Imperative Programmierung (IPR)

Kapitel 3: Listen

Univ.-Prof. Dr.-Ing. habil. Gero Mühl

Lehrstuhl für Architektur von Anwendungssystemen (AVA)
Fakultät für Informatik und Elektrotechnik (IEF)
Universität Rostock

Universität Rostock

Traditio et Innovatio

AVA

Inhalte

Kapitel 3.1

Einführung

Was ist eine Liste?

Definition 1 (Liste)

*Eine **Liste** speichert eine Sequenz von Elementen und bietet Operationen für die Manipulation der Liste sowie für den Zugriff auf die Elemente an.*

- Aus der Mathematik ist die Liste auch als **Tupel** bekannt.
- Die Elemente einer Liste können prinzipiell einem beliebigen primitiven Datentyp (z. B. `int`) angehören.
- Ein Element kann aber auch selbst wieder eine Liste sein oder einem anderen zusammengesetzten Datentyp angehören.
- Im Folgenden wird zunächst die Spezifikation der Liste betrachtet.
- Diese beschreibt präzise, *was* die Operationen einer Liste tun, ohne dabei festzulegen, *wie* sie es tun.
- Danach werden mehrere Implementierungsvarianten von Listen (Array-basierte Liste, verkettete Liste) besprochen.

Kapitel 3.2

Spezifikation

Operationen einer Liste

Operation	Beschreibung der Operation
`init`	Erzeugt neue Liste
`insert(e,l)`	Fügt Element **vorne** an Liste an
`empty(l)`	Prüft, ob Liste leer
`length(l)`	Liefert Länge der Liste
`head(l)`	Liefert vorderstes Element der Liste
`last(l)`	Liefert letztes Element der Liste
`nth(n,l)`	Liefert *n*-tes Element der Liste
`isin(e,l)`	Prüft, ob Element in Liste enthalten
`tail(l)`	Liefert Liste ohne vorderstes Element
`append(l,m)`	Hängt Liste *m* **hinten** an Liste *l* an

Ansatz für die Spezifikation

- Spezifikation legt die **von außen sichtbaren Eigenschaften** der Operationen fest.
- Für jede Listenoperation, die einen „sichtbaren“ Wert liefert, muss präzise festgelegt werden, *welchen* Wert sie liefert.
- Von außen sichtbar bei einer Liste sind
  - Boolesche Werte (Ergebnisse von `empty` und `isin`)
  - Listenelemente (Ergebnisse von `head`, `last` und `nth`)
  - Natürliche Zahlen (Ergebnis von `length`)
- *Nicht* sichtbar ist der interne Aufbau der Werte vom Typ `List` (Ergebnisse von `init`, `insert`, `tail` und `append`).
- Allerdings müssen die Eigenschaften dieser Funktionen trotzdem festgelegt werden.

Schrittweise Entwicklung der Spezifikation

- Der Datentyp `List` greift zurück auf Boolesche Werte, auf natürliche Zahlen und auf die Elemente, die in einer Liste vorliegen können:

$$[\mathbb{B}, \mathbb{N}, \mathit{Element}]$$

- Innerhalb von *Element* soll es auch ein **Fehlerelement** geben (z. B. für den Fall, dass auf die leere Liste zugegriffen wird):

$$\mid \quad \mathit{errorelement} \in \mathit{Element}$$

- Zur einfacheren Darstellung der Eigenschaften wird noch definiert:

$$\mid \quad \mathit{Element}_V = \mathit{Element} \setminus \{\mathit{errorelement}\}$$

Schrittweise Entwicklung der Spezifikation

- Die Menge der Listen wird definiert:

$$[List]$$

- Listen können mit folgenden Operationen erzeugt werden:

$$\begin{array}{|l} init : List \\ insert : Element \times List \longrightarrow List \end{array}$$

- Das Hinzufügen des Fehlerelements verändert eine Liste nicht:

$$\begin{array}{|l} \forall\, l : List \bullet \\ \qquad insert(errorelement, l) = l \end{array}$$

- Mit Hilfe der beiden Operationen *init* und *insert* können die Eigenschaften der weiteren Listenoperationen definiert werden.

Listenoperationen: `empty`

$$
\begin{array}{|l}
\hline
empty : List \longrightarrow \mathbb{B} \\
\hline
\forall\, e : Element_V;\ l : List \bullet \\
\quad empty(init) = True \\
\quad empty(insert(e, l)) = False
\end{array}
$$

Beispiel 1 (`empty`)

$$empty(init) = True$$

$$empty(insert(\underbrace{3}_{e}, \underbrace{insert(4, insert(5, init))}_{l})) = False$$

Listenoperationen: head

$$head : List \longrightarrow Element$$

$$\forall\, e : Element_V;\ l : List \bullet$$

$$head(init) = errorelement$$
$$head(insert(e, l)) = e$$

Beispiel 2 (head)

$$head(insert(\underbrace{3}_{e}, \underbrace{insert(4, insert(5, init))}_{l})) = 3$$

Listenoperationen: `tail`

- Obwohl *tail* keinen nach außen sichtbaren Wert zurück gibt, muss auch diese Funktion präzise spezifiziert werden.
- Sonst wäre z. B. das Ergebnis von $head(tail(l))$ unbestimmt.

$$tail : List \longrightarrow List$$

$$\forall\, e : Element_V;\ l : List \bullet$$

$$tail(init) = init$$
$$tail(insert(e, l)) = l$$

Beispiel 3 (`tail`)

$$tail(insert(\underbrace{3}_{e}, \underbrace{insert(4, insert(5, init))}_{l}))$$

$$= insert(4, insert(5, init))$$

Listenoperationen: `isin`

$$isin : Element \times List \longrightarrow \mathbb{B}$$

$$\forall\, e, f : Element_V;\ l : List \mid e \neq f \bullet$$

$$isin(errorelement, l) = False$$
$$isin(e, init) = False$$
$$isin(e, insert(e, l)) = True$$
$$isin(e, insert(f, l)) = isin(e, l)$$

Beispiel 4 (`isin`)

$$isin(\underbrace{4}_{e}, insert(\underbrace{3}_{f}, \underbrace{insert(4, insert(5, init))}_{l}))$$

$$= isin(\underbrace{4}_{e}, insert(\underbrace{4}_{e}, , \underbrace{insert(5, init)}_{l}))$$

$$= True$$

Listenoperationen: nth

$$nth : \mathbb{N} \times List \longrightarrow Element$$

$$\forall\, e : Element_V;\ n : \mathbb{N};\ l : List \bullet$$

$$nth(n, init) = errorelement$$
$$nth(0, insert(e, l)) = e$$
$$nth(n+1, insert(e, l)) = nth(n, l)$$

Beispiel 5 (nth)

$$nth(\underbrace{1}_{n+1}, insert(\underbrace{3}_{e}, \underbrace{insert(4, insert(5, init))}_{l}))$$

$$= nth(0, insert(\underbrace{4}_{e}, \underbrace{insert(5, init)}_{l}))$$

$$= 4$$

Listenoperationen: append

$$append : List \times List \longrightarrow List$$

$$\forall\, e : Element_V;\ l, m : List \bullet$$

$$append(init, m) = m$$
$$append(insert(e, l), m) = insert(e, append(l, m))$$

Beispiel 6 (append)

- Die Liste m wird *hinten* an die Liste l angehängt:

$$append(insert(\underbrace{1}_{e}, \underbrace{insert(2, init)}_{l}), \underbrace{insert(3, insert(4, init))}_{m})$$
$$= insert(1, append(insert(2, init), insert(3, insert(4, init))))$$
$$= insert(1, insert(2, append(init, insert(3, insert(4, init)))))$$
$$= insert(1, insert(2, insert(3, insert(4, init))))$$

Listenoperationen: `last`

$$last : List \longrightarrow Element$$

$$\forall\, e, f : Element_V;\ l : List \bullet$$

$$last(init) = errorelement$$
$$last(insert(e, init)) = e$$
$$last(insert(e, insert(f, l))) = last(insert(f, l))$$

Beispiel 7 (`last`)

$$last(insert(\underbrace{3}_{e}, insert(\underbrace{4}_{f}, \underbrace{insert(5, init)}_{l})))$$

$$= last(insert(\underbrace{4}_{e}, insert(\underbrace{5}_{f}, \underbrace{init}_{l})))$$

$$= last(insert(\underbrace{5}_{e}, init))$$

$$= 5$$

Listenoperationen: `length`

$$length : List \longrightarrow \mathbb{N}$$

$$\forall\, e : Element_V;\ l : List \bullet$$

$$length(init) = 0$$
$$length(insert(e, l)) = 1 + length(l)$$

Beispiel 8 (`length`)

$$\begin{aligned}
&length(insert(3, insert(4, insert(5, init))))\\
&= 1 + length(insert(4, insert(5, init)))\\
&= 1 + 1 + length(insert(5, init))\\
&= 1 + 1 + 1 + length(init)\\
&= 1 + 1 + 1 + 0\\
&= 3
\end{aligned}$$

Beweis von Eigenschaften

- Mit den definierten Funktionen können Eigenschaften des Datentyps Liste mit den üblichen Beweismethoden gezeigt werden.
- Für nicht-leere Listen $m = \mathit{insert}(e, l)$ lässt sich zum Beispiel die folgende Eigenschaft nachweisen:

$$\mathit{insert}(\mathit{head}(m), \mathit{tail}(m)) = m$$

- Beweis:

$$\begin{aligned}
&\mathit{insert}(\mathit{head}(m), \mathit{tail}(m)) \\
&= \mathit{insert}(\mathit{head}(\mathit{insert}(e, l)), \mathit{tail}(\mathit{insert}(e, l))) && \text{[Annahme über } m\text{]} \\
&= \mathit{insert}(e, \mathit{tail}(\mathit{insert}(e, l))) && \text{[Def. } \mathit{head}\text{]} \\
&= \mathit{insert}(e, l) && \text{[Def. } \mathit{tail}\text{]} \\
&= m
\end{aligned}$$

Ausführbare Spezifikation der Liste

```
module List where

import Prelude hiding (init,tail,head,last,length)
type Element = Int
errorelement = -1

data List = Empty | App(Element,List)
     deriving Show

init   :: List
insert :: (Element,List) -> List

init = Empty

insert(e,l) = if e == errorelement then l
              else App(e,l)
```

Ausführbare Spezifikation der Liste

```
isin   :: (Element, List) -> Bool
empty  :: List -> Bool
head   :: List -> Element
tail   :: List -> List

empty(Empty)       = True
empty(App(e,l))    = False

head(Empty)        = errorelement
head(App(e,l))     = e

tail(Empty)        = Empty
tail(App(e,l))     = l

isin(e,Empty)      = False
isin(e,App(f,l)) = if e == f then True
                   else isin(e,l)
```

Ausführbare Spezifikation der Liste

```
nth    :: (Element, List) -> Element
append :: (List, List) -> List
last   :: List -> Element

nth(n,Empty)       = errorelement
nth(0,App(e,l))    = e
nth(n,App(e,l))    = nth(n-1,l)

append(Empty,m)    = m
append(App(e,l),m) = App(e,append(l,m))

last(Empty)              = errorelement
last(App(e,Empty))       = e
last(App(e,App(f,l)))    = last(insert(f,l))
```

Ausführbare Spezifikation der Liste

```
length :: List -> Element

length(Empty)     = 0
length(App(e,l)) = 1 + length(l)

tuplelist(Empty)          = []
tuplelist(App(e,Empty)) = [e]
tuplelist(App(e,l))       = [e] ++ tuplelist(l)
```

- Mit diesen Definitionen liefert beispielsweise

  $$\mathit{tuplelist}(\mathit{append}(\mathit{insert}(3, \mathit{insert}(4, \mathit{init})), \mathit{insert}(5, \mathit{insert}(6, \mathit{init}))))$$

  die Ausgabe

  $$[3, 4, 5, 6].$$

- Hausaufgabe: Besorgen Sie sich einen Haskell-Interpreter oder -Compiler (`www.haskell.org`) und probieren Sie das selber aus!

Kapitel 3.3

Vorüberlegungen zur Implementierung

Grundsätzliches zur Implementierung

- Im Folgenden werden einige Implementierungsvarianten von Listen näher besprochen.
- Die Implementierung erfolgt in der Programmiersprache C.
- Die exakte Semantik der Listenoperationen muss bei imperativer Programmierung besonders beachtet werden.
- Ausgangspunkt ist die Spezifikation des abtrakten Datentyps Liste.
- Teilweise wird aber von der Spezifikation leicht abgewichen.

Nicht-Destruktive Listenoperationen

- Betrachten wir exemplarisch die Operation `append`:

```
list* list_append(list* l, list* m)
```

- Laut Spezifikation soll `append` eine Liste *n* zurückgeben, welche die Elemente der Listen *m* und *l* enthält.
- Angenommen *m* und *l* werden hierbei nicht verändert.
- Dann können alle drei Listen unabhängig voneinander und entsprechend der Spezifikation weiter genutzt werden.
- Eine derartige Implementierung einer Operation wird **nicht-destruktiv** genannt.
- Nicht-destruktive Implementierungen sind oft ineffizient, da sie meist das **Kopieren von Datenstrukturen** erfordern.

Destruktive Listenoperationen

- Eine Alternative sind **destruktive** Operationen, die einige oder auch alle Parameter durch ihre Ausführung verändern.
- Oft können die Parameter dann nicht mehr sinnvoll genutzt werden.
- Beispielsweise könnte `append` die Elemente der Liste *m* an die Liste *l* anhängen und einen Zeiger auf die veränderte Liste *l* zurückliefern.
- In diesem Fall ist die ursprüngliche Liste *l* nicht mehr vorhanden.
- Die Liste *m* ist hingegen unversehrt geblieben.
- Eine destruktive Implementierung von `append` entspricht weitgehend der **objektorientierten Programmierung**.
- Bei dieser wird ein Objekt (hier die Liste *l*) durch die Ausführung einer Operation manipuliert.

Kapitel 3.4

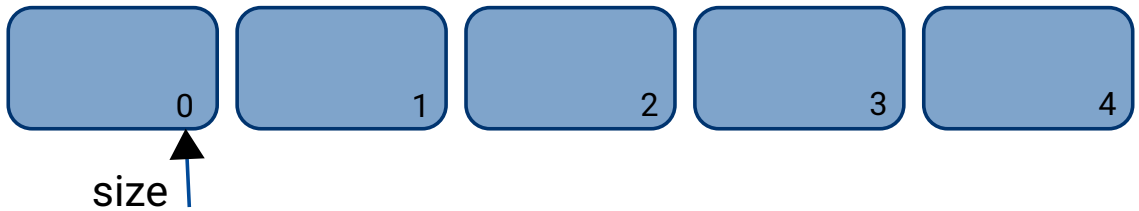
Array-basierte Liste

Realisierung als Array

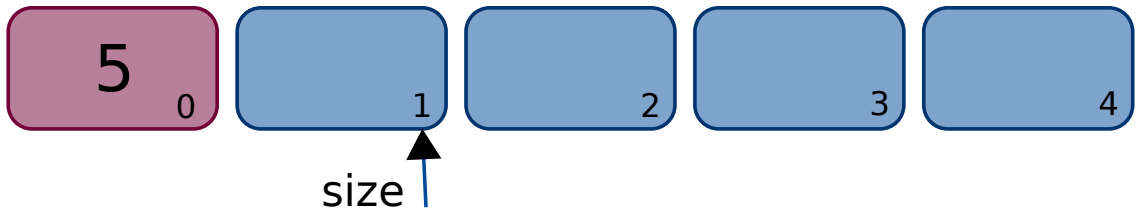
- Die Array-basierte Liste verwendet ein Array zur Speicherung der Listenelemente, verbirgt dieses aber vor den Nutzern.
- Der Zugriff erfolgt ausschließlich über **Schnittstellenmethoden**, die die Implementierung kapseln.
- Ein neues Element wird beim Einfügen *hinten* in das Array nach dem bisher *letzten* Element des Arrays eingefügt.
- **Achtung: Das erste Element der Liste ist also jeweils das letzte belegte Element des Arrays und umgekehrt.**
- Dieses Vorgehen hat den Vorteil, dass die Operationen `insert` und `tail` effizient (mit konstantem Aufwand) ausgeführt werden können.

Realisierung als Array (head, tail)

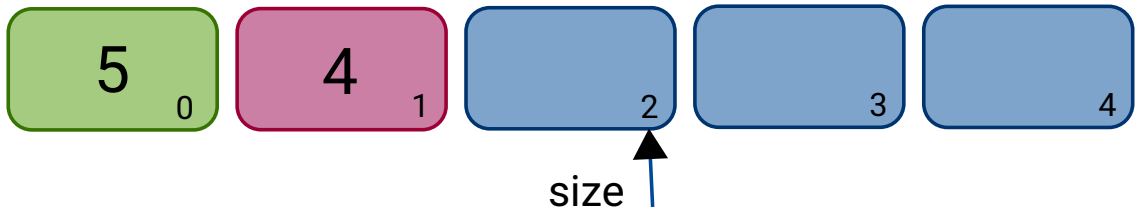
- Anfangszustand



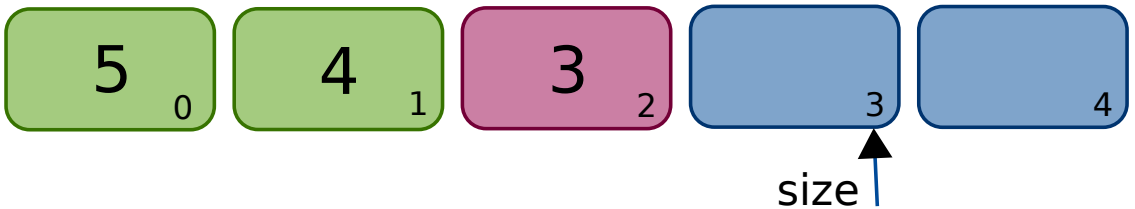
- Zustand nach Einfügen der Zahl 5



- Zustand nach Einfügen der Zahlen 5 und 4



- Zustand nach Einfügen der Zahlen 5, 4 und 3



Implementierung

```
#define LIST_ERROR_ELEMENT INT_MIN
#define LIST_MAX_ELEMENTS 100

typedef int element;

struct _list {
  int size;
  element elements[LIST_MAX_ELEMENTS];
};

typedef struct _list list;

static int list_no_mallocs = 0;
static int list_no_frees = 0;
```

Implementierung

```
void* list_malloc(size_t size) {
  list_no_mallocs++;
  return malloc(size);
}

void list_free(void* ptr) {
  if (ptr != NULL) list_no_frees++;
  free(ptr);
}

int list_get_mallocs() {
  return list_no_mallocs;
}

int list_get_frees() {
  return list_no_frees;
}
```

Implementierung

```
void list_print_mallocs_frees() {
  printf("list_no_mallocs = %i\n", list_get_mallocs());
  printf("list_no_frees = %i\n", list_get_frees());
  if (list_get_mallocs() > list_get_frees())
    fprintf(stderr,
        "memory leak in list implementation\n");
  else if (list_get_mallocs() < list_get_frees())
    fprintf(stderr,
        "double free in list implementation!\n");
}
```

Implementierung init, destroy und empty

```
list* list_init() {
  list* l = list_malloc(sizeof(list));
  l->size = 0;
  return l;
}

void list_destroy(list* l) {
  list_free(l);
}

int list_empty(list* l) {
  if (l == NULL)
    fprintf(stderr, "empty: list is NULL\n");
  return l->size == 0;
}
```

Implementierung insert

```
list* list_insert(element e, list* l) {
  if (e == LIST_ERROR_ELEMENT)
    fprintf(stderr,
        "insert: trying to insert error element!\n");
  else if (l->size >= LIST_MAX_ELEMENTS) {
    fprintf(stderr,
        "insert: list capacity exceeded!\n");
  else
    l->elements[l->size++] = e;

  return l;
}
```

Implementierung nth

```
element list_nth(int n, list* l) {
  if (n < 0)
    fprintf(stderr, "nth: negative index!\n");
  else if (n < l->size)
    return l->elements[l->size - n - 1];
  else
    fprintf(stderr, "nth: list too short!\n");
  return LIST_ERROR_ELEMENT;
}
```

Implementierung head und tail

```
element list_head(list* l) {
  return list_nth(0, l);
}

list* list_tail(list* l) {
  if (!list_empty(l))
    l->size--;
  return l;
}
```

Implementierung isin

```
int list_isin(element e, list* l) {
  for (int i = 0; i < l->size; i++)
    if (l->elements[i] == e)
      return 1;
  return 0;
}
```

Implementierung append

```
list* list_append(list* l, list* m) {
  if (l->size + m->size > LIST_MAX_ELEMENTS)
    fprintf(stderr,
        "append: list capacity exceeded!\n");
  else if (l == m)
    fprintf(stderr,
        "append: appending the list to itself!\n");
  else if (m->size != 0) {
    l->size = l->size + m->size;
    for (int i = l->size - 1; i >= 0; i--)
      l->elements[i] = i >= m->size ?
          l->elements[i - m->size] : m->elements[i];
  }
  return l;
}
```

Implementierung last und length

```
element list_last(list* l) {
  if (!list_empty(l))
    return l->elements[0];
  fprintf(stderr, "last: empty list!\n");
  return LIST_ERROR_ELEMENT;
}

int list_length(list* l) {
  return l->size;
}
```

Implementierung show

```
void list_show(list* l) {
  if (list_empty(l)) {
    printf("<>\n");
    return;
  }
  printf("<");
  for (int i = l->size - 1; i >= 0; i--)
    if (i != 0)
      printf("%i,", l->elements[i]);
    else
      printf("%i>\n", l->elements[i]);
}
```

Implementierung print und println

```
void list_print(list* l) {
  for (int i = l->size - 1; i >= 0; i--)
    printf("insert(%i, ", l->elements[i]);
  printf("init");
  for (int i = 0; i < l->size; i++)
    printf(")");
}

void list_println(list* l) {
  list_print(l);
  printf("\n");
}
```

Exemplarische main-Methode

```
int main() {
  list* l = list_init(); // <>
  l = list_insert(4, l); // <4>
  l = list_insert(5, l); // <5, 4>
  l = list_insert(6, l); // <6, 5, 4>

  list* m = list_init(); // <>
  m = list_insert(7, m); // <7>
  m = list_insert(8, m); // <8, 7>

  list* a = list_append(l, m);
  list_show(a);          // prints <6, 5, 4, 8, 7>

  list_destroy(l);
}
```

Bemerkungen zur Implementierung

- Diverse Operationen haben als Parameter eine Liste und geben eine Liste als Ergebnis zurück (z. B. `insert` und `tail`).
- Die zurückgegebene Liste ist aber keine *neue* Liste, sondern *dieselbe*.

- Die Operation `append` hat zwei Listen als Parameter.
- Hierbei bezeichnen die Liste, die als erster Parameter übergeben wird, und die Liste, die zurückgegeben wird, auch *dieselbe* Liste.

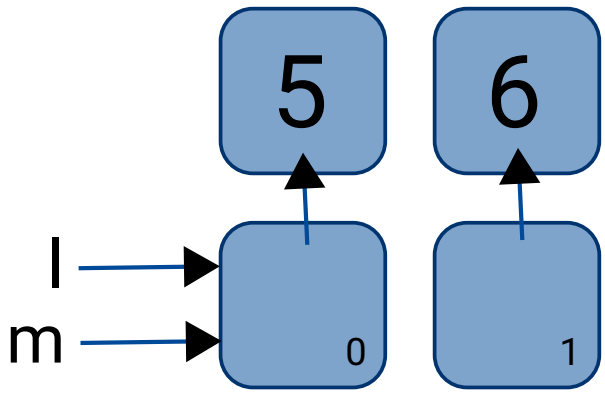
- Eine neue Liste wird also nur mittels `init` erzeugt.
- Diese Vorgehensweise wird aus Effizienzgründen gewählt, da sonst bei jeder der o. g. Operation eine Kopie der Liste erzeugt werden müsste.

Bemerkungen zur Implementierung

Beispiel 9

```
list* l = list_insert(7,list_insert(6,
                      list_insert(5,list_init())));
printf("%i\n", list_length(l));                    // 3
list* m = list_tail(l);
printf("%i\n", list_length(l));                    // 2
printf("%i\n", list_length(m));                    // 2
printf("%s\n", (m == l) ? "1" : "0");              // 1
```

- Die folgende Abbildung zeigt den Zustand nach der dritten Anweisung



Implementierung copy

- Soll die ursprüngliche Liste trotz destruktiver Operationen erhalten bleiben, muss diese vor dem Aufruf kopiert werden.
- Hierzu wird die Operation copy eingeführt.

```
list* list_copy(list* l) {
  // create new list
  list* m = list_init();
  // copy elements from old to new list
  for (int i = 0; i < l->size; i++)
    m->elements[i] = l->elements[i];
  m->size = l->size;
  // return new list
  return m;
}
```

Verwendung von copy

Beispiel 10

```
list* k = list_insert(7,list_insert(6,
                      list_insert(5,list_init())));
list* j = list_tail(list_copy(k));
printf("%i\n", list_length(k));                    // 3
printf("%i\n", list_length(j));                    // 2
printf("%s\n", (k == j) ? "1" : "0");              // 0
```

Bewertung der Implementierung

- Die feste Größe des Arrays macht Array-basierte Listen inflexibel und verursacht hohen Speicherverbrauch bei kleinen Listen.
- Anpassung bei wachsenden (schrumpfenden) Listen erfordert Umkopieren vom aktuellen in ein größeres (kleineres) Array.
- Das Einfügen von Elementen erfordert in diesen Fällen dann linearen Aufwand.
- Das Aneinanderhängen von Listen und das Suchen eines Elements erfordern in jedem Fall linearen Aufwand.
- Das Einfügen eines Elements an das Ende der Liste sowie das Löschen eines Elements an beliebiger Stelle wären ebenfalls ineffizient.
- Beides erfordert das Verschieben der restlichen Listenelemente.
- Ein Vorteil neben der einfachen Implementierung ist der Zugriff auf beliebige Elemente mit konstantem Aufwand.

Aufwand für die Listenoperationen

Funktion	Aufwand
`insert(e,l)`	konstant
`empty(l)`	konstant
`length(l)`	konstant
`head(l)`	konstant
`last(l)`	konstant
`nth(n,l)`	konstant
`isin(e,l)`	linear
`tail(l)`	konstant
`append(l,m)`	linear

Kapitel 3.5

Verkettete Liste

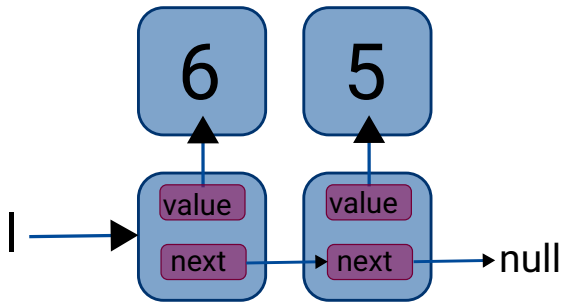
Realisierung als verkettete Liste

- Eine verkettete Liste verwendet statt eines Arrays eine mittels Zeigern verkettete **dynamische Datenstruktur**.
- Sie basiert auf der Idee: Eine Liste ist entweder leer oder sie besteht aus einem Element und einem Zeiger, der wieder auf eine Liste zeigt.

```
struct _list {
  element value;
  struct _list* next;
};

typedef struct _list list;
```

- Liste nach Einfügen der Elemente 5 und 6:



Implementierung

```
#define LIST_ERROR_ELEMENT INT_MIN

typedef int element;

struct _list {
  element value;
  struct _list* next;
};

typedef struct _list list;
```

Implementierung `init` und `empty`

```
list* list_init() {
  return NULL;
}

int list_empty(list* l) {
  return l == NULL;
}
```

- Die leere Liste wird also durch `NULL` repräsentiert.

Implementierung insert

```
list* list_insert(element e, list* l) {
  if (e == LIST_ERROR_ELEMENT) {
    fprintf(stderr,
        "insert: trying to insert error element!\n");
    return l;
  }
  list* m = list_malloc(sizeof(list));
  m->value = e;
  m->next = l;
  return m;
}
```

Implementierung head und tail

```
element list_head(list* l) {
  if (!list_empty(l))
    return l->value;

  fprintf(stderr, "head: empty list!\n");
  return LIST_ERROR_ELEMENT;
}

list* list_tail(list* l) {
  return !list_empty(l) ? l->next : l;
}
```

Implementierung nth

```
element list_nth(int n, list* l) {
  if (n < 0) {
    fprintf(stderr, "nth: negative index!\n");
    return LIST_ERROR_ELEMENT;
  }
  if (list_empty(l)) {
    fprintf(stderr, "nth: list too short!\n");
    return LIST_ERROR_ELEMENT;
  }
  if (n == 0)
    return l->value;

  return list_nth(n - 1, list_tail(l));
}
```

Implementierung isin

```
int list_isin(element e, list* l) {
  if (list_empty(l))
    return 0;

  return list_head(l) == e ?
              1 : list_isin(e, list_tail(l));
}
```

Implementierung last und length

```
element list_last(list* l) {
  if (list_empty(l)) {
    fprintf(stderr, "last: empty list!\n");
    return LIST_ERROR_ELEMENT;
  }

  return list_empty(l->next) ?
              l->value : list_last(l->next);
}

int list_length(list* l) {
  return list_empty(l) ?
              0 : list_length(list_tail(l)) + 1;
}
```

Implementierung copy

```
list* list_copy(list* l) {
  if (list_empty(l))
    return list_init();

  return list_insert(list_head(l),
                     list_copy(list_tail(l)));
}
```

Implementierung destroy

```
void list_destroy(list* l) {
  if (!list_empty(l))
    list_destroy(list_tail(l));

  list_free(l);
}
```

- Durch die Rekursion werden die Listenelemente von hinten nach vorne freigegeben.
- Das letzte Element wird also zuerst und das erste zuletzt freigegeben.

Implementierung append

```
list* list_append(list* l, list* m) {
  if (list_empty(l))
    return m;

  return list_insert(
              list_head(l),
              list_append(list_tail(l), m));
}
```

- append soll die Elemente der Liste *m* hinten an die Liste *l* anhängen.
- Elemente können aber nur (mittels `insert`) vorne in eine Liste eingefügt werden.
- Daher wird von der Liste *m* ausgegangen und die Elemente der Liste *l* rekursiv vorne in *m* eingefügt $\rightarrow$ linearer Aufwand.
- Die Listen *l* bleibt und *m* bleiben dabei intakt.

Erläuterung von append

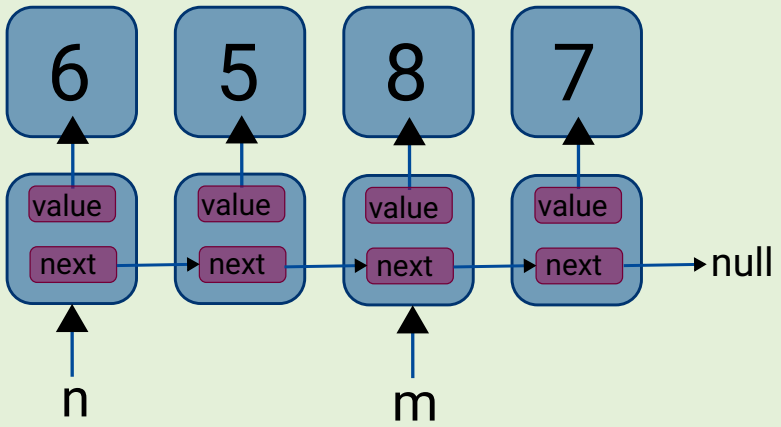
Beispiel 11

```
list* l = list_init();
l = list_insert(5, l);
l = list_insert(6, l);

list* m = list_init();
m = list_insert(7, m);
m = list_insert(8, m);

list* n = list_append(l, m);

list_destroy(l);
list_destroy(n); // also destroys the elements of m
```



Exemplarische main-Methode

```
int main(int argc, char* argv[]) {
  list* l = list_init(); // <>
  l = list_insert(4, l); // <4>
  l = list_insert(5, l); // <5, 4>
  l = list_insert(6, l); // <6, 5, 4>
  list_show(l);
  list_println(l);

  printf("head(l) = %i\n", list_head(l)); // prints 6
  printf("last(l) = %i\n", list_last(l)); // prints 4

  list_destroy(l);
 }
```

Bewertung der Implementierung

- Die vorgestellte Implementierung lehnt sich mit ihrer rekursiven Implementierung der Operationen stark an die Spezifikation an.
- Daher ist ihre Korrektheit relativ einfach nachzuvollziehen.
- Allerdings sind iterative Implementierungen häufig effizienter.

- Sie unterscheidet nicht zwischen Listen und Listenelementen.
- Daher können keine Metainformationen (z. B. die Länge der Liste) in der Liste gespeichert werden. Um die Länge der Liste zu bestimmen, muss daher die gesamte Liste durchlaufen werden.
- Für das Einfügen von Elementen an das Ende der Liste wäre die gesamte Liste zu durchlaufen, da kein Zeiger auf das letzte Element der Liste gespeichert werden kann.

Bewertung der Implementierung

- Problematisch ist, dass bei dieser Art der Implementierung Listenelemente in mehrerer Listen enthalten sein können.
- Das Löschen von Elementen führt in diesem Fall zu Problemen, da das Element aus mehreren Listen gelöscht werden würde.
- Beim Freigeben einer Liste kann es passieren, dass Listenelemente freigegeben werden, die auch Teil anderer Listen sind.
- Wird eine solche andere Liste dann freigegeben, kommt es in der Regel zu einem Speicherzugriffsfehler.
- Alternativ verhält sich das Programm unvohersagbar.

Aufwand für die Listenoperationen

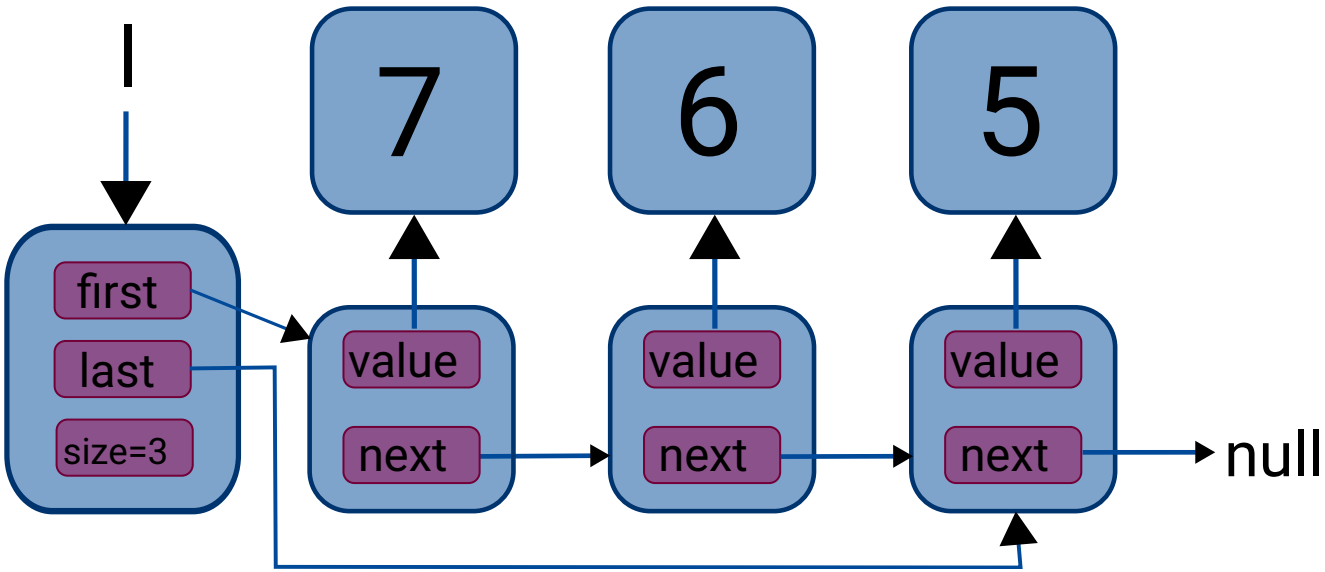
Funktion	Aufwand
`insert(e,l)`	konstant
`empty(l)`	konstant
`length(l)`	linear
`head(l)`	konstant
`last(l)`	linear
`nth(n,l)`	linear
`isin(e,l)`	linear
`tail(l)`	konstant
`append(l,m)`	linear

Kapitel 3.6

Verkettete Liste mit gekapselten Elementen

Verkettete Liste mit gekapselten Elementen

- Diese Implementierung unterscheidet zwischen einer Liste und ihren Listenelementen.
- Die Liste hat einen Zeiger `first` auf ihr erstes und einen Zeiger `last` auf ihr letztes Element. Sie speichert auch ihre aktuelle Länge.
- Die Elemente sind untereinander mittels des Zeigers `next` verkettet.
- Die Abb. zeigt eine Liste `l` nach Einfügen der Elemente 5, 6 und 7:



Verkettete Liste mit gekapselten Elementen

Neue Funktion	Beschreibung der Funktion
`add(e, l)`	Anfügen eines Elements hinten an die Liste
`delete(e, l)`	Löschen des ersten Vorkommens eines Elements aus der Liste

- Bei den Operationen `insert`, `add`, `delete` und `tail` wird die Liste, die als Parameter übergeben wird, verändert und zurückgegeben.
- Bei der Operation `append` gilt dies für die erste Liste, die als Parameter übergeben wird. Die zweite Liste bleibt unverändert.
- Daher sind alle Zeiger, die aus einer Liste hervorgehen, identisch.
- Neue Listen können also wie bei der Array-basierten Liste nur mittels `init` erzeugt werden.
- Mit `copy` kann eine Kopie einer Liste erzeugt werden.

Implementierung

```
#define LIST_ERROR_ELEMENT INT_MIN

typedef int element;

struct _node {
  element value;
  struct _node* next;
};
typedef struct _node node;

struct _list {
  int size;
  node* first;
  node* last;
};
typedef struct _list list;
```

Implementierung init und empty

```
list* list_init() {
  list* l = list_malloc(sizeof(list));
  l->size = 0;
  l->first = NULL;
  l->last = NULL;
  return l;
}

int list_empty(list* l) {
  return l->size == 0;
}
```

- Ob eine Liste leer ist, wird hier also aus ihrer Größe abgeleitet.

Implementierung insert

```
list* list_insert(element e, list* l) {
  if (e == LIST_ERROR_ELEMENT) {
    fprintf(stderr,
        "insert: trying to insert error element!\n");
    return l;
  }
  node* n = list_malloc(sizeof(node));
  n->value = e;
  n->next = l->first;
  if (list_empty(l))
    l->last = n;
  l->first = n;
  l->size++;
  return l;
}
```

Implementierung head

```
element list_head(list* l) {
  if (!list_empty(l))
    return l->first->value;

  fprintf(stderr, "head: empty l!\n");
  return LIST_ERROR_ELEMENT;
}
```

Implementierung nth

```
element list_nth(int n, list* l) {
  if (n < 0) {
    fprintf(stderr, "nth: negative index!\n");
    return LIST_ERROR_ELEMENT;
  }
  if (n >= l->size) {
    fprintf(stderr, "nth: list too short!\n");
    return LIST_ERROR_ELEMENT;
  }

  node* no = l->first;
  for (; n > 0; n--)
    no = no->next;
  return no->value;
}
```

Implementierung isin

```
int list_isin(element e, list* l) {
  if (e == LIST_ERROR_ELEMENT) {
    fprintf(stderr,
        "isin: searching for error element!\n");
    return 0;
  }

  for (node* n = l->first; n != NULL; n = n->next)
    if (n->value == e)
      return 1;

  return 0;
}
```

Implementierung last und length

```
element list_last(list* l) {
  if (!list_empty(l))
    return l->last->value;

  fprintf(stderr, "last: empty list!\n");
  return LIST_ERROR_ELEMENT;
}

int list_length(list* l) {
  return l->size;
}
```

Implementierung add

```
list* list_add(element e, list* l) {
  if (e == LIST_ERROR_ELEMENT) {
    fprintf(stderr,
        "add: trying to insert error element!\n");
    return l;
  }
  node* n = list_malloc(sizeof(node));
  n->value = e;
  n->next = NULL;
  if (list_empty(l))
    l->first = n;
  else
    l->last->next = n;
  l->last = n;
  l->size++;
  return l;
}
```

Implementierung delete

```
list* list_del(element e, list* l) {
  node* prev = NULL;
  node* cur = l->first;
  for (; cur != NULL; prev = cur, cur = cur->next) {
    if (cur->value == e) {
      if (prev == NULL)
        l->first = cur->next;
      else
        prev->next = cur->next;
      l->size--;
      list_free(cur);
      return l;
    }
  }
  return l;
}
```

Implementierung `tail`

```
list* list_tail(list* l) {
  if (!list_empty(l)) {
    node* tmp = l->first;
    l->first = l->first->next;
    list_free(tmp);
    l->size--;
  }
  return l;
}
```

- Bei dieser Implementierung löscht *tail* also tatsächlich das erste Listenelement der Liste und gibt dessen Speicher frei.

Implementierung append

```
list* list_append(list* l, list* m) {
  for (node* n = m->first; n != NULL; n = n->next)
    list_add(n->value, l);

  return l;
}
```

- append hängt mittels add nacheinander die Elemente der Liste *m* hinten an die Liste *l* an.
- Die Liste *m* bleibt intakt.

Implementierung destroy

```
void list_destroy(list* l) {
  while (!list_empty(l))
    list_tail(l);

  list_free(l);
}
```

- destroy löscht solange mittels *tail* das jeweils erste Listenelement, bis die Liste leer ist.
- Im Anschluss wird noch die Liste selber freigegeben.

Implementierung copy

```
list* list_copy(list* l) {
  list* m = list_init();
  for (node* n = l->first; n != NULL; n = n->next)
    list_add(n->value, m);

    return m;
}
```

- *copy* erzeugt mittels *init* eine neue Liste *m* und fügt in diese nacheinander die Elemente der Liste *l* ein.
- Da *l* von vorne nach hinten durchgegangen wird, muss das jeweilige Element mittels *add* hinten an die neue Liste *m* angefügt werden.

Bewertung der Implementierung

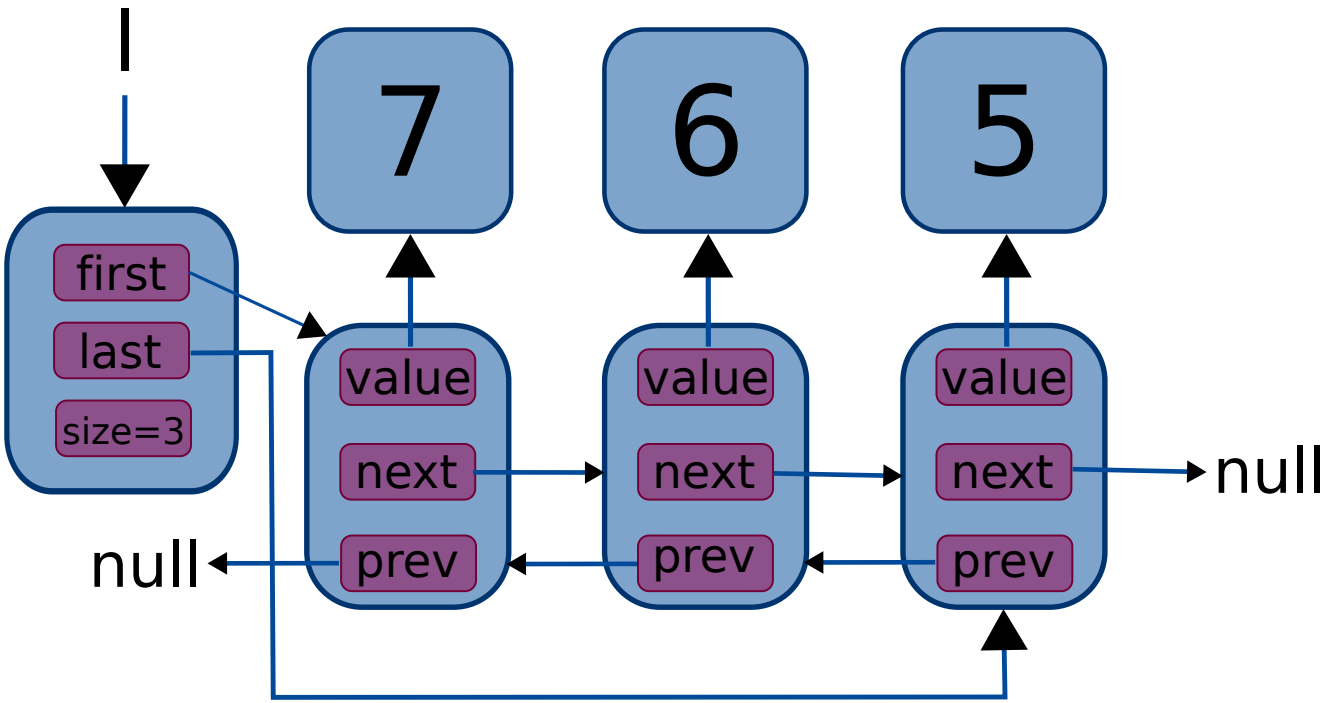
- Die Implementierung vermeidet Rekursion.
- Sie setzt stattdessen auf Iteration.
- Durch die Unterscheidung zwischen einer Liste und ihren Elementen können Metainformationen gespeichert werden.
- Hier: der Zeiger auf das letzte Element sowie die Länge der Liste.
- Daher erfordern `add` und `size` nur konstanter Aufwand.
- Daneben wird das Löschen von Elementen mit `delete` unterstützt.

Aufwand für die Listenoperationen

Funktion	Aufwand
`insert(e,l)`	konstant
`empty(l)`	konstant
`length(l)`	konstant
`head(l)`	konstant
`last(l)`	konstant
`nth(n,l)`	linear
`isin(e,l)`	linear
`tail(l)`	konstant
`append(l,m)`	linear
`add(e,l)`	konstant
`delete(e,l)`	linear

Doppelt verkettete Listen

- Die Elemente enthalten zusätzlich zum Zeiger auf das nächste Element auch einen Zeiger auf das vorige Element.



Exemplarische Fragen zur Lernkontrolle

1. Wozu dient eine Liste?
2. Welche Operation bietet eine Liste typischerweise an?
3. Spezifizieren Sie alle grundlegenden Listenoperationen!
4. Erläutern Sie die drei Implementierungsvarianten für Listen!
5. Worin besteht der Unterschied zwischen desktruktiven und nicht-destruktiven Operationen?
6. Welche Implementierungvariante orientiert sich am stärksten an der Spezifikation?
7. Warum soll auf eine Liste (oder auch einen anderen Datentyp) nur über die Schnittstellenoperationen zugegriffen werden?

Vielen Dank für Ihre Aufmerksamkeit!

Univ.-Prof. Dr.-Ing. Gero Mühl

gero.muehl@uni-rostock.de
https://www.ava.uni-rostock.de