LISTEN ALS ABSTRAKTE DATENTYPEN

Listen

Formal:

Sei A eine geg. Menge von Datenelementen;

<u>Liste</u> $l = \langle a_1, a_2, \ldots, a_n \rangle$ endliche Folge von Elementen aus A, mit

n: ***Länge*** der Liste l

a_1: ***erstes Element*** von l (head)

$\langle a_2, \ldots, a_n \rangle$: ***Rest*** der Liste l (tail)

$\langle\ \rangle$: ***leere*** Liste (Länge $n = 0$)
(wir schreiben auch $\langle a_1, a_2, \ldots, a_n \rangle$ und $n = 0$).

a_i: ***Vorgänger*** von a_{i+1} in l, $1 \le i < n$

a_j: ***Nachfolger*** von a_{j-1} in l, $1 < j \le n$

p: ***Position*** (Index, Stelle) von a_p in l, $1 \le p \le n$

Jeder Liste l ist eine Menge $IT(l) = \{it_1, it_2, \ldots\}$ von **Iteratoren** zugeordnet, die zu ihrer Durchmusterung und Modifikation dienen. Ein Iterator $it \in IT(l)$ hat einen Wert $val(it) \in \{1, 2, \ldots, n\} \cup \{nil\}$.

1. Ist $val(it) \neq nil$, so ist *val(it)* der Index eines Listenelementes aus l.
2. Der spezielle Wert $val(it) = nil$ bedeutet, daß *it* auf kein Listenelement verweist.

Ist $it \in IT(l)$, so wird durch die unten beschriebenen Operationen jeweils $1 \le val(it) \le n$ bzw. $val(it) = nil$ sichergestellt. Sollte das nicht möglich sein (beispielsweise werden durch das Löschen eines Listenelementes alle Iteratoren, die auf dieses Listenelement verweisen, ungültig), so wird *it* aus dieser Menge entfernt. Es ist also nur erlaubt, Iteratoren $it \in IT(l)$ zu verwenden.

Es muß sichergestellt sein, daß der Elementtyp A die folgenden Operationen implementiert, die im ADT Elem definiert sind:

ADT Elem

Sei *e* von Typ Elem: Die folgenden Operationen sind definiert:

Elem copy(): Kopieroperation;

(V) -

(N) *e*.copy() = *e*

boolean equals(Elem e_1): Test auf Gleichheit;

(V) -

(N) *e*.equals(e_1) = **true** gdw. $e = e_1$

boolean less(Elem e_1): Test auf “Kleiner”;

(V) -

(N) *e*.less(e_1) = **true** gdw. $e < e_1$, wobei < eine auf dem Datentyp Elem definierte Halbordnung ist

String toString(): Konvertierungsoperation;

(V) -

(N) *e*.toString() = < textuelle Repräsentation von *e* >

ADT: List (*A*)

LIST (*A*) besteht aus:

- Der Menge der Listen (endliche Folgen) über *A*.
- Einem ADT Iterator, dessen Elemente die Durchmusterung und Modifikation von Listen erlauben. Jedem $l \in List(A)$ ist eine Menge *IT(l)* von Iteratoren zugeordnet.
- Die folgenden Operationen[1] sind für jedes $l \in List(A)$ definiert:

List(): Konstruktor-Methode;

(V) -

(N) ist *l* eine durch *List()* kreierte Liste, so gilt $l = \langle\ \rangle$, $IT(l) = \varnothing$

1. Achtung: Diese Operationen modifizieren u. U. nicht nur *l*, sondern ebenso *IT(l)*.

clear(): Löschen - macht *l* zur leeren Liste (Re-initialisierung);

(V) -

(N) $l' = \langle\ \rangle, IT(l') = \{it \in IT(l) : val(it) = nil\}$

boolean isEmpty(): Test auf die leere Liste;

(V) -

(N) *l*.isEmpty() = **true** gdw. $l = \langle\ \rangle$

int length(): Länge der Liste;

(V) -

(N) *l*.length = *n*, falls $l = \langle a_1, a_2, \ldots, a_n\rangle \wedge n > 0$

l.length = 0, falls $l = \langle\ \rangle$

append (*Elem e*): Anhängen von e an die Liste;

(V) -

(N) ist $l = \langle a_1, a_2, \ldots, a_n\rangle \wedge n > 0$,
so gilt $l' = \langle a_1, a_2, \ldots, a_n, e\rangle$
ist $l = \langle\ \rangle$,
so gilt $l' = \langle e\rangle$

connect(List l_1*)*: Verketten zweier Listen (Ergebnis über *l* bereitstellen);

(V) -

(N) für $l = \langle a_1, a_2, \ldots, a_n\rangle$ und $l_1 = \langle b_1, b_2, \ldots, b_m\rangle$ gilt
$l' = \langle a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m\rangle$

Anm.: bei *m* = 1 wie *l*.append

Iterator getIterator(): Erzeugen eines passenden Iterators;

(V) -

(N) Ist *it* ein durch *l*.getIterator() kreierter Iterator, so gilt
$val(it) = nil$ und $IT(l') = IT(l) \cup \{it\}$

Sei $l \in List(A)$ und sei $it \in IT(l)$ ein Iterator der Liste *l*.

insert(Iterator it, Elem e): Ein neues Element e vor dem indizierten Element einfügen;

(V) -

(N) ist $val(it) = nil$ und $l = \langle\ \rangle$, so gilt
$l' = \langle e\rangle$, $val(it') = nil$, und $IT(l') = IT(l)$,
anderenfalls gilt für $l = \langle a_1, a_2, \ldots, a_n\rangle$, $(n > 0)$
$l' = \langle e, a_1, a_2, \ldots, a_n\rangle$, falls $val(it) = nil$,
$l' = \langle a_1, a_2, \ldots, e, a_{val(it)}, \ldots, a_n\rangle$ sonst;

$val(it') = nil$, falls $val(it) = nil$,
$val(it') = val(it) + 1$, sonst;

$IT(l') = \{it_1 \in IT(l) :$
$val(it_1) = nil \vee val(it_1) < val(it)\} \cup \{it\}$

locate(Iterator it, Elem e): Setzt *it* auf den nächstgrößeren Index $i > val(it)$ von *l*, so daß $a_i = e$ gilt, falls ein solcher Index existiert;

(V) -

(N) *Sei* $l = \langle a_1, a_2, \ldots, a_n\rangle$ für $n \geq 0$;

1. falls $(val(it) \neq nil \wedge \forall i\,(val(it) < i \leq n \Rightarrow a_i \neq e))$
oder $(val(it) = nil \wedge \forall i\,(1 \leq i \leq n \Rightarrow a_i \neq e))$ gilt,
so ist $val(it') = nil$
2. andernfalls,
wenn $val(it) = nil$ gilt und *i* derjenige Index mit
$a_i = e \wedge \forall j(1 \leq j < i \Rightarrow a_j \neq e)$
ist,
oder $val(it) \neq nil$ gilt und *i* derjenige Index mit
$a_i = e \wedge \forall j(val(it) < j < i \Rightarrow a_j \neq e)$
ist,
so gilt $val(it') = i$

Elem retrieve(Iterator it): Zugriff - indiziertes Element bereitstellen;

(V) $val(it) \neq nil$

(N) *für* $l = \langle a_1, a_2, \ldots, a_n \rangle$ ist *l*.retrieve$(it) = a_{val(it)}$

delete (*Iterator it*): Entfernen - indiziertes Element löschen;

(V) $val(it) \neq nil$

(N) ist
$l = \langle a_1, \ldots, a_{val(it)-1}, a_{val(it)}, a_{val(it)+1}, \ldots, a_n \rangle$,
so gilt $l' = \langle a_1, \ldots, a_{val(it)-1}, a_{val(it)+1}, \ldots, a_n \rangle$,

$val(it') = nil$, falls $val(it) = n$
$val(it') = val(it)$, sonst

$IT(l') = \{it_1 \in IT(l):$
$val(it_1) = nil \vee val(it_1) < val(it)\} \cup \{it\}$

replace(Iterator it, Elem e): Ersetzen - indiziertes Element austauschen;

(V) $val(it) \neq nil$

(N) $a_{val(it)} = e$ für $l = \langle a_1, a_2, \ldots, a_n \rangle$

Die folgenden Operationen stellen sicher, daß List(*A*) selbst Implementierung des ADT Elem sein kann. Damit sind Listen von Listen der Form List(List(*A*)) möglich.

List copy(): Kopieroperation;

(V) -

(N) *l*.copy() = *l*, *IT*(*l*.copy()) = $\varnothing$

boolean equals(Elem e): Test auf Gleichheit;

(V) -

(N) *l*.equals(l_1) = true gdw. $l = l_1$, d. h. für
$l = \langle a_1, a_2, \ldots, a_n \rangle$, $l_1 = \langle b_1, b_2, \ldots, b_m \rangle$ gilt
$n = m \wedge \forall i (1 \leq i \leq n \Rightarrow a_i = b_i)$

boolean less(Elem e): Test auf “Kleiner”;

(V) -

(N) *l*.less(l_1) = **true** gdw. $l < l_1$, d. h. für
$l = \langle a_1, a_2, \ldots, a_n \rangle$, $l_1 = \langle b_1, b_2, \ldots, b_m \rangle$ gilt
$n = m \wedge \forall i (1 \leq i \leq n \Rightarrow a_i < b_i)$

String toString(): Konvertierungsoperation;

(V) -

(N) *l*.toString() = < textuelle Repräsentation von *l* >

ADT Iterator

Sei $l \in List(A)$ und sei $it \in IT(l)$ ein Iterator der Liste *l*. Die folgenden Operationen[2] sind definiert:

first(): Erstes Listenelement indizieren;

(V) -

(N) $val(it') = nil$, falls *l*.length()= 0,
$val(it') = 1$, sonst

2. Achtung: Diese Operationen modifizieren u. U. *it, l* und *IT*(*l*).

last(): Letztes Listenelement indizieren;

(V) -

(N) $val(it') = nil$, falls *l*.length()= 0,
$val(it') =$ *l*.length(), sonst

boolean hasMore(): Test auf *val(it)* = *nil*;

(V) -

(N) *it*.hasMore() = **true** gdw. $val(it) \neq nil$

next(): Inkrementieren;

(V) $val(it) \neq nil$

(N) $val(it') = val(it) + 1$, falls $val(it) <$ *l*.length(),
$val(it') = nil$, sonst

boolean hasPrev(): Test auf Vorgänger;

(V) -

(N) *it*.hasPrev() = **true** gdw. $val(it) \neq nil$ and $val(it) > 1$

prev(): Dekrementieren;

(V) $val(it) \neq nil$

(N) $val(it') = val(it) - 1$, falls $1 < val(it) \leq$ *l*.length(),
$val(it') = nil$, sonst

Iterator copy(): Kopieroperation;

(V) -

(N) Ist $it_1 =$ *it*.copy(), so gilt
$IT(l') = IT(l) \cup \{it_1\}$ und $val(it_1) = val(it)$

setToIndex(int i): Auf den Index *i* setzen;

(V) $1 \leq i \leq$ *l*.length()

(N) $val(it') = i$

IMPLEMENTIERUNG - ARRAYLISTT, OPERATIONEN FÜR LISTT[3]

(1) void clear()

size := 0 0(1)

(2) boolean isEmpty()

return (size = 0) 0(1)

(3) int length()

return (size) 0(1)

(4) void append(ElemT e)

if (size < maxSize) 0(1)
then
contents [size] := e;
size ++
endif

(5) void connect(ListT l)

falls genügend freier Platz (n+m ≤ maxSize)

⇨ Umspeichern der Elemente der anzuhängenden Liste 0(n)

3. skizziert, nicht ausprogrammiert

OPERATIONEN FÜR ITERATORT[4]

private class ArrayListIteratorT implements IteratorT
    protected int current = NIL;

Operationen

(6) void first()

    falls die Liste nicht leer: 0(1)
      current := 0

(7) void last()

    falls die Liste nicht leer: 0(1)
      current := size - 1

(8) boolean hasMore()

    return (current ≠ NIL) 0(1)

(9) void next()

    falls noch nicht Listenende erreicht: 0(1)
      current ++
    sonst: current := NIL

(10) boolean hasPrev()

    return (current ≠ NIL AND current ≠ 0) 0(1)

(11) void prev()

    falls current ≠ NIL: 0(1)
      current --

(12) Iterator copy()

    erzeuge einen neuen gleichwertigen Iterator O(1)

(13) setToIndex(int i)

    setze current auf i O(1)

4. lokale Klasse innerhalb ArrayListT

ITERATOR-GESTEUERTE LISTENOPERATIONEN

(14) IteratorT getIterator()

    return (new ArrayListIteratorT()) 0(1)

(15) void insert (IteratorT it, ElemT e)

    falls noch Platz: 0(n)
      size ++;
      $a_{val(it)}$, ... , a_n um eine Position nach rechts;
      e in Lücke ablegen

(16) void locate (IteratorT it, ElemT e)

    sequentielle Suche nach e ab der Iterator-Position 0(n)

(17) ElemT retrieve (IteratorT it)

    falls Iterator definiert, direkter Zugriff möglich: 0(1)
      return (contents [it.current])

(18) void delete (IteratorT it)

    die Array-Elemente rechts vom gelöschten Element 0(n)
      ($a_{val(it)+1}$, ... , a_n)
    um eine Position nach links rücken;
    size --

(19) void replace (IteratorT it, ElemT e)

    falls Iterator definiert: 0(1)
      contents [it.current] := e

Anmerkung:

- Die direkte Implementierung von replace ist effektiver (0(1))
  als die indirekte Implementierung über delete und insert (0(n)).
- allg. Suche benötigt O(n);
  für sortierte Folgen gibt es effektivere Verfahren mit O(log n);

ARRAYLISTT, HAUPTNACHTEILE

- Einfügen und Löschen sind relativ teuer (hier O(n));
- maximale Listengröße (relativ) fest eingestellt;
- gesamte Speicherbereich fest reserviert
  (d. h. kein dynamisches Wachsen/Schrumpfen);

  in Java kann jedoch bei Bedarf dynamisch ein größeres Array allokiert werden;
  -> das ist dann jedoch sehr aufwendig !

↳
VERKETTET GESPEICHERTE LISTEN

IMPLEMENTIERUNG - POINTERLIST1T, OPERATIONEN FÜR ITERATORT

```
private class PointerList1IteratorT implements IteratorT
        protected NodeT current = NULL;

(6)     void first()
            current := anchor;                                          0(1)

(7)     void last()
            Listenende suchen                                           0(n)

(8)     boolean hasMore()
            current ≠ NULL                                              0(1)

(9)     void next()
            if  ( current.next ≠ NULL )                                 0(1)
            then    current = current.next
            else    current := NULL
            endif                                   -> current := current.next

(10)    boolean hasPrev()
            es gibt einen Vorgänger, falls current                      0(1)
            nicht auf das 1. Listenelement (wie der Anker) zeigt:
              (current ≠ anchor AND current ≠ NULL)

(11)    void prev()
            aufwendig; mit einem Hilfszeiger p:                         0(n)
              1. suche nach p.next = current;
              2. current := p

(12)    Iterator copy()
            erzeuge einen neuen gleichwertigen Iterator                 O(1)

(13)    setToIndex(int i)
            i-te Listenelement durch abzählen suchen                    O(n)
```

POINTERLIST1T, HAUPTNACHTEILE

- i. allg. müssen viele Spezialfälle unterschieden werden
  -> leere Liste, Listenende;
- bei Operationen, die sich auf das Listenende beziehen
  (append, connect, last)
  muß man sich zunächst bis zum Listenende durchhangeln;
  -> O(n)

Bei den folgenden Implementierungen entfallen diese Schwierigkeiten teilweise

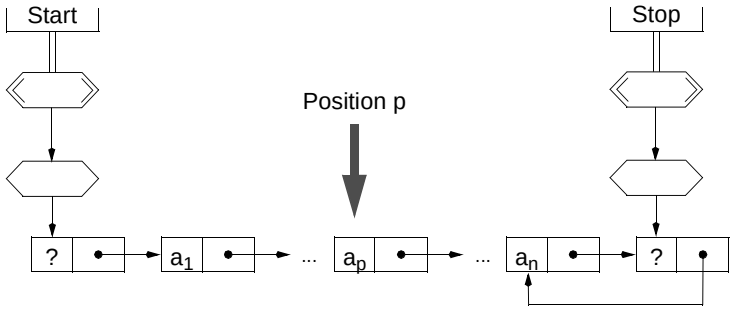
- eine dummy-Klammerung reduziert die Spezialfälle
- Zeiger zum Listenende reduziert die Komplexität

↳

POINTERLIST2T
POINTERLIST3T
POINTERLIST4T

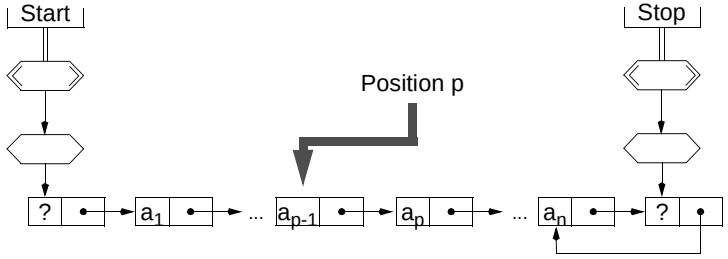
IMPLEMENTIERUNG - POINTERLIST2T

- Liste mit Start- und Stop-Dummy-Element
- Position p
  -> Zeiger auf Knoten a_p



IMPLEMENTIERUNG - POINTERLIST3T

- Liste mit Start- und Stopp-Dummy-Element;
- Position p
  -> Zeiger auf Knoten, dessen next-Komponente
  einen Zeiger auf den Knoten mit dem Element a_p enthält



IMPLEMENTIERUNG - POINTERLIST4T

- Liste mit Start- und Stopp-Dummy-Element;
- jedes Element der Zeigerstruktur hat zwei Zeiger
  -> doppelt-verkettet Liste
- Position p
  -> Zeiger auf Knoten a_p

ZUSAMMENFASSUNG/ÜBERBLICK ZEIGER-IMPLEMENTIERUNGSVARIANTEN:

	Start/Stop-Element	Position	Verkettung
Zeiger 1	nein	direkt auf a_p	einfach
Zeiger 2	ja		
Zeiger 3		Vorgänger von a_p	
Zeiger 4		direkt auf a_p	doppelt

VERGLEICH: WELCHE IMPLEMENTIERUNG IST BESSER?

HÄNGT VOM SATZ DER BENÖTIGTEN OPERATIONEN AB!

allg. Kriterien	Array	Zeiger, allg.
Flexibilität	- (max!)	+
Speicherplatz - insgesamt	- (1 .. max) fest belegt	+ (belegt nur, was benötigt)[1)]
Speicherplatz - pro Element	+	- (Zeiger)

1) aber Haldenverwaltung notwendig

ZEITKOMPLEXITÄT DER OPERATIONEN:

- kein Unterschied
  (bei unterschiedlichen Implementierungen)
- clear, isEmpty, length, retrieve, first, next, ... : O (1)
  locate, toString: O (n)

Operation	Array	Zeiger 1	Zeiger 2	Zeiger 3	Zeiger 4
append	O (1)	O (n)[2)]→	O (1)	O (1)	O (1)
insert	O (n)[1)]	O (1)	O (1)	O (1)	O (1)
delete	O (n)[1)]	O (n)	O (n)[3)] →	O (1)	O (1)
last	O (1)	O (n)[2)] →	O (1)	O (1)	O (1)
previous	O (1)	O (n)	O (n)	O (n)[4)] →	O (1)
connect	O (n)[1)]	O (n)[2)] →	O (1)	O (1)	O (1)

1) und viele Adreßänderungen

2) durch Suchen des Listenendes - O (n)
(diese Operationen werden mit “Zeiger 2” verbessert zu O (1))

3) durch Suchen des Vorgängers - O (n)
(diese Operation wird mit “Zeiger 3” verbessert zu O (1))

4) durch doppelte Verkettung nun Vorgänger direkt verfügbar!

GÄNGIGE SPEZIALFÄLLE DER LINEAREN LISTE:

- wenn jeweils nur am Anfang bzw. Ende der Liste
  entfernt / gelesen bzw.
  hinzugefügt / geschrieben
  wird (d. h. allg. Insert bzw. Delete an beliebiger Position nicht notwendig)

ADT	Zugriffs prinzip	Nächstes Element	Entfernen (spez. Delete)	Hinzufügen (spez. Insert)
Stapel S **Iterator itS** nur an einem Ende wird entfernt/ hinzugefügt	LIFO (1) (2)	**Top** itS.First (); S.Retrieve (itS); itS.Last (); S.Retrieve (itS);	**Pop**/Entkellern itS.First (); S.Delete (itS) itS.Last (); S.Delete (itS)	**Push**/Kellern itS.First (); S.Insert (itS, x); S.Append (x);
WS Q **Iterator itQ** an einem Ende wird entfernt, am anderen hinzugefügt	FIFO	**Front** itQ.First (); Q.Retrieve (itQ);	**Dequeue** itQ.First (); Q.Delete (itQ);	**Enqueue** Q.Append (x);

(1) “oberstes” = erstes Element

(2) “oberstes” = letztes Element

∑: Listen mit kontrollierten Zugriffspunkten, so daß alle Operationen leicht mit konstantem Zeitaufwand implementiert werden können.