Imperative Programmierung

Übung 11: Listen

Justin Kreikemeyer
Informatik, Uni Rostock

Leitfragen

- Wiederholung: Wie beschreibt man einen ADT?
- Was ist eine Liste?
- Wie spezifiziert man eine Liste?
- Wie implementiert man Listen (imperativ) in C?

Motivation

```c
// Implementierung 1
typedef double vector2d[2];
void scale(vector2d v, double c) {
  v[0] *= c; v[1] *= c;
}

// Implementierung 2
typedef struct { int x; int y; }* vector2d;
void scale(vector2d v, double c) {
  v->x *= c; v->y *= c;
}
```

Gibt es einen Unterschied zwischen den beiden Implementierungen?

Motivation

```c
// Implementierung 1
typedef double vector2d[2];
void scale(vector2d v, double c) {
  v[0] *= c; v[1] *= c;
}

// Implementierung 2
typedef struct { int x; int y; }* vector2d;
void scale(vector2d v, double c) {
  v->x *= c; v->y *= c;
}
```

Gibt es einen Unterschied zwischen den beiden Implementierungen?

→ Nur in der Repräsentation des Vektors!

→ Semantik der Operation gleich

→ Motivation ADT: Spezifikation der Operationen unabhängig von Repräsentation

Abstrakte Datentypen (ADTs)

- Beschreiben, die Semantik (**Was** ein Algorithmus tun soll)
- Aber **nicht**, die Implementierung (**Wie** es der Algorithmus tun soll)

- ADT ist ein Paar (Σ,E)
  - Signatur Σ ist ein Paar (S, F)
    - S ist eine Menge von Sorten (S = {$\mathbb{B}$} | S = {$\mathbb{N}$})
    - F ist eine Menge von Operatorsymbolen ($add : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$)
    - Jedes Operatorsymbol besitzt einen Vorbereich und einen Zielbereich.
  - Gesetze E

Wiederholung: Spezifikation ADT Vektor

Sorten: $[\mathbb{R}, \mathit{Vektor}]$
Definition Sorte: $[\mathit{Vector}]$:

$$
\begin{array}{l}
\mathit{vec} : \mathbb{R} \times \mathbb{R} \to \mathit{Vector} \\
\mathit{scale} : \mathit{Vector} \times \mathbb{R} \to \mathit{Vector} \\
\mathit{add} : \mathit{Vector} \times \mathit{Vector} \to \mathit{Vector} \\
\hline
\forall\, a, b, c \in \mathbb{R} \bullet \mathit{scale}(\mathit{vec}(a, b), c) = \mathit{vec}(ca, cb) \\
\forall\, a_1, a_2, b_1, b_2 \in \mathbb{R} \bullet \mathit{add}(\mathit{vec}(a_1, a_2), \mathit{vec}(b_1, b_2)) = \mathit{vec}(a_1 + b_1, a_2 + b_2)
\end{array}
$$

Hinweis: Spezifikationssprache ähnlich zur Z Notation (Wikipedia | Handbuch | ISO Norm als zipped pdf)

Wiederholung: Termersetzung

$[Vector]$

$$
\begin{array}{l}
vec : \mathbb{R} \times \mathbb{R} \to Vector \\
scale : Vector \times \mathbb{R} \to Vector \\
add : Vector \times Vector \to Vector \\
\hline
\forall\, a, b, c \in \mathbb{R} \bullet scale(vec(a, b), c) = vec(ca, cb) \\
\forall\, a_1, a_2, b_1, b_2 \in \mathbb{R} \bullet add(vec(a_1, a_2), vec(b_1, b_2)) = vec(a_1 + b_1, a_2 + b_2)
\end{array}
$$

Frage 1: Was sind die Konstruktoren der Sorte *Vector*?
Frage 2: Ist $add(vec(2, 3), 9)$ ein gültiger Term? Warum?
Frage 3: Vereinfachen Sie den Term $add(vec(0, 1), scale(vec(2, 1), 2))$ bis zur Normalform!

ADT Liste: Motivation

- Aneinanderreihung von Elementen in einer bestimmten Reihenfolge
- Head: Das erste (Hier und VL: *zuletzt hinzugefügte*) Element
- Tail: Die Liste ohne den Head

$insert(5, insert(4, insert(3, insert(2, insert(1, Empty))))) \equiv 1, 2, 3, 4, 5$
Head: ??
Tail: ??

ADT Liste: Motivation

- Aneinanderreihung von Elementen in einer bestimmten Reihenfolge
- Head: Das erste (Hier und VL: *zuletzt hinzugefügte*) Element
- Tail: Die Liste ohne den Head

$insert(5, insert(4, insert(3, insert(2, insert(1, Empty))))) \equiv 1, 2, 3, 4, 5$
Head: 5
Tail: 1, 2, 3, 4

ADT Liste: Operationen

init Erzeugt eine neue leere Liste
insert Fügt ein Element *vorne* (am Head) an die Liste an
empty Prüft, ob die Liste leer ist
length Bestimmt die Länge der Liste
head Bestimmt das vorderste Element der Liste
tail Bestimmt die Liste ohne das vorderste Element
last Bestimmt das letzte Element der Liste
nth Bestimmt das n-te Element der Liste
isin Prüft, ob ein Element in der Liste enthalten ist
append hängt zwei Listen aneinander

ADT Liste: (teilweise) Spezifikation

$[Elem]$

$$errorelem \in Elem;\ Elem_v = Elem \setminus \{errorelem\}$$

$[List]$

$$\begin{array}{l} init : List \\ insert : Elem \times List \to List \\ head : List \to Elem \\ tail : List \to List \\ isin : Elem \times List \to \mathbb{B} \\ maxel : List \to Elem \\ \hline \forall\, l \in List \bullet insert(errorelem, l) = l \\ head(init) = errorelem \\ \forall\, l \in List;\ e \in Elem \bullet head(insert(e, l)) = e \end{array}$$

ADT Liste: *tail*

Aufgabe: Definiere *tail*!

ADT Liste: *tail*

Aufgabe: Definiere *tail*!

$$tail : List \to List$$

$$tail(init) = init$$
$$\forall\, e \in Elem_v;\ l \in List \bullet tail(insert(e, l)) = l$$

ADT Liste: *isin*

Aufgabe: Definiere *isin*!

ADT Liste: *isin*

Aufgabe: Definiere *isin*!

$$isin : List \to \mathbb{B}$$

$$\forall\, e \in Elem_v \bullet isin(e, init) = False$$
$$\forall\, e, f \in Elem_v, e \neq f;\ l \in List \bullet isin(e, insert(f, l)) = isin(e, l)$$
$$\forall\, e \in Elem_v;\ l \in List \bullet isin(e, insert(e, l)) = True$$
$$isin(errorelem, init) = False$$

ADT Liste: *maxel*

Aufgabe: Definiere *maxel*!

ADT Liste: *maxel*

Aufgabe: Definiere *maxel*!

$$
\begin{array}{l}
maxel : List \to Elem \\
max : Elem \times Elem \to Elem \\
\hline
maxel(init) = errorelem \\
\forall\, e \in Elem_v;\ l \in List \bullet maxel(insert(e, l)) = max(e, maxel(l)) \\
\forall\, a, b \in Elem_v, a < b \bullet max(a, b) = b \\
\forall\, a, b \in Elem_v, a \leq b \bullet max(a, b) = a \\
\forall\, a \in Elem \bullet max(a, errorelem) = a \\
\forall\, a \in Elem \bullet max(errorelem, a) = a
\end{array}
$$

Implementierung

Implementierung des ADTs Liste

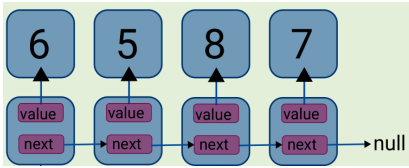
- Mit Array
- Als verkettete Liste
- Als verkettete Liste mit gekapselten Elementen
- Als doppelt verkettete Liste
- ...

Implementierung mit Array

```
typedef int element;
typedef struct _list{
  int length;
  element* data;
} list;
```

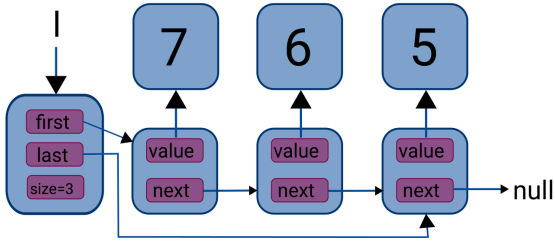
Implementierung mit einfacher Verkettung

```
typedef int element;
typedef struct _list{
  element value;
  _list* next;
} list;
```



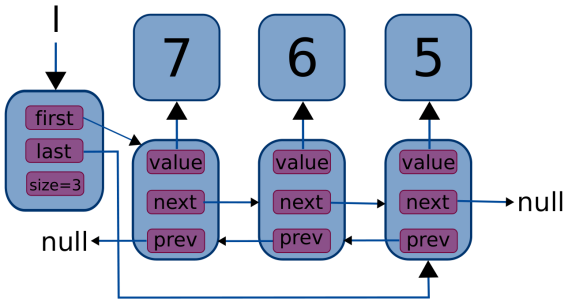
Implementierung mit einfacher Verkettung & Kapselung

```
typedef int element;
typedef struct _node{
  element val;
  _node* next;
} node;
typedef struct _list{
  int length;
  _node* first;
  _node* last;
} list;
```



Implementierung mit doppelter Verkettung

```
typedef int element;
typedef struct _node{
  element val;
  _node* next;
  _node* prev;
} node;
typedef struct _list{
  int length;
  _node* first;
  _node* last;
} list;
```



Welche Implementierung ist die Richtige?

Es kommt drauf an...

- Mit Array: wahlfreier Zugriff (*nth*) effizient, *isin* linear, ...
- Als verkettete Liste: Vergrößerung bei (*insert*) effizient, ...
- Als verkettete Liste mit gekapselten Elementen: +*last* effizient, ...
- Als doppelt verkettete Liste: +Iteration in beide Richtungen, ...
- ...

→ Welche Implementierung die “richtige” ist, hängt von der Anwendung ab!

Fragen?

Aufgaben

Lösen Sie die folgenden Aufgaben! Nutzen Sie dazu die Konzepte aus dieser Übung!
Bearbeitungszeit: bis 10 Minuten vor Schluss. Dann Besprechung von häufigen Problemen.

Aufgaben: Spezifikation ADT "Set"

Ein Set (engl. Menge) ist eine Liste an Elementen, wobei jedes Element nur einmal vorkommen kann. Die Spezifikation könnte wie folgt beginnen:

Sorten: $[Elem, \mathbb{B}]$

[Set]

$init : Set$	
$isin : Elem \times Set \to \mathbb{B}$	– True, wenn Element in Set vorkommt
$setadd : Elem \times Set \to Set$	– Fügt Element hinzu (falls noch nicht vorhanden)
$setremove : Elem \times Set \to Set$	– Entfernt Element aus dem Set
`TODO...`	

Definieren Sie die Gesetze, die für den ADT Set gelten!

Aufgaben: Implementierung ADT “Set”

Implementieren Sie den Datentyp Set mit ganzzahligen Elementen in C auf Basis einer einfach verketteten Liste mit gekapselten Elementen!

Lösungen

Lösung: Spezifikation ADT "Set"

$init : Set$

$isin : Elem \times Set \to \mathbb{B}$ – True, wenn Element in Set vorkommt

$setadd : Elem \times Set \to Set$ – Fügt Element hinzu (falls noch nicht vorhanden)

$setremove : Elem \times Set \to Set$ – Entfernt Element aus dem Set

---

$$\forall\, e, f, g \in Elem, e \neq f, f \neq g, e \neq g;\ s \in Set \bullet$$

$$isin(e, init) = False$$

$$isin(e, setadd(e, s)) = True$$

$$isin(e, setadd(f, s)) = isin(e, s)$$

$$isin(e, s) \Rightarrow setadd(e, setadd(f, s)) = setadd(f, s)$$

$$setremove(e, init) = init$$

$$setremove(e, setadd(f, init)) = setadd(f, init)$$

$$setremove(e, setadd(e, init)) = init$$

$$setremove(e, setadd(f, setadd(g, s))) = setadd(f, setremove(e, setadd(g, s)))$$

$$setremove(e, setadd(f, setadd(e, s))) = setadd(f, s)$$