindet. Weiter sieht man, dass jedes Element, bis auf das erste, einen eindeutigen Vorgänger hat.

Diese Form der Darstellung werden wir im weiteren Verlauf immer wieder aufgreifen und Dir damit beispielsweise die Operationen auf Listen bildlich verdeutlichen. Nachdem wir nun die Listenelemente beschrieben haben, stellt sich noch die Fra-

ge, was uns jetzt noch zur gesamten Liste fehlt. Wir müssen wissen, wo unsere Liste anfängt und wo sie endet, also welches Element das erste und welches das letzte ist.



Definition (Liste)

Eine Liste besteht aus einer beliebigen Anzahl an Listenelementen. Es existiert zudem ein Zeiger auf das erste Element der Liste, den wir Kopf der Liste nennen wollen. Das letzte Element der Liste ist das einzige Element, dass keinen Zeiger auf ein Nächstes besitzt. Auch auf dieses Element gibt es einen Zeiger, den wir mit Fuß bezeichnen wollen.

Durch diese Definition wird unsere Liste eindeutig definiert, da wir ausgehend vom Kopf der Liste alle Elemente miteinander verkettet haben. Dies könnte dann zum Beispiel so aussehen:

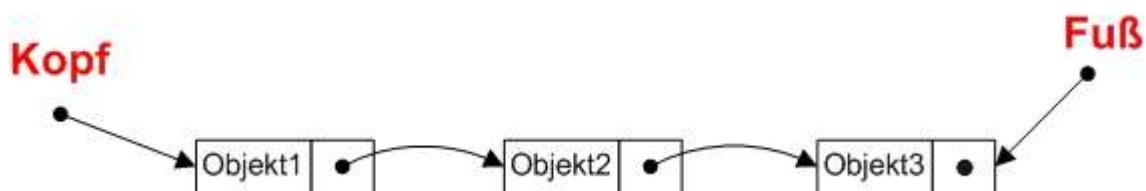


Abbildung 2.3: Dreielementige Liste

Die Zeiger werden hier als Pfeile zwischen den Elementen dargestellt. Der Pfeil zwischen dem ersten und dem zweiten Element bedeutet, dass das erste Element auf das zweite „zeigt“. Da das letzte Element (hier Objekt3) keinen Nachfolger besitzt, existiert auch kein Zeiger auf ein weiteres Element. An dieser Stelle ist unsere Liste damit zu Ende. Kopf und Fuß sind keine Elemente, sondern sind Zeiger auf das erste bzw. das letzte Element. Nachdem wir einiges an Theorie dazu gemacht haben, wird es wieder Zeit, etwas praktischer zu werden.



Aufgabe 2.1:

Du wirst von Deinen Großeltern gebeten, für sie Einkaufen zu gehen. Deine Großmutter gibt Dir telefonisch ihre Einkaufsliste durch. Auf der Liste stehen

- ein Brot,
- 500g Butter,
- 10 Frikadellen,
- 2 kg Rinderbraten,

- eine Gurke,
 - eine Schale Pilze und
 - 1 kg Tomaten,
 - ein Glas Marmelade.
- (a) Schreibe den Einkaufszettel Deiner Großeltern in der Notation auf, die wir gerade eingeführt haben. Dabei soll ein Listenelement folgende Informationen für jedes Lebensmittel separat speichern:

- Name
- Menge

Ein Listenelement sieht also wie folgt aus:



Zeichne auch die Zeiger für Kopf und Fuß ein.

- (b) Ist die folgende Aussage richtig? Wenn nein, dann verbessere sie!

Der Kopf ist das erste und der Fuß das letzte Element der Liste.

- (c) Was unterscheidet die Listenelemente aus der Definition mit den Listenelementen, die wir für die Lösung dieser Aufgabe vorgegeben haben. Was schließt Du daraus?

Hinweis: Schau Dir dazu die Bemerkung nach der Definition vom Listenelement an.

Bevor jetzt uns jetzt endlich mit den „Operationen auf Listen“ beschäftigen, gibt es zum Abschnittsende noch eine weitere ähnliche Aufgabe, auf die wir im Verlaufe des Kapitels noch zurückgreifen wollen.



Aufgabe 2.2:

Wir betrachten nun wieder den Posteingang aus Kapitel 1. Wie könnten wir diese Liste nun mit der oben vorgestellten graphischen Notation darstellen? Versuche, eine Liste mit fünf Elementen auf Papier aufzuschreiben. Da es sich bei den Objekten in den Listenelementen um E-Mails handelt, sollten diese mit „E-Mail1“, „E-Mail2“, usw. beschriftet werden.

2.2 Operationen auf Listen

Um nun mit unserer formal eingeführten Liste arbeiten zu können, müssen wir noch einige Funktionsweisen der Liste genauer erklären. Wir müssen die Funktionen so definieren, dass durch Verändern der Liste diese immer noch formal korrekt arbeitet, also zum Beispiel, dass jedes Element weiterhin einen Zeiger auf seinen Nachfolger besitzt und Kopf und Fuß jeweils auf das erste beziehungsweise letzte Element zeigen. Diese angesprochenen Funktionen werden, gerade in der Programmierung, auch Operationen genannt.

2.2.1 Suchen

Als Erstes wollen wir uns damit beschäftigen, wie wir in einer Liste bestimmte Elemente suchen können. Dies erscheint gerade bei Listen mit wenigen Elementen sehr einfach, man überschaut die Liste und sieht, ob ein Element vorhanden ist und an welcher Stelle es steht. Wir müssen uns hiermit allerdings aus zwei Gründen etwas genauer beschäftigen. Zum einen könnte unsere Liste sehr groß sein, so dass wir auf Anhieb nicht sehen, ob ein Element in der Liste ist, und zum anderen, weil ein Computer die Möglichkeit nicht besitzt, die Liste zu überschauen, da der Computer nicht alle Elemente zur selben Zeit betrachten kann. Wir müssen uns daher einen Algorithmus überlegen, wie wir systematisch ein Element in einer Liste suchen können.

Wir gehen wie folgt vor:



Definition (Suchen eines Listenelementes)

- (1) Wir suchen das erste Element der Liste. Dieses können wir sehr einfach finden, da vom Kopf der Liste direkt ein Zeiger auf dieses Element zeigt.
- (2) Nun schauen wir in dieses Element hinein und prüfen, ob es, das von uns Gesuchte ist. Wenn ja, haben wir unser Element schon gefunden.
- (3) Wenn dieses Element nicht das Gesuchte ist, betrachten wir den Nachfolger des Elementes, wenn es einen Nachfolger gibt. Dieses können wir durch den vorhandenen Zeiger auf das nächste Element ebenfalls einfach finden.

- (4) Jetzt prüfen wir, ob dieses Element das Gesuchte ist. Wenn ja, haben wir unser Element gefunden. Wenn nein, gehen wir zurück zu Schritt 3 und durchlaufen so die Liste bis an die Stelle, wo sich das gesuchte Element befindet.
- (5) Durchlaufen wir die Liste bis zum Ende, ohne das Gesuchte gefunden zu haben, wissen wir, dass unsere Liste nicht das gesuchte Element enthält.

Dies wollen wir anhand eines Beispiels noch einmal erläutern. Wir betrachten wieder den Posteingang unseres E-Mail-Accounts, wollen diesen aber direkt in unserer formalen Schreibweise ansehen.

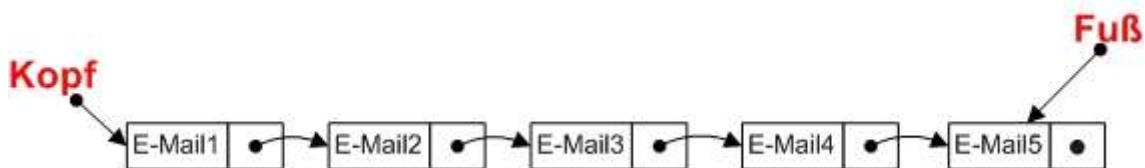


Abbildung 2.4: Ein Posteingang in formaler Schreibweise

Wir möchten nun alle E-Mails finden, die vom Absender „Dietmar Schulz“ gesendet wurden. Bei Betrachtung der Liste stellen wir fest, dass bei den E-Mails gar kein Absender vorhanden ist. Wir müssen also in jedes Listenelement hineinschauen, ob der Absender der von uns gesuchte ist. Wie dies genau zu tun ist, wird im vierten Kapitel bei der Implementierung der Liste genauer behandelt. Wir wollen uns jetzt vorerst damit begnügen, dass wir den Absender in der E-Mail sehen können. In diesem Beispiel wollen wir davon ausgehen, dass die Objekte „E-Mail3“ und „E-Mail5“ den Absender „Dietmar Schulz“ enthalten, was wir aber vorher nicht wissen und es daher suchen müssen. Wir gehen wie folgt vor:

- Gehe vom Kopf der Liste zur ersten Element („E-Mail1“).
- Prüfe, ob der Absender von „E-Mail1“, der von uns gesuchte ist. Da er es nicht ist, betrachten wir den Nachfolger („E-Mail2“).
- Auch hier prüfen wir, ob dies das gesuchte Element ist. Da dies ebenfalls nicht das Gesuchte ist, betrachte wieder den Nachfolger („E-Mail3“).
- Hier stellen wir beim Prüfen fest, dass wir das richtige Element gefunden haben!

Wir wissen also nun, dass das Objekt „E-Mail3“ im dritten Listenelement den Absender „Dietmar Schulz“ besitzt.



Aufgabe 2.3:

Wir haben also nun einen Algorithmus, um Elemente in einer Liste zu suchen. Ist das Suchergebnis in unserem Beispiel vollständig? Ergänze den oben genannten Algorithmus so, dass das Suchergebnis immer vollständig ist!

2.2.2 Einfügen

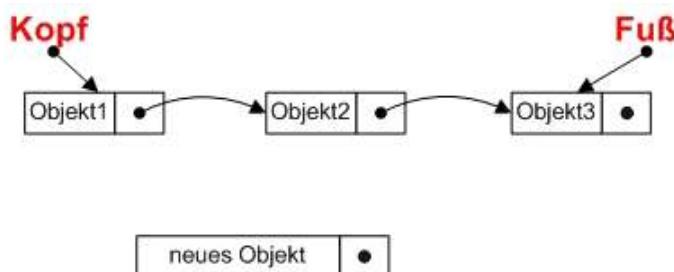
Damit wir überhaupt etwas suchen können, muss die Liste einige Einträge enthalten. Wie kommen die Einträge in die Liste? Durch Einfügen! Um ein Element in eine Liste einzufügen gehen wir wie folgt vor:



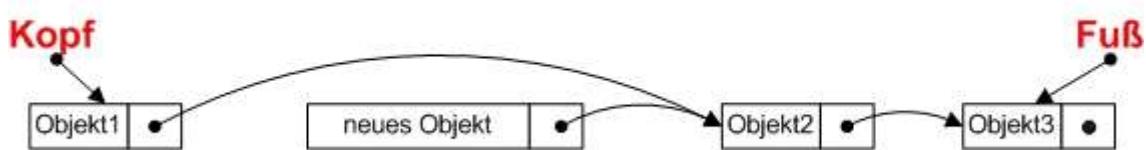
Definition (Einfügen eines Listenelementes)

- (1) Zuerst müssen wir die Stelle suchen, an der das Element eingefügt werden soll (siehe Kapitel 2.2.1), wir merken uns sowohl den Vorgänger als auch den Nachfolger des einzufügenden Elementes.
- (2) Setze den Zeiger des Elementes auf den Nachfolger.
- (3) Der Zeiger des Vorgängers, der ebenfalls noch auf den Nachfolger des eingefügten Elementes zeigt, muss nun auf das neu eingefügte Element gesetzt werden.

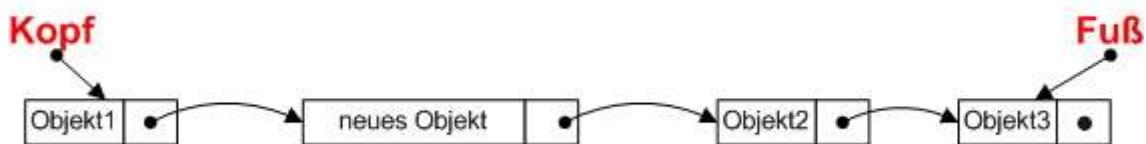
Da diese Definition sehr abstrakt ist, wollen wir diese noch einmal graphisch darstellen. Hierzu betrachten wir eine dreielementige Liste und ein neues Listenelement, dass wir an zweiter Stelle der Liste einfügen wollen.



Zuerst müssen wir die Stelle finden, an der wir das neue Element einfügen wollen. Um die zweite Stelle zu finden müssen wir, wie in 2.2.1 beschrieben, die Liste vom ersten Element aus durchlaufen, bis wir das Element gefunden haben, vor dem wir einfügen wollen. Dies geht hier natürlich recht schnell, da wir schon an der zweiten Stelle einfügen wollen. Das Objekt „neues Objekt“ muss nun den Nachfolger „Objekt2“ erhalten, da es an der zweiten Stelle eingefügt werden soll.



Es besteht nun das Problem, dass das Element „neues Objekt“ ausgehend vom Kopf noch nicht erreichen können. Zudem besitzen zwei Elemente den Nachfolger „Objekt2“, unsere Liste ist jetzt nicht mehr eindeutig. Daher wird der Zeiger von „Objekt1“ auf „neues Objekt“ gesetzt. Damit ist „neues Objekt“ der Nachfolger des ersten Elementes und unsere Liste wieder eindeutig. Das eingefügte Element ist als Teil der Liste an zweiter Stelle erreichbar.



Wenn wir die graphischen Veränderungen wieder in die Art und Weise der Speicherung im Computer übertragen, so werden die Listenelemente nicht verschoben, sondern nur ihre Zeiger auf die Nachfolger verändert und das neue Listen- element an einer freien Stelle im Speicher abgelegt.

Je nach Verwendungszweck der Liste kann es erforderlich sein, immer an einer festen Stelle einzufügen, beispielsweise am Anfang oder am Ende der Liste.



Aufgabe 2.4:

Was müssen wir beachten, wenn wir ganz am Anfang oder am Ende der Liste ein Element einfügen? Schaue Dir hierfür noch einmal die Definition unserer Liste an. Entscheide Dich, ob Du diese Aufgabe für Anfang oder Ende der Liste bearbeiten möchtest.

- (a) Schreibe formal, also analog wie in der Definition „Suchen eines Listen-elementes“, auf wie Du vorgehen würdest.
- (b) Zeichne in unserer Notation zu Deiner Lösung aus dem vorherigen Auf-gabenteil ein Beispiel, wo bei einer Liste mit drei Einträgen ein vierter hinzugefügt wird.

2.2.3 Löschen

Eine weitere Funktion die eine Liste besitzen sollte, ist das Löschen von Elementen. Wir müssen uns also Gedanken machen, wie wir ein Element aus der Liste entfernen können, ohne dass dabei die Struktur der Liste verloren geht.



Aufgabe 2.5:

Schreibe auf ein Blatt Papier auf, wie eine Liste aufgebaut ist beziehungsweise wodurch sie charakterisiert ist! Solltest dies nicht mehr ganz im Kopf haben, kannst natürlich noch einmal am Anfang des Kapitels nachsehen. Schreibe Dir zudem eine beliebige Liste mit vier Elementen in der oben vorgegebenen grafischen Notation auf.

Zuerst wollen wir uns überlegen, wie wir ein Element, dass mitten in der Liste, also nicht am Anfang oder am Ende steht, entfernen können. Dies erfolgt auf folgende Weise:



Definition (Löschen eines Listenelementes)

- (1) Zuerst müssen wir das Element suchen, dass gelöscht werden soll (siehe Kapitel 2.2.1), wir merken uns sowohl den Vorgänger als auch den Nachfolger des zu löschenen Elementes. Finden wir dieses Element nicht in der Liste brauchen wir nichts weiter zu tun, ansonsten müssen die beiden nächsten Schritte durchgeführt werden.
- (2) Nun entfernen wir das zu löschen Element aus der Liste.

- (3) Der Zeiger des Vorgängerelementes muss nun auf das Nachfolgeelement gesetzt werden.



Aufgabe 2.6:

In der letzten Aufgabe solltest Du Dir eine beliebige Liste mit vier Elementen aufschreiben. Versuche nun das zweite Element dieser Liste mit der gerade vorgestellten Methode aus der Liste zu entfernen. Schreibe die Liste nach jedem durchgeföhrten Schritt neu auf.

Wir haben jetzt ein Element aus unserer Liste gelöscht und unsere Listenstruktur weiterhin erhalten, da wir die „Lücke“, die durch das Entfernen des Elementes entstanden ist, geschlossen haben. Jetzt müssen wir nur noch klären, wie das erste beziehungsweise das letzte Element der Liste gelöscht werden kann.



Aufgabe 2.7:

Überlege Dir, wie man das erste und das letzte Element aus einer Liste löschen kann. Versuche diese Vorgehensweise möglichst genau aufzuschreiben. Was muss an diesen beiden Stellen beachtet werden?

Man sieht, dass das Löschen am Anfang viel schneller möglich ist, als das Löschen am Ende, da wir durch den Kopf, der auf das erste Element zeigt, direkt wissen, wo sich das Element befindet und wir es nicht suchen müssen. Wir wissen durch den Fuß zwar auch, wo sich das letzte Element der Liste befindet, können dieses aber ohne vorheriges Suchen des Vorgängerelementes dieses nicht löschen.

Als Sonderfall wäre noch eine Liste zu erwähnen, wo das zu löschende Element sowohl das erste als auch das letzte Element ist, also unsere Liste nur aus einem Element besteht. In diesem Fall haben wir nach dem Löschen des letzten Elementes eine leere Liste, das heißt sowohl Kopf als auch Fuß zeigen auf kein Element.

2.2.4 Sortieren

Als letzte Operation wollen wir nun - zumindest in Ansätzen - das Sortieren einer Liste erklären. Bevor wir dieses tun, sollte man sich noch über den Sinn und die

Vorteile einer sortierten Liste Gedanken machen. Hierzu wollen wir wieder einmal den Posteingang eines Mail-Accounts betrachten. Bei den meisten Mailanbietern oder Mailprogrammen können die angekommenen Mails nach verschiedenen Kriterien sortiert werden, zum Beispiel nach dem Absender, der Betreffzeile oder der Ankunftszeit, was normalerweise als Standard eingesetzt wird. Die Sortierung bringt allerdings nicht nur für denjenigen, der vor dem Computer sitzt und die Mails betrachtet Vorteile, sondern auch für den Computer, wenn er die Mails im Posteingang weiterverarbeiten möchte.



Aufgabe 2.8:

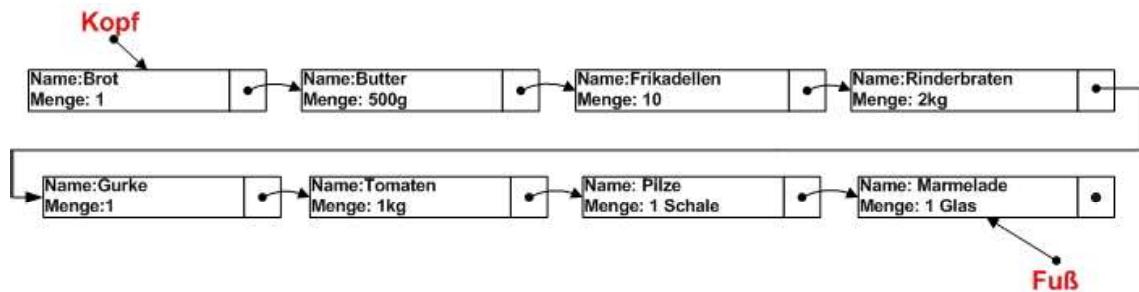
Überlege Dir, welche Vorteile eine sortierte Liste bei der Durchführung der oben beschriebenen Operationen bringen! Beachte besonders das Suchen eines Listenelements.

Als kleinen Nachteil sollten wir allerdings noch erwähnen, dass das Erstellen einer sortierten Liste etwas aufwendiger ist, als die Elemente beliebig anzuordnen, da man zuerst die passende Stelle in der Liste Suchen muss, bevor man ein Element einfügt oder löscht. Gerade bei besonders großen Listen ist dieser Nachteil nicht gravierend, da die Vorteile der sortierten Liste diesen Aufwand rechtfertigen. Leider ist es nicht einfach eine unsortierte in eine sortierte Liste umzuwandeln. Man muss hier schrittweise vorgehen und immer paarweise zwei Elemente vertauschen. Dies können wir durch Löschen und Einfügen der Elemente erreichen. Vielleicht hast oder wirst Du Dich noch im weiteren Verlauf des Informatikunterrichts mit dem Problem des Sortierens auseinandersetzen. Einen konkreten Algorithmus zu entwerfen, der eine Liste komplett sortiert, wollen wir hier nicht aufführen, da dies hier den Rahmen sprengen würde und auch kein spezielles Problem der Liste ist.

2.3 Lösungen der Aufgaben

Lösung Aufgabe 2.1

(a) Der Einkaufszettel sollte wie folgt aussehen:



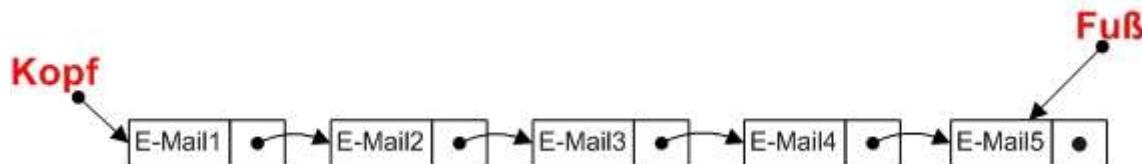
Der Zeilenumbruch nach dem vierten Element geschieht hier aus Platzgründen, wichtig ist, dass die einzelnen Elemente durch Zeiger korrekt verkettet sind.

(b) Die Aussage ist nicht ganz korrekt. Der Kopf ist nicht das erste Element, sondern ein Zeiger auf dieses. Genauso verhält es sich beim Fuß.

(c) In dieser Aufgabe haben wir in das Listenelement direkt die Eigenschaften (Name und Menge) hineingeschrieben. Dies haben wir bei der Definition nicht gemacht, da man einerseits sehr schwer allgemeine Eigenschaften festlegen kann und andererseits uns die Struktur der Liste wichtiger ist als die genauen Eigenschaften oder deren Art der Speicherung.

Lösung Aufgabe 2.2

Die Liste sollte wie folgt aussehen:



Lösung Aufgabe 2.3

Das Suchergebnis ist unter Umständen nicht vollständig, da nicht auszuschließen ist, dass auch noch weitere Elemente in der Liste den gleichen Absender haben. In unserem Beispiel sollte „E-Mail5“ ebenfalls den gesuchten Absender enthalten, allerdings wird dieser nicht mehr gefunden, da wir vorher schon ein Element mit dem gesuchten Absender gefunden haben.

Um dieses Problem zu vermeiden, müssen wir den Suchalgorithmus im vierten Schritt verändern:

- (4) Jetzt prüfen wir, ob dieses Element das Gesuchte ist. Wenn ja, haben wir **ein gesuchtes** Element gefunden. Jetzt gehen wir zurück zu Schritt 3 und Durchlaufen so die Liste bis an die Stelle, wo sich **nächste gesuchte Element** befindet. **Dies wiederholen wir solange bis die Liste zu Ende ist.**

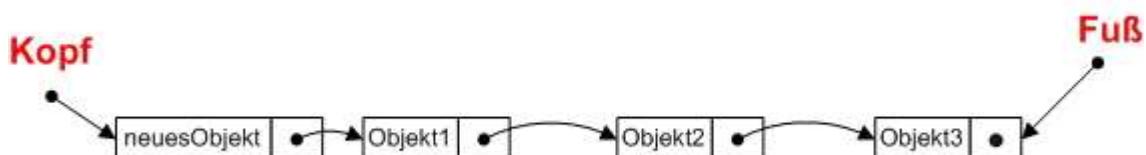
Wir müssen die Liste also in jedem Fall bis zum letzten Element untersuchen, damit unser Suchergebnis vollständig ist.

Lösung Aufgabe 2.4

Einfügen am Anfang:

(a) Es muss darauf geachtet werden, dass der Kopf dann auf das neu eingefügte Element zeigt. Ansonsten wäre unser „altes“ erstes Element immer noch das Erste in der Liste, da immer das Element das Erste ist, wo der Kopf hinzeigt.

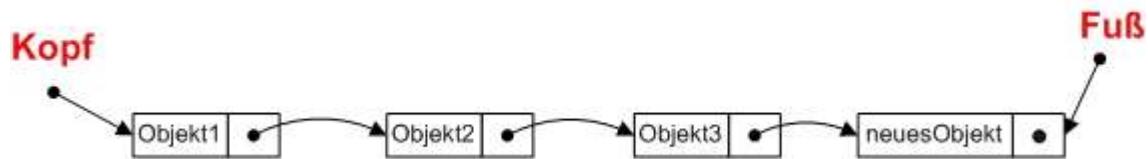
(b) Ein korrektes Einfügen könnte z.B. so aussehen:



Einfügen am Ende:

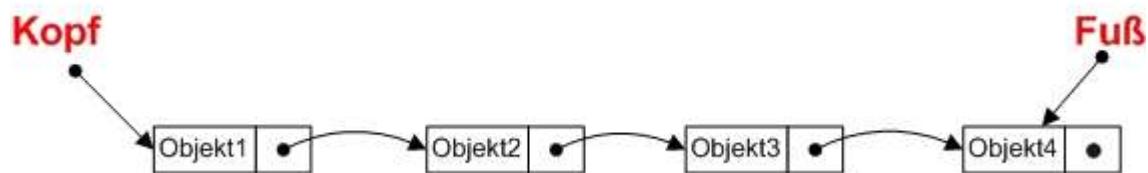
(a) Beim Einfügen an der letzten Stelle muss natürlich dementsprechend der Zeiger auf das neu eingefügte Element zeigen. Da das „alte“ letzte Element keinen Zeiger auf ein Nachfolgeelement besitzt, hat unser „neues“ letztes Element auch keinen Nachfolger.

(b) Dies könnte dann so aussehen:

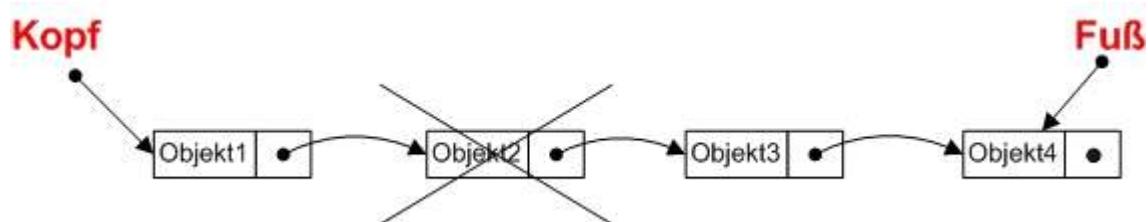


Lösung Aufgabe 2.5

Eine Liste besteht aus einer Verkettung von Listenelementen, die wiederum aus einem Objekt, also einem Inhalt und einem Zeiger auf ihren Nachfolger bestehen. Damit wir wissen, wo unsere Liste anfängt und endet, haben wir zudem noch einen Kopf und einen Fuß definiert, die jeweils auf das erste und letzte Element zeigen. Eine solche Liste könnte zum Beispiel so aussehen:



Lösung Aufgabe 2.6

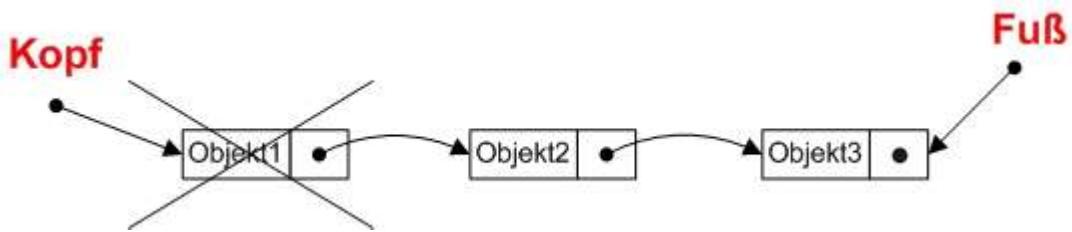




Lösung Aufgabe 2.7

Auch hier müssen wieder der Kopf- und der Fußzeiger beachtet werden.

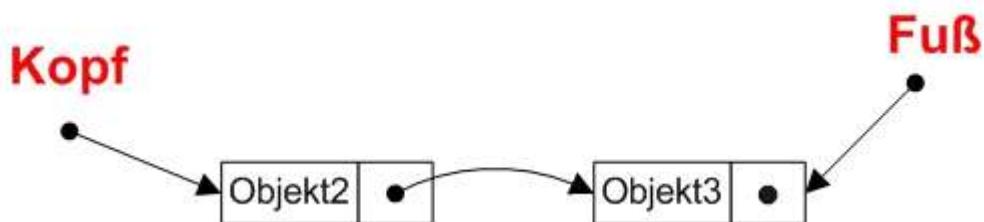
Löschen des ersten Elementes:



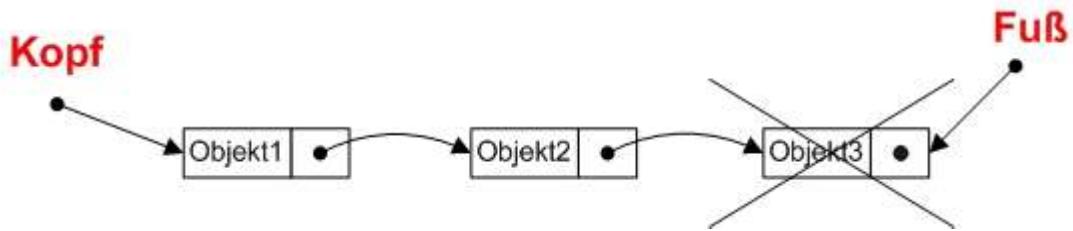
Lösche das erste Element. Da dieses Element kein Vorgängerelement besitzt, brauchen wir diesen nicht, wie in der Definition unter Schritt 3 beschrieben, auf das Nachfolgeelement zu setzen.



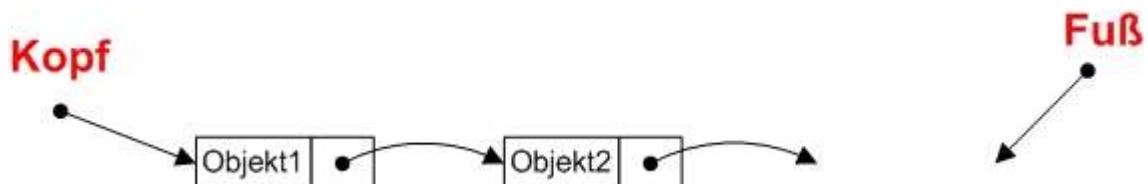
Setze der Kopfzeiger auf das zweite Element.



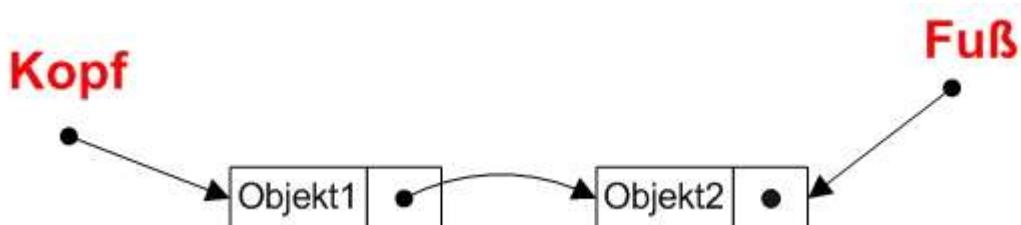
Löschen des letzten Elementes:



Lösche das letzte Element. Per Definition müsste nun der Zeiger des Vorgängers auf den Nachfolger zeigen. Da das letztes Element allerdings auf kein weiteres mehr zeigt, zeigt nun auch der Vorgänger, das „neue“ letzte Element, auf kein weiteres Element.



Setze nun den Fußzeiger auf das „neue“ letzte Element.



Lösung Aufgabe 2.8

Folgende Vorteile entstehen durch eine sortierte Liste:

- Beim Suchen eines Elementes brauchen wir in einer sortierten Liste nur bis zu der Stelle zu suchen, wo wir von der Sortierung her, das zu suchende Element erwartet hätten. Suchen wir beispielsweise in unserem Posteingang, der nach dem Namen des Absenders sortiert ist, eine Mail von „Max Mustermann“ brauchen wir nur Mails bis einschließlich dem Buchstaben „M“

zu durchsuchen. Wenn wir bis dahin den Namen nicht gefunden haben, wissen wir, dass dieser nicht in der Liste vorhanden ist.

⇒ Wir können die Liste schneller durchsuchen!

- Beim Einfügen und Löschen von Listenelementen benötigen wir vorher immer das Suchen eines Elementes. Von daher gehen auch diese beiden Operationen schneller.

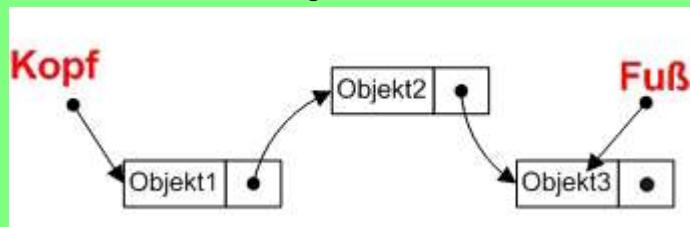
2.4 Lernfortschrittskontrolle



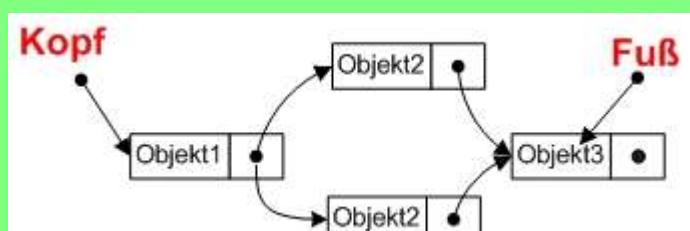
Lernfortschrittskontrolle 2.1:

Welche der hier dargestellten Listen, sind nach unserer Definition KEINE Listen? Schau Dir die Listen genau an!

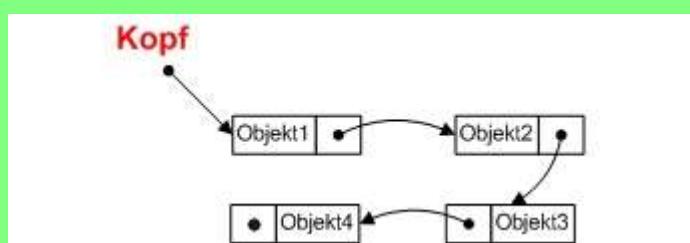
(a)



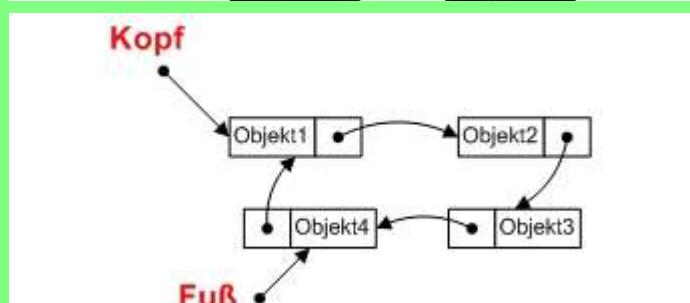
(b)



(c)



(d)





Lernfortschrittskontrolle 2.2:

Vervollständige folgende Aussage über eine Liste:

Eine Liste besteht aus Listenelementen, die durch einen _____ miteinander verkettet sind.

Damit wir wissen, wo unsere Liste anfängt und endet, existiert zudem ein Zeiger auf das erste Element, den wir _____ nennen, und ein Zeiger, der auf das _____ Element zeigt, den wir Fuß nennen.



Lernfortschrittskontrolle 2.3:

Welche der aufgeführten Schritte müssen durchgeführt werden, um ein Element in eine Liste einzufügen?

- (1) Zeiger des Vorgängerelementes auf das neue zeigen lassen.
- (2) Zeiger des neuen Elementes auf seinen Vorgänger zeigen lassen.
- (3) Zeiger des neuen Elementes auf seinen Nachfolger zeigen lassen.
- (4) Suchen der Stelle, wo das Element eingefügt werden soll.
- (5) Zeiger des Vorgängerelementes immer auf das letzte Element zeigen lassen.

2.5 Lösungen der Lernfortschrittkontrolle

Aufgabe 2.1

- (a) Dies ist gemäß unserer Definition eine Liste, auch wenn sie etwas ungewöhnlich aufgeschrieben ist.
- (b) Da „Objekt1“ zwei Nachfolger und „Objekt3“ zwei Vorgänger besitzen, ist dies keine formal korrekte Liste.
- (c) Hier haben fast eine korrekte Liste, allerdings fehlt der Fußzeiger.
- (d) Dies ist keine korrekte Liste, da „Objekt4“ einen Zeiger auf „Objekt1“ besitzt. Dieses ist aber wegen des Kopfes unser erstes Element und hat daher keinen Vorgänger.

Aufgabe 2.2

Eine Liste besteht aus Listenelementen, die durch einen **Zeiger** miteinander verbunden sind. Damit wir wissen, wo unsere Liste anfängt und endet, existiert zudem ein Zeiger auf das erste Element, den wir **Kopf** nennen, und ein Zeiger, der auf das **letzte** Element zeigt, den wir Fuß nennen.

Aufgabe 2.3

Die Schritte müssen in der Reihenfolge (4), (3), (1) durchgeführt werden. (2) und (5) sind falsch.

2.6 Additum: Die doppelt verkettete Liste

In diesem Additum wollen wir die Liste, die Du bereits kennengelernt hast, erweitern.



Aufgabe 2.9:

- (a) Beschreibe formal, wie Du vorgehen würdest, wenn Du für einen Listeneintrag in unserer Listennotation den jeweiligen Vorgänger suchen müsstest.
- (b) Überlege Dir, wie man die Liste erweitern könnte, damit man das Vorgängerelement sofort (d.h. ohne suchen) bestimmen kann.



Definition (doppelt verkettete Liste)

Eine doppelt verkettete Liste ist eine einfach verkettete Liste mit folgender Erweiterung:

Jedes Element der Liste hat nicht nur einen Zeiger auf das nachfolgende, sondern auch einen zusätzlichen zweiten Zeiger auf das Vorgängerelement.

Auch hier ist wieder ein Sonderfall zu beachten: Das erste Element hat keinen Vorgänger und hat folglich auch keinen Zeiger auf ein Vorgängerelement.

Diese Erweiterung bringt uns den Vorteil, dass wir zum Einem Vorgängerelemente schnell und einfach finden können und zum Anderen, dass wir nun in beide Richtungen über die Liste laufen können. Es gibt allerdings einen kleinen Nachteil an dem neuen Komfort: Der zusätzliche Zeiger belegt Speicherplatz, so dass wir insgesamt mehr Speicherplatz für die Liste brauchen. Um uns nun die Veränderung der Operationen bezüglich der einfach verketteten Liste zu überlegen, wollen wir uns auch hier erstmal eine graphische Notation überlegen, die die interne Arbeit der Computers aufzeigt.



Aufgabe 2.10:

Überlege Dir eine graphische Schreibweise für die doppelt verkettete Liste analog zu der Notation, die Dir schon von der einfach verketteten Liste bekannt ist.



Aufgabe 2.11:

Erinnere dich an die Einkaufsliste Deiner Großeltern aus Aufgabe 2.1. Stelle diese Einkaufsliste als doppelt verkettete Liste dar. Nutze dazu die Notation aus der Lösung zu Aufgabe 2.10.

Da es zu ausführlich wäre alle Operationen jetzt noch einmal zu erklären, wollen wir uns mit Einfügen eines Elementes begnügen. Es stellt sich nun die Frage, was wir nun anderes tun müssen als bei der einfach verketteten Liste. Und die Antwort ist recht einfach: Wir müssen nichts anders machen, sondern nur zusätzlich die Zeiger zum Vorgängerelement bearbeiten.



Aufgabe 2.12:

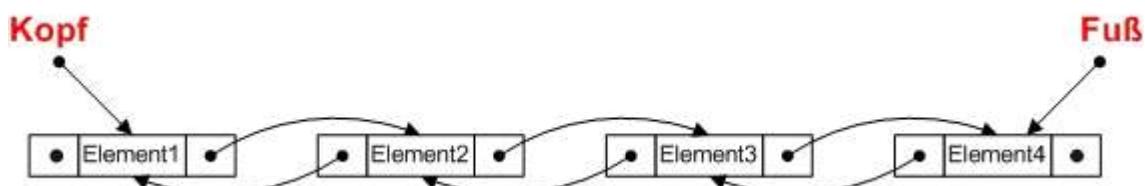
Überlege Dir, wie die Zeiger auf Vorgängerelemente gesetzt werden müssen, wenn ein neues Element eingefügt wird!

Lösung Aufgabe 2.9

- (a) Wenn wir das Vorgängerelement bestimmen wollen, müssen wir die Liste vom Anfang bis zu dem Element durchlaufen, von dem wir den Vorgänger ermitteln wollen, und uns beim Durchlaufen immer das „vorherige“ Element merken. Dadurch haben wir dann den Vorgänger ermittelt. Es dauert also sehr lange ein Vorgängerelement zu bestimmen, vor allem wenn es eher am Ende der Liste steht, da wir dann fast die komplette Liste durchlaufen müssen.
- (b) siehe Definition (doppelt verkettete Liste)

Lösung Aufgabe 2.10

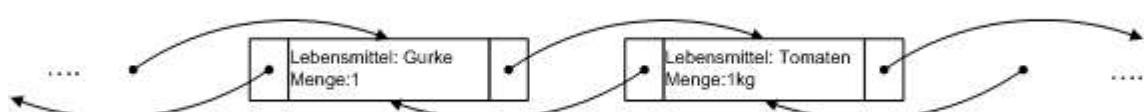
Eine Möglichkeit die doppelt verkettete Liste darzustellen wäre die folgende:



Natürlich gibt es durchaus auch noch weitere Möglichkeiten diese darzustellen, allerdings wollen wir hier weiterhin die Notation benutzen, die wir auch schon bei der einfach verketteten Liste benutzt haben.

Lösung Aufgabe 2.11

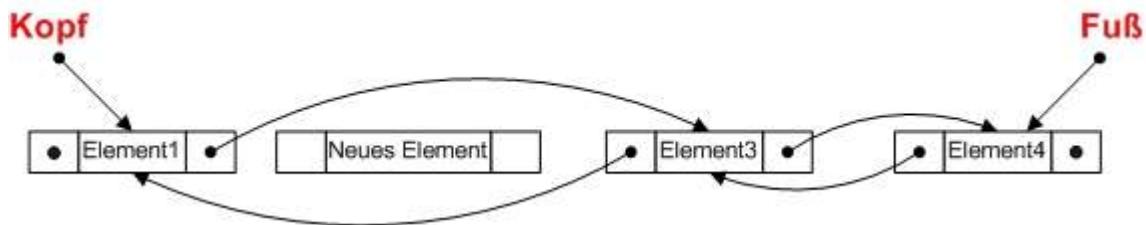
Die Einkaufsliste aus Aufgabe 2.1 könnte wie folgt aussehen:



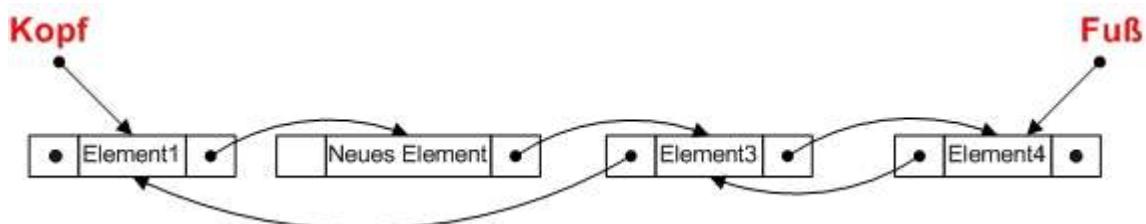
Der Übersicht wegen haben wir hier nicht die ganze Liste, sondern nur exemplarisch zwei Elemente dargestellt.

Lösung Aufgabe 2.12

Was zu tun ist, wollen wir uns an folgender Liste anschauen, wo „neues Element“ eingefügt werden soll:



Als erstes werden die Zeiger auf die Nachfolgeelemente bearbeitet, genauso wie bei einer einfach verketteten Liste:



Jetzt müssen wir nur noch die Zeiger, die auf die Vorgängerelemente zeigen, bearbeiten. Man sieht eigentlich sehr schnell, was noch getan werden muss: Der Vorgängerzeiger von „Element3“ muss auf „neues Element“ und der von „neues Element“ auf „Element1“ gesetzt werden.

