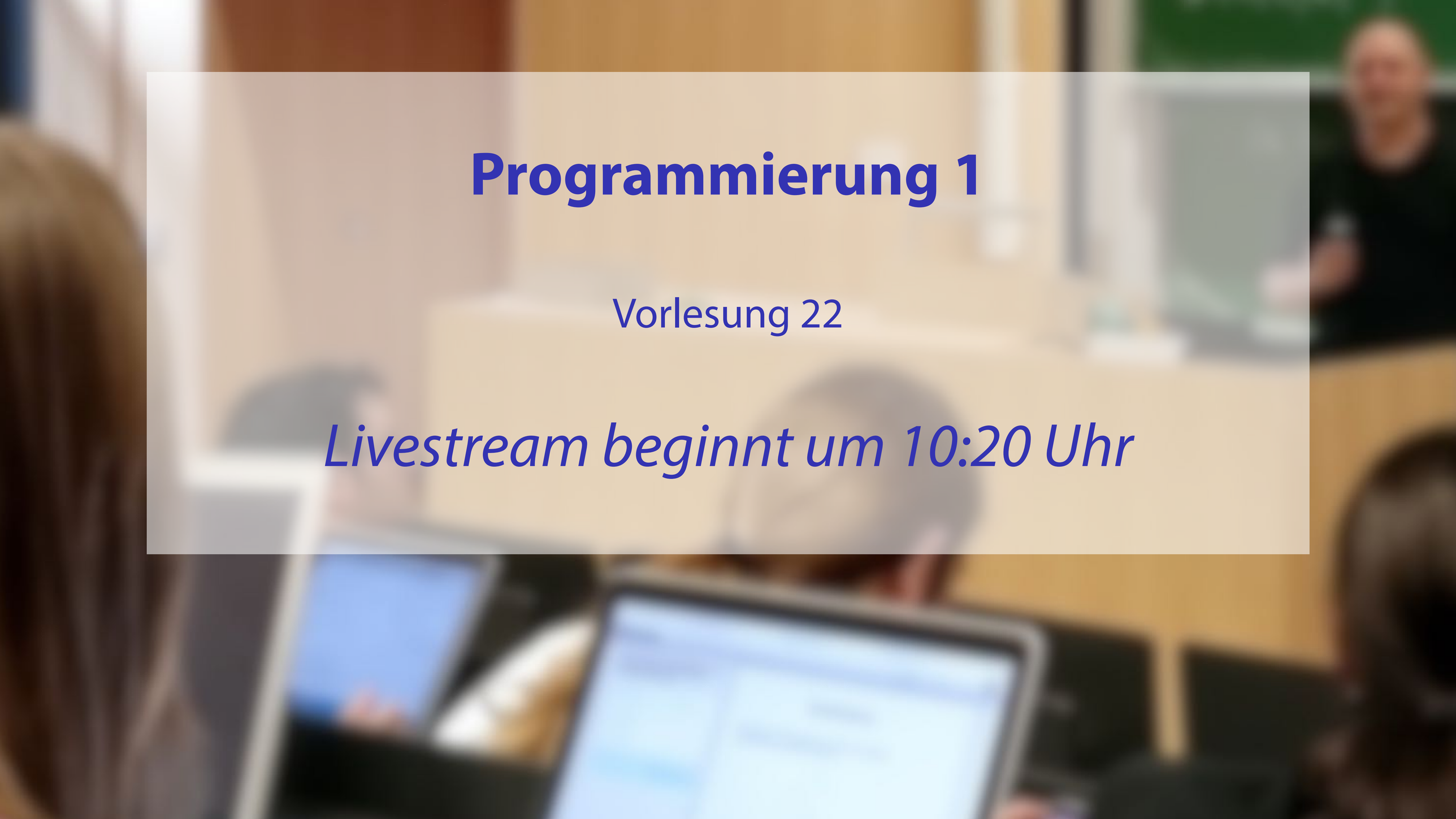


Programmierung 1

Vorlesung 22

Livestream beginnt um 10:20 Uhr

Datenstrukturen
Teil 2

Programmierung 1

Implementierung von Datenstrukturen

- Strukturdeklarationen sind dazu da, die **Implementierung** einer Datenstruktur zu **kapseln**.

Beispiel: (Endliche) Mengen von ganzen Zahlen

```
structure ISet = struct
    type set = int list
    fun set xs = xs
    fun union xs ys = xs@ys
    fun elem ys x = List.exists (fn y => y=x) ys
    fun subset xs ys = List.all (elem ys) xs
end
```

Verbergen von Implementierungsdetails

- Problem: Die **Darstellungsgleichheit** (Gleichheit von Listen) stimmt nicht mit der **abstrakten Gleichheit** (Gleichheit von Mengen) überein!
- Lösung: **Verbergen von Implementierungsdetails** durch einen **Signaturconstraint:**

```
signature ISET = sig
    type set
    val set    : int list -> set
    val union  : set -> set -> set
    val subset : set -> set -> bool
end
```

Nur die in dem Signaturconstraint **angegebenen Felder** sind für den Benutzer der Signatur **sichtbar.**

Verbergen von Implementierungsinformation

```
signature ISET = sig
    type set
    val set    : int list -> set
    val union  : set -> set -> set
    val subset : set -> set -> bool
end

structure ISet :> ISET = struct
    type set = int list
    fun set xs = xs
    fun union xs ys = xs@ys
    fun elem ys x = List.exists (fn y => y=x) ys
    fun subset xs ys = List.all (elem ys) xs
end
```

abstrakter Typ
Gleichheitstest wird verborgen.
(sichtbar bei eqtype)

Typannahmen
werden überprüft

Implementierung darf allgemeiner sein als Signatur.

polymorphe Implementierung

elem ist **verborgen**

Strukturen und Signaturen

$$\begin{aligned} &\mathit{structure}\ \langle \mathit{Bezeichner} \rangle\ :>\ \langle \mathit{Signaturausdruck} \rangle\ = \\ &\qquad \mathit{struct}\ \langle \mathit{Deklaration} \rangle\ \ldots\ \langle \mathit{Deklaration} \rangle\ \mathit{end} \end{aligned}$$

- Eine **Signatur** beschreibt die **Schnittstelle** zwischen der **Benutzung** und der **Implementierung** einer Struktur.
- **Signaturen** verhalten sich zu **Strukturen** so, wie sich **Typen** zu **Werten** verhalten.
  Wenn eine Struktur ohne Signaturconstraint eingeführt wird, sieht sie der Benutzer gemäß einer automatisch abgeleiteten Signatur, die keine Implementierungsdetails verbirgt.
- Per **Konvention** schreibt man Bezeichner für Strukturen und Signaturen **groß**. Bezeichner für **Signaturen** schreibt man darüber hinaus **nur mit Großbuchstaben**.
- In **let-Ausdrücken** ist die Deklaration von Strukturen und Signaturen unzulässig.

Ist die folgende Deklaration zulässig?

```
structure S :> sig eqtype t end =
struct type t = int -> int end
```

A: Ja ✗

B: Nein ✓

Abstrakte Datenstrukturen

- Man unterscheidet zwischen der **Spezifikation** und der **Implementierung** einer Datenstruktur.
- Die **Spezifikation einer Datenstruktur** beschreibt die Typen und Operationen der Datenstruktur **aus Benutzersicht**, ohne sich auf eine bestimmte Implementierung festzulegen.
  Wir sprechen von einer **abstrakten Datenstruktur** oder einem **abstrakten Datentyp (ADT).**
- Abstrakte Datenstrukturen können durch eine Signatur und eine **Modellimplementierung** beschrieben werden.
  Diese legt die Semantik der Operationen (d.h. die Funktionsweise) fest, nicht aber die Laufzeiten.
- Verschiedene Implementierungen sind aus Benutzersicht (bis auf die Laufzeiten) von der Modellimplementierung **ununterscheidbar.**

Trade-Offs

Oft können einzelne Operationen einer Datenstruktur **auf Kosten anderer** Operationen schneller gemacht werden.

Beispiel ISet:

- **Laufzeit der Modellimplementierung:**
  set **konstant**
  union **linear** (in der Länge des ersten Arguments)
  subset **quadratisch** (in der Summe der Längen)
- **Laufzeit einer alternativen Implementierung mit strikt sortierten Listen:**
  set **linear-logarithmisch**
  union **linear** (in der Summe der Längen)
  subset **linear** (im Minimum der Längen)

Welche Implementierung insgesamt am effizientesten ist, hängt davon ab, auf welchen Argumenten und wie oft die verschiedenen Operationen angewandt werden.

Vektoren

- **Listen** sind eine Datenstruktur für endliche Folgen.
  - Anfügen an eine Liste (::): konstante Laufzeit
  - **Zugriff auf Position** (*List.nth*): **lineare Laufzeit**
- **Vektoren:** alternative Datenstruktur für endliche Folgen
  - Anfügen an einen Vektor (*Vector.concat*): lineare Laufzeit
  - **Zugriff auf Position** (*Vector.sub*): **konstante Laufzeit**

Trade-Off:

- Werden Folgen **Schritt-für-Schritt** aufgebaut:
  Listen vorteilhaft
- Wird **oft** auf **spezifische Elemente** der Folge zugegriffen:
  Vektoren vorteilhaft

Signatur der Standardstruktur Vektor (Ausschnitt)

$$eqtype\ \alpha\ vector$$

linear	$val\ fromList$	$:$	$\alpha\ list \rightarrow \alpha\ vector$
linear	$val\ tabulate$	$:$	$int * (int \rightarrow \alpha) \rightarrow \alpha\ vector$
konstant	$val\ sub$	$:$	$\alpha\ vector * int \rightarrow \alpha \quad (* Subscript *)$
konstant	$val\ length$	$:$	$\alpha\ vector \rightarrow int$
linear	$val\ map$	$:$	$(\alpha \rightarrow \beta) \rightarrow \alpha\ vector \rightarrow \beta\ vector$
linear	$val\ foldl$	$:$	$(\alpha * \beta \rightarrow \beta) \rightarrow \beta \rightarrow \alpha\ vector \rightarrow \beta$
linear	$val\ foldr$	$:$	$(\alpha * \beta \rightarrow \beta) \rightarrow \beta \rightarrow \alpha\ vector \rightarrow \beta$
linear	$val\ concat$	$:$	$\alpha\ vector\ list \rightarrow \alpha\ vector$

besser als Listen (konstant vs. linear)

gleich gut (linear)

schlechter als Listen (linear in Länge des ersten Arguments vs. linear in Summe der Längen)

Suche in sortierten Folgen

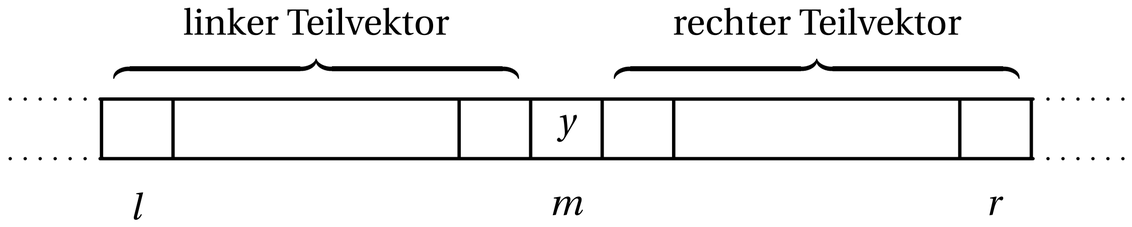
- **Lineare Suche** auf einer sortierten Liste:

```
fun linearsearch compare ls x =
let
   fun position(n, nil) = NONE
     | position(n, y::yr) = case compare (y,x) of
                              EQUAL => SOME n
                            | LESS => position(n+1, yr)
                            | GREATER => NONE
in
   position(0, ls)
end
```

```
linearsearch Int.compare [1,2,5,6,7,8,10] 8;
> val it = SOME 5 : int option
```

Laufzeit hat **lineare Komplexität** in der Länge der Liste.

Suche in sortierten Folgen



- **Binäre Suche** auf einem sortierten Vektor:

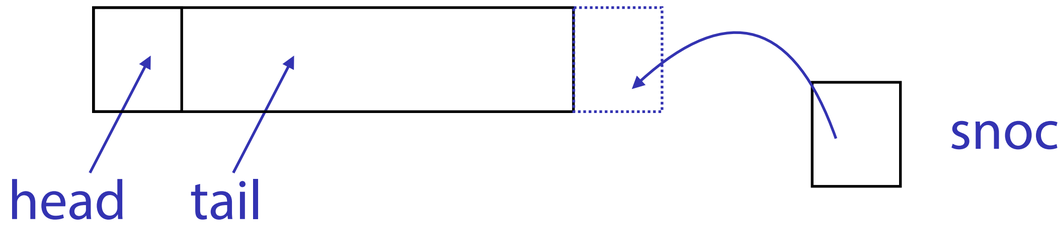
```
fun binarysearch compare v x =
let
  fun position l r =
       if l>r then NONE
       else let val m = (l+r) div 2
                val y = Vector.sub(v,m)
            in case compare(x,y) of
                          EQUAL => SOME m
                        | LESS => position l (m-1)
                        | GREATER => position (m+1) r
            end
in
  position 0 (Vector.length v - 1)
end
```

Laufzeit hat **logarithmische Komplexität** in der Länge des Vektors.

Schlangen

- **Listen** realisieren Folgen nach dem **last-in-first-out (LIFO)** Prinzip
- **Schlangen** realisieren Folgen nach dem **first-in-first-out (FIFO)** Prinzip

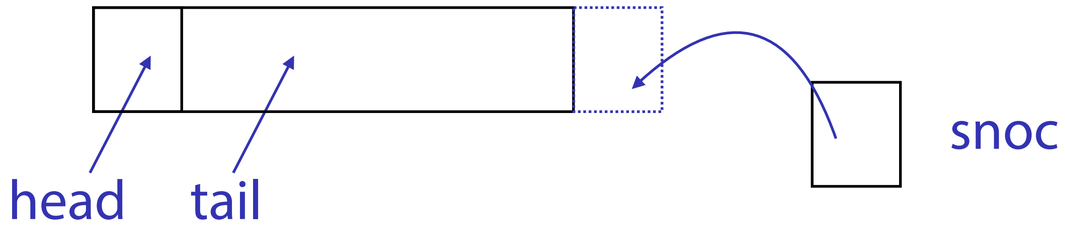
```
signature QUEUE = sig
    type 'a queue
    val empty  : 'a queue
    val snoc : 'a queue -> 'a -> 'a queue
    val head   : 'a queue -> 'a                (* Empty *)
    val tail   : 'a queue -> 'a queue          (* Empty *)
end
```



Modellimplementierung

```
signature QUEUE = sig
    type 'a queue
    val empty  : 'a queue
    val snoc : 'a queue -> 'a -> 'a queue
    val head   : 'a queue -> 'a                (* Empty *)
    val tail   : 'a queue -> 'a queue          (* Empty *)
end

structure Queue :> QUEUE = struct
    type 'a queue = 'a list
    val empty = nil
    fun snoc q x = q@[x]
    val head = hd
    val tail = tl
end
```



Was ist die Komplexität der Laufzeit einer Folge von *n* beliebigen Operationen der Modellimplementierung?

A: O(1) ✗

B: O(*n*) ✗

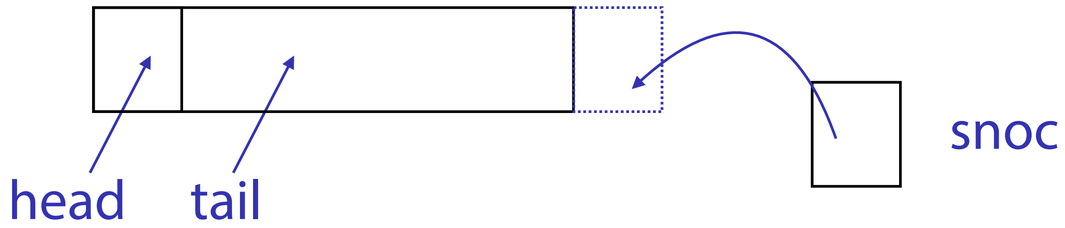
C: O(*n* log *n*) ✗

D: O(n^2) ✓

Laufzeit

- In der Modellimplementierung hängt die Laufzeit von *snoc* **linear** von der Länge der zu erweiternden Schlange ab.
- Für eine Folge von n Erweiterungsschritten mit der Operation *snoc*, die nacheinander auf die leere Schlange angewendet werden, fällt damit **quadratische** Laufzeit $O(n^2)$ an.

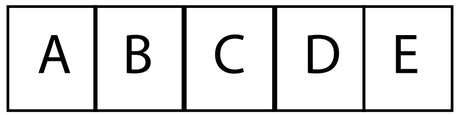
```
structure Queue :> QUEUE = struct
    type 'a queue = 'a list
    val empty = nil
    fun snoc q x = q@[x]
    val head = hd
    val tail = tl
end
```



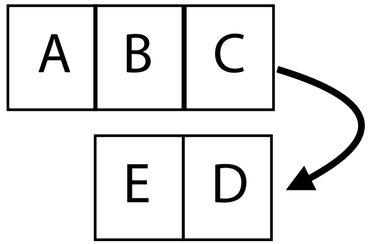
Effiziente Implementierung von Schlangen

```
structure FQueue :> QUEUE = struct
    type 'a queue = 'a list * 'a list
    val empty = ([],[])
    fun snoc ([],_) x = ([x],[])
      | snoc (q, r) x = (q, x::r)
    fun head (q, r) = hd q
    fun tail ([x], r) = (rev r, [])
      | tail ( q , r) = (tl q, r)
end
```

Modell-
implementierung:



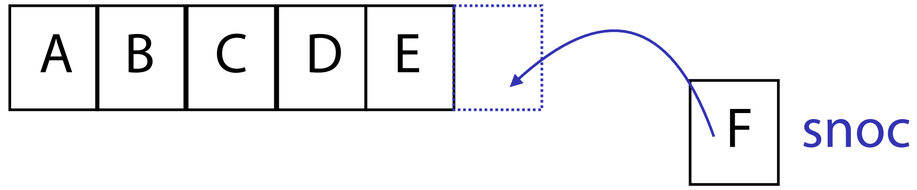
effiziente
Implementierung:



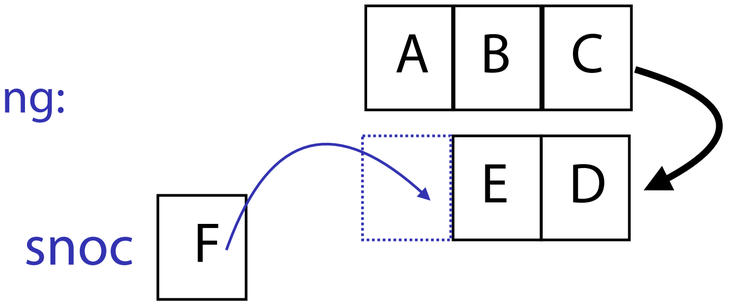
Effiziente Implementierung von Schlangen

```
structure FQueue :> QUEUE = struct
    type 'a queue = 'a list * 'a list
    val empty = ([],[])
    fun snoc ([],_) x = ([x],[])
      | snoc (q, r) x = (q, x::r)
    fun head (q, r) = hd q
    fun tail ([x], r) = (rev r, [])
      | tail ( q , r) = (tl q, r)
end
```

Modell-
implementierung:



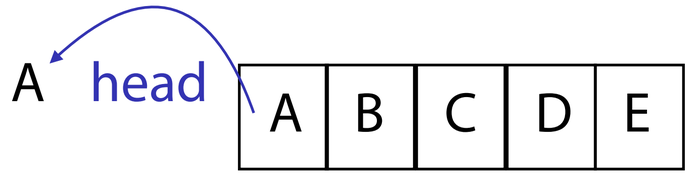
effiziente
Implementierung:



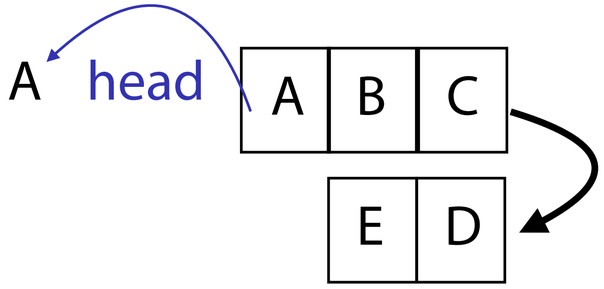
Effiziente Implementierung von Schlangen

```
structure FQueue :> QUEUE = struct
    type 'a queue = 'a list * 'a list
    val empty = ([],[])
    fun snoc ([],_) x = ([x],[])
      | snoc (q, r) x = (q, x::r)
    fun head (q, r) = hd q
    fun tail ([x], r) = (rev r, [])
      | tail ( q , r) = (tl q, r)
end
```

Modell-implementierung:



effiziente Implementierung:

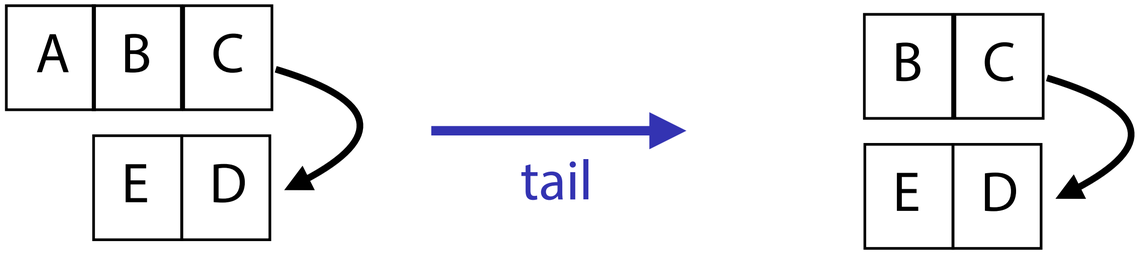


Effiziente Implementierung von Schlangen

```
structure FQueue :> QUEUE = struct
    type 'a queue = 'a list * 'a list
    val empty = ([],[])
    fun snoc ([],_) x = ([x],[])
      | snoc (q, r) x = (q, x::r)
    fun head (q, r) = hd q
    fun tail ([x], r) = (rev r, [])
      | tail ( q , r) = (tl q, r)
end
```

Modell-implementierung:

effiziente Implementierung:

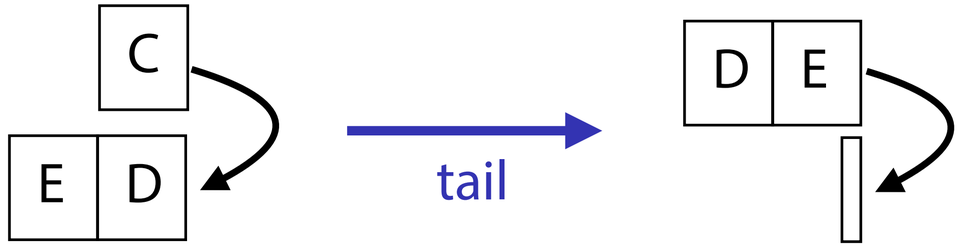


Effiziente Implementierung von Schlangen

```
structure FQueue :> QUEUE = struct
    type 'a queue = 'a list * 'a list
    val empty = ([],[])
    fun snoc ([],_) x = ([x],[])
      | snoc (q, r) x = (q, x::r)
    fun head (q, r) = hd q
    fun tail ([x], r) = (rev r, [])
      | tail ( q , r) = (tl q, r)
end
```

Modell-implementierung:

effiziente Implementierung:

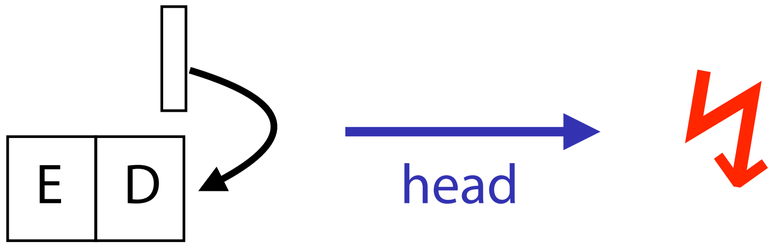


Laufzeit

- *empty*, *snoc*, und *head* haben **konstante Laufzeit.**
- Laufzeit von *tail* ist **nicht uniform**: konstant außer wenn erste Liste einelementig, dann linear in zweiter Liste.
- **Beobachtung:** Reversionskosten sind durch Anzahl vorausgehender *snoc* Operationen beschränkt.
- **Akkumuliert betrachtet haben damit alle Operationen konstante Laufzeit.**

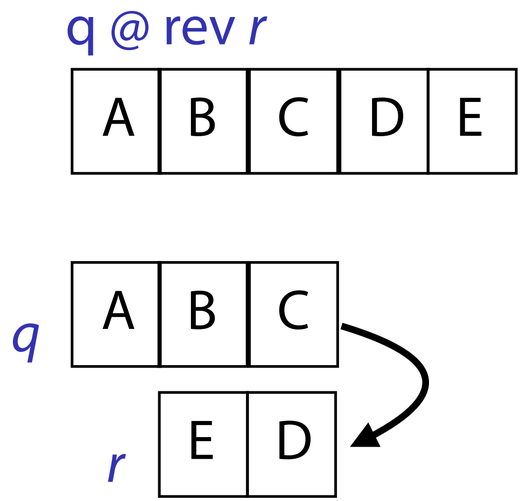
Korrektheit?

```
structure FQueue :> QUEUE = struct
    type 'a queue = 'a list * 'a list
    val empty = ([],[])
    fun snoc ([],_) x = ([x],[])
      | snoc (q, r) x = (q, x::r)
    fun head (q, r) = hd q
    fun tail ([x], r) = (rev r, [])
      | tail ( q , r) = (tl q, r)
end
```



Darstellungsinvariante

```
structure FQueue :> QUEUE = struct
    type 'a queue = 'a list * 'a list
    val empty = ([],[])
    fun snoc ([],_) x = ([x],[])
      | snoc (q, r) x = (q, x::r)
    fun head (q, r) = hd q
    fun tail ([x], r) = (rev r, [])
      | tail ( q , r) = (tl q, r)
end
```



Die Implementierung von Schlangen als Paar (*q*,*r*) erfüllt die folgende **Darstellungsinvariante**:

- Die Liste *q* @ rev *r* stellt die **Schlange** dar
- Wenn *q* **leer** ist, dann ist auch *r* **leer**.

- Beweis per Induktion.

Modularisierung in ML

- **Signaturen:**

```
signature ISET = sig
    type set
    val set    : int list -> set
    val union  : set -> set -> set
    val subset : set -> set -> bool
end
```

- **Strukturen:**

```
structure ISet :> ISET = struct
    type set = int list
    fun set xs = xs
    fun union xs ys = xs@ys
    fun elem ys x = List.exists (fn y => y=x) ys
    fun subset xs ys = List.all (elem ys) xs
end
```

- **Funktoren:** **parametrierte** Datentypen

Ein Funktor für endliche Mengen

```
functor Set
(
    type t
    val compare : t * t -> order
)
:>
sig
    type set
    val set : t list -> set
    val union : set -> set -> set
    val subset : set -> set -> bool
end
=
struct
    type set = t list
    val set = ssort compare
    val union = smerge compare
    val subset = ssublist compare
end
```

Funktoranwendung

```
structure StringSet = Set(type t = string
                          val compare = String.compare)
```

$$
\begin{aligned}
&\mathit{type\ set}\\
&\mathit{val\ set}: \mathit{string\ list} \rightarrow \mathit{set}\\
&\mathit{val\ union}: \mathit{set} \rightarrow \mathit{set} \rightarrow \mathit{set}\\
&\mathit{val\ subset}: \mathit{set} \rightarrow \mathit{set} \rightarrow \mathit{bool}
\end{aligned}
$$

Kapitel 15
Speicher und veränderliche Objekte

Unveränderliche und veränderliche Objekte

- **Mathematische Objekte,** wie z.B. 13, { λ *x* . *x*, λ *x* . 5 }
  sind **unveränderliche Objekte.**
- **Physikalische Objekte,** wie z.B. Uhren oder Computer,
  sind **veränderliche Objekte.**
  Sie ändern Ihren **Zustand** im Laufe der Zeit.

- Ein **Speicher** ist ein physikalisches Objekt, das in **Zellen** unterteilt ist.
  In jeder **Zelle** kann ein **Wert** dargestellt werden.
  Die Zellen des Speichers sind nummeriert.
  Die **Nummern** werden als **Referenzen** bezeichnet.

Zellen und Referenzen

- Für **Programme** ist der Speicher als **abstrakte Datenstruktur** sichtbar:

$eqtype\ \alpha\ ref$	**Referenztypen**
$ref\ :\ \alpha \rightarrow \alpha\ ref$	**Allokation**
$!\ \ :\ \alpha\ ref \rightarrow \alpha$	**Dereferenzierung**
$:=\ :\ \alpha\ ref * \alpha \rightarrow unit$	**Zuweisung**

- ***ref x*** wählt eine bisher **nicht benutzte Zelle** aus, **schreibt** die Darstellung des Wertes von *x* in die Zelle und **liefert die Referenz** der Zelle.
- ***!r*** liefert zu einer **Referenz** *r* den in der Zelle dargestellten **Wert.**
- ***r := x*** **schreibt** die Darstellung des Wertes x in eine **bereits allozierte** Zelle.

Was ist das Ergebnis von

```
let val x = ref 25
in (x := !x + 5; !x + !x) end ?
```

A: 25 ✗

B: 50 ✗

C: 60 ✓

D: () ✗

Referenzen

- **Zuweisung :=** bezeichnen wir auch als **Schreiben**.
  Wir **setzen** eine Referenz **auf einen Wert.**
  Wir **weisen** der **Referenz** einen Wert **zu**.

- **Zuweisung :=** bindet **schwächer** als jeder andere Operator,
  ausser der Deklaration mit =

```
val v = f x := y + 4          val v = ( (f x) := (y+4) )
```

- **Dereferenzierung !** bezeichnen wir auch als **Lesen.**
  Der **Wert einer Referenz** ist der Wert in der durch die Referenz
  bezeichneten Zelle.

- **Dereferenzierung !** und **Allokation *ref*** werden **wie Bezeichner**
  behandelt.

```
!(!(ref(ref 1)))
```

Ambige Deklarationen

```
let
    val r = ref (fn x => x)
in
    r:= (fn () => ()) ;
    1 + (!r 4)
end
```

Der Ausdruck ist aus folgenden Gründen **unzulässig**:

- Die Deklaration von *r* ist **ambig**, da ihre rechte Seite eine Applikation ist. Also muss *r* mit einem Typen $(t \rightarrow t)\ ref$ typisiert werden.
- Das **erste** benutzende Auftreten von *r* verlangt den Typ $(unit \rightarrow unit)\ ref$.
- Das **zweite** benutzende Auftreten von *r* verlangt einen Typ $(int \rightarrow int)\ ref$.

Ambige Deklarationen

```
let
    val r = ref (fn x => x)
in
    r:= (fn () => ()) ;
    1 + (!r 4)
end
```

Angenommen *r* wäre polymorph getypt:

$$\forall \alpha.\ (\alpha \rightarrow \alpha)\ ref$$

- Dann wäre der Ausdruck gemäß der **statischen Semantik** zulässig.
- Bei der Ausführung von !*r* 4 käme es aber zu einem **Typfehler.**

Funktionale und imperative Werte

- Ein **funktionaler Wert** beinhaltet **keine Referenzen.**
  Funktionale Werte sind **unveränderliche** Objekte.
- Ein **imperativer Wert** beinhaltet **Referenzen**.
  Imperative Werte sind **veränderliche** Objekte.
  Z.B. Referenzen, Paare von Referenzen sind imperative Werte
- Der **Zustand eines imperativen Objektes** ergibt sich aus dem **Speicherzustand.**
- Wenn die Ausführung einer Phrase den Speicherzustand verändert sprechen wir von dem **Speichereffekt.**
  - **Allokation** einer Zelle
  - **Zuweisung** eines Wertes an eine bereits allozierte Zelle
- Für Speichereffekte verwendet man oft **Prozeduren** mit Ergebnistyp **unit.**

Speichereffekt

```
fun swap r r' = r:= #1(!r', r':= !r)
```

val swap : α ref → α ref → unit

1. Beschaffe den Wert der Referenz r'.
2. Beschaffe den Wert der Referenz r.
3. Weise r' den Wert aus (2) zu.
4. Weise r den Wert aus (1) zu.

```
let val (r,r') = (ref 2, ref 3) in swap r r' ; (!r, !r') end
```

*(3,2) : int * int*

App

Die Prozedur **app** wendet eine Prozedur *p*
von links nach rechts auf die Elemente einer Liste an:

```
fun app p nil = ()
  | app p (x::xr) = (p x : unit ; app p xr)
```

val app : (α → unit) → α list → unit

```
val xs = map ref [2, 3, 6]
```

val xs = [ref, ref, ref] : int ref list

```
map ! xs
```

[2, 3, 6] : int list

```
app (fn r => r:= !r+7) xs
```

() : unit

```
map ! xs
```

[9, 10, 13] : int list

Imperative Prozeduren

Die **imperative Prozedur** **counter** zählt in einer Speicherzelle mit, wie oft sie aufgerufen wird:

```
val r = ref 0
```
val r : int ref

```
fun counter () = (r:= !r+1 ; !r)
```
val counter : unit → int

```
counter()
```
1 : int

```
(counter(),  counter(), counter())
```
*(2, 3, 4) : int * int * int*

```
counter() + counter()
```
11 : int

www.prog1.saarland