Übersicht

9 Abstrakte Datentypen
- Einführung
- Abstrakte Datentypen
- Beispiele: Grundlegende Datenstrukturen
- Beispiele – ADT in OOP (Java)

9 Abstrakte Datentypen
- Einführung
- Abstrakte Datentypen
- Beispiele: Grundlegende Datenstrukturen
- Beispiele – ADT in OOP (Java)

Einführung

- Beispiele für Datenstrukturen (aus den Übungen)
  - Rationale Zahlen
  - Matrizen
  - ...
  - *Klassen von Objekten*
- Motivation:
  Strukturierung und Wiederverwendbarkeit von Software
- Ziel:
  Beschreibung von Datenstrukturen *unabhängig* von Implementierung (d.h. unabhängig von Programmiersprache)
- Konzept:
  Beschreibung von **Abstrakten Datentypen (ADT)**

9 Abstrakte Datentypen

- Einführung
- Abstrakte Datentypen
- Beispiele: Grundlegende Datenstrukturen
- Beispiele – ADT in OOP (Java)

Abstrakter Datentyp

Definition (Abstrakter Datentyp)

Ein *Abstrakter Datentyp (ADT)* ist ein Verbund von Daten zusammen mit der Definition aller zulässigen Operationen auf diesen Daten.

- Kapselung: Zugriff nur über Operationen (Schnittstelle)
- Geheimnisprinzip: Realisierung bleibt verborgen
- ADT
  - beschreibt *Semantik* von Operationen „Was?“
  - beschreibt nicht Implementierung ~~„Wie?“~~
  - ist mathematisches („abstraktes“) Modell für Klasse von Daten
- Objektorientierte Programmiersprachen unterstützen dieses Prinzip!

Schnittstelle: Signatur

- Eine Signatur Σ ist definiert durch ein Paar

$$\Sigma \;=\; (S, \Omega) \quad \text{wobei gilt}$$

  - S ist eine Menge von *Sorten* (Typen).
  - $\Omega = \{ f : s_1 \times \cdots \times s_n \to s \,|\, s_i, s \in S \text{ für } i, n \geqslant 0 \}$ ist eine Menge von *Funktionen*.

- Nullstellige Funktionen (ohne Argumente) heißen *Konstanten*
  - Für *Methoden* in der objektorientierten Programmierung: Selbstreferenz `this` ist „unsichtbares erstes" Argument!
- Signatur legt formale Schnittstelle zu einem Datentyp fest
  - Menge der Funktionen (Methoden) – *jeweils mit*
  - Anzahl und Typ der Argumente und
  - Typ des Rückgabewerts

Signatur: Beispiel

Definiere ADT `Nat`: natürliche Zahlen $x, y \in \mathbb{N}_0$

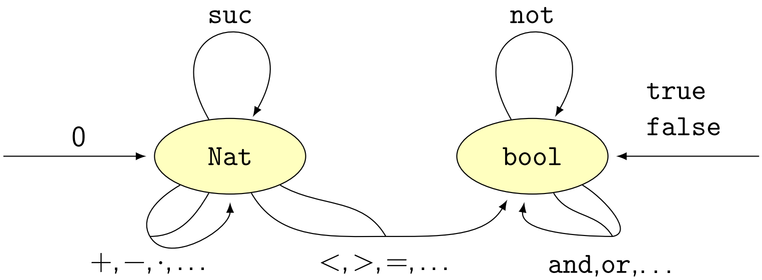
- *Konstante* Nullfunktion 0
- Nachfolgerfunktion $\mathrm{suc}(x) \equiv x + 1$ (*successor*)
- Addition $\mathrm{add}(x, y) \equiv x + y$

```
type Nat
operators
   0: → Nat
   suc: Nat → Nat
   add: Nat×Nat → Nat
```

ADT `Nat` (erweitert um $+, -, \cdot, \ldots, <, >, =, \ldots$) und `bool`

- Konstanten 0 und `true`, `false`
- Operatoren

$$
\begin{aligned}
+, -, \cdot, \ldots : \quad & \mathtt{Nat} \times \mathtt{Nat} \to \mathtt{Nat} \\
<, >, =, \ldots : \quad & \mathtt{Nat} \times \mathtt{Nat} \to \mathtt{bool}
\end{aligned}
$$



Definition von Semantik

- Signatur beschreibt Schnittstelle des ADT
- Zur vollständigen Beschreibung fehlt noch die Semantik
  - Beschreibung des Verhaltens der einzelnen Funktionen in Ω
  - *Was wird berechnet?* (Aber nicht: „*Wie?*“ !!)
  - z.B. „`add` berechnet die Summe von zwei natürlichen Zahlen.“
  - Beschreibung kann andere Typen und deren Spezifikation mit einbeziehen („importieren“), hier z.B. `bool`
- Es gibt verschiedene Möglichkeiten, Semantik zu spezifizieren.
  - Verschiedene Möglichkeiten der (formalen) Beschreibung
  - Verschiedene Grade an „Komplexität“ solcher Spezifikation (Detailgrad, Vollständigkeit)

- Definiere Semantik von $\texttt{add} : \texttt{Nat} \times \texttt{Nat} \to \texttt{Nat}$
- Informell

$\texttt{add}(x, y)$ *liefert die Summe von x und y.*

- Algebraisch

$$\texttt{add}(x, y) = \begin{cases} x & \text{falls } y = 0 \\ \texttt{suc}(\texttt{add}(x, z)) & \text{falls } y = \texttt{suc}(z) \end{cases}$$

- Axiomatisch

```
add( x, 0 )           =  x
add( x, suc(y) )      =  suc( add(x,y) )
```

- `type`
  Name des neuen ADT ggf. mit Parametern
- `operators`
  Signaturen der auf dem ADT definierten Funktionen.
  Für jede Funktion
  - Name, Typ und Reihenfolge der Argumente, Rückgabetyp
  - ggf. Funktion partiell oder total?
  - ggf. asymptotischer Aufwand (schränkt *Implementierung* ein!)
- `axioms`
  Axiome, die Eigenschaften der Funktionen beschreiben
  (i.d.R. als Gleichungen)
- `preconditions`
  Vorbedingung für jede *partielle* Funktion, die angibt, wann
  diese definiert ist

Beispiel: Bool

```
type Bool

operators

  true  : → Bool
  false : → Bool

  not   : Bool → Bool
  ...

axioms ∀ b : Bool

  not(true)   = false
  not(not(b)) = b
  ...
```

Beispiel: Nat

```
type Nat
import Bool
operators
    0   : → Nat
    suc : Nat → Nat
    add : Nat×Nat → Nat
    > : Nat × Nat → Bool
    ...
axioms  (∀x,y : Nat)

    add(x,0)          =* x
    add(x,suc(y))     =* suc(add(x,y))

    0 > 0             = false
    0 > suc(y)        = false
    suc(x) > 0        = true
    suc(x) > suc(y) = x > y
    ...
```

- *Gleichheitsrelation* $=^\star$ ist noch unbekannt!
- Sie muss ebenfalls definiert werden!
- Ergänze dazu

```
operators

  =* : Nat × Nat → Bool

axioms

  0 =* 0            =  true
  0 =* suc(x)       =  false
  suc(x) =* 0       =  false
  suc(x) =* suc(y)  =  x =* y
```

- *Wir sparen uns das bei einigen Beispielen.*

Beispiel: `Rational` (1)

ADT Zur Darstellung von *rationalen Zahlen*

```
type Rational
import Bool, Nat

operators

  0        : → Rational
  create   : Nat × Nat → Rational
  nom      : Rational → Nat
  denom    : Rational → Nat
  add      : Rational × Rational → Rational
  normal   : Rational → Rational
  is_equal : Rational × Rational → Bool
  is_zero  : Rational → Bool
```

nominator=Zähler, *denominator*=Nenner, `normal` → gekürzte „Normalform“

Beispiel: Rational (2)

```
axioms  (∀ p,q : Rational), (∀ x,y : Nat)

  is_zero(0) = true
  is_zero(create(0,y)) = true
  is_zero(create(suc(x),y)) = false
  nom(create(x,y)) = x
  denom(create(x,y)) = y

  nom(add(p,q))   = add(mult(nom(p),denom(q)),
                        mult(nom(q),denom(p)))  ⋆
  denom(add(p,q)) = mult(denom(p),denom(q))
  ...

preconditions

  create(x,y) : y ≠ 0
```

⋆ add und mult auf der rechten Seite sind Funktionen von Nat!

Beispiel: Rational (3)

```
axioms  (∀ p,q : Rational), (∀ x,y : Nat)
  ...

  gcd(nom(normal(p)),
      denom(normal(p))) = 1

  is_equal(p,q) =
    ( is_zero(p) ∧ is_zero(q) ) ∨
    ( nom(normal(p))   = nom(normal(q)) ∧
      denom(normal(p)) = denom(normal(q)) )
```

- Vergleich auf gekürzten Brüchen
- Sonderfall 0 (Nenner beliebig)
- `gdc(m,n)` berechnet $\mathrm{ggT}(m,n)$

Beispiel: Set

ADT für Mengen (*set*) von Daten des Typs `Item`

- Typ `Item` (Element/Einheit) ist *Parameter* des ADT `Set`
- Spezifikation von `Set` ist unabhängig von `Item`

```
type Set(Item)
import Bool
operators
  empty_set : → Set
  is_empty  : Set → Bool
  insert    : Set × Item → Set
  is_in     : Set × Item → Bool
axioms  (∀s : Set), (∀i,j : Item)
 is_empty(empty_set)    = true
 is_empty(insert(s,i))  = false
 is_in(empty_set,i)     = false
 is_in(insert(s,i),j)   = (i=j) ∨ is_in(s,j)
```

Denkmodell zur `is_in`

- Axiome sind keine Rechenvorschriften!
- Deshalb ist Betrachtung "*rückwärts*" zulässig
- **Beispiel**: Die Menge $\{1,2,3\}$ lässt sich konstruieren als

  ```
  insert(insert(insert(empty_set,1),2),3)
  ```

  Damit

  ```
      is_in( insert(insert(insert(empty_set,1),2),3), 0)
  ⇒   is_in( insert(insert(empty_set,1),2), 0)
  ⇒   is_in( insert(empty_set,1), 0)
  ⇒   is_in( empty_set, 0)
  ⇒   false
  ```

- Der komplexere Ausdruck auf *linken Seite* wird durch einen einfacheren (*rechts*) ersetzt ("*komplexer*" = "größere Menge").
- Mit Hilfe des Axioms *kann jede* Auswertung von `is_in` auf den Trivialfall reduziert werden!
- *Die Axiome selbst spezifizieren dazu keinen Algorithmus!*

Klassifikation der Funktionen (und Beispiele)

- Konstruktoren erzeugen Instanzen von ADT
  - `true`: $\rightarrow$ `Bool`
  - `0`: $\rightarrow$ `Nat`
  - `empty_set`: $\rightarrow$ `Set`

  Dabei werden ggf. einzelne „Teile“ des ADT erzeugt.
- Selektoren „zerlegen in Einzelteile“
  - `nom (Rational)`
  - `denom (Rational)`

  oder liefern *Aussagen* über Instanz
  - `is_zero (Rational)`
  - `is_empty (Set)`
- Manipulatoren liefern *neue Instanzen*
  - `add (Nat,Rational)`
  - `normal (Rational)`
  - `insert (Set)`

Spezifikation von Semantik

- Wie komme ich zu System von Axiomen?
- Wann habe ich genug Gleichungen?
- Wann fehlt mir ein Axiom?

Leider keine einfache Antwort!

- So einfach wie möglich
- So komplex/umfangreich wie nötig
- Im folgenden Hinweise zu systematischem Vorgehen

Systematischer Entwurf von ADT (1)

- Festlegung der Konstruktoren
- Definition von geeigneten Selektoren
  Semantik durch Zurückführen auf Konstruktorterme
- Manipulatoren festlegen
- Fehlersituationen abfangen (`preconditions`)

Systematischer Entwurf von ADT (2)

Festlegen der Manipulatoren: *Semantik*? *Axiome*?

- Manipulatoren wenn möglich direkt auf Konstruktoren oder Selektoren zurückführen.
- Regeln von links nach rechts als Ersetzungsregeln aufbauen
  - Rechte Seite ist *einfacher* (=bekannt) als linke Seite.
  - Bei rekursiven Regeln bedeutet das
    - Argumente sind *einfacher* (d.h. von Bekanntem abgeleitet).
    - Terminierung ist garantiert! (i.d.R. als Spezialfall/Trivialfall)

    z.B. `add` (`Nat`), `is_empty` (`Set`)
  - Vorgehen analog zu *funktionaler Programmierung*
- Wichtig: *vollständige* Fallunterscheidung, d.h. *jeder* Konstruktor muss bei Parametern berücksichtigt werden!

Übersicht

9 Abstrakte Datentypen

- Einführung
- Abstrakte Datentypen
- Beispiele: Grundlegende Datenstrukturen
- Beispiele – ADT in OOP (Java)

- List – Listen von Elementen
- Stack – Stapel zur Ablage
- Queue – Warteschlange [,die Dinge weiterleitet]

ADT List (Liste)

- Liste als einfache und grundlegende Datenstruktur, wie sie oft in funktionalen Programmiersprachen verwendet wird

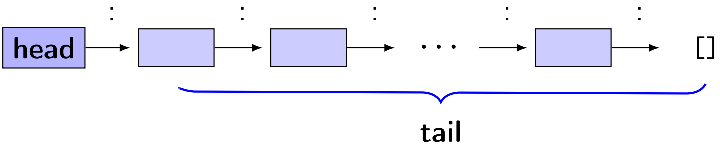
```
type List(Item)
import Nat
operators
  []      :    → List
  _ : _   :    Item × List → List
  head    :    List → Item
  tail    :    List → List
  length:      List → Nat
axioms   (∀L : List), (∀i : Item)
 head(i : L)     =  i
 tail([])        =*  []
 tail(i : L)     =*  L
 length([])      =  0
 length(i : L)  =  1+length(L)
preconditions
 head(L) : length(L)>0
```

ADT List (Liste)

```
type List(Item)
import Nat
operators
  []     :     → List
  _ : _  :     Item × List → List
  head   :     List → Item
  tail   :     List → List
  length :     List → Nat
```

ADT `List` (Liste)

- Konstruktor `[]` erzeugt *leere* Liste
- Operator `_ : _` fügt neues Element am Beginn (als *head*) ein
  Äquivalent z.B. zu Funktion `insert_head:Item`$\times$`List`$\rightarrow$`List`



- Beispiele für Listen über natürlichen Zahlen (`Item=Nat`)

```
[]
1 : []
1 : ( 1 : [] )              ==  1 : 1 : []
1 : ( 2 : ( 3 : [] ) )      ==  1 : 2 : 3 : []
```

ADT List (Liste)

```
axioms  (∀L : List), (∀i : Item)

 head(i : L)    =   i
 tail([])       =*  []
 tail(i : L)    =*  L
 length([])     =  0
 length(i : L)  =   suc( length(L) )

preconditions

 head(L) : length(L)>0
```

- Hier fehlt der Vergleich von Listen $=^\star$
- Wie könnte man $=^\star$: List×List→Bool spezifizieren?

ADT List (Liste)

- Liste als einfache und grundlegende Datenstruktur, wie sie oft in funktionalen Programmiersprachen verwendet wird

```
type List(Item)
import Nat
operators
  []      :    → List
  _ : _   :    Item × List → List
  head    :    List → Item
  tail    :    List → List
  length:      List → Nat
axioms   (∀L : List), (∀i : Item)
 head(i : L)      =   i
 tail([])         =*  []
 tail(i : L)      =*  L
 length([])       =   0
 length(i : L) =   1+length(L)
preconditions
 head(L) : length(L)>0
```

Beispiel zu Axiomen (List)

- Wie könnte das *letzte* Element charakterisiert werden?
- Hypothetische Erweiterung um Funktion

`last : List→Item`

- Welche Axiome gelten für `last`?

1. Undefiniert für *leere* Liste

```
preconditions
  last(L) : length(L)>0
```

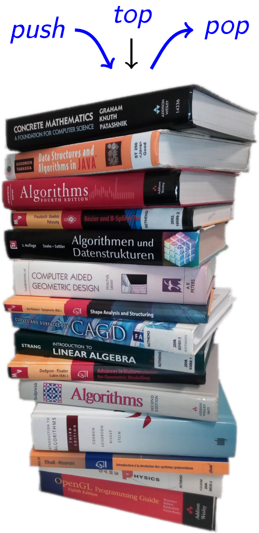
2. Trivialfall: Liste mit *einem* Element — Rekursionsende!

```
  last( x : [] ) = x
```

3. Allgemeiner Fall — rekursive Definition!

```
  last( x : L ) = last(L)
```

ADT Stack (Stapel)



- ADT `Stack`
  dt. auch: Stapel(-speicher), Keller(-speicher)
- `top` : `Stack` $\to$ `Item`
  oberstes (*top*) Element auslesen
- `push` : `Stack` $\times$ `Item` $\to$ `Stack`
  Element auf dem Stapel ablegen
- `pop` : `Stack` $\to$ `Stack`
  oberstes Element (`top`) vom Stapel nehmen
- `is_empty` : `Stack` $\to$ `Bool`
  Ist Stapel leer?
- Für leeren Stapel sind `top` & `pop` *undefiniert*!

ADT Stack (Stapel)

```
type Stack(Item)

import Bool

operators

  empty_stack : → Stack
  top         : Stack → Item
  push        : Stack × Item → Stack
  pop         : Stack → Stack
  is_empty    : Stack → Bool
```

ADT `Stack` (Stapel)

```
axioms  (∀S : Stack), (∀i : Item)

  top(push(S,i)) =  i
  pop(push(S,i)) =* S
  is_empty(empty_stack) = true
  is_empty(push(S,i)) = false

preconditions

  top(S) : ¬is_empty(S)
  pop(S) : ¬is_empty(S)
```

ADT Stack (Stapel)

```
type Stack(Item)
import Bool
operators
  empty_stack : → Stack
  top         : Stack → Item
  push        : Stack × Item → Stack
  pop         : Stack → Stack
  is_empty    : Stack → Bool
axioms  (∀S : Stack), (∀i : Item)
  top(push(S,i)) =  i
  pop(push(S,i)) =* S
  is_empty(empty_stack) = true
  is_empty(push(S,i)) = false
preconditions
  top(S) : ¬is_empty(S)
  pop(S) : ¬is_empty(S)
```

Stack und List

- `Stack` kann mit Hilfe von/als `List` implementiert werden.

Stack		List
`empty_stack`	$\equiv$	`[]`
`top(S)`	$\equiv$	`head(L)`
`push(S,i)`	$\equiv$	`i : L`
`pop(S)`	$\equiv$	`tail(L)`
`is_empty(S)`	$\equiv$	`length(L)==0`

- Wenn eine Implementierung des ADT `List` gegeben ist, kann diese zur Implementierung von `Stack` genutzt werden.

Anwendungsbeispiel für Stapel

- Betrachte *arithmetische Ausdrücke* über $\mathbb{Z}$
  - z.B. $2 \times (3+4)$
  - Es gilt die *Punkt-vor-Strich* Regel.
  - Bei Abweichung müssen *Klammern* gesetzt werden.
  - Diese übliche Notation heißt auch *Infix-Notation*.

- *Umgekehrte Polnische Notation* (*Postfix-Notation*)
  - Alternative Notation, die auf Klammern verzichtet!
  - Auch UPN oder RPN (*Reversed Polish Notation*)
  - Auswertung mit Hilfe eines Stack

  - Früher üblich in (teuren) wissenschaftlichen Taschenrechnern
  - Emulation z.B. durch das UNIX tool `dc` (*desk calculator*) oder auch Emacs calculator

Umgekehrte Polnische Notation (UPN)

- Betrachte binären Operator $\oplus : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$
- Anwendung des Operators in

Standard Infix-Notation	Postfix-Notation (UPN)
$x \oplus y$	$x\, y\, \oplus$

- Beispiele

$2+3$	$2\ \ 3\ +$
2×3	$2\ \ 3\ \times$
$2 \times 3 + 4$	$\underbrace{2\ \ 3\ \times}\ \ 4\ +$
$2 \times (3+4)$	$2\ \ \underbrace{3\ \ 4\ +}\ \times$

- UPN benötigt keine Klammern!
  - Auswertreihenfolge eindeutig definiert.
  - ($\underbrace{\boxed{}}$ stehen nur zur Verdeutlichung!)

- Beispiel: $2 \times (3+4)$ bzw. $2\ 3\ 4\ +\ \times$ in UPN
- Gegeben: Term als Liste `L = 2 : 3 : 4 :+:×: []`
- Auswertung mit Hilfe eines Stapels

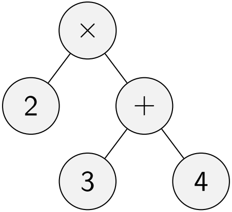
```
S:=empty_stack

while ¬is_empty(L) do
  x:=head(L); L:=tail(L)
  if x∈ℤ then
    S:=push(S,x)
  else
    b:=top(S); S:=pop(S);
    a:=top(S); S:=pop(S);
    S:=push(S, a ⊕[x] b )
  fi
od
output top(S)
```

Liste `L`	Stack `S`
$2:3:4:+:\times:$ `[]`	`empty_stack`
$3:4:+:\times:$ `[]`	$\mid 2$
$4:+:\times:$ `[]`	$\mid 2 \mid 3$
$+:\times:$ `[]`	$\mid 2 \mid 3 \mid 4$
$\times:$ `[]`	$\mid 2 \mid 7$
`[]`	$\mid 14$

Infix $\leftrightarrow$ Postfix

- Wie kommt man *algorithmisch* von der Infix- zur Postfix-Darstellung?
- Antwort im nächsten Semester!



Stapel als grundlegende Datenstruktur

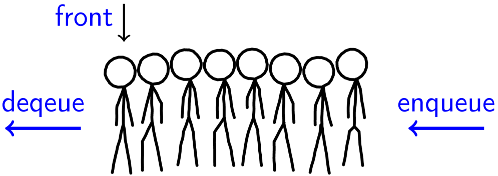
- Auswertung von *Programmcode* ähnlich (z.B. in Java VM)
- Stapel zur Speicherung von Zwischenergebnissen

- Nötig für Funktionsaufrufe
  - Speichern von Argumenten und Ergebnis
  - Speichern des „Rücksprungpunkts“
- Gilt insbesondere für *Rekursion*!

- Mit Hilfe eines Stapels kann *jede* rekursiv definierte Funktion auch iterativ dargestellt werden!
  - Schleife und Stapel ersetzen Rekursion
  - Folgevorlesung *Algorithmen und Datenstrukturen*

ADT `Queue` (Warteschlange)

- Für Stapel gilt *last in, first out (LIFO)*
- Prinzip ungeeignet z.B. zur Erledigung von Aufgaben
  - typisches Schreibtischproblem:
    unterste Papiere/Aufgaben bleiben liegen
- Besser: Aufgaben der Reihe nach abarbeiten, also *first in, first out* oder FIFO-Prinzip



- Warteschlange (*queue*) mit Operationen
  - *enqueue*: Element hinten anstellen
  - *dequeue*: Element vorne entfernen
  - *front*: vorderstes Element auslesen

ADT `Queue` (Warteschlange)

```
type Queue(Item)

import Bool

operators

  empty_queue : → Queue
  front       : Queue → Item
  enqueue     : Queue × Item → Queue
  dequeue     : Queue → Queue
  is_empty    : Queue → Bool
```

ADT Queue (Warteschlange)

```
axioms  (∀Q : Queue), (∀i,j : Item)

  front(enqueue(empty_queue,i)) = i
  front(enqueue(enqueue(Q,i),j)) =
      front(enqueue(Q,i))

  dequeue(enqueue(empty_queue,i)) =* empty_queue
  dequeue(enqueue(enqueue(Q,i),j)) =*
      enqueue(dequeue(enqueue(Q,i)),j)

  is_empty(empty_queue) = true
  is_empty(enqueue(Q,i)) = false

preconditions
  front(Q)   : ¬is_empty(Q)
  dequeue(Q) : ¬is_empty(Q)
```

ADT Queue (Warteschlange)

```
type Queue(Item)
import Bool
operators
  empty_queue : → Queue
  front       : Queue → Item
  enqueue     : Queue × Item → Queue
  dequeue     : Queue → Queue
  is_empty    : Queue → Bool
axioms  (∀Q : Queue), (∀i,j : Item)
  front(enqueue(empty_queue,i)) = i
  front(enqueue(enqueue(Q,i),j)) =
      front(enqueue(Q,i))
  dequeue(enqueue(empty_queue),i) =* empty_queue
  dequeue(enqueue(enqueue(Q,i),j)) =*
      enqueue(dequeue(enqueue(Q,i)),j)
  is_empty(empty_queue) = true
  is_empty(enqueue(Q,i)) = false
preconditions
  front(Q)   : ¬is_empty(Q)
  dequeue(Q) : ¬is_empty(Q)
```

Stack und Queue: Bezeichnungen

- Beim *stack* i.d.R. Bezeichnungen *push* und *pop*
- Für *queues* sind verschiedene Bezeichnungen üblich, z.B.

enqueue	*dequeue*
push	*pop*
unshift	*shift*
offer	*poll*
enter	*leave*
...	...

- *Queues* werden oft auch als *FIFO* (*buffers*) bezeichnet
  - Oft: Transport von Daten z.B. zwischen Prozessen
  - z.B. (Unix) *pipes*, *message queues*,...

Übersicht

9 Abstrakte Datentypen

- Einführung
- Abstrakte Datentypen
- Beispiele: Grundlegende Datenstrukturen
- Beispiele – ADT in OOP (Java)

ADT und objektorientierte Programmierung (OOP)

- ADT erzwingen Kapselung und Geheimnisprinzip
  - Interaktion mit ADT nur über *Schnittstelle*
  - Interne Realisierung (Implementierung) bleibt *verborgen*

- OOP unterstützt/erzwingt diese Eigenschaften ebenfalls.
  - ADT als Grundlage des Prinzips der OOP

- ADT als Abstrakte Basisklasse oder *interface* (z.B. in Java)

Typen/ADT	$\rightarrow$	Klassen (oder *interfaces*)
Funktionen	$\rightarrow$	Methoden
konkrete Implementierung	$\rightarrow$	abgeleitete Klassen

  - Methoden statt Funktionen:
    Selbstreferenz `this` als implizites Argument

Fehlerbehandlung

- Wir schließen undefinierte Konfigurationen bis jetzt aus:

  `preconditions`

- In der Praxis: Fehlerbehandlung durch Methoden des ADT
  - Kann ungültige Eingabewerte i.d.R. nicht ausschließen.
  - (Wenn doch: `assert`)
  - Wie soll Methode grundsätzlich reagieren?

- Mögliche Varianten
  - Rückgabe eines Fehlerwerts (z.B. `return null;`)
  - Generierung und Aufwerfen einer Ausnahme (*exception*)

Ausnahmen (*exceptions*)

- Versuche (*try*), einen Block/eine Sequenz auszuführen.
- Ausnahme = Objekt, das einen Fehlerzustand signalisiert
  - In Java abgeleitet von `java.lang.Throwable` / `Exception`
  - Unterscheidung von Fehlern nach *Typ* (Klasse) der Ausnahme
- Aufwerfen (*throw* / *raise*) → „Sprung“ zu Fehlerbehandlung
  - Konstruktion eines Ausnahme-Objekts
  - „Aufwerfen“ des Objekts
- Abfangen (*catch*) der Ausnahme durch Fehlerbehandlung
  - Das kann auch in einer *aufrufenden* Methode passieren!
  - Programmabbruch, falls keine (passende) Fehlerbehandlung
  - Abfangen von *bestimmten Klassen* von Fehlern

Einfaches Beispiel: Ausnahmen in Java

```
try {
 ...
 if ( error_condition ) {
   throw new RuntimeException("error message");
 }
 ...
} catch (Exception e) {
  System.err.println(e.getMessage());
}
```

- Es können auch mehrere `catch` Blöcke stehen.
- Optionaler `finally` Block wird *immer* ausgeführt:
  nach `try` und vor `catch` (falls `throw`)
- *Abzufangende* Ausnahmen optional mit `throws` spezifizieren
- Siehe auch online Dokumentation!

Warum Ausnahmen?

- Ausnahmen *können* helfen, Code besser zu strukturieren.
  - Trenne Fehlerbehandlung vom „Hauptteil" des Codes
  - Sicherstellen, dass Ressourcen freigegeben werden (z.B. Dateien schließen, in C++ auch Speicher freigeben)
- Fehlerbehandlung muss nicht lokal sein
  - `throw` in Funktion f_k
  - `catch` in *beliebiger* aufrufender Funktion
    $f_0 \to f_1 \to \ldots \to f_{k-1} \to f_k$ ($f_0 \equiv$ `main`)
- Klassifizierung von Fehlern und Fehlertypen
  - als Klassenhierarchie
  - mit Basisklasse `java.lang.Exception`

Parametrisierung von ADT

- z.B. Typ `Item` als Parameter von `Set`, `List`, `Stack`, `Queue`
- Möglichkeiten zur Realisierung
  1. Vererbung und Polymorphie: Item ≡ `Object` und *type cast*
     z.B. Klasse `List` definiert `public Object head()`
     `List list; ... MyItem x=(MyItem) list.head()`
  2. Generische Typen (in Java: *generics*) Definiere
     `class List<Item>` mit `public Item head()`
- Bemerkungen zu Java *generics*
  - `class OperandStack<T extends Number>`
  - Diverse Einschränkungen (im Vergleich zu C++ *templates*)
- Siehe auch online Dokumentation

Code Beispiele

- Beispiele zu ADT online !
  - `List`
  - `Stack`
  - `Queue`
- Verschiedene Möglichkeiten der Implementierung
  - Alle benutzen Felder vom Typ `Item[]`.
  - Implementierung von `List` ist ineffizient!
  - Gleiches gilt naive Implementierung von `Stack` mittels `List`!
- Nächstes Semester (*Algorithmen und Datenstrukturen*) u.a.
  - Alternative zu Feldern: verkettete Listen
  - Effiziente Implementierung von Mengen (`Set`)

ADT List in Java

```java
public abstract class AbstractList<Item> {
    /// empty list []
    public AbstractList<Item>();
    /// get length
    public abstract int length();
    /// get head (fails if length()==0)
    public abstract Item head() throws
        RuntimeException;
    /// get tail as a new list
    public abstract AbstractList<Item> tail();
    /// append x as new head
    public abstract void append(Item x);
}
```

- Manipulator `append()` ändert Zustand: `void` Methode
- Implementierung als Unterklasse
  `public class List extends AbstractList`

```
abstract public class AbstractStack<Item> {
    /// default constructor creates empty stack!
    public AbstractStack() {}

    public abstract boolean is_empty();

    public abstract Item top() throws
        RuntimeException;

    public abstract void pop();

    public abstract void push(Item x);
}
```

- Oft bequemer: `Item pop()` liefert den entfernten Eintrag

Beispiel: Termauswertung in UPN in Java

```
void main(String[] args) {
  Stack<Integer> stack;

  for (int i=0;i<args.length;++i) {
    char c=args[i].charAt(0);
    if (args[i].length()==1 && (c=='+' || c=='*')) {
      int b=stack.top().intValue(); stack.pop();
      int a=stack.top().intValue(); stack.pop();
      stack.push(Integer.new((c=='+') ? a+b : a*b));
    }
    else
      stack.push(Integer.new(args[i]));
  }
  System.out.println(stack.top());
}
```

- Eingabe in Feld `args`
- Hier Beschränkung auf Summe und Produkt

Zusammenfassung

- Allgemeine Ziele im Software-Design:
  - Strukturierung
  - Wiederverwendbarkeit
  - Erweiterbarkeit
- ADT: Trenne Beschreibung von Implementierung
  - Kapselung: Interaktion nur über Schnittstelle
  - Geheimnisprinzip: Realisierung bleibt verborgen
- Objektorientierte Programmierung nutzt gleiche Prinzipien!
- Formale Beschreibung von ADT
  - Signatur $=$ Schnittstelle
  - Spezifikation von Semantik z.B. durch System von Axiomen
- Grundlegende Datenstrukturen
  - Stack
  - Queue