Algorithmen und Datenstrukturen

SoSe 2008 in Trier

Henning Fernau
Universität Trier
fernau@uni-trier.de

Algorithmen und Datenstrukturen

Gesamtübersicht

- Organisatorisches / Einführung
- Grundlagen: RAM, $\mathcal{O}$-Notation, Rekursion, Datenstrukturen
- Sortieren
- Wörterbücher und Mengen
- Graphen und Graphalgorithmen

Ebenen der Abstraktion I

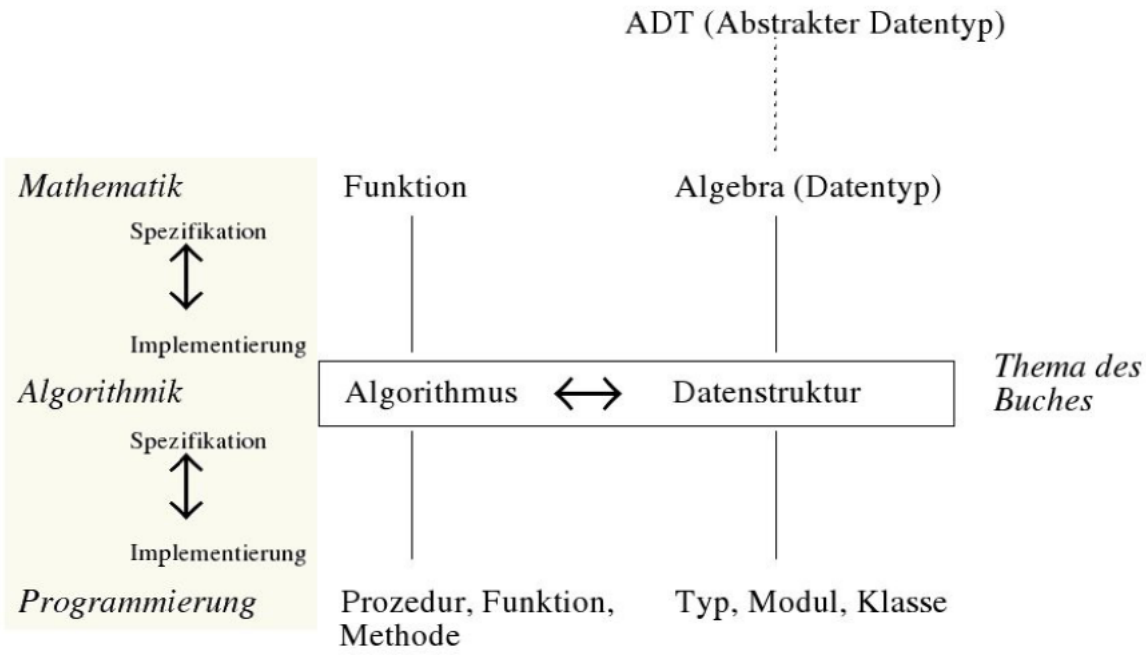
Was zu tun ist. . .

Mathematik Funktionen, (logische etc.) Spezifikation der Arbeitsweise

Algorithmik Algorithmus (in Pseudocode)

Programmierung Implementierung als konkrete Prozedur in einer Programmiersprache

Aus dem Güting-Buch. . .



Pseudocode

Wir werden im Folgenden hoffentlich leicht einsichtige Programmablaufkonstrukte verwenden, wie sie von der LaTeX algorithm.sty Umgebung geliefert werden.

Dazu gehören: Bedingungsabfragen, Schleifen, Rekursionen

Die einzelnen Konstrukte werden hoffentlich anhand der Beispiele klar werden.

Für die Zuweisung verwenden wir $\leftarrow$, um das Gleichheitszeichen in üblicher mathematischer Bedeutung verwenden zu können.

Algorithm 1 Sortieren durch Mischen: mergesort

Input(s): an array $A : \mathbb{R}[1..n]$

Output(s): a sorted array $\mathbb{R}[1..n]$

if $n = 1$ **then**
  return A
else
  Find midpoint m.
  Let $B : \mathbb{R}[1..m]$, $C : \mathbb{R}[1..(n-m)]$.
  $B \leftarrow mergesort(A[1..m])$;
  $C \leftarrow mergesort(A[m+1..n])$;
  return $merge(B, C)$

Algorithm 2 Mischen: merge

Input(s): two sorted arrays $B : \mathbb{R}[1..n], C : \mathbb{R}[1..m]$

Output(s): a sorted array $\mathbb{R}[1..n+m]$

Let $i, j \in \mathbb{Z}$ with $i \leftarrow 1, j \leftarrow 1$.
Let $A : \mathbb{R}[1..n+m]$ be the array to be returned.
while $(i \leq n) \wedge (j \leq m)$ **do**
  if $B[i] \leq C[j]$ **then**
    $A[i+j-1] \leftarrow B[i]; i++;$
  else
    $A[i+j-1] \leftarrow C[j]; j++;$
end while
if $i > n$ **then**
  $A[n+j..n+m] \leftarrow C[j..m];$
else
  $A[m+i..m+n] \leftarrow B[i..n];$

Ebenen der Abstraktion II

Womit etwas zu tun ist. . .

Mathematik Algebra (Datentyp): syntaktische und semantische Sicht

Algorithmik Datenstruktur

Programmierung Implementierung als konkreter Typ (Modul, Klasse)

Elementare Datentypen

Alle gängigen Programmiersprachen bieten Typen wie “ganze Zahlen”, “reelle Zahlen”, “boolean” an.

Auf diesen Typen gibt es Operationen wie “Addition”, “logisches UND”,

Auf der Mathematik- bzw. Algorithmik-Ebene werden wir uns (meist) im Folgenden der üblichen mathematischen Notation bedienen und schreiben $x : \mathbb{Z}$, wenn wir ausdrücken wollen, dass die Variable x ganze Zahlen enthalten kann.

Dabei / dadurch abstrahieren wir (meist) von Überlaufproblemen etc.

Datenstrukturen

bauen auf *(nicht notwendigerweise elementaren) Datentypen* auf, z.B. kann ein *Feld* (engl.: *Array*) A von 20 reellen Zahlen durch $A : \mathbb{R}[1..20]$ eingeführt (*deklariert*) werden (womit auch der Indexbereich klar ist).

Als einzige Operation gestattet ein Feld den *Zugriff* (lesend und schreibend) auf einzelne Elemente durch *Indizierung*.
Beispiel: $A[8]$ greift auf das achte Element von A zu.
Vereinfachend gestatten wir auch den Zugriff auf ganze Bereiche des Feldes, z.B. durch $A[3..5]$.
In unserer Vorstellung liegen die Elemente eines Feldes nacheinander im Speicher des Rechners.
Man beachte die Ähnlichkeit mit dem “Speichermodell” einer RAM.

Weitere Datenstrukturen

Keller (engl.: (meist) Stack)
Schlangen
Listen
Mengen
Bäume

Ein **statischer Keller** wird beschrieben durch:
Maximalgröße m;
Datentyp D der Kellerelemente;
Feld $\mathrm{K} : \mathrm{D}[1..\mathrm{m}]$;
Index des obersten Kellerzeichens: $\mathrm{top} \in \{0, \ldots, \mathrm{m}\}$
Hierbei $\mathrm{top} = 0$ gdw. Keller ist leer.

Kellerspeicher arbeiten nach dem LIFO-Prinzip (Last In, First Out).

Keller-Operationen und ihre Umsetzung für Keller S über Datentyp D:

$S.clear()$	$top \leftarrow 0$
$S.empty()$	return $(top = 0)$
$S.push(x)$	$top{+}{+}; S.K[top] \leftarrow x;$
$S.pop()$	return $S.K[top]; top{-}{-};$
$S.top()$	return $S.K[top]$

Achtung: Überlauf / Unterlauf (Overflow / Underflow) bislang unbeachtet.
Dafür könnte man zusätzliche (Test-)Operationen einführen.
Beobachte: (1) Ist D elementar (z.B., $D = \mathbb{Z}$), so lassen sich alle Operationen in $\mathcal{O}(1)$ im RAM-Modell umsetzen.
(2) top zeigt immer auf das oberste Kellerzeichen, sofern vorhanden.

Keller-Operationen und ihre Eigenschaften

1. Nach $S.\mathtt{clear}()$ ist $S.\mathtt{empty}()$ stets wahr.

2. Tritt kein Überlauf auf, so gilt: $S.\mathtt{top}(S.\mathtt{push}(x)) = x$.

3. $x \leftarrow S.\mathtt{top}()$ is äquivalent zu: $x \leftarrow S.\mathtt{pop}(); S.\mathtt{push}(x)$.

4. Ist der Keller nicht leer, so gilt: $S.\mathtt{push}(S.\mathtt{pop}())$ verändert den Keller nicht.

Beobachte: Auf die "Implementierung" des Kellers als Feld wird hier nicht mehr Bezug genommen. $\rightsquigarrow$ Altenativer (abstrakterer, mathematischerer) Ansatz über *abstrakte Datentypen / Algebren*

Keller-Algebra

(A) Syntaktische Beschreibung:

Ein *Keller* wird beschrieben durch ein Tupel $(D, \chi, \epsilon, \downarrow, \uparrow, \tau)$ mit
D Datentyp (Sorte); daraus abgeleitet: $stack(D) \rightsquigarrow$ mehrsortige Signatur
$\chi :\rightarrow stack(D)$
$\epsilon : stack(D) \rightarrow \mathbb{B}$
$\downarrow: stack(D) \times D \rightarrow stack(D)$
$\uparrow: stack(D) \rightarrow stack(D) \times D$
$\tau : stack(D) \rightarrow D$

(B) Semantik: Die obigen Symbolfolgen werden als Funktionen gedeutet.
Es gelten insbesondere folgende Gesetze (Axiome):
$\forall S \in stack(D) \forall x \in D(\uparrow (\downarrow (S, x)) = (S, x))$
$\forall S \in stack(D) \forall x \in D(\tau(\downarrow (S, x)) = x)$
$\forall S \in stack(D)(\neg\epsilon(S) \Rightarrow (\downarrow (\uparrow (S)) = S))$

Dynamische Keller

Beobachte: Die Keller-Algebra idealisiert, indem keine Unterläufe oder Überläufe expliziert werden.
Jedoch erscheinen Unterläufe unvermeidlich, wenn auf leerem Keller ein pop oder top ausgeführt wird.
Überläufe können (jedenfalls im Prinzip) vermieden werden, wenn man nicht die Maximalgröße des Kellers von vornherein festlegt (wie bei der statischen Implementierung oben), sondern dynamisch (flexibel) anlegt.
Dynamische Keller kann man mit Hilfe von Listen (s.u.) darstellen.

Was Keller können

Sehr schöne Beschreibungen zu diesem Thema finden Sie in “matheprisma”, genauer:
`http://www.matheprisma.de/Module/LinDatSt/pages/node8.htm`

Dort finden Sie auch hübsche Animationen für andere Datentypen wie Liste und Schlange.

Listen

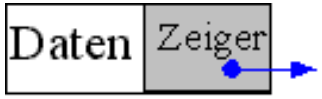
Hinweis: Listen in der Programmiersprache LISP

Listen sind neben den Einzelwerten (Atomen) die Basisdatenstruktur der Programmiersprache LISP.

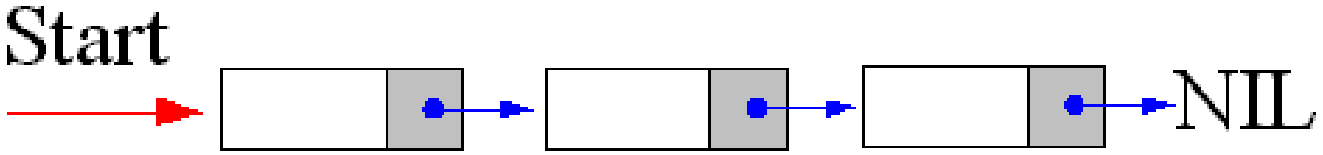
Programme sind Listen von Listen.

Man unterscheidet
einfach verkettete und doppelt verkettete Listen

Einfach verkettete Listen



Die einzelnen *Listenelemente* unterteilen sich in:
Speicherplatz für die Daten (Werte) sowie
einen *Verweis* (Zeiger, Referenz, Pointer) auf das nächste Listenelement.



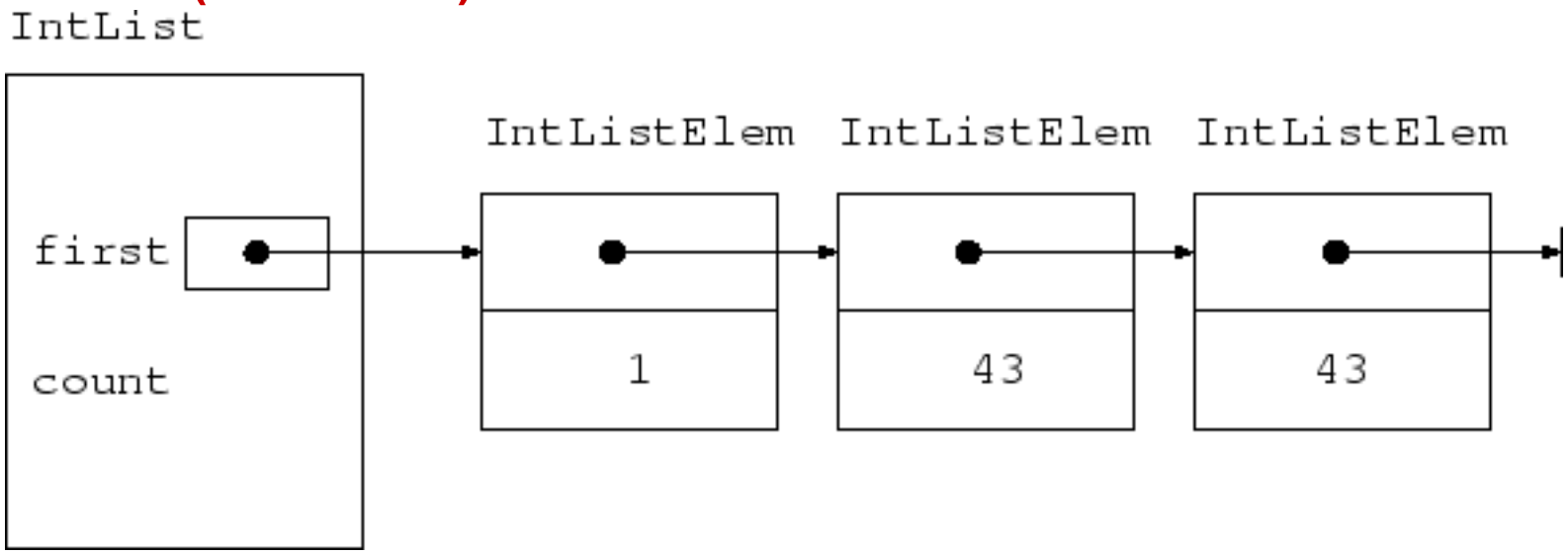
Sonderfälle:
Der *Kopf* (Start) ist ein Verweis auf das erste Listenelement.
Das letzte Listenelement enthält einen *leeren Verweis* (Nil, Null).

Listenelemente

Zugriff für Listenelement elem:
elem.val greift auf die Daten zu
elem.next enthält den Verweis auf den Nachfolger (oder Null).
C++ Beispiel für ein Listenelement einer Liste ganzer Zahlen:

```
struct IntListElem {
    int val;              // Daten zu diesem Element
    IntListElem* next;    // Zeiger auf nächstes Element
} ;
```

Liste im Bild (mit Zähler)



Zeiger

Verweise oder Zeiger kann man *dereferenzieren*, um auf die Objekte, auf die verwiesen wird, zugreifen zu können. Hinweis: Indirekte RAM-Adressierung

In C++ Notation schreiben wir

```
p->val
```

für einen Zugriff auf den Wert eines Listenelementes, dessen Adresse in der Zeigervariablen `p` gespeichert ist.
`p` ist also vom Typ `IntListElem*` (im vorigen Beispiel).
Die *leere Liste* wird meist durch einen Nullzeiger implementiert, und dies kann in C++ auch einfach durch einen Vergleich mit Null abgefragt werden.

Durchsuchen einer Liste nach (erstem) Vorkommen von Datum `x`

Ein kleines Programm in C++

```
struct IntList {
    int count;           // Anzahl Elemente in der Liste
    IntListElem* first; // Zeiger auf erstes Element der Liste
} ;
IntListElem* find_first_x (IntList l, int x)
{
    for (IntListElem* p=l.first; p!=0; p=p->next)
        if (p->val==x) return p;
    return 0;
}
```

C++ Notation ist sehr kompakt;
versuchen wir nochmals **das Ganze in Pseudo-Code** mit Tafelbeispiel:

Algorithm 3 Durchsuchen einer Liste ganzer Zahlen

Input(s): a list $l : \mathbb{Z} - list$, $x : \mathbb{Z}$

Output(s): pointer p to list element containing x (if found), NULL otherwise

Let p be a pointer to list element variable and $cont : \mathbb{B}$.

$p \leftarrow l.first$; $cont \leftarrow$ TRUE;

while $(p \neq \text{NULL}) \wedge cont$ **do**

  $cont \leftarrow (p \rightarrow val) \neq x$:

  $p \leftarrow (p \rightarrow next)$;

end while

return p;

Dynamische Keller und Listen

Wir gehen aus von Datentypen D-List und D-ListElem (analog zu eben).
Es sei also K vom Typ D-List und e von Typ D-ListElem.

$\chi :\rightarrow \mathrm{stack(D)}$	$\mathrm{K.count} \leftarrow 0;\ \mathrm{K.first} \leftarrow \mathrm{NULL};$
$\epsilon : \mathrm{stack(D)} \rightarrow \mathbb{B}$	$(0 = \mathrm{K.count});$
$\downarrow\!: \mathrm{stack(D)} \times \mathrm{D} \rightarrow \mathrm{stack(D)}$	$e \leftarrow (x, \mathrm{K.first});\ \mathrm{K.first} \leftarrow \pi(e);\ \mathrm{K.count}++;$
$\uparrow\!: \mathrm{stack(D)} \rightarrow \mathrm{stack(D)} \times \mathrm{D}$	$x \leftarrow \mathrm{K.first} \rightarrow val;\ \mathrm{K.first} \leftarrow \mathrm{K.first} \rightarrow next;\ \mathrm{K.count}--$
$\tau : \mathrm{stack(D)} \rightarrow \mathrm{D}$	$x \leftarrow \mathrm{K.first} \rightarrow val;$

Wieso ist das (evtl.) eine korrekte Implementierung der Kelleroperationen ?

$\pi(e)$ soll übrigens einen Verweis (Pointer) auf e liefern.

Speicherverwaltung allgemein

Hinweis: Die Einschränkung (“evtl.”) bei der Korrektheitsbehauptung bezieht sich auf die Speicherverwaltung.

Wenn wir richtig mit dynamischen Datentypen umgehen wollen, müssen wir (eigentlich) immer wieder neue Speicherbereiche anfordern (*allozieren*) und wieder freigeben.
Bei manchen Programmiersprachen (wie JAVA) ist das explizite (Anfordern und) Freigeben nicht nötig, weil eine *Speicherbereinigung* (“Speicher-Müllabfuhr”: *Garbage Collection*) dafür Sorge trägt, dass automatisch Speicherbereiche, auf die nicht mehr referenziert wird, freigegeben werden.
Diese Müllabfuhr sorgt außerdem (meist) dafür, dass tatsächlich benötigte (also aktuell referenzierte) Daten hintereinander weg im Speicher liegen.
Explizites Speicheranfordern und -freigeben (wie bei C[++]) ist programmtechnisch aufwändiger und fehleranfälliger, produziert aber oft effizienteren Code.

Speicherverwaltung konkret

Problem:

$\downarrow$: $\mathrm{stack(D)} \times \mathrm{D} \to \mathrm{stack(D)}$ sollte implementiert werden durch:

$e \leftarrow (\mathrm{x}, \mathrm{K}.first); \mathrm{K}.first \leftarrow \pi(e); \mathrm{K}.count + +;$

Nämlich: e ist “ganz normale Variable”, die später natürlich überschrieben werden könnte, und damit wird auch geändert, was über den Zeiger $\pi(e)$ angesprochen wird.

Diese Seiteneffekte sollte man tunlichst vermeiden.

Dazu sollte man sich neue Speicherbereiche mit Hilfe von `new` (in C++) besorgen und mit `delete` auch wieder aufräumen.

Speicherverwaltung konkret: Ein Exkurs

Betrachten wir folgendes C++ Programm

```
struct rational {
    int n; // Nenner
    int z; // Zähler
} ;
void main ()
{
    rational  q;
    rational* p;

    p = &q;         // p enthält Verweis auf den Speicherplatz von q
    (*p).n = 5;     // Zuweisung an Komponente n von q
    p->n   = 5;     // identische Abkürzung
}
```

Dynamische Speicherverwaltung mit *dynamischen Variablen* in C++: Exkurs II
Diese werden vom Programmierer explizit erzeugt und vernichtet.
Dazu dienen die Operatoren `new` und `delete`.
Dynamische Variablen haben keinen Namen und können nur indirekt über Zeiger bearbeitet werden:

```
int m;
rational* p;
p = new rational;
p->n = 4; p->z = 5;
m = p->n;
delete p; // Was wird überleben ?
```

Die Anweisung `p = new rational` erzeugt eine Variable vom Typ rational und weist deren Adresse dem Zeiger `p` zu. Man sagt auch, dass die Variable *dynamisch allokiert* wurde.
Dynamische Variablen sind notwendig, um Strukturen im Rechner zu erzeugen, deren Größe sich während der Rechnung ergibt.

Eine neue Implementierung von “push”

$\downarrow: \mathrm{stack}(\mathrm{D}) \times \mathrm{D} \to \mathrm{stack}(\mathrm{D})$ kann man besser implementieren durch:

```
IntListElem* p;
p = new IntListElem;
p->val = x; p->next = K.first;
K.first = p;
K.count++;
```

Überlegen Sie, wie “pop” ganz entsprechend dargestellt werden kann (und benutzen Sie “delete”).

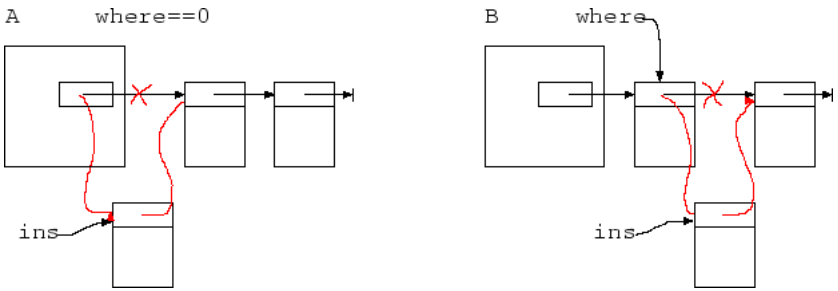
Was (einfach verkettete) Listen noch können

Zur Verwaltung von Listen wollen wir folgende Operationen vorsehen

- Erzeugen einer leeren Liste.
- Einfügen von Elementen an beliebiger Stelle.
- Entfernen von (beliebigen) Elementen.
- Durchsuchen der Liste.

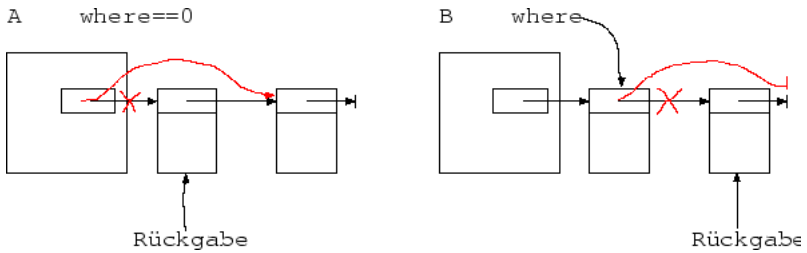
Interessant sind Punkt 2 und 3, wenn wir nicht nur den “Listen-Kopf” betrachten.

Einfügen in Liste




```
void insert_in_list (IntList* list,
IntListElem* where, IntListElem* ins)
{
  if (where==0) // A: füge am Anfang ein
 {
    ins->next = list->first;
    list->first = ins;
    list->count = list->count + 1;
 }
  else          // B: füge nach where ein
 {
    ins->next = where->next;
    where->next = ins;
    list->count = list->count + 1;
  }
}
```

Löschen aus Liste




```
IntListElem* remove_from_list
(IntList* list, IntListElem* where)
{
  IntListElem* p;// das entfernte Element
  if (where==0) {// A: entferne erstes Elem.
      p = list->first;
      if (p!=0) {
         list->first = p->next;
         list->count = list->count - 1;
  }  }
  else { // entferne Element nach where
      p = where->next;
      if (p!=0) {
         where->next = p->next;
         list->count = list->count - 1;
  }  }
  return p;
}
```

Algebraische Sicht

Eine (einfach verkettete) Liste ist ein Tupel $(D, \chi, \epsilon, \phi, \rho, \alpha, \kappa, \nu)$
D Datentyp (Sorte) für Elemente; daraus abgeleitet: $list(D) \rightsquigarrow$ mehrsortige Signatur
$\chi :\rightarrow list(D)$ (create, clear)
$\epsilon : list(D) \rightarrow \mathbb{B}$ (emptiness test)
$\phi : list(D) \rightarrow D$ (first element)
$\rho : list(D) \rightarrow list(D)$ (rest list)
$\alpha : list(D) \times D \rightarrow list(D)$ (append)
$\kappa : list(D) \times list(D)$ (concat)
$\nu : list(D) \rightarrow \mathbb{N}$ (number)
Wieder kann man diverse Gesetze (Axiome) fordern, wie z.B.:
Falls $\epsilon(L_1)$, so $\kappa(\alpha(L_1, x), L_2) = \alpha(L_2, x)$.
$\epsilon(L)$ gdw. $\nu(L) = 0$ gdw. $\phi(L)$ ist undef. gdw. $\rho(L)$ ist undef.
$\alpha(\rho(L), \phi(L)) = L$, $\nu(\kappa(L_1, L_2)) = \nu(L_1) + \nu(L_2)$, $\nu(\alpha(L, x)) = \nu(L) + 1$
$\epsilon(\chi())$ ist wahr.
Überlegen Sie: Mit welcher Komplexität lassen sich die Operationen implementieren ?

Literatur

Natürlich finden Sie viele (gute) Sachen (Lehrmittel) im Internet.
So orientiert sich meine Darstellung an:
`http://hal.iwr.uni-heidelberg.de/lehre/inf1-ws02/html/`

Die abstraktere Sicht finden Sie u.a. im Buch von Ralf Hartmut Güting.

Dort finden Sie auch eine alternative Implementierung einfach verketteter Listen.