Implementierung - Array, Operationen für ListT[1]

(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)

1. skizziert, nicht ausprogrammiert

Operationen für IteratorT[2]

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)

2. 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);

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)

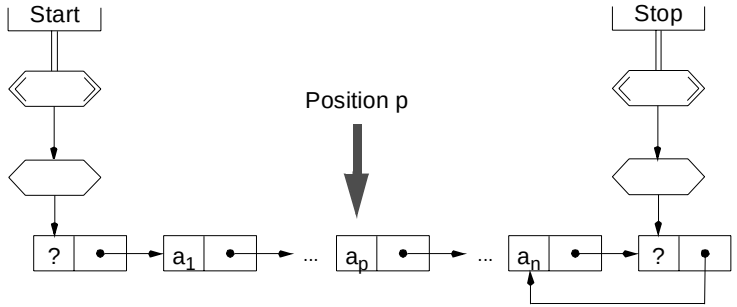
Bei den folgenden Implementierungen entfallen diese Schwierigkeiten teilweise

- 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