Algorithmen und Datenstrukturen

Listen-Abstraktionen

Prof. Justus Piater, Ph.D.

23. März 2025

Dieser Kurs folgt in weiten Teilen dem sehr empfehlenswerten Lehrbuch *Data Structures and Algorithms in Java* [Goodrich u. a. 2014].

Inhaltsverzeichnis

Einführung

In diesem Kapitel beschäftigen wir uns mit zwei verschiedenen abstrakten Datentypen für Listen, nämlich der *Liste* und der *positionsbasierten Liste*. Es handelt sich wohlgemerkt um abstrakte Datentypen und nicht um Datenstrukturen wie verkettete Listen. Halten Sie diese beiden Konzepte sauber auseinander!

Der abstrakte Datentyp ***Liste***, englisch *list*, dient der Repräsentation einer Rangliste, wie z.B. im Sport. Beim Kletter-Worldcup gibt einen Erstplatzierten, einen Zweitplatzierten, usw.

Das macht eine *Liste* als abstrakten Datentyp aus: Wir reden vom 1. Element, vom 2. Element, oder vom 7. Element. Es kann höchstens ein k-tes Element geben, und falls es existiert und $k > 1$ ist, dann muss es auch das $k - 1$-te Element geben. Eine Liste darf also keine Lücken enthalten. Jeder Rang zwischen dem 1. und dem letzten Element muss belegt sein. Und wenn wir ein neues Element irgendwo in der Mitte einfügen, dann erhöht sich ab hier für alle Elemente der Rang um eins.

Eine ***positionsbasierte Liste***, auf englisch *positional list*, ist dagegen wie eine Seilschaft. Jede Schlaufe im Seil ist eine *Position*, und jede ist mit seinem *Vorgänger* und seinem *Nachfolger* verbunden.

In einer positionsbasierten Liste haben die Elemente keine Ränge, sondern nur Vorgänger und Nachfolger. Wir können also vom 1. und vom letzten Element reden, aber nicht vom 2. oder 7.

Knüpfen wir hier hinter mir eine neue Schlaufe ein, wird diese zu meiner neuen Nachfolger-Position, aber meine eigene Position ändert sich nicht. Ich hänge ja nach wie vor an derselben Schlaufe!

Für beide Listen-Abstraktionen werden wir wieder verschiedene Datenstrukturen und Algorithmen betrachten. Es wird sich herausstellen, dass man den abstrakten Datentyp

Liste im Allgemeinen am effizientesten auf Basis eines sogenannten dynamischen Arrays implementiert, und nicht etwa auf Basis einer verketteten Liste.

Abschließend werden wir den *Iterator* einführen, ein wichtiges Entwurfsmuster zum Iterieren über die Elemente einer Datenstruktur.

1 ADT Liste und DS ArrayList

Video 1 beginnt hier.

ADT: Liste [Slide 1]

```
get(i)     // Returns the element of the list at rank i;
           // error if i is not in range [0, size() −1].

set(i, e)  // Replaces the element at rank i with e,
           // and returns the old element; error if i is not in range.

add(i, e)  // Inserts a new element e into the list at rank i,
           // increasing the rank of all subsequent elements by 1;
           // error if i is not in range.

remove(i)  // Removes and returns the element at rank i,
           // decreasing the rank of all subsequent elements by 1;
           // error if i is not in range.

size()     // Returns the number of elements in the list .
isEmpty()  // Returns a boolean indicating whether the list is empty.
```

Indizierte, ***lückenlose*** Sequenz: Index = ***Rang***

Den ADT Liste nicht mit der Datenstruktur verkettete Liste verwechseln! Dies sind zwei völlig verschiedene Konzepte.

Dieser ADT entspricht dem Interface `java.util.List`.

Definition

Der abstrakte Datentyp ***Liste*** definiert eine Sequenz, deren Elemente über ihren ***Rang*** angesprochen werden. Mit anderen Worten, wir können z.B. das dritte Element abfragen. Oder, wenn wir das 4. Element entfernen, dann rücken alle nachfolgenden Elemente auf: das bisher 5. Element ist nun das 4., das bisher 6. ist nun das 5., und so weiter. Umgekehrt können wir ein neues Element z.B. an 2. Stelle einfügen. Dann ist das bisher 2. Element nun das 3., und so weiter.

Es ergibt jedoch keinen Sinn, ein 7. Element einzufügen, wenn die Liste nur 5 oder weniger Elemente enthält. Ein solcher Versuch löst einen Fehler aus.

Eine Liste ist also eine indizierte, lückenlose Sequenz, wobei der Index als Rang verstanden wird.

Sie stellt zwei Zugriffsmethoden zur Verfügung, die die Länge der Liste nicht verändern. Die Methode `get(i)` liefert das i-te Element zurück (und belässt es in der Liste). Die Methode `set(i, e)` liefert das i-te Element zurück und ersetzt es in der Liste durch das Element e, das von nun an den i-ten Rang einnimmt.

Die beiden anderen charakteristischen Methoden der Liste verändern ihre Länge. Die Methode `add(i, e)` fügt das Element e am Rang i in die Liste ein, wodurch sich die Ränge aller nachfolenden Elemente um 1 erhöhen. Umgekehrt entfernt die Methode `remove(i)`

das i-te Element aus der Liste, wodurch sich die Ränge aller nachfolenden Elemente um 1 erniedrigen, und liefert das entfernte Element zurück.

Wie immer haben wir natürlich auch die beiden Methoden `size()` und `isEmpty()`.

ADT Liste: `add(`i`, `e`)` und `remove(`i`)` [Slide 2]

Animation

Veranschaulichen wir uns, wie eine Liste funktioniert. In unsere zunächst leere Liste fügen wir hier als Erstes das Element A ein. Dies muss zwangsläufig am Rang 0 geschehen. Ein zweites Element können wir entweder am Rang 0 oder am Rang 1 einfügen. Hier fügen wir das Element B am Rang 0 ein, und anschließend noch das Element C, ebenfalls am Rang 0. A hat nun den Rang 2. Nun fügen wir das Element D am Rang 3 ein, und anschließend das Element E am Rang 1, wofür eine Lücke geschaffen werden muss.

Nun entfernen wir das Element am Rang 2. Die hinterlassene Lücke wird geschlossen, indem die nachfolgenden Elemente aufrücken. Nun entfernen wir noch das Element am Rang 3, welches nun das letzte ist, und das Element am Rang 0.

Implementierung: Array (ArrayList) [Slide 3]

C	B	A	D						
0	1	2	3	4	5	6	7	8	9

`add(1, E)`:

C	E	B	A	D					
0	1	2	3	4	5	6	7	8	9

`remove(2)`:

C	E	A	D						
0	1	2	3	4	5	6	7	8	9

Komplexität der Methoden `add(`i`, `e`)` und `remove(`i`)`?

Erklärung

Wegen des Zugriffs per Rang oder Index bietet sich ein Array als Datenstruktur für eine Liste an. Eine Array-basierte Implementierung einer Liste wird oft als ArrayList bezeichnet.

Dieselbe Illustration, die wir in der Animation des abstrakten Datentyps Liste verwendet haben, dient uns hier zur Veranschaulichung der Datenstruktur Array.

Die Algorithmen für eine Array-basierte Implementierung der Methoden der Liste müssen wir hier wohl nicht im Detail ausformulieren; dies ist eine einfache Übung für Sie.

Welche Laufzeiten unserer Methoden können wir erzielen?

`get()` und `set()` können in konstanter Zeit über den Index auf das gewünschte Element zugreifen und lassen sich somit in konstanter Zeit implementieren.

`add()` und `remove()` hingegen müssen in der Regel vorher Platz schaffen bzw. hinterher die Lücke schließen. Wird das Element am Rang 0 entfernt oder eingefügt, müssen alle n bzw. die verbleibenden $n-1$ Elemente bewegt werden. Wird das Element am Rang $n-1$ entfernt oder am Rang n eingefügt, muss überhaupt kein Element bewegt werden. Im Erwartungsfall muss also die Hälfte der Elemente bewegt werden. Daher ist die erwartete Laufzeit von `add()` und `remove()` jeweils $\Theta(n)$.

Quiz [Slide 4]

Den ADT Liste kann man auch als verkettete Liste implementieren.

A: Ja, und zwar genauso gut oder besser (Einfügen in $O(1)$, etc.).

B: Ja, aber eine ArrayList eignet sich besser.

C: Nein.

D: weiß nicht

Noch ein Quiz [Slide 5]

Mit einer ArrayList haben `add(0, e)` und `remove(0)` (oder `add(N-1, e)` und `remove(N-1)`) zwangsläufig eine Laufzeit von $\Theta(n)$.

A: Tja, so ist es. Konstant geht's nur an einem Ende eines Arrays, nicht an beiden.

B: Falsch; man kann auch an beiden Enden der ArrayList konstante Laufzeit dieser beiden Methoden erzielen.

C: Konstant geht nicht an beiden Enden, aber logarithmische Laufzeit ist möglich, dank Binärsuche.

D: weiß nicht

2 Dynamische Arrays

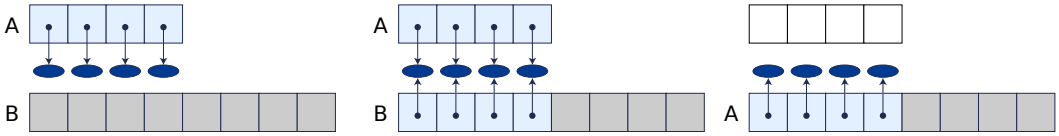
Video 2 beginnt hier.

Dynamische Arrays [Slide 6]

Ist das Array voll, wird es automatisch durch ein größeres ersetzt.

Warum kann man nicht einfach das bestehende Array vergrößern?

```
Algorithm grow(A, c)
  Allocate a new array B of capacity c
  B[k] ← A[k] for k = 0, ..., n − 1
  A ← B
```



Erklärung

Wir haben nun bereits mehrere arraybasierte Implementationen verschiedener abstrakter Datentypen kennengelernt. Bei allen stellt sich das Problem, was passiert, wenn das Array voll ist. Die einfache Antwort lautet: Vergrößern wir das Array!

In der Regel können wir ein bestehendes Array nicht einfach vergrößern, sondern wir müssen es durch ein neues, größeres ersetzen. Dies impliziert, dass wir die Elemente aus dem alten in das neue Array hinüberkopieren müssen. Anschließend können wir das alte

Array freigeben und das neue, hier B genannt, an den Bezeichner des alten zuweisen, hier A, so dass A nun das neue Array bezeichnet.

Wegen der Notwendigkeit, die n Elemente des alten Arrays zu kopieren, hat unsere Vergrößerungsaktion also eine asymptotische Laufzeit von $\Theta(n)$. Dies erscheint ziemlich teuer, insbesondere wenn unser neues Array auch wieder voll wird und wir es deshalb wieder vergrößern. Dann müssen wir die nun noch größere Anzahl Elemente kopieren.

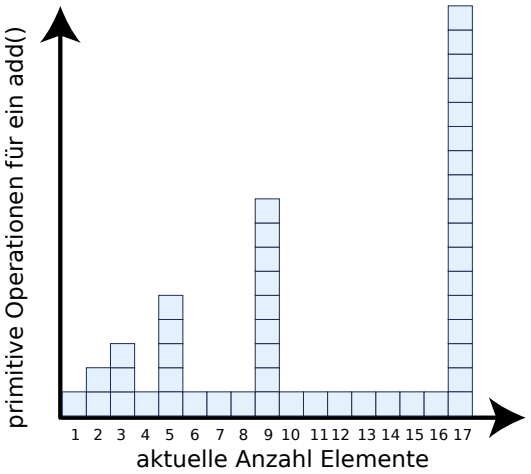
Daher ist es vielleicht etwas überraschend, dass wir die Vergrößerungen so gestalten können, das sie *überhaupt keinen Einfluss* auf die asymptotische Laufzeit haben. Eine gegebene Vergrößerung kostet natürlich $\Theta(n)$ Zeit. Das ist unvermeidbar. Aber über viele Einfügeoperationen hinweg können wir die Kosten sämtlicher Array-Vergrößerungen amortisieren!

Das besagt Folgendes: Nehmen wir an, wir erstellen ein Array der Kapazität N, wir füllen es mit N Aufrufen von `add()` nach und nach auf, und dies beansprucht insgesamt eine Zeit von $O(k)$. Dann ist diese Gesamtzeit auch $O(k)$, wenn wir mit einem beliebig kleineren Array beginnen und es bei Bedarf mehrfach vergrößern!

Um dies zu verstehen, müssen wir uns genauer anschauen, wie viel Zeit uns jede Einfügeoperation `add()` in einer Sequenz von `add()`s kostet.

Wie teuer ist ein `add()`? [Slide 7]

Bei `grow(A, 2 * A.size())`:



Beispiel

Hier beginnen wir mit einem Array der Größe 1, und immer, wenn seine Kapazität nicht mehr ausreicht, verdoppeln wir seine Größe.

Das erste `add()` dauert eine Zeiteinheit. In der Grafik wird diese Zeiteinheit durch das blaue Kästchen unten links dargestellt.

Dann ist das Array voll.

Beim 2. `add()` wird daher das Array auf Länge 2 verdoppelt. Das 2. `add()` dauert also zwei Zeiteinheiten, eine für das Kopieren des ersten Elements, und eine für das Einfügen des zweiten Elements.

Beim 3. `add()` wird das Array wiederum verdoppelt. Das 3. `add()` dauert nun drei Zeiteinheiten, zwei für das Kopieren der beiden bestehenden Elemente (der Turm), und eine für das Einfügen des neuen Elements (die Basis).

Nun hat das Array eine Kapazität von 4 Elementen. Das 4. `add()` erfolgt also wieder in einer einzigen Zeiteinheit.

Beim 5. `add()` müssen wir allerdings wieder das Array verdoppeln. Damit dauert es insgesamt 5 Zeiteinheiten.

Das Array hat nun eine Kapazität von 8 Elementen. Die nächste Verdoppelung erfolgt also beim 9. `add()`.

Eine Verdoppelung erfolgt also immer beim 2^i+1. `add()`. Dieses `add()` dauert dann 2^i+1 Zeiteinheiten. Danach ist dann jeweils für die nächsten 2^i-1 Zeiteinheiten Ruhe.

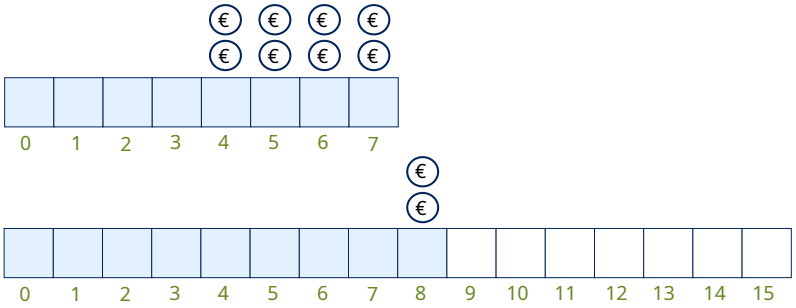
Nehmen wir nun an, jede Zeiteinheit koste 1 €. Stellen Sie sich vor, Sie sägen jeden dieser Kostentürme oberhalb von 2 € ab und lassen ihn nach rechts kippen. Jeder dieser Türme passt dann genau in die Lücke bis zum jeweils nächsten Turm. Damit bilden die gefällten Kostentürme eine zweite 1 €-Schicht oberhalb der ersten, und die Kosten für jedes `add()` bleiben für immer konstant, nämlich – amortisiert – bei 2 € pro `add()`!

Amortisierte Laufzeit von `add()` [Slide 8]

Proposition: Die Gesamtzeit, n Elemente am Ende des Arrays einzufügen, ist $O(n)$.

Beweisskizze (durch Amortisation):

- Angenommen, die $i$te Erweiterung des Arrays koste $k = 2^i$ €.
- Veranschlagen wir für jedes `add()` 2 €.
- Das angesparte Geld reicht für das nächste `grow()`.



Erklärung

Formulieren wir diese Erkenntnis als formale Proposition: Die Gesamtzeit, n Elemente am Ende eines zunächst beliebig kleinen Arrays einzufügen, ist $O(n)$.

Um dies zu beweisen, nehmen wir an, das Kopieren eines Elements koste 1 €. Wir veranschlagen nun für jedes `add()` 2 €. Mit dem 1. `add()` haben wir damit 2 € eingenommen; davon können wir das Kopieren des ersten Elements bei der 1. Verdoppelung locker bezahlen. Vom übrigen 1 € können wir uns eine Kugel Eis kaufen.

Für das 2. `add()` veranschlagen wir wieder 2 €. Mit diesen 2 € können wir bei der nächsten Verdoppelung die Kopie der ersten beiden Elemente exakt bezahlen.

Für das 3. und 4. `add()` legen wir wieder jeweils 2 € zurück. Nun haben wir 4 € angespart, mit denen wir bei der 3. Verdoppelung die Kopie der 4 Elemente bezahlen können.

Nun haben wir nichts mehr auf der hohen Kante, aber bis zur 4. Verdoppelung legen wir wieder für jedes der 4 `add()`s jeweils 2 € zurück. Von diesen angesparten 8 € können wir dann die Kopie der 8 Elemente bezahlen, und so weiter.

Auf diese Weise nehmen wir bei *konstanten* Kosten für jedes `add()` bis zur nächsten Verdoppelung jeweils ausreichend ein, um die Kopie der bestehenden Elemente zu bezahlen.

Für ein weiteres Eis bleibt allerdings nichts mehr übrig.

Diese Kostenrechnung funktioniert, weil das Array bei jeder Vergrößerung *verdoppelt* wird. Der Schlüssel ist, dass das Wachstum des Arrays ***proportional*** zur aktuellen Arraygröße ist. Damit ist auch die Anzahl der vor jeder Vergrößerung neu hinzugekommenen Elemente proportional zur Anzahl der zu kopierenden Elemente, und konstante Kosten für jedes `add()` können diese Kopierkosten decken.

Folglich haben wir gezeigt, dass bei *multiplikativer* Vergrößerung die Gesamtlaufzeit von n `add()`s linear in n ist.

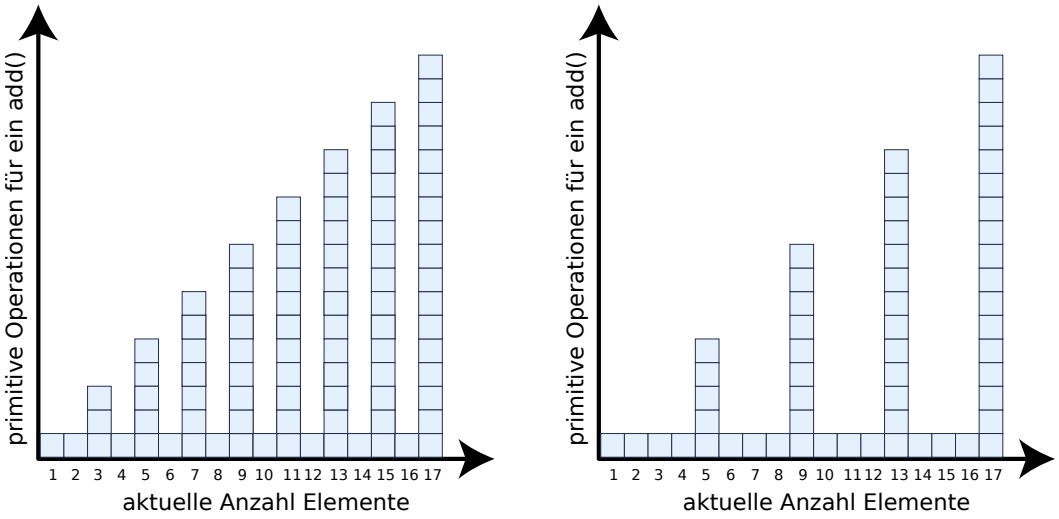
Die *amortisierte* Laufzeit eines `add()`, also die Summe der Laufzeiten aller `add()`s dividiert durch die Anzahl dieser `add()`s, ist also konstant.

Konstante amortisierte Laufzeit von `add()` [Slide 9]

Dies funktioniert für multiplikatives `grow()` um einen beliebigen, konstanten Faktor größer 1, also bei *geometrischen* Folgen von Arraygrößen.

Schlüssel: Das Wachstum muss ***proportional*** zur aktuellen Arraygröße sein. Damit ist die Anzahl der zu kopierenden Elemente proportional zur Anzahl der seit dem letzten `grow()` hinzugekommenen Elemente.

Arithmetische (additive) Folgen von Arraygrößen [Slide 10]



Erklärung

Bei *additiver* Vergrößerung ist die Anzahl der `add()`-Aufrufe bis zur nächsten Vergrößerung allerdings *konstant*. Da die Kopierkosten mit jeder Vergrößerung wachsen, können wir diese nicht mehr mit konstanten Einnahmen für jedes `add()` decken.

Links sehen wir die wachsenden Kopierkosten bei einer additiven Vergößerung um 2 Elemente, und rechts um 4 Elemente. Wie wir sehen, passt jeweils nur der erste Kopierkostenturm in die Lücke bis zur nächsten Vergrößerung. Danach übersteigen die Kopierkosten die zwischenzeitlichen Einnahmen.

Laufzeit bei additivem `grow()` [Slide 11]

Proposition: Die Gesamtzeit, n Elemente am Ende des Arrays einzufügen, ist $\Omega(n^2)$.

Beweis: Sei $c > 0$ der konstante Term, um den `grow()` das Array additiv vergrößert. Dann haben nach n `add()` $m = \lceil \frac{n}{c} \rceil$ `grow()` stattgefunden, die insgesamt eine Zeit linear in $c + 2c + 3c + \cdots + mc$ beansprucht haben. Nun,

$$\sum_{i=1}^{m} ci = c \sum_{i=1}^{m} i = c\frac{m(m+1)}{2} \geq c\frac{\frac{n}{c}(\frac{n}{c}+1)}{2} \geq \frac{1}{2c}n^2$$

Also benötigen n `add()` eine Gesamtzeit von $\Omega(n^2)$.

Speicherbedarf dynamischer Arrays [Slide 12]

- Ein (multiplikatives) dynamisches Array benötigt Platz proportional zur Anzahl seiner Elemente.
- Es sei denn, diese sinkt und bleibt klein.
- Man kann ein dynamisches Array schrumpfen, bei konstanten amortisierten Laufzeiten für `add()` und `remove()`.

  Dieses Prinzip für `remove()` folgt dem Besprochenen für `add()`.

3 Positionsbasierte Listen

Video 3 beginnt hier.

Position [Slide 13]

Wo bin ich *relativ* zu meinen Nachbarn? (Kein Rang, keine Indizes.)

[KarlKu auf Pixabay]

Definition

Positionsbasierte Listen haben mit unserem abstrakten Datentyp Liste gemeinsam, dass sie eine lückenlose Sequenz beschreiben. Im Gegensatz zur Liste, die ihre Elemente über ihren Rang zugänglich macht, lokalisiert die positionsbasierte Liste ihre Elemente durch ihre Nachbarschaftsbeziehungen.

In einer gewöhnlichen Liste identifiziere ich ein Element über seinen Rang, also eine ganze Zahl ≥ 0. In einer positionsbasierten Liste identifiziere ich ein Element über seine Position. Eine Position ist keine ganze Zahl, sondern ein abstraktes Gebilde, das ich auffordern kann: Zeige mir das in dir gespeicherte Element! Und: Gibt mir die Position deines Vorgängers! Gib mir die Position deines Nachfolgers!

Eine positionsbasierte Liste kann man sich also vorstellen wie eine Seilschaft. Jeder an das Seil geknüpfte Karabiner ist eine Position. Jeder Karabiner hat einen Vorgänger und einen Nachfolger am Seil. Jede Position verweist auf sein Element, nämlich den am Karabiner hängenden Bergsteiger.

Diese Relationen werden durch das Seil zusammengehalten und ändern sich nicht, auch wenn sich die Seilschaft durch das Gelände bewegt.

Wir können jedoch einen Bergsteiger aushaken und dafür eine andere Bergsteigerin einhaken, die dann seine Position übernimmt.

Oder wir können einen Karabiner komplett ausknüpfen. Dann gehört weder der Karabiner noch der angehängte Bergsteiger noch zur Seilschaft, und sein ehemaliger Vorgänger wird zum Vorgänger seines ehemaligen Nachfolgers, und sein ehemaliger Nachfolger wird zum Nachfolger seines ehemaligen Vorgängers.

Wir können auch einen neuen Karabiner einknüpfen und dort eine weitere Bergsteigerin einhaken. In diesen Fällen ändern sich die lokalen Nachbarschaftsverhältnisse, aber die *Positionen* der Bergsteiger ändern sich nicht; jeder bleibt an seinem Haken. Dies ist im Gegensatz zur Liste, wo sich nach dem Einfügen oder Entfernen eines Elements die Ränge sämtlicher nachfolgenden Elemente ändern.

ADT: Position [Slide 14]

```
getElement() // Returns the element stored at this position.
```

Sei Element e an Position p in Liste L gespeichert.

- p ändert sich nicht, wenn Listenelemente eingefügt oder gelöscht werden.
- p ändert sich nicht, wenn das mit ihr assoziierte Element e geändert oder ersetzt wird.
- p bleibt gültig, bis es aus L entfernt wird (zusammen mit e).

Erklärung

Was genau ist also eine ***Position***? Ich habe sie salopp als abstraktes Gebilde bezeichnet. In der Tat handelt es sich bei der Position um einen abstrakten Datentyp. Dieser stellt lediglich eine Methode zur Verfügung, nämlich `getElement()`, um das in dieser Position gespeicherte Element zu erfragen.

Dies ist der Hauptzweck einer Position, nämlich die Identifikation eines bestimmten Datenelements, wie es in der gewöhnlichen Liste der Rang tut.

Darüber hinaus können über die Position Nachbarschaftsverhältnisse erfragt werden. Diese hängen von der Natur des positionsbasierten abstrakten Datentyps ab. Um diese Abfragen zu unterstützen, muss der abstrakte Datentyp durch eine geeignete Datenstruktur implementiert werden, die eng mit der Datenstruktur des positionsbasierten abstrakten Datentyps zusammenhängt.

Eine Position ist nur als Teil ihrer Datenstruktur gültig. Daher gehören die Methoden zur Abfrage und Manipulation der Nachbarschaftsverhältnisse nicht zur Position, sondern zum positionsbasierten abstrakten Datentyp.

ADT: Positionsbasierte Liste *(Positional List)* [Slide 15]

```
// Access methods:

first()   // Returns the position of the first element (or null if empty).
last()    // Returns the position of the last element (or null if empty).

before(p) // Returns the position immediately before position p
          // (or null if p is the first position)
after(p)  // Returns the position immediately after position p
          // (or null if p is the last position)

size()    // Returns the number of elements in the list
isEmpty() // Returns a boolean indicating whether the list is empty.


// Update methods:

...
```

Definition

Schauen wir uns nun als unseren ersten positionsbasierten abstrakten Datentyp die positionsbasierte Liste an. Sie verfügt über 6 Abfragemethoden. `size()` und `isEmpty()` sind bereits gute Bekannte. `first()` und `last()` liefern jeweils die erste bzw. letzte Position der positionsbasierten Liste zurück.

`before()` und `after()` dienen der Abfrage von Nachbarschaftsbeziehungen: Sie liefern jeweils die Position des Vorgängers bzw. Nachfolgers der als Argument übergebenen Position.

ADT: Positionsbasierte Liste *(Positional List)* Forts. [Slide 16]

```
// Access methods:
...
// Update methods:

addFirst(e) // Inserts a new element e at the front,
            // returning the position of the new element.
addLast(e)  // Inserts a new element e at the rear,
            // returning the position of the new element.

addBefore(p, e) // Inserts a new element e just before position p,
                // returning the position of the new element.
addAfter(p, e)  // Inserts a new element e just after position p,
                // returning the position of the new element.

remove(p)   // Removes and returns the element at position p,
            // invalidating p.

set(p, e)   // Replaces the element at position p with e,
            // returning the element formerly at p.
```

Definition

Kommen wir nun zu den Änderungsmethoden der positionsbasierten Liste. `addFirst()` und `addLast()` verpacken das übergebene Element in eine neue Position und fügen diese am Anfang bzw. am Ende der Liste ein.

`addBefore()` und `addAfter()` verpacken ebenfalls das übergebene Element in eine neue Position, und reihen diese vor bzw. nach der übergebenen Position der Liste ein.

`remove()` entfernt die übergebene Position aus der Liste, wodurch *ihr* ehemaliger Vorgänger und Nachfolger nun Vorgänger und Nachfolger *voneinander* werden.

`set()` ersetzt das Element an der gegebenen Position. Die Nachbarschaftsverhältnisse ändern sich dabei nicht.

Implementation als doppelt verkettete Liste [Slide 17]

Erklärung

Für den abstrakten Datentyp positionsbasierte Liste bietet sich als Datenstruktur eine doppelt verkettete Liste an. Als Datenstrukur für die Position dient hier der Knoten der Liste, ausgestattet mit Zeigern auf das Element, die Vorgängerposition (also den Vorgängerknoten), und die Nachfolgerposition.

Animation

Folgen wir hier derselben Sequenz an Einfüge- und Entfernoperationen, mit der wir bereits den abstrakten Datentyp Liste illustriert haben. Allerdings verwenden wir hier keine Ränge, sondern Positionen, um uns auf Elemente zu beziehen.

Beginnen wir mit `addFirst(A)`, gefolgt von `addFirst(B)`, `addFirst(C)` und `addLast(D)`. Anschließend rufen wir `addAfter(first(), E)` auf.

Nun wollen wir das Element B aus der Liste entfernen. In der Praxis tun wir dies über eine Referenz auf seine Position `p`, so dass wir `remove(p)` aufrufen können. Oder, vielleicht wollen wir tatsächlich den Vorgänger von Position `q` entfernen, die das Element A enthält; dann rufen wir äquivalent `remove(before(q))` auf.

Abschließend entfernen wir das letzte und das erste Element der positionsbasierten Liste mit `remove(last())` und `remove(first())`.

Die asymptotischen Laufzeiten aller Methoden des abstrakten Datentyps positionsbasierte Liste implementiert mittels einer doppelt verketteten Liste sollten Sie sich selbst überlegen.

Zu guter Letzt muss noch bemerkt werden, dass man eine positionsbasierte Liste auch mit anderen Datenstrukturen implementieren kann. Die doppelt verkettete Liste bietet sich hier zwar sehr an, ist aber nicht die einzige Möglichkeit.

Einige mögliche Fehler [Slide 18]

- $p =$`null`
- p wurde aus der Liste entfernt
- p gehört zu einer anderen Liste

Auf welche Fehler sollte man testen?

Wo es in $O(1)$ Zeit möglich ist.

Java-Interface [Slide 19]

```
public interface Position<E> {
  /**
   * Returns the element stored at this position.
   *
   * @return the stored element
   * @throws IllegalStateException if position no longer valid
   */
  E getElement() throws IllegalStateException;
}

public interface PositionalList<E> extends Iterable<E> {

  /**
   * Returns the number of elements in the list.
   * @return number of elements in the list
   */
  int size();

  /**
   * Tests whether the list is empty.
   * @return true if the list is empty, false otherwise
   */
  boolean isEmpty();

  /**
   * Returns the first Position in the list.
   *
   * @return the first Position in the list (or null, if empty)
   */
  Position<E> first();

  /**
   * Returns the last Position in the list.
   *
   * @return the last Position in the list (or null, if empty)
   */
  Position<E> last();

  /**
   * Returns the Position immediately before Position p.
   * @param p a Position of the list
   * @return the Position of the preceding element (or null, if p is first)
   * @throws IllegalArgumentException if p is not a valid position for this list
   */
  Position<E> before(Position<E> p) throws IllegalArgumentException;

  /**
   * Returns the Position immediately after Position p.
   * @param p a Position of the list
   * @return the Position of the following element (or null, if p is last)
```

```
     * @throws IllegalArgumentException if p is not a valid position for this list
     */
    Position<E> after(Position<E> p) throws IllegalArgumentException;

    /**
     * Inserts an element at the front of the list.
     *
     * @param e the new element
     * @return the Position representing the location of the new element
     */
    Position<E> addFirst(E e);

    /**
     * Inserts an element at the back of the list.
     *
     * @param e the new element
     * @return the Position representing the location of the new element
     */
    Position<E> addLast(E e);

    /**
     * Inserts an element immediately before the given Position.
     *
     * @param p the Position before which the insertion takes place
     * @param e the new element
     * @return the Position representing the location of the new element
     * @throws IllegalArgumentException if p is not a valid position for this list
     */
    Position<E> addBefore(Position<E> p, E e)
      throws IllegalArgumentException;

    /**
     * Inserts an element immediately after the given Position.
     *
     * @param p the Position after which the insertion takes place
     * @param e the new element
     * @return the Position representing the location of the new element
     * @throws IllegalArgumentException if p is not a valid position for this list
     */
    Position<E> addAfter(Position<E> p, E e)
      throws IllegalArgumentException;

    /**
     * Replaces the element stored at the given Position and returns the replaced element.
     *
     * @param p the Position of the element to be replaced
     * @param e the new element
     * @return the replaced element
     * @throws IllegalArgumentException if p is not a valid position for this list
     */
    E set(Position<E> p, E e) throws IllegalArgumentException;
```

```
  /**
   * Removes the element stored at the given Position and returns it.
   * The given position is invalidated as a result.
   *
   * @param p the Position of the element to be removed
   * @return the removed element
   * @throws IllegalArgumentException if p is not a valid position for this list
   */
  E remove(Position<E> p) throws IllegalArgumentException;

  /**
   * Returns an iterator of the elements stored in the list.
   * @return iterator of the list's elements
   */
  Iterator<E> iterator();

  /**
   * Returns the positions of the list in iterable form from first to last.
   * @return iterable collection of the list's positions
   */
  Iterable<Position<E>> positions();
}
```

Java-Implementation: Doppelt verkettete Liste [Slide 20]

```
public class LinkedPositionalList<E> implements PositionalList<E> {
  //---------- nested Node class ----------
  /**
   * Node of a doubly linked list, which stores a reference to its
   * element and to both the previous and next node in the list.
   */
  private static class Node<E> implements Position<E> {

    /** The element stored at this node */
    private E element;            // reference to the element stored at this node

    /** A reference to the preceding node in the list */
    private Node<E> prev;         // reference to the previous node in the list

    /** A reference to the subsequent node in the list */
    private Node<E> next;         // reference to the subsequent node in the list

    /**
     * Creates a node with the given element and next node.
     *
     * @param e the element to be stored
     * @param p reference to a node that should precede the new node
     * @param n reference to a node that should follow the new node
     */
    public Node(E e, Node<E> p, Node<E> n) {
      element = e;
      prev = p;
      next = n;
    }

    // public accessor methods
    /**
     * Returns the element stored at the node.
     * @return the stored element
     * @throws IllegalStateException if node not currently linked to others
     */
    public E getElement() throws IllegalStateException {
      if (next == null)                    // convention for defunct node
        throw new IllegalStateException("Position no longer valid");
      return element;
    }

    /**
     * Returns the node that precedes this one (or null if no such node).
     * @return the preceding node
     */
    public Node<E> getPrev() {
      return prev;
    }
```

```
  /**
   * Returns the node that follows this one (or null if no such node).
   * @return the following node
   */
  public Node<E> getNext() {
    return next;
  }

  // Update methods
  /**
   * Sets the node's element to the given element e.
   * @param e    the node's new element
   */
  public void setElement(E e) {
    element = e;
  }

  /**
   * Sets the node's previous reference to point to Node n.
   * @param p    the node that should precede this one
   */
  public void setPrev(Node<E> p) {
    prev = p;
  }

  /**
   * Sets the node's next reference to point to Node n.
   * @param n    the node that should follow this one
   */
  public void setNext(Node<E> n) {
    next = n;
  }
} //---------- end of nested Node class ----------

// instance variables of the LinkedPositionalList
/** Sentinel node at the beginning of the list */
private Node<E> header;                    // header sentinel

/** Sentinel node at the end of the list */
private Node<E> trailer;                   // trailer sentinel

/** Number of elements in the list (not including sentinels) */
private int size = 0;                      // number of elements in the list

/** Constructs a new empty list. */
public LinkedPositionalList() {
  header = new Node<>(null, null, null); // create header
  trailer = new Node<>(null, header, null); // trailer is preceded by header
  header.setNext(trailer);                 // header is followed by trailer
}

// private utilities
```

```
/**
 * Verifies that a Position belongs to the appropriate class, and is
 * not one that has been previously removed. Note that our current
 * implementation does not actually verify that the position belongs
 * to this particular list instance.
 *
 * @param p  a Position (that should belong to this list)
 * @return   the underlying Node instance at that position
 * @throws IllegalArgumentException if an invalid position is detected
 */
private Node<E> validate(Position<E> p) throws IllegalArgumentException {
  if (!(p instanceof Node)) throw new IllegalArgumentException("Invalid p");
  Node<E> node = (Node<E>) p; // safe cast
  if (node.getNext() == null) // convention for defunct node
    throw new IllegalArgumentException("p is no longer in the list");
  return node;
}

/**
 * Returns the given node as a Position, unless it is a sentinel, in which case
 * null is returned (so as not to expose the sentinels to the user).
 */
private Position<E> position(Node<E> node) {
  if (node == header || node == trailer)
    return null; // do not expose user to the sentinels
  return node;
}

// public accessor methods
/**
 * Returns the number of elements in the list.
 * @return number of elements in the list
 */
@Override
public int size() { return size; }

/**
 * Tests whether the list is empty.
 * @return true if the list is empty, false otherwise
 */
@Override
public boolean isEmpty() { return size == 0; }

/**
 * Returns the first Position in the list.
 *
 * @return the first Position in the list (or null, if empty)
 */
@Override
public Position<E> first() {
  return position(header.getNext());
}
```

```
/**
 * Returns the last Position in the list.
 *
 * @return the last Position in the list (or null, if empty)
 */
@Override
public Position<E> last() {
  return position(trailer.getPrev());
}

/**
 * Returns the Position immediately before Position p.
 * @param p  a Position of the list
 * @return the Position of the preceding element (or null, if p is first)
 * @throws IllegalArgumentException if p is not a valid position for this list
 */
@Override
public Position<E> before(Position<E> p) throws IllegalArgumentException {
  Node<E> node = validate(p);
  return position(node.getPrev());
}

/**
 * Returns the Position immediately after Position p.
 * @param p  a Position of the list
 * @return the Position of the following element (or null, if p is last)
 * @throws IllegalArgumentException if p is not a valid position for this list
 */
@Override
public Position<E> after(Position<E> p) throws IllegalArgumentException {
  Node<E> node = validate(p);
  return position(node.getNext());
}

// private utilities
/**
 * Adds an element to the linked list between the given nodes.
 * The given predecessor and successor should be neighboring each
 * other prior to the call.
 *
 * @param pred      node just before the location where the new element is inserted
 * @param succ      node just after the location where the new element is inserted
 * @return the new element's node
 */
private Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {
  Node<E> newest = new Node<>(e, pred, succ); // create and link a new node
  pred.setNext(newest);
  succ.setPrev(newest);
  size++;
  return newest;
}
```

```
// public update methods
/**
 * Inserts an element at the front of the list.
 *
 * @param e the new element
 * @return the Position representing the location of the new element
 */
@Override
public Position<E> addFirst(E e) {
  return addBetween(e, header, header.getNext()); // just after the header
}

/**
 * Inserts an element at the back of the list.
 *
 * @param e the new element
 * @return the Position representing the location of the new element
 */
@Override
public Position<E> addLast(E e) {
  return addBetween(e, trailer.getPrev(), trailer); // just before the trailer
}

/**
 * Inserts an element immediately before the given Position.
 *
 * @param p the Position before which the insertion takes place
 * @param e the new element
 * @return the Position representing the location of the new element
 * @throws IllegalArgumentException if p is not a valid position for this list
 */
@Override
public Position<E> addBefore(Position<E> p, E e)
                          throws IllegalArgumentException {
  Node<E> node = validate(p);
  return addBetween(e, node.getPrev(), node);
}

/**
 * Inserts an element immediately after the given Position.
 *
 * @param p the Position after which the insertion takes place
 * @param e the new element
 * @return the Position representing the location of the new element
 * @throws IllegalArgumentException if p is not a valid position for this list
 */
@Override
public Position<E> addAfter(Position<E> p, E e)
                          throws IllegalArgumentException {
  Node<E> node = validate(p);
  return addBetween(e, node, node.getNext());
```

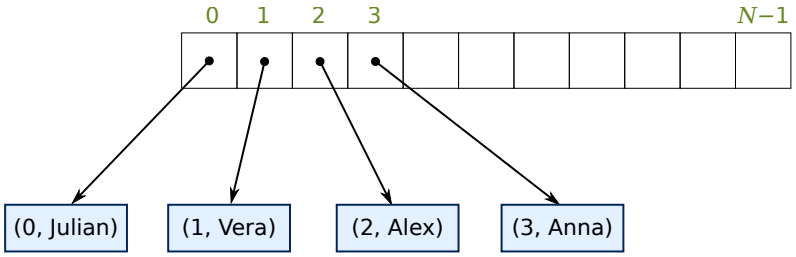
```
  }

  /**
   * Replaces the element stored at the given Position and returns the replaced element.
   *
   * @param p the Position of the element to be replaced
   * @param e the new element
   * @return the replaced element
   * @throws IllegalArgumentException if p is not a valid position for this list
   */
  @Override
  public E set(Position<E> p, E e) throws IllegalArgumentException {
    Node<E> node = validate(p);
    E answer = node.getElement();
    node.setElement(e);
    return answer;
  }

  /**
   * Removes the element stored at the given Position and returns it.
   * The given position is invalidated as a result.
   *
   * @param p the Position of the element to be removed
   * @return the removed element
   * @throws IllegalArgumentException if p is not a valid position for this list
   */
  @Override
  public E remove(Position<E> p) throws IllegalArgumentException {
    Node<E> node = validate(p);
    Node<E> predecessor = node.getPrev();
    Node<E> successor = node.getNext();
    predecessor.setNext(successor);
    successor.setPrev(predecessor);
    size--;
    E answer = node.getElement();
    node.setElement(null);       // help with garbage collection
    node.setNext(null);          // and convention for defunct node
    node.setPrev(null);
    return answer;
  }
}
```

Array-basierte Implementierung [Slide 21]

Worin besteht hier eine `Position`?

Es reicht nicht der Index, denn dieser verändert sich, falls Elemente davor eingefügt oder entfernt werden.

Eine `Position` ist eine eigene Datenstruktur, die einen Zeiger auf die Nutzerdaten sowie den Rang innerhalb des Arrays enthält; damit bleibt sie (aus Nutzersicht) immer unverändert:



Laufzeiten der Methoden?

- `before(p)`: $O(1)$
- `addBefore(p, e)`: $O(n)$ für Verschiebung der nachfolgenden Elemente und Inkrementierung ihrer Ränge
- `remove(p)`: $O(n)$ für Verschiebung der nachfolgenden Elemente und Dekrementierung ihrer Ränge

4 Iteratoren

Video 4 beginnt hier.

Erklärung

Wir haben nun bereits eine ganze Anzahl verschiedener abstrakter Datentypen und jeweils mehrere Implementierungsmöglichkeiten auf Basis verschiedener Datenstrukturen kennengelernt. Bei allen handelt es sich um sogenannte *Container* oder *Collections*, also Behälter oder Sammlungen, denn sie alle enthalten Elemente.

Eine häufige Operation auf Containern ist das Iterieren über die enthaltenen Elemente. Sie haben sicher alle schon Code geschrieben, der über die Elemente eines Arrays iteriert, indem ein Index hochgezählt wird.

Genauso wie abstrakte Datentypen Methoden auf Daten definieren, ohne auf eine bestimmte zugrunde liegende Datenstruktur Bezug zu nehmen, ist es wünschenswert, das Iterieren über die Elemente eines Containers zu ermöglichen, unabhängig vom Typ des Containers oder der zugrunde liegenden Datenstruktur.

Stellen Sie sich vor, Sie haben ein Array und eine verkettete Liste, und Sie möchten jeweils über deren Elemente iterieren. Statt für das Array einen Index hochzuzählen und in der verketteten Liste Zeigern zu folgen, schreiben Sie für beides denselben, kompakten Code, der nur ausdrückt, dass Sie iterieren, aber keinen Bezug auf die Datenstrukturen nimmt!

ADT: Iterator [Slide 22]

```
hasNext() // Returns true if there is at least one
          // additional element in the sequence.

next()    // Returns the next element in the sequence.

// Optional:
remove()  // Removes from the collection the element
          // returned by the most recent call to next().
```

Typische Nutzung in Java:

```
Iterator<String> iter = collection.iterator();
while (iter.hasNext()) {
  String string = iter.next();
  System.out.println(string);
}

for (String string : collection) {
  System.out.println(string);
}
```

Definition

Um dies zu ermöglichen, führt man einen sogenannten Iterator ein. Ein Iterator ist ein eigener abstrakter Datentyp, der Datenstruktur-unabhängige Methoden definiert, aber intern Datenstrukturen enthält, die eng mit den Datenstrukturen des Containers abgestimmt sind, über deren Elemente er iteriert. Hierin ähnelt er dem abstrakten Datentyp Position.

Ein Iterator definiert mindestens die beiden Methoden `hasNext()` und `next()`. Ein Container kann eine Methode bereitstellen, die einen Iterator zurückliefert. Auf diesem Iterator kann man die Methode `next()` wiederholt aufrufen. Bei jedem Aufruf von `next()` wird ein weiteres Element des Containers zurückgegeben. Solange noch Elemente vorhanden sind, die noch nicht zurückgegeben wurden, liefert `hasNext()` den Wert `true`.

Ein Iterator hält also einen *Zustand*, der sich bei jedem Aufruf von `next()` ändert.

Beispiel

Eine typische Verwendung eines Iterators sehen wir hier in Java. Die Variable `collection` ist eine Referenz auf einen Container. Die Natur dieses Containers spielt keine Rolle. Es könnte ein Array sein, eine verkettete Liste, oder eine Baumstruktur, oder etwas völlig anderes. Seine Methode `iterator()` liefert einen Iterator über die Elemente dieses Containers. Wir können nun über diese Elemente iterieren, indem wir die Methode `next()` des Iterators aufrufen, bis `hasNext()` falsch wird.

Im oberen Beispiel ist der Iterator explizit. Er wird durch einen expliziten Aufruf der Methode `iterator()` der `collection` instanziiert, und seine Methoden `hasNext()` und `next()` werden explizit aufgerufen.

Da diese Verwendung sehr häufig ist, bietet Java eine verkürzte Syntax für denselben Inhalt. Im unteren Codebeispiel ist der Iterator *implizit*. Er wird durch die spezielle `for`-Schleifen-Syntax mit dem Doppelpunkt erzeugt. Hier, genau wie im ersten Codebeispiel, enthält die Variable `string` bei jeder Iteration den nächsten String der `collection`.

Erklärung

Warum verwenden wir einen Iterator separat vom Container? Warum stellt der Container nicht selbst die Methoden `next()` und `hasNext()` zur Verfügung, plus eine Methode `start()`, um eine neue Iteration zu starten?

Die Antwort ist, dass in diesem Fall nur eine Iteration zur Zeit möglich wäre. Der Zustand der Iteration wäre an den Container gekoppelt. Häufig benötigt man jedoch gleichzeitig mehrere, unabhängige Iteratoren. Zum Auflisten aller Paare von Elementen eines Containers benötigt man beispielsweise zwei verschachtelte Schleifen, die beide jeweils über alle Elemente des Containers iterieren. Durch das Separieren des Iterators vom Container können die äußere und die innere Schleife unabhängig voneinander iterieren.

In unserem Beispiel werden die Elemente des Containers lediglich angezeigt. Häufig möchte man jedoch das eine oder andere Element aus dem Container löschen, während man darüber iteriert. Dies kann natürlich gefährlich sein. Wenn ich über eine verkettete Liste iteriere, und ich entferne das aktuelle Element, dann zerstöre ich möglicherweise die Datenstruktur, die der Iterator benötigt, um das nächste Element zu bestimmen.

Um das Löschen von Elementen während einer Iteration gefahrlos zu ermöglichen, kann ein Iterator eine Methode `remove()` zur Verfügung stellen. `remove()` entfernt das aktuelle Element aus dem Container, das heißt, das Element, das beim letzten Aufruf von `next()` zurückgeliefert wurde. Dadurch ist der Iterator bei korrekter Verwendung immer informiert, wenn während der Iteration Elemente gelöscht werden, und kann entsprechende Maßnahmen ergreifen.

Wie ist ein Iterator intern aufgebaut?

Ein Iterator über ein Array wird hauptsächlich eine Index-Variable enthalten, die bei der Instanziierung des Iterators mit 0 initialisiert wird. Diese Index-Variable wird dann bei jedem Aufruf von `next()` inkrementiert.

Ein Iterator über eine verkettete Liste wird hauptsächlich einen Zeiger auf den nächsten Knoten der Liste enthalten, der zu Beginn auf den Kopf der Liste zeigt. Bei jedem Aufruf von `next()` wird dieser Zeiger auf den Nachfolgezeiger des aktuellen Knotens gesetzt. Die Implementation eines Iterators folgt also denselben Mechanismen wie die gewöhnliche Implementierung einer Schleife von Hand.

Der große Vorteil des Iterators ist jedoch, dass diese Implementationsdetails nicht mehr vom Anwendungsprogrammierer selbst für jede Iteration neu codiert werden müssen. Dies ist fehleranfällig, und der Code ist schwer zu warten, da er von den Details der Datenstrukturen abhängt. Der Iterator versteckt die Abhängigkeit von den Datenstrukturen des Containers und den Code, den man ansonsten jedes Mal neu schreiben müsste, ein für alle Mal hinter einem generischen, trivial verwendbaren abstrakten Datentyp.

Iterator separat von DS [Slide 23]

Iteratoren werden *separat von der Datenstruktur* implementiert, über deren Elemente sie iterieren, damit mehrere Iterationen parallel und unabhängig voneinander stattfinden können.

Eine Java-Klasse, die einen Iterator anbietet, implementiert das Interface `Iterable`:

```
iterator() // Returns an iterator of the elements in the collection
```

Aufruf von `remove()` [Slide 24]

```
ArrayList<Double> data; // populate with random numbers (not shown)
Iterator<Double> walk = data.iterator();
while (walk.hasNext())
  if (walk.next() < 0.0)
    walk.remove();
```

In der `for`-Schleifensyntax ist der Iterator anonym; damit ist `remove()` nicht aufrufbar.

Zwei verschiedene Implementationstypen [Slide 25]

***Schnappschuss*-Iterator**	***Fauler* Iterator**
Instantiierung erstellt lokale Kopie der Daten	–
Instantiierung $O(n)$	Instantiierung $O(1)$
–	Was, wenn die Datenstruktur während der Iteration (nicht über `remove()` des Iterators) geändert wird?

Implementation für `ArrayList` [Slide 26]

```
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * Realization of a list by means of a dynamic array. This is a simplified version
 * of the java.util.ArrayList class.
 *
 * @author Michael T. Goodrich
 * @author Roberto Tamassia
 * @author Michael H. Goldwasser
 */
public class ArrayList<E> implements List<E> { // Excerpts
  private E[] data;                    // generic array used for storage

  /** Current number of elements in the list. */
  private int size = 0;                // current number of elements

  //------------------ nested ArrayIterator class ------------------
  /**
   * A (nonstatic) inner class. Note well that each instance contains an implicit
   * reference to the containing list, allowing it to access the list's members.
   */
  private class ArrayIterator implements Iterator<E> {
    private int j = 0;               // index of the next element to report
    private boolean removable = false; // can remove be called at this time?

    /**
     * Tests whether the iterator has a next object.
     * @return true if there are further objects, false otherwise
     */
    public boolean hasNext() { return j < size; } // size is field of outer instance

    /**
     * Returns the next object in the iterator.
     *
     * @return next object
     * @throws NoSuchElementException if there are no further elements
     */
    public E next() throws NoSuchElementException {
      if (j == size) throw new NoSuchElementException("No next element");
      removable = true; // this element can subsequently be removed
      return data[j++]; // post-increment j, so it is ready for future call to next
    }

    /**
     * Removes the element returned by most recent call to next.
     * @throws IllegalStateException if next has not yet been called
     * @throws IllegalStateException if remove was already called since recent next
     */
    public void remove() throws IllegalStateException {
```

```
      if (!removable) throw new IllegalStateException("nothing to remove");
      ArrayList.this.remove(j-1); // that was the last one returned
      j--;                        // next element has shifted one cell to the left
      removable = false;          // do not allow remove again until next is called
    }
  } //---------- end of nested ArrayIterator class ----------

  /**
   * Returns an iterator of the elements stored in the list.
   * @return iterator of the list's elements
   */
  public Iterator<E> iterator() {
    return new ArrayIterator(); // create a new instance of the inner class
  }
}
```

Iteratoren für `LinkedPositionalList` [Slide 27]

Frage: Iterieren wir über Positionen oder über Elemente?

Antwort: Der Iterator einer Datenstruktur iteriert immer über ihre Elemente:

```
for (String guest : waitlist)
```

Wir können jedoch *zusätzlich* einen Iterator über Positionen anbieten:

```
for (Position<String> p : waitlist.positions())
```

Damit diese Syntax funktioniert, liefert `positions()` einen `Iterable` (und nicht direkt einen Iterator).

Implementation für `LinkedPositionalList` [Slide 28]

```
public class LinkedPositionalList<E> implements PositionalList<E> {
  // Iterator code only

  // support for iterating either positions and elements
  //------------------ nested PositionIterator class ------------------
  /**
   * A (nonstatic) inner class. Note well that each instance
   * contains an implicit reference to the containing list,
   * allowing us to call the list's methods directly.
   */
  private class PositionIterator implements Iterator<Position<E>> {

    /** A Position of the containing list, initialized to the first position. */
    private Position<E> cursor = first(); // position of the next element to report
    /** A Position of the most recent element reported (if any). */
    private Position<E> recent = null; // position of last reported element

    /**
     * Tests whether the iterator has a next object.
     * @return true if there are further objects, false otherwise
     */
    public boolean hasNext() { return (cursor != null); }

    /**
     * Returns the next position in the iterator.
     *
     * @return next position
     * @throws NoSuchElementException if there are no further elements
     */
    public Position<E> next() throws NoSuchElementException {
      if (cursor == null) throw new NoSuchElementException("nothing␣left");
      recent = cursor;        // element at this position might later be removed
      cursor = after(cursor);
      return recent;
    }

    /**
     * Removes the element returned by most recent call to next.
     * @throws IllegalStateException if next has not yet been called
     * @throws IllegalStateException if remove was already called since recent next
     */
    public void remove() throws IllegalStateException {
      if (recent == null) throw new IllegalStateException("nothing␣to␣remove");
      LinkedPositionalList.this.remove(recent);  // remove from outer list
      recent = null;            // do not allow remove again until next is called
    }
  } //---------------- end of nested PositionIterator class ----------------

  //------------------ nested PositionIterable class ------------------
  private class PositionIterable implements Iterable<Position<E>> {
```

```
    public Iterator<Position<E>> iterator() { return new PositionIterator(); }
  } //-------------- end of nested PositionIterable class --------------

  /**
   * Returns an iterable representation of the list's positions.
   * @return iterable representation of the list's positions
   */
  @Override
  public Iterable<Position<E>> positions() {
    return new PositionIterable();  // create a new instance of the inner class
  }

  //------------------- nested ElementIterator class -------------------
  /* This class adapts the iteration produced by positions() to return elements. */
  private class ElementIterator implements Iterator<E> {
    Iterator<Position<E>> posIterator = new PositionIterator();
    public boolean hasNext() { return posIterator.hasNext(); }
    public E next() { return posIterator.next().getElement(); } // return element!
    public void remove() { posIterator.remove(); }
  }

  /**
   * Returns an iterator of the elements stored in the list.
   * @return iterator of the list's elements
   */
  @Override
  public Iterator<E> iterator() { return new ElementIterator(); }
}
```

5 Zusammenfassung

Zusammenfassung [Slide 29]

ADT:

- Liste
- Positionsbasierte Liste
- Iterator

DS:

- ArrayList (als dynamisches Array)
- Verkettete Liste