Algorithmen und Datenstrukturen

Praktische Einführung und Programmierung

Stefan Bosse

Universität Koblenz - AG Praktische Informatik

Listen

Im Gegensatz zu Tabellenstrukturen werden hier dynamische Strukturen eingeführt:

1. Einfach verkettete Listen
2. Doppelt verkettete Listen
3. Skip Listen

Basisoperationen auf Listen:

1. Einfügen
2. Suchen
3. Entfernen

Dynamische Strukturen

Die Probleme statischer Datenstrukturen lassen sich durch die Verwendung dynamischer Strukturen umgehen.

- Dynamisch bedeutet dabei, dass sie zur Laufzeit »wachsen« oder »schrumpfen« können und sich dadurch an den tatsächlichen Speicherbedarf anpassen lassen, anders als tabellarische Strukturen.

Ein typischer Vertreter einer dynamischen Datenstruktur ist die **verkettete Liste**.

- Eine solche Liste besteht aus einer Menge von Knoten, die untereinander »verzeigert« (referenziert) oder auch »verkettet« sind.
  - Jeder Knoten besteht damit aus einem Verweis (in Java eine Objektreferenz) auf das eigentliche zu speichernde Element sowie einem Verweis auf das nachfolgende Element.

Im Vergleich zu Tabellen sind Listen nicht für echtzeitfähige Systeme geeignet da ihre Laufzeiten nicht im voraus bestimmbar sind.

Entwurf von Datentypen

Etwas systematischer als heuristisch: Wir haben bisher einige einfache Beispiele für abstrakte Datentypen kennen gelernt und deren Spezifikationen mit Gleichungen angegeben.

- Vorgehensweise, mit der man gerade einfache Datentypen gut spezifizieren kann. Hier empfiehlt sich folgende Vorgehensweise:

1. Die Festlegung der **Konstruktorfunktionen** legt den Bereich der Basisterme fest, die den später implementierten Werten entsprechen, und sollte daher mit besonderer Sorgfalt erfolgen.
   - Im Stack-Beispiel benötigt man einen Konstruktor für den leeren Stack und die push-Operation, um alle zu realisierenden Stack-Werte zu definieren.
2. Eventuell ist es nötig, auch Axiome zur Gleichsetzung von Konstruktortermen einzuführen. Der Set-Datentyp ist ein gutes Beispiel hierfür. 3.Die Definition geeigneter Selektorfunktionen erlaubt den lesenden Zugriff auf die konstruierten Werte, ohne einen neuen Wert zu konstruieren. Typischerweise ist das Ergebnis einer Selektion daher ein Wert eines importierten Datentyps.
3. Dann müssen die weiteren Operationen festgelegt werden, sozusagen die Manipulatoren der Datenwerte. Manipulatorfunktionen haben als Ergebnistyp den neu spezifizierten Datentyp.
4. Der letzte Schritt besteht darin, Fehlersituationen abzufangen.

Listen

Ein typischer Vertreter einer dynamischen Datenstruktur ist die verkettete Liste.

- Eine solche Liste besteht aus einer Menge von Knoten, die
  - untereinander »verzeigert« oder auch »verkettet« sind.
  - Jeder Knoten besteht damit aus einem Verweis (in Java eine Objektreferenz) auf das eigentliche zu speichernde Element sowie einem Verweis auf das nachfolgende Element.

$$K =< \text{data}|\text{next} \rightarrow K >$$

$$L = \{k_i \in K, \forall i = 1..n \wedge k_i|\text{next} = k_{i+1} \forall i = 1..n-1\}$$

Listen sind keine Mengen! Listen sind geordnet bzw. haben eine feste Reihenfolge der Knoten, wonhingegen Mengen ungeordnet sind und jede Permutation der Reihenfolge von Knotenelementen möglich und zuässig ist.

Das letzte Element verweist demzufolge auf null. In der Liste selbst muss nur noch der erste Knoten direkt verankert sein, alle anderen Knoten sind durch das Navigieren über die Zeiger erreichbar. Der »Anker« für den Beginn der Liste wird mit head bezeichnet.

Einfach verkettete Listen

[addp]

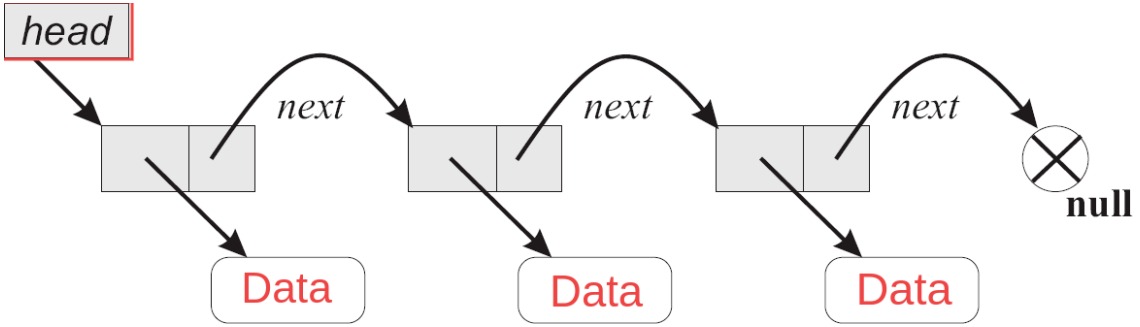

Abb. 1. Prinzip der einfach verketteten Liste. Dier Daten werden i.A. separat gespeichert und in der Elementstruktur gibt es nur einen Zeiger auf die Daten. Eine Integatrion der Daten (z.B. ganze Zahlen) ist aber auch möglich.

ZeigerArrays

- Ein Spezialfall von Listen sind Zeigerarrays, wie sie typischerweise in allen dynamisch typisierten Programmiersprachen wie javaScript, Python oder Lua vorkommen.

Arrays sind in dynamische typisierten Programmiersprachen polymorph und mehrsortig, d.h. jedes Element kann einen Wert mit anderen Datentyp besitzen. Es bedarf daher einer getrennten Speicherung von Organisationsstruktur (Tabelle) und Daten.

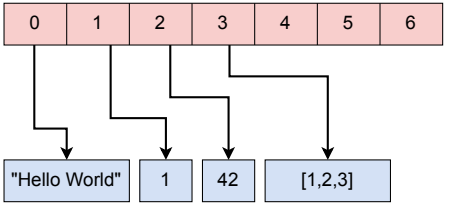

Abb. 2. Einfache (einelementige) Listen in mehrsortigen Arrays - oder Urform der Bereichslisten

Einfach verkettete Listen

Bei Stack und Queue gab es ein Paar aus (verschiedenen) Operation um Elemente einzufügen und zu entfernen. Diese können auch bei verketteten Listen eingesetzt werden:

1. `addHead(L,x)` fügt das Objekt x als erstes Element in die Liste ein.
2. `getFirst(L)` liefert das erste Element der Liste.
3. `removeFirst(L)` entfernt das erste Element der Liste und gibt es gleichzeitig als Ergebnis zurück.
4. `addLast(L,x)` hängt das Objekt x als letztes Element an die Liste an.
5. `getLast(L)` liefert das letzte Element der Liste.
6. `removeLast(L)` entfernt das letzte Element der Liste und gibt es gleichzeitig als Ergebnis zurück.

- Wir haben i.A. Kopf- oder Endlisten oder beides. Das wird durch die Anker bestimmt. Eine Kopfliste hat den *head* Zeiger, die Endliste den *tail* Zeiger.

Einfach verkettete Listen

Einfügen und Löschen am Anfange

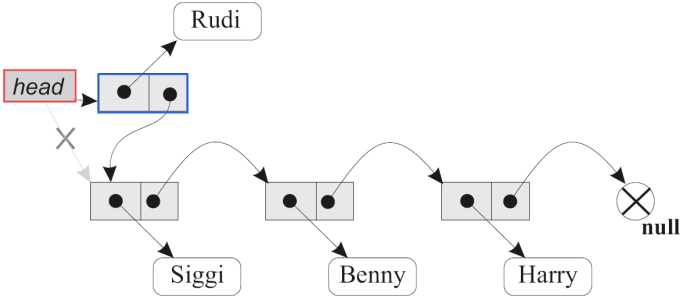

Abb. 3. Einfügen eines Elements *x* am Anfang ist durch den Anker *head* einfach in O(1) durch Änderung des *head* Zeigers auf *x* möglich. Das gleiche gilt für das Löschen des Kopfelements, ebenfalls nur eine Änderung von *head*.

Einfach verkettete Listen

Einfügen und Löschen am Ende

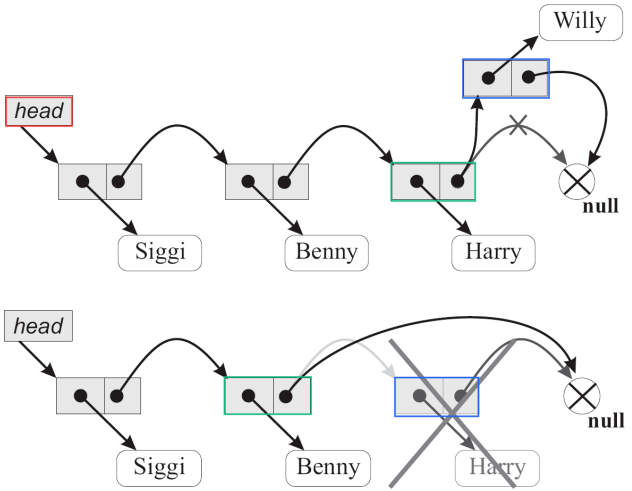

Abb. 4. Einfügen und Löschen eines Elements *x* am Ende ist durch den Anker *head* nicht möglich. Stattdessen benötigt man das letzte und vorletzte Element beim Löschen, nur durch Suche (Iteration) in O(N) möglich! Auch ein *tail* Anker hilft beim Löschen nicht, beim Einfügen schon!

Einfach verkettete Listen

- Die Operationen zur Manipulation des Listenendes sind dagegen etwas komplexer.
  - Da nur der Listenanfang bekannt ist, muss für das Anhängen eines Elementes zunächst der letzte Knoten (tail) durch Verfolgen der next-Zeiger ermittelt werden.
  - Dieser ist dadurch gekennzeichnet, dass sein next-Zeiger auf null verweist.
  - Ist der letzte Knoten gefunden, wird dessen next-Zeiger auf den neu angelegten Knoten mit dem anzuhängenden Element gesetzt
- Zum Löschen des letzten Elementes wird dagegen auch der vorletzte Knoten benötigt, da dessen Verweis auf null gesetzt werden muss.
  - Entsprechend muss der Knoten gesucht werden, dessen Nachfolgerknoten auf null verweist.
  - Da in jedem Fall die gesamte Liste durchlaufen werden muss, ist der Aufwand abhängig von der Anzahl N der Elemente in der Liste und beträgt für diese Operationen entsprechend O(N).

Einfach verkettete Listen

Komplexität

Operation	Komplexität (head)	Komplexität (head+tail)
addFirst	O(1)	O(1)
getFirst	O(1)	O(1)
removeFirst	O(1)	O(1)
addLast	O(N)	O(1)
getlast	O(N)	O(1)
removeLast	O(N)	O(N)

Tab. 1. Der Aufwand aller Listenoperationen für einfach verkettete Listen

Einfach verkettete Listen

Algorithmus und Datenstruktur

```
L={head:null}

function Node(data,next) {
  return { data:data, next:next }
}

function addFirst(L,x)  {
  node=Node(x,L.head)
  L.head=node
}
function getFirst(L)  {
  return L.head.data
}
function removeFirst(L,x)  {
  if (!L.head) return;
  first=L.head
  L.head=L.head.next

  return first
}
```

```
function addLast(L,x)  {
  node=L.head
  while (node.next)
    node=node.next
  node.next=Node(x,null)
}
function getLast(L,x)  {
  node=L.head
  while (node.next)
    node=node.next
  return node.data
}
function removeLast(L,x)  {
  node=L.head
  while (node.next && node.next.next)
    node=node.next
  last=node.next
  node.next=null
  return last
}
```

Alg. 1. Einfach verkettete Liste

Einfach verkettete Listen

Doppelt verkettete Listen

Ein Problem der Implementierung der verketteten Liste ist der mit dem Zugriff auf das letzte Element verbundene Aufwand.

- Da jeder Knoten nur mit seinem Nachfolger verkettet ist, muss das letzte Element immer erst durch Navigieren vom Listenkopf ausgehend über alle Knoten ermittelt werden.
  - Zwar könnte man die Liste um einen tail-Zeiger auf das letzte Element ergänzen und so das Einfügen von Elementen am Ende der Liste vereinfachen.
  - Dies würde beispielsweise die Implementierung der Klasse Queue vereinfachen.
  - Trotzdem ist beim Löschen des letzten Elementes immer noch das Durchlaufen aller Knoten zum Auffinden des vorletzten Knotens notwendig.
- Uns es gibt noch zwei weitere Basisoperationen:

1. `insert(L,x,pos)` fügt das Element x an oder hinter *pos* in der Liste ein.
2. `remove(L,x)` entfernt ein beliebiges Element x aus der Liste.

Doppelt verkettete Listen

Eine Verbesserung bietet daher die doppelt verkettete Liste, bei der jeder Knoten nicht nur seinen Nachfolger, sondern auch seinen Vorgänger kennt

- Es gilt formal:

$$K =< \text{data}|\text{prev} \rightarrow K|\text{next} \rightarrow K >$$

$$L = \{k_i \in K, \forall i = 1..n \wedge k_i|\text{next} = k_{i+1} \forall i = 1..n-1 \wedge k_i|\text{prev} = k_{i-1} \forall i = 2..n\}$$

Doppelt verkettete Listen

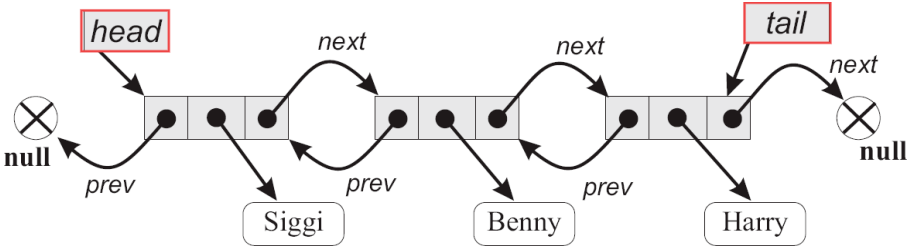

Abb. 5. Doppelt verkettete Liste

Für eine doppelt verkettete Liste werden zwei spezielle Knoten, *head* für den Listenanfang und *tail* für das Listenende, verwaltet. Durch die Verfügbarkeit des Listenendes und der rückwärts gerichteten Verweise vereinfachen sich die Operationen am Listenende. So kann das letzte Element für die Operation getLast direkt über *tail* mit konstantem Aufwand O(1) ermittelt werden. Auch beim Anhängen eines Elementes mit *addLast* kann über tail sofort der bisher letzte Knoten bestimmt werden, dessen *next*-Zeiger nun auf den neuen Knoten zeigt. Zusätzlich muss noch der *prev*-Zeiger des neuen Knotens auf den bisherigen letzten Knoten verweisen

Einfach verkettete Listen

Einfügen und Löschen am Anfange und Ende

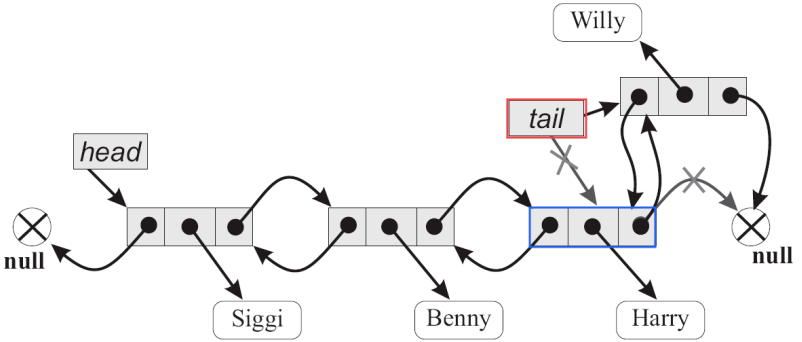

Abb. 6. Anhängen am Ende einer doppelt verketteten Liste (analog Löschen, am Anfang wie 1 VKL)

Doppelt verkettete Listen

- Beim Löschen (removeLast) wird zunächst über den prev-Zeiger des letzten Knotens der vorletzte Knoten bestimmt, dessen next-Zeiger dann auf null gesetzt wird.
- Auch für die Operationen addLast und removeLast beträgt der Aufwand daher O(1).
- Ebenso einfach ist das Einfügen und Löschen von Elementen in der Mitte der Liste.
    - Allerdings muss hierzu zunächst die entsprechende Position in der Liste gefunden werden, indem beispielsweise vom Anfang der Liste ausgehend dasjenige Element gesucht wird, vor oder nach dem das neue Element eingefügt werden soll.

Doppelt verkettete Listen

Komplexität

Operation	Komplexität (head+tail)
addFirst	O(1)
getFirst	O(1)
removeFirst	O(1)
addLast	O(1)
getlast	O(1)
removeLast	O(1)
insert	O(N)
remove	O(N)

Tab. 2. Der Aufwand aller Listenoperationen für deppelt verkettete Listen

Einfach verkettete Listen

Algorithmus und Datenstruktur

```
L={head:null,tail:null}
function Node(x,prev,next) { return { data:x, prev:prev, next:next }

function addLast(L,x) {
  node=Node(x,L.tail,null)
  if (L.tail) L.tail.next=node
  L.tail=node
  if (!L.head) L.head=L.tail

}
function getLast(L,x) {
  return L.tail.data
}
function removeLast(L,x) {
  if (!L.tail) return;
  last=L.tail
  L.tail.prev.next=null
  L.tail=L.tail.prev
  return last
}
```

```
function insert(L,x,pos) {
  node=L.head
  while(pos && node) {
    node=node.next
    pos--
  }
  if (!node) return;
  next=node.next;
  node.next=Node(x,node,next)
  if (next) next.prev=node.next
}
function remove(L,x) {
  node=L.head
  while (node.data!==x) node=node.next;
  if (!node) return;
  removed=node;
  node.prev.next=node.next
  if (node.next)
    node.next.prev=node.prev
  return removed
}
```

Alg. 2. Doppelt verkettete Liste (addHead, getHead, removeHead wie 1 VKL)

Bereichslisten (Skiplisten)

Ein Nachteil von verketteten Listen gegenüber Feldern ist, dass die binäre Suche nicht einsetzbar ist, da man nicht beliebig »springen« kann. Eine Alternative ist es, mehrere Listen übereinanderzulegen, die dieses Springen ermöglichen. Eine derartige Datenstruktur wird als Skip-Liste bezeichnet.

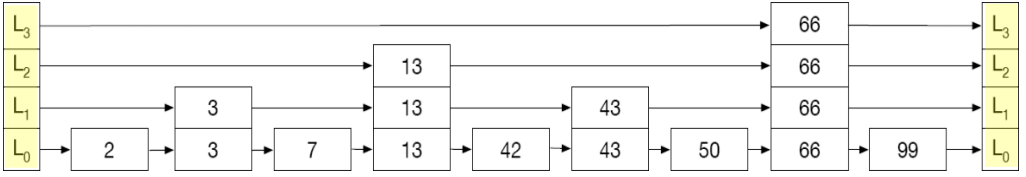

Abb. 7. Aufbau einer Skip-Liste

- Es wird nun aber ein Schlüssel $k(x)$ zu jedem Listenelement (bzw. zu den Daten) benötigt!

Bereichslisten (Skiplisten)

Es gibt jeweils einen initialen und einen terminalen Listenknoten, die die Werte $-\infty$ und $+\infty$ tragen. Grundsätzlich gibt es zwei Möglichkeiten zur Speicherung der Listenelemente auf mehreren Ebenen:

- Der Wert wird in separaten Listenelementen auf allen betroffenen Ebenen abgespeichert.
- Die Listenelemente werden verschmolzen, so dass Listenelemente mehrere Nachfolger haben können (ja nach abgedeckten Ebenen). Auf diese Weise haben wir in der Abbildung diese Elemente schon zusammengeschoben.
  - Einfach verkettete listen haben einen, doppelt verk. zwei, und diese Skiplisten bis zu p Nachfolger.

Bereichslisten (Skiplisten)

Suche

Man startet in der höchsten Ebene mit dem initialen Knoten. Wenn man einen Knoten ansieht, gibt es drei Möglichkeiten:

- Der aktuelle Wert ist der Suchwert. Dann ist man fertig.
- Der aktuelle Wert ist kleiner als der Suchwert. Dann wechselt man zum nächsten Knoten derselben Ebene.
- Der aktuelle Wert ist größer als der Suchwert. Befinden wir uns auf der Ebene L0, ist der Suchwert nicht gespeichert. Befinden wir uns auf einer höheren Ebene, gehen wir zurück zum letzten Knoten und gehen dann eine Ebene tiefer.

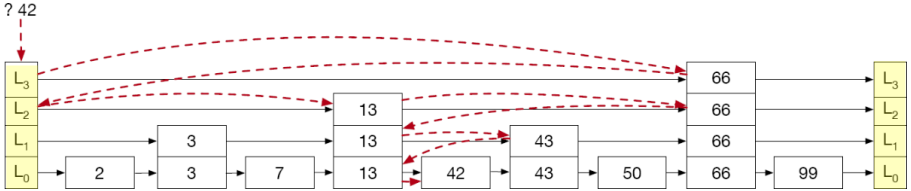

Abb. 8. Suche in einer Skip-Liste

Bereichslisten (Skiplisten)

- Die bisher gezeigte Liste hat eine statische Struktur mit festen Schrittlängen, die durch das Einfügen und Löschen von Werten zerstört werden würde (etwa wenn die Werte 8 und 9 eingefügt werden).
- Allerdings ist der Algorithmus der Suche davon nicht betroffen: Selbst wenn 8 und 9 auf Ebene L0 eingefügt würden, würde die Suche weiterhin funktionieren – jedoch bei der Suche nach der 9 mehrere Schritte auf der untersten Ebenen durchlaufen.
- Daher reicht es aus, wenn wir nur eine gute Näherung an feste Schrittlängen haben.
  - Eine Möglichkeit ist es dabei, neu eingefügte Werte zufällig einer Ebene zuzuweisen, wobei im Mittel die Hälfte der Werte der Ebene L0 zugeordnet wird, ein weiteres Viertel der Ebene L1, ein Achtel dann L2 etc.
  - Derartige Skip-Listen nennt man randomisiert.

Bereichslisten (Skiplisten)

Die zuerst eingeführten Skip-Listen mit Zweierpotenzen als festen Schrittlängen werden zur Abgrenzung von **randomisierten Skip-Listen** als **perfekte Skip-Listen** bezeichnet. Sie können insbesondere genutzt werden, wenn die Werte der Skip-Listen nicht mehr geändert oder ergänzt werden.

Bereichslisten (Skiplisten)

Algrotihmik

Das Iterator-Konzept

Die Implementierungen der Listen aus den vorigen Abschnitten weist noch ein Manko auf, welches die praktische Verwendbarkeit einschränkt. So ist es oft notwendig, eine Kollektion zu »durchwandern«, d.h. über alle Elemente zu navigieren.

- Dieses Navigieren ist zunächst abhängig von der Implementierung: Während beispielsweise ein Feld mittels einer Indexvariablen durchlaufen wird, ist für verkettete Listen das Verfolgen der next-Zeiger der einzelnen Knoten notwendig.
- Auch im Hinblick auf die Erhaltung des Prinzips der **Kapselung** ist daher ein Konzept wünschenswert, das eine einheitliche Behandlung des Navigierens unabhängig von der internen Realisierung unterstützt.
- Das ist das Konzept der Iteratoren. Hierbei handelt es sich um Objekte zum Iterieren über Kollektionen.

Das Iterator-Konzept

- Ein Iterator verwaltet einen internen Zeiger auf die aktuelle Position in der zugrunde liegenden Datenstruktur.
- Auf diese Weise ist es möglich, dass mehrere Iteratoren gleichzeitig auf einer Kollektion arbeiten.

Vier Funktionen von Iteratoren:

- `hasNext(I)` prüft, ob noch weitere Elemente in der Kollektion verfügbar sind. In diesem Fall wird true geliefert. Ist dagegen das Ende erreicht, wird false zurückgegeben.
- `next(I)` liefert das aktuelle Element zurück und setzt den internen Zeiger des Iterators auf das nächste Element.
- `remove(I)` erlaubt es, das zuletzt von `next()` gelieferte Element zu löschen, wobei pro next-Aufruf nur einmal remove aufgerufen werden darf.
- `reset(I)` setzt den internen Zustand zurück (auf Anfang).

Text Lexer und Parser arbeiten nach dem Iteratorprinzip (hier über Textzeichen oder Texttokens).

Das Iterator-Konzept

[addp]

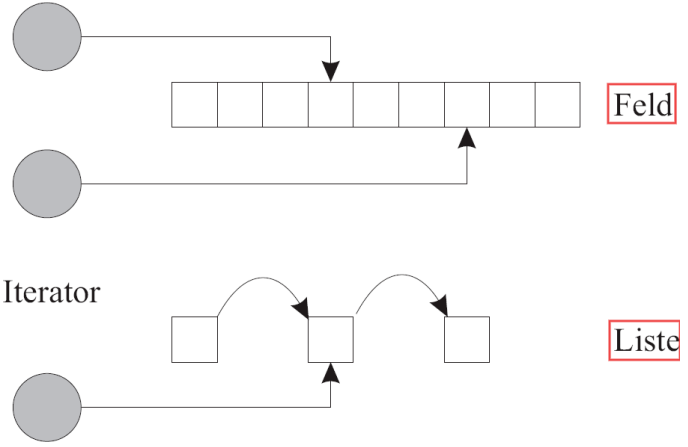

Abb. 9. Iterator-Konzept

Das Iterator-Konzept

Algorithmik

```
I={iterable:List,next:head(List)}
function next(I) {
  node=I.next
  I.next=node.next
  return node
}

I={iterable:Array,next:0}
function next(I) {
  node=I.iterable[I.next]
  I.next++
  return node
}
```

Alg. 3. Iteratoren