Listen*

Hans Jürgen Ohlbach

23. März 2018

Keywords: Einfach verkettete Liste, doppelt verkettete Listen

Empfohlene Vorkenntnisse: Arrays, C kann hilfreich sein.

Inhaltsverzeichnis

1 Einleitung **2**

2 Die abstrakte Sichtweise **3**

3 Die konkrete Sichtweise **4**

4 Implementierung **4**

5 Ausblick **10**

*Dieser Text ist Teil einer Sammlung von Miniskripten zur Einführung in die Informatik. Er ist in erster Linien für Nichtinformatiker gedacht, kann aber natürlich auch als erste Einführung für Informatiker nützlich sein.

1 Einleitung

Listen sind, ähnlich wie Arrays, Folgen von Objekten. Man benutzt sie insbesondere

- wenn die Objekte unterschiedlich lang sind, oder
- wenn man häufiger Objekte an beliebigen Stellen in der Liste einbauen oder löschen will.

In beiden Fällen könnte man im Prinzip auch Arrays benutzen. Allerdings wären die Operationen u.U. sehr aufwendig. Um z.B. in einem Array ein Element ganz vorne einzufügen, müsste man alle anderen Elemente nach hinten schieben, damit für das neue Element Platz geschaffen wird.

In Listen sind dagegen die Objekte nicht direkt hintereinander gespeichert, sondern können beliebig im Speicher angeordnet werden. Die Sequenz der Elemente erhält man, indem bei jedem Element auch die Adresse auf das nächste Element gespeichert wird. Der Nachteil ist jedoch, dass man die Adressen der Elemente nicht direkt berechnen kann. Um an ein bestimmtes Element zu kommen, muss man die ganze Liste bis zu diesem Element durchlaufen.

Man unterscheidet

einfach verkettete Listen: Diese speichern bei jedem Element die Adresse des nachfolgenden Elements, und

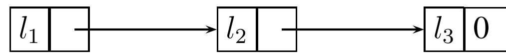
doppelt verkettete Listen: Diese speichern bei jedem Element sowohl die Adresse des nachfolgenden als auch des vorherigen Elements.

Einfach verkettete Listen kann man nur von vorne nach hinten durchlaufen, während doppelt verkettete Listen in beide Richtungen durchlaufen werden können.

Typische Verwendungen von Listen sind auch Schlangen, z.B. Warteschlangen (engl. queues), wo man neue Elemente hinten anhängt, und Elemente vorne wegnimmt.

2 Die abstrakte Sichtweise

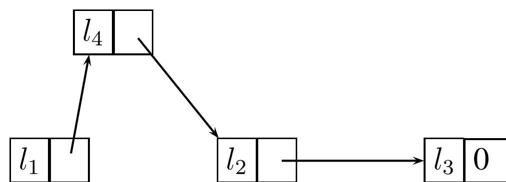
Ganz abstrakt kann man eine einfach verkette Liste als eine Folge von Tupeln
(Element, Zeiger auf nächstes Element) darstellen.



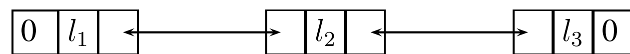
l_1, l_2 und l_3 sind die Listenelemente. Die Pfeile symbolisieren den Verweis von einem Listenelement auf das nachfolgende. Das Ende der Liste wird mit der Adresse 0 angezeigt.

Die Adresse des ersten Listenelements ist damit gleichzeitig die Adresse der ganzen Liste. Man kann sich ja vom ersten Listenelement aus durch die ganze Liste hangeln.

Angenommen, wir wollen jetzt ein weiteres Element l_4 zwischen l_1 und l_2 einfügen. Dazu muss man nur die neue Doppelzelle für l_4 erzeugen und die Verweise entsprechend angleichen.



Doppelt verkette Listen: Diese haben neben den Vorwärtsverweisen bei jedem Element auch noch Rückwärtsverweise.



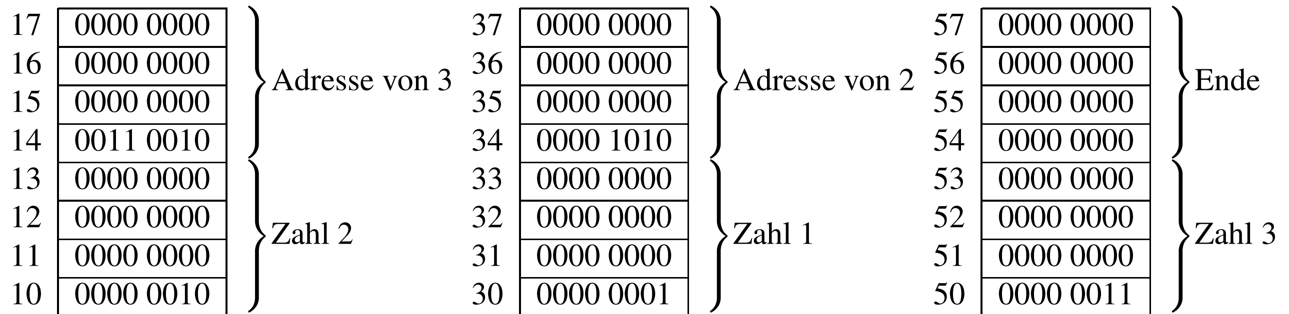
Mit den Rückwärtsverweisen kann man vom Ende der Liste rückwärts durch die Liste iterieren. Hierfür wäre es nützlich, sich auch die Adresse des letzten Listenelements zu merken.

Folgende Operationen sind typisch für Listen:

- ein neues Listenelement erzeugen, mit leeren Zellen für die Verweise,
- ein Listenelement einfügen: vorne, hinten oder in der Mitte,
- ein Listenelement löschen: vorne, hinten oder in der Mitte,
- die Länge der Liste berechnen,
- das n-te Listenelement suchen,
- ein bestimmtes Listenelement suchen,
- eine Schleife über alle Listenelemente laufen lassen, entweder von vorne nach hinten, oder, bei doppelt verkettete Listen auch von hinten nach vorne.

3 Die konkrete Sichtweise

Anders als bei Arrays können die Listenelemente an beliebigen Stellen im Speicher liegen. Eine mögliche Speicherbelegung für eine einfach verkettete Liste mit den drei int-Zahlen 1,2,3 könnte, bei einer 32-Bit Architektur, so aussehen:



Die Liste beginnt an der Adresse 30.
Der Speicherbereich 30-33 enthält die Zahl 1, gefolgt von der Adresse der Zahl 2, nämlich 10.
Der Speicherbereich 10-13 enthält die Zahl 2, gefolgt von der Adresse der Zahl 3, nämlich 50.
Der Speicherbereich 50-53 enthält die Zahl 3, gefolgt von der Endemarkierung (0).
Dabei wird angenommen, dass kein Listenelement jemals an der Adresse 0 abgelegt wird. (Dafür sorgt das Betriebssystem.)

Bei doppelt verketteten Listen hätten man für jedes Listenelement weitere 4 Bytes für die Adresse des Vorgängerelements.

Bei 64-Bit Architekturen bräuchte man für die Adresswerte jeweils 8 Bytes.

4 Implementierung

Das beste Verständnis für die internen Abläufe bei den jeweiligen Operationen, und deren Rechenaufwand erhält man durch eine konkrete Implementierung. Hier wird eine C-Implementierung für einfach verkettete int-Listen vorgestellt. Mit der Programmiersprache C ist man sehr nahe an der Prozessorarchitektur, und kann sehr genau verfolgen, was alles gemacht wird, um die Funktionen zu realisieren.

Wir starten mit ein paar Definitionen:

```
#include <stdlib.h>
#include <stdio.h>

struct ListNode {
  int element;
  struct ListNode* next;};

typedef struct ListNode* List;

#define ERROR -2147483648
```

Zunächst werden die beiden Bibliotheken stdlib und stdio geladen.

`struct ListNode ...` definiert einen Datentyp `ListNode`, bestehend aus einem int-Wert und der *Adresse* eines ListNodes. Der * in `ListNode*` besagt, dass es die Adresse, nicht der Wert selbst ist.

`typedef struct ListNode* List;` gibt der Adresse einer ListNode einen neuen Namen `List`. Variablen des Typs List enthalten ab sofort die Adresse einer ListNode.

Die Direktive `#define ERROR -2147483648` an den Präprozessor des C-Compilers legt fest, dass die Zeichenkette ERROR durch die Zahl -2147483648 zu ersetzen ist. Dies ist die kleinste negative int-Zahl. Sie wird benutzt, um einen Fehler anzuzeigen. Dabei wird angenommen, dass keine Liste diese Zahl als Element enthält.

Als erstes definieren wir eine Funktion, die einen neuen Listenknoten erzeugt.

```
List newNode(int item) {
  List list = (List) malloc(sizeof(struct ListNode));
  list->element = item;
  list->next = 0;
  return list;}
```

`malloc(sizeof(struct ListNode))` reserviert einen freien Speicherblock für ein ListNode-Element[1]. Das Ergebnis von malloc ist die Adresse dieses Speicherbereichs, der mit `(List) malloc(sizeof(struct ListNode))` ab sofort als ListNode-Adresse interpretiert wird.

`list->element = item;` weist dem Element der Liste das item zu.
`list->next = 0;` bewirkt, dass es zunächst kein Nachfolgeelement gibt.
Das Ergebnis ist die Adresse dieses neu erzeugten Listenelements.

Ein Aufruf `List l = newNode(5);` würde also eine Einerliste mit der Zahl 5 erzeugen.

Jetzt fügen wir einer Liste ein neues erstes Element hinzu:

```
List addFirst(int item, List list) {
  List l = newNode(item);
  l->next = list;
  return l;}
```

`l->next = list;` sorgt dafür, dass das Nachfolgeelement des ersten Listenelements die alte Liste ist. Damit wird das neue Listenelement automatisch erstes Element der neuen Liste.

Die Berechnungskosten für diese Operation hängen nicht von der Länge der Liste ab, sind also konstant für alle Listen. Mathematisch sagt man dazu, die Komplexität der Operation ist $\mathcal{O}(1)$.

Ein Aufruf `addFirst(5,0)` würde nun ebenfalls eine Einerliste mit der Zahl 5 erzeugen.

`addFirst(10,addFirst(5,0))` würde die Liste (10,5) erzeugen.

Jetzt können wir Listen erzeugen, und möchten die auch mal ausdrucken.

```
void printList(List list) {
  if(list == 0) {printf("empty list");}
  else {
    while(list != 0) {
      printf("%i \n", list->element);
      list = list->next;}}}
```

Die while-Schleife läuft über die Listenelemente, druckt jeweils die Zahl aus, und schaltet mit `list = list->next;` die Liste auf die Nachfolgelist weiter. Wenn die next-Zelle gleich 0 ist,

[1]Bei einer 64-Bit Architektur wären das 12 Bytes, 4 Bytes für den int-Wert, und 8 Bytes für die Adresse. Tatsächlich wird aber alles auf 8-Byte Blöcke normiert, so dass weitere 4 Bytes nutzlos reserviert werden. Der gesamte Platzbedarf ist daher 16 Bytes.

wird aufgehört.

Man kann auch ein neues Element an das Ende der Liste anhängen.

```
List addLast(int item, List list) {
  List l = newNode(item);
  if(list == 0) {return l;} // Einerliste
  List tail = list;
  while(tail->next != 0) {tail = tail->next;}
  tail->next = l;
  return list;}
```

Das Problem hierbei ist, dass man das letzte Element der alten Liste braucht, um ein neues Element anzuhängen. Die while-Schleife läuft dazu durch die Liste, bis sie auf ein Element stößt, welches keinen Nachfolger hat. Das ist das letzte Element der Liste. An dieses wird das neue Element angehängt.

Der Rechenaufwand für diese Operation hängt daher von der Länge der Liste ab. Die Komplexität ist daher $\mathcal{O}(n)$ wobei n die Länge der Liste ist. Das ist nicht gut! Besser wäre es, man merkt sich separat die Adresse des letzten Elements. Dann kann man ein neues letztes Element mit $\mathcal{O}(1)$ Rechenaufwand direkt anhängen.

Die beiden nächsten Funktionen löschen das erste bzw. das letzte Element der Liste.

```
List removeFirst(List list) {
  if(list == 0) {return 0;} // leere Liste
  List next  = list->next;
  free(list);
  return next;}
```

In der Programmiersprache C ist es wichtig, frei gewordenen Speicher auch explizit wieder frei zu geben. Der Aufruf `free(list)` sorgt dafür, dass der Speicher für das erste Listenelement zu anderweitigen Verwendung wieder frei gegeben wird. (In Java besorgt das der Garbage Collector.)

Die Funktion `removeLast` muss zunächst zum vorletzten Element laufen, um dort die Adresse `next` auf 0 zu setzen. Das letzte Element wird wieder frei gegeben. Der Rechenaufwand dafür ist wiederum $\mathcal{O}(n)$.

```
List removeLast(List list) {
  if(list == 0) {return 0;}
  if(list->next == 0) {free(list); return 0;}
  List tail = list;
  while(tail->next->next != 0) {tail = tail->next;}
  free(tail->next); // tail ist jetzt das vorletzte Element
  tail->next = 0;
  return list;}
```

Eine für das Weitere nützlich Funktion ist die Folgende:

```
List getTail(int index, List list) {
  if(list == 0) {return 0;}
  int i;
  for(i = 0; i < index; ++i) {list = list->next;}
  return list;}
```

Sie läuft eine Liste bis zu einem bestimmten Index ab, und liefert dann die Liste ab diesem Index. `getTail(3,list)` würde z.B. die Restliste ab dem 3. Element liefern.

Die nächste Funktion löscht das Element an einem bestimmten Index. `removeAtIndex(3, list)` würde also das 3. Element löschen. Dafür muss die Funktion das Element vor dem zu löschenden Element finden, und dort die `next`-Adresse auf das übernächste Element setzen. Der Berechnungsaufwand ist wieder $\mathcal{O}(n)$.

```
List removeAtIndex(int index, List list) {
  if(list == 0) {return 0;}  // leere Liste
  if(index == 0) {  // erstes Element loeschen
    List next = list->next;
    free(list);
    return next;}

  List tail = getTail(index-1,list); // tail ab dem Element vor dem index.
  if(tail == 0) {return list;}
  List removeMe = tail->next;     // zu loeschendes Element
  if(removeMe == 0) {return list;}
  tail->next = removeMe->next;    // zu loeschendes Element ueberspringen
  free(removeMe);
  return list;}
```

Die nächste Funktion zählt die Elemente der Liste.

```
int length(List list) {
  if(list == 0) {return 0;} // leere Liste
  int laenge = 1;
  while(list->next != 0) {
    ++laenge;
    list = list->next;}
  return laenge;}
```

Wir möchten natürlich auch Zugriff auf einzelne Elemente der Liste. Das machen die folgenden Funktionen.

```
int getFirst(List list) {
  if(list == 0) {return ERROR;}
  return list->element;}

int getLast(List list) {
  if(list == 0) {return ERROR;}
  while(list->next != 0) {list = list->next;}
  return list->element;}

int get(int index, List list) {
  List tail = getTail(index,list);
  if(tail == 0) {return ERROR;}
  return tail->element;}
```

`getFirst` hat wieder konstanten Berechnungsaufwand, während die anderen beiden Funktionen durch die Liste laufen müssen. Daher ist der Aufwand wieder $\mathcal{O}(n)$.

Die nächste Funktion testet, ob die Liste ein bestimmtes Element enthält.

```
_Bool contains(int item, List list) {
  while(list != 0) {
    if(list->element == item) {return 1;}
    list = list->next;}
  return 0;}
```

Die letzte Funktion schließlich ändert den Wert des Listenelements an einem gegebenen Index.

```
void set(int index, int item, List list) {
  List tail = getTail(index,list);
  if(tail == 0) {return;}
  tail->element = item;}
```

Da Listen eine sehr häufig gebrauchte Datenstruktur sind, enthalten viele Bibliotheken fertige Implementierungen. In Java gibt es dazu insbesondere die generische Klasse LinkedList. Sie implementiert doppelt verkettete Listen eines festen Objekttyps (keine primitiven Datentypen).

5 Ausblick

In vielen Anwendungen kommen die Objekte, die man in eine Liste oder ein Array einfügen kann, mit einer Ordnungsrelation, mit der man sie in eine sortierte Reihenfolge bringen kann. Zahlen kann man der $<$-Relation vergleichen, Strings kann man lexikographisch vergleichen, usw. Will man keine vorgegebene Reihenfolge in einer Liste oder Array, dann kann man die Ordnungsrelation für die Bestimmung der Reihenfolge benutzen, z.B. Zahlen nach ihrer Größe sortieren, oder Personen nach der lexikographischen Ordnung ihrer Namen. In diesem Fall gibt es spezielle der Operationen auf den Arrays und Listen, die diese Ordnungsrelation nutzen, um die sortierte Reihenfolge zu halten. Darauf wird im Miniskript zur Sortierung eingegangen.

Ein weiterer Ansatz ist das *Hashing*. Hier ist die Idee, Objekte zunächst auf eine Zahl abzubilden, und diese Zahl als Index für ein Array zu nutzen. Darauf wird ebenfalls in einem Miniskript eingegangen.