Universität Rostock

Traditio et Innovatio

Imperative Programmierung

Übung 10: Abstrakte Datentypen

Justin Kreikemeyer

Informatik, Uni Rostock

Aufwärmübung:

Testat zur Selbstkontrolle

Löst die Aufgaben Handschriftlich!

Leitfragen

- Warum Abstraktion?
- Wie definiert man formal einen abstrakten Datentypen?
- Wie implementiert man einen ADT in C?

Motivation

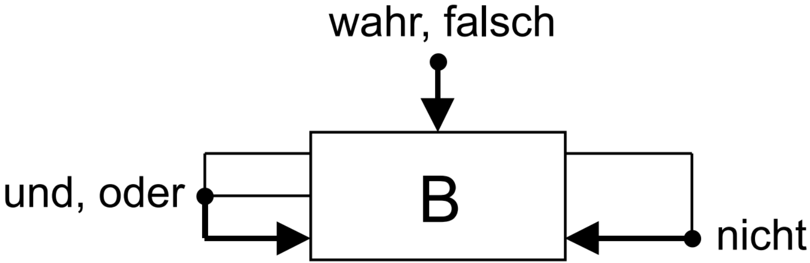
- Definieren nur “Wie muss sich mein Datentyp verhalten?” $\rightarrow$ Unabhängigkeit von der Implementierung
- Implementierung der selben Spezifikation auf verschiedene Arten möglich
- Unabhängigkeit von der Programmiersprache und sogar dem Programmierparadigma
- Heben Details für später auf und konzentrieren uns auf das Wesentliche
- Formale Aussagen möglich; Mittel der Kommunikation

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

Boolsche Werte



Boolsche Werte

- Signatur:
  - Sorten: $S = \{\mathbb{B}\}$
  - Operatorsymbole: $F = \{$
    - $T : \emptyset \to \mathbb{B}$ (alternativ: $T : \mathbb{B}$)
    - $F : \emptyset \to \mathbb{B}$ (alternativ: $F : \mathbb{B}$)
    - $not : \mathbb{B} \to \mathbb{B}$
    - $and : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$
    - $or : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

    $\}$

Terme

- $and(T,F)$
- $not(T)$
- $or(T)$

Ungültig, da der Vorbereich von or genau 2 boolsche Variablen entgegen nimmt

- $not\left(not\left(or(T,F)\right)\right)$
- $and\left(T,not(F)\right)$
- $or\left(not(F),and(T,T,F)\right)$
- $and(and(not(F),not\left(not(T)\right)),or(F,not(F))$

Ungültig, da der Vorbereich von and genau 2 boolsche Variablen entgegen nimmt

Boolsche Werte

> Gesetze:

1. $not(T) = F$
2. $not(F) = T$
3. $\forall x \in \mathbb{B} : not(not(x)) = x$

4. $\forall x \in \mathbb{B} : and(T, x) = x$
5. $\forall x \in \mathbb{B} : and(F, x) = F$
6. $\forall x, y \in \mathbb{B} : and(x, y) = and(y, x)$

7. $\forall x \in \mathbb{B} : or(T, x) = T$
8. $\forall x \in \mathbb{B} : or(F, x) = x$
9. $\forall x, y \in \mathbb{B} : or(x, y) = or(y, x)$

Termersetzung

$not(T) = F \mid not(F) = T \mid \forall x \in \mathbb{B} : not(not(x)) = x$

$\forall x \in \mathbb{B} : and(T, x) = x \mid \forall x \in \mathbb{B} : and(F, x) = F \mid \forall x, y \in \mathbb{B} : and(x, y) = and(y, x)$

$\forall x \in \mathbb{B} : or(T, x) = T \mid \forall x \in \mathbb{B} : or(F, x) = x \mid \forall x, y \in \mathbb{B} : or(x, y) = or(y, x)$

> $not\left(or\left(and(F, and(T, y)), or\left(not(not(F)), or(not(F), z)\right)\right)\right)$

Termersetzung

$not(T) = F \mid not(F) = T \mid \forall x \in \mathbb{B} : not(not(x)) = x$

$\forall x \in \mathbb{B} : and(T,x) = x \mid \forall x \in \mathbb{B} : and(F,x) = F \mid \forall x,y \in \mathbb{B} : and(x,y) = and(y,x)$

$\forall x \in \mathbb{B} : or(T,x) = T \mid \forall x \in \mathbb{B} : or(F,x) = x \mid \forall x,y \in \mathbb{B} : or(x,y) = or(y,x)$

> $not\left(or\left(and\left(F, and(T,y)\right), or\left(not\left(not(F)\right), or\left(not(F), z\right)\right)\right)\right)$

Gesetz 2

> $not\left(or\left(and\left(F, and(T,y)\right), or\left(not\left(not(F)\right), or(T, z)\right)\right)\right)$

Gesetz 7

> $not\left(or\left(and\left(F, and(T,y)\right), or\left(not\left(not(F)\right), T\right)\right)\right)$

Gesetz 3

> $not\left(or\left(and\left(F, and(T,y)\right), or(F,T)\right)\right)$

Gesetz 9

> $not\left(or\left(and\left(F, and(T,y)\right), or(T,F)\right)\right)$

Gesetz 7

> $not\left(or\left(and\left(F, and(T,y)\right), T\right)\right)$

Gesetz 4

> $not(or(and(F,y),T))$

Gesetz 5

> $not(or(F,T))$

Gesetz 6

> $not(or(T,F))$

Gesetz 7

> $not(T) = F$

Gemeinsames Beispiel: Natürliche Zahlen (Spezifikation)

Definieren Sie den ADT `Integer`, welcher eine natürliche Zahl repräsentiert und folgende Operationen unterstützt:

- *zero*: Gibt das Nullelement zurück
- *suc*: Gibt den Nachfolger einer natürlichen Zahl zurück
- *add*: Addiert zwei natürliche Zahlen

Gemeinsames Beispiel: Natürliche Zahlen (Implementierung)

- `int` (fertig! :D)
- "Wörtlich" der Spezifikation folgen
- Irgendwas dazwischen…

Die "richtige" Implementierung hängt von der Anwendung ab!

Noch nicht überzeugt?

Weiteres Beispiel: ADT “Liste” (Jetzt Überblick, Details später)

ADT – Liste

- init → Erzeugt eine neue Liste
- insert → Fügt ein Element **vorne** 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

Implementierung

Implementierung des ADTs Liste

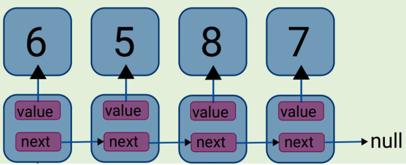
- Mit Array
- Als verkettete Liste
- Als verkettete Liste mit gekapselten Elementen
- Als doppelt verkettete Liste
- ...

Implementierung mit Array

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

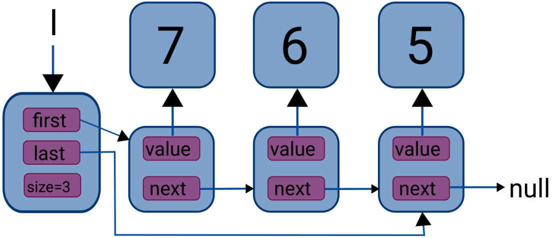
Implementierung mit einfacher Verkettung

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



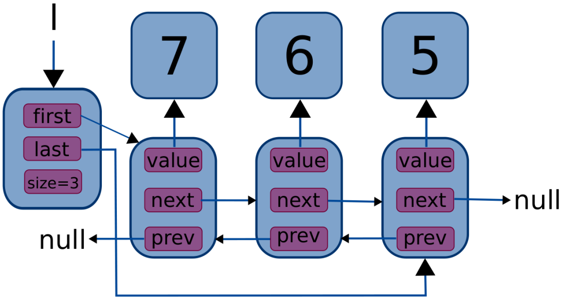
Implementierung mit einfacher Verkettung & Kapselung

```
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 schnell, …
- Als verkettete Liste: Vergrößerung 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: Termersetzung

Vereinfachen Sie die folgenden Ausdrücke bis zur Normalform. Die Spezifikationen der zugehörigen ADT finden Sie in den folgenden Folien.

1. $less\Big(mult\big(suc(zero), suc(zero)\big), suc\big(suc(zero)\big)\Big)$
2. $eq\Big(add\big(suc(zero), suc(zero)\big), mult\big(suc(suc(zero)), suc(zero)\big)\Big)$

Natürliche Zahlen

- Signatur:
  - Sorten: S = $\{\mathrm{b}, \mathrm{n}\}$
  - Operatorsymbole: F = {
    - $zero : \emptyset \rightarrow \mathrm{n}$ (alternativ: $\mathrm{zero} : \mathrm{n}$)
    - $suc : \mathrm{n} \rightarrow \mathrm{n}$
    - $add : \mathrm{n} \times \mathrm{n} \rightarrow \mathrm{n}$
    - $mult : \mathrm{n} \times \mathrm{n} \rightarrow \mathrm{n}$
    - $div : \mathrm{n} \times \mathrm{n} \rightarrow \mathrm{n}$
    - $eq : \mathrm{n} \times \mathrm{n} \rightarrow \mathrm{b}$
    - $noteq : \mathrm{n} \times \mathrm{n} \rightarrow \mathrm{b}$
    - $less : \mathrm{n} \times \mathrm{n} \rightarrow \mathrm{b}$
    - $lesseq : \mathrm{n} \times \mathrm{n} \rightarrow \mathrm{b}$
    - $more : \mathrm{n} \times \mathrm{n} \rightarrow \mathrm{b}$
    - $moreeq : \mathrm{n} \times \mathrm{n} \rightarrow \mathrm{b}$
    - $even : \mathrm{n} \rightarrow \mathrm{b}$
    - $odd : \mathrm{n} \rightarrow \mathrm{b}$

    }

Natürliche Zahlen

- Gesetze: $(x, y \in X_n)$
  1. $add(x, zero) = x$
  2. $add(x, suc(y)) = suc(add(x, y))$
  3. $mult(x, zero) = zero$
  4. $mult(x, suc(y)) = add(mult(x, y), x)$
  5. $eq(zero, zero) = T$
  6. $eq(zero, suc(x)) = F$
  7. $eq(suc(x), zero) = F$
  8. $eq(suc(x), suc(y)) = eq(x, y)$

Natürliche Zahlen

> Gesetze: $(x, y \in X_n)$

9. $less(x, zero) = F$
10. $less(\mathrm{zero}, \mathrm{suc}(\mathrm{x})) = \mathrm{T}$
11. $less\big(suc(x), suc(y)\big) = less(x, y)$

12. $even(zero) = T$
13. $even\big(suc(zero)\big) = F$
14. $even(suc\big(suc(x)\big) = even(x)$

15. $noteq(x, y) = not\big(eq(x, y)\big)$
16. $lesseq(or(less(x, y), eq(x, y))$
17. $more(x, y) = not\big(lesseq(x, y)\big)$
18. $moreeq(x, y) = not\big(less(x, y)\big)$
19. $odd(x) = not(even(x))$

Aufgaben: Eigener ADT

Definieren Sie den formalen Abstrakten Datentypen `Vector2D` zum Umgang mit Vektoren. Folgende Operationen sollen unterstützt werden:

- *make*: Erstellt einen neuen Vektor aus seinen zwei Komponenten
- *null*: Gibt den Nullvektor $(0, 0)$ zurück
- *e1*: Gibt den Basisvektor Entlang der ersten Achse $\vec{e}_1 = (1, 0)$ zurück
- *e2*: Gibt den Basisvektor Entlang der zweiten Achse $\vec{e}_2 = (0, 1)$ zurück
- *first*: Gibt die erste Komponente des Vektors zurück
- *second*: Gibt die zweite Komponente des Vektors zurück
- *add*: Berechnet die (komponentenweise) Summe zweier Vektoren
- *scale*: Skaliert einen Vektor entsprechend eines reellwertigen Faktors
- *product*: Berechnet das Skalarprodukt zweier Vektoren

Hinweis: Nutzen Sie gerne die Vorlage auf der nächsten Folie.
Reduzieren Sie schließlich den Term $\mathit{add}(\mathit{scale}(e1, 4), \mathit{scale}(e2, 2))$ bis zu einem simplen Vektor!

Aufgabe: Eigener ADT (Vorlage)

Sorten S = $\{ \qquad\qquad\qquad\qquad\qquad\qquad \}$

Operatorsymbole F = $\{$

- $make:$
- $null:$
- $e1:$
- $e2:$
- $first:$
- $second:$
- $add:$
- $scale:$
- $product:$

$\}$

Aufgabe: Eigener ADT (Vorlage)

1. $make(a, b) =$
2. $add(x, null) =$
3. $add((x_1, x_2), (y_1, y_2)) =$
4. $first((x_1, x_2)) =$
5. $second((x_1, x_2)) =$
6. $scale((x_1, x_2), a) =$
7. $product((x_1, x_2), (y_1, y_2)) =$

Aufgaben: Implementierung ADT

Implementieren Sie Ihre Spezifikation des ADT `Vektor2D` auf folgende Weisen:

- Als eigener Datentyp mittels eines zweielementigen Arrays.
- Als eigener Datentyp mittels `struct`.

Prüfen Sie anhand von Beispielen, ob ihre Implementierung alle ihre Gesetzmäßigkeiten erfüllt.

Fun Fact: Jingle Bells

- Der ASCII-Charakter mit der Nr. 7 bzw. Repräsentation `\a` produziert in manchen Umgebungen einen Glocken- oder Piepton

```
#include <stdio.h>
int main(int argc, char* argv[]) {
  printf("\a"); // beep!
  return 0;
}
```

"Back in the dark ages [. . .] a terminal was a large chunk of iron that hammered ink onto paper [. . .]. In case an operator fell asleep to the soothing noises of it hammering out text, it had an actual bell it could ring. [. . .] As terminals became smaller and implemented with few or no moving parts, the physical bell was replaced by a beeper. Exactly what your terminal emulator [. . .] does when it is asked to display that control character is not well standardized today. It ought to make a noise or flash the window, but your mileage will vary."
– Quelle: https://stackoverflow.com/a/3456213/5627083

Lösungen

Termersetzung bis zur Normalform – Lösung

> $less(add(mult(suc(zero), zero), suc(zero)), suc(suc(zero)))$

4

> $less(mult(suc(zero), suc(zero)), suc(suc(zero)))$

3

> $less(add(zero, suc(zero)), suc(suc(zero)))$

2

> $less(suc(add(zero, zero)), suc(suc(zero)))$

1

> $less(suc(zero), suc(suc(zero)))$

11

> $less(zero, suc(zero))$

10

> T

Termersetzung bis zur Normalform – Lösung

> $eq(add(suc(zero), suc(zero)), mult(suc(suc(zero)), suc(zero)))$

4

> $eq(add(suc(zero), suc(zero)), add(mult(suc(suc(zero)), zero), suc(suc(zero))))$

3

> $eq(add(suc(zero), suc(zero)), add(zero, suc(suc(zero))))$

2

> $eq(add(suc(zero), suc(zero)), suc(add(zero, suc(zero))))$

2

> $eq(add(suc(zero), suc(zero)), suc(suc(add(zero, zero))))$

1

> $eq(add(suc(zero), suc(zero)), suc(suc(zero)))$

2

> $eq(suc(add(suc(zero), zero)), suc(suc(zero)))$

1

> $eq(suc(suc(zero)), suc(suc(zero)))$

8

> $eq(suc(zero), suc(zero))$

8

> $eq(zero, zero)$

5

> T

Lösung: Eigener ADT (Signatur)

$(Vektor : \mathbb{R}^2)$
Sorten S = $\{Vektor, \mathbb{R}\}$
Operatorsymbole F = $\{$

- $make : \mathbb{R} \times \mathbb{R} \rightarrow Vektor$
- $null : \emptyset \rightarrow Vektor$
- $e1 : \emptyset \rightarrow Vektor$ (alternativ: $e1 : Vektor$)
- $e2 : \emptyset \rightarrow Vektor$ (alternativ: $e2 : Vektor$)
- $first : Vektor \rightarrow \mathbb{R}$
- $second : Vektor \rightarrow \mathbb{R}$
- $add : Vektor \times Vektor \rightarrow Vektor$
- $scale : Vektor \times \mathbb{R} \rightarrow Vektor$
- $product : Vektor \times Vektor \rightarrow \mathbb{R}$

$\}$

Lösung: Eigener ADT (Gesetze/Vereinfachung)

$\forall x, y \in \mathit{Vektor}$ und $\forall a, b \in \mathbb{R}$, wobei $x = (x_1, x_2)$ und $y = (y_1, y_2)$

1. $\mathit{make}(a, b) = (a, b)$
2. $\mathit{add}(x, \mathit{null}) = x$
3. $\mathit{add}(x, y) = (x_1 + y_1, x_2 + y_2)$
4. $\mathit{first}(x) = x_1$
5. $\mathit{second}(x) = x_2$
6. $\mathit{scale}(x, a) = (ax_1, ax_2)$
7. $\mathit{product}(x, y) = (x_1 y_1, x_2 y_2)$

$\mathit{add}(\mathit{scale}(e1, 4), \mathit{scale}(e2, 2))$
$\mathit{add}((4 * 1, 4 * 0), \mathit{scale}(e2, 2))$
$\mathit{add}((4, 0), \mathit{scale}(e2, 2))$
$\mathit{add}((4, 0), (0, 2))$
$(4 + 0, 0 + 2)$
$(4, 2)$