Programmieren II

Dr. Klaus Höppner

Hochschule Darmstadt – Sommersemester 2010

Dynamische Listen: Wahlfreier Zugriff

Collections

Listen mit wahlfreiem Zugriff

In der letzten Vorlesung wurde eine einfache Stapelliste implementiert.

In der Praxis sind allerdings Listen mit einem Zugriff auf die Elemente über einen Index viel relevanter. Für diese sollen dann folgende Operationen möglich sein:

- Lesen/Setzen eines Elements an einem bestimmten Index
- Einfügen eines neuen Elements am Ende
- Einfügen eines neuen Elements an einem bestimmten Index
- Löschen eines Elements an einem bestimmten Index

Ansätze für die Implementierung

Für die Realisierung solch eines Datencontainers werden üblicherweise zwei Ansätze verfolgt:

- Realisierung als *Array*, der bei Bedarf durch einen größeren ersetzt wird.
  Hier liegen die Daten somit in einem zusammen hängenden Speicherbereich entsprechend ihrer Reihenfolge.
- Realisierung als *doppelt verkettete Liste*.
  Hier „kennt“ jedes Listenelement seinen Vorgänger und Nachfolger, die Speicherorte der Daten können im Arbeitspeicher verstreut sein.

Ansatz 1: Array

Beide Ansätze sollen nun im Folgenden kurz skizziert werden.

Die Grundlage für die Array-basierte dynamische Liste wurde bereits mehrfach im Praktikum gelegt. Hier wurde jeweils ein Array mit einer bestimmten Maximalkapazität angelegt, der dann nach und nach gefüllt wurde, und hierbei jeweils eine Zählvariable hochgezählt wurde.

Entscheidender, noch fehlender Punkt ist das Wachsen der Liste über die (bisherige) Kapazität hinaus.

Liste 1 (mit Array)

```
public class List1<T> {
    private Object[] data;
    private int size = 0;
    private int max = 16;

    public List1() {
        data = new Object[max];
    }

    public int getSize() {
        return size;
    }
```

Liste 1 (Forts.)

```
    public int getMax() {
        return max;
    }

    public void add(T value) {
        if (size==max) {
            Object[] alt = data;
            max *= 2;
            data = new Object[max];
            for (int index=0; index<size; index++) {
                data[index] = alt[index];
            }
        }
        data[size++] = value;
    }
```


Liste 1 (Forts.)

```
    public T get(int index) {
        if (index<0 || index>=size)
            throw new IndexOutOfBoundsException();
        return (T) data[index];
    }

    public void set(int index, T value) {
        if (index>=size)
            throw new ArrayIndexOutOfBoundsException();
        data[index] = value;
    }
}
```


Analyse

Besonders interessant am Array-Index ist die Betrachtung der Laufzeiten:

- Die Zugriffszeit eine Element (mit `get` oder `set`) ist wie der Indexoperator von Arrays unabhängig von der Größe des Containers, also *konstant*.
- Beim Anhängen eines Elements ist die Lage komplexer. Wenn noch freie Positionen vorhanden sind, ist der Aufwand zum Besetzen des neuen Elements konstant. Beim Vergrößern der Kapazität müssen die bisherigen Elemente umkopiert werden. Dieser Aufwand verdoppelt sich jeweils analog zur Verdopplung der Kapazität, allerdings verdoppelt sich auch die Schrittzahl, bis die nächste Kapazitätserhöhung notwendig wird.
  Im Mittel ist der Aufwand zum Anhängen daher ebenfalls *konstant*.

Ansatz 2: Verkettete Liste

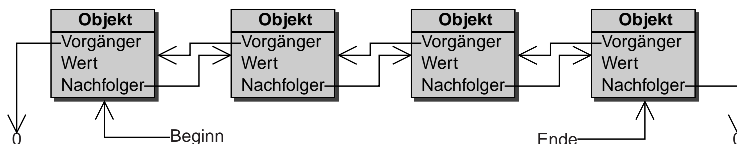
Der Ansatz der doppelt verketteten Liste erinnert an die Stapelliste aus der letzten Vorlesung.

Auch die verkettete Liste besteht aus Listenelementen. Diese enthalten neben dem eigentlichen Speicherwert auch Referenzen auf das vorherige und nachfolgende Element.

Diese Listenelemente werden innerhalb einer Klasse verwaltet, die sich das erste und letzte Listenelement merkt, und die Methoden zum lesenden und schreibenden Zugirff auf den Wert an einem bestimmten Index zur Verfügung stellt.

Dynamische Listen: Wahlfreier Zugriff Collections

Schaubild: Verkettete Liste



Dynamische Listen: Wahlfreier Zugriff Collections

Klasse `Element`

```
class Element<T> {
    private T value;
    private Element<T> next = null;
    private Element<T> prev = null;

    Element(T value) {
        this.value = value;
    }

    T getValue() {
        return value;
    }

    void setValue(T value) {
        this.value = value;
    }
```

Klasse Element (Forts.)

```
    Element<T> getNext() {
        return next;
    }

    void setNext(Element<T> next) {
        this.next = next;
    }

    Element<T> getPrev() {
        return prev;
    }

    void setPrev(Element<T> prev) {
        this.prev = prev;
    }
}
```


Klasse List2 (verkettete Liste)

```
public class List2<T> {
    private int size = 0;
    private Element<T> first = null;
    private Element<T> last = null;

    public int getSize() {
        return size;
    }

    public void add(T value) {
        Element<T> neu = new Element<T>(value);
        neu.setPrev(last);
        if (last!=null) last.setNext(neu);
        last = neu;
        if (first==null) first = last;
        size++;
    }
```


Klasse List2 (Forts.)

```
    private Element<T> elementAt(int index) {
        if (index<0 || index>=size)
            throw new IndexOutOfBoundsException();
        Element<T> result = first;
        while (index-- != 0) {
            result = result.getNext();
        }
        return result;
    }

    public T get(int index) {
        return elementAt(index).getValue();
    }
    public void set(int index, T value) {
        elementAt(index).setValue(value);
    }
}
```

Analyse

Auch die doppelt verkettete Liste kann man bezüglich ihres Laufzeitverhaltens betrachten:

- Wenn man auf ein Element mit bestimmten Index zugreifen will, geht dies nicht direkt, sondern durch Abzählen startend beim ersten Element (bzw. bei einer besseren Implementation für Elemente in der zweiten Hälfte vom letzten Element aus). In diesem Fall ist nur die Zugriffszeit auf das erste und letzte Element *konstant*, der Aufwand zum Zugriff auf ein Element in der Mitte ist jedoch hoch und wächst *linear* mit der Größe der Liste.
- Das Anhängen eines Elements hinten ist unabhängig von der Containergröße und daher *konstant*.

Laufzeitverhalten beim Einfügen/Löschen

In den beiden Implementationen wurden nur Methoden zum Zugriff auf die Elemente des Containers und das Anhängen neuer Werte realisiert. Aber auch ohne Implementation der Methoden zum Einfügen und Löschen an anderen Stellen kann man deren Laufzeitverhalten beurteilen.

- Beim Einfügen oder Löschen eines Wertes am Anfang müssen bei Ansatz 1 (Array) alle nachfolgenden Elemente verschoben werden. Der Aufwand hierfür wächst *linear* mit der Größe des Containers.
- Demgegenüber muss zum Einfügen eines Wertes vorne bei Ansatz 2 (verkettete Liste) eine neue Instanz von `Element` erzeugt/zersört und Referenzen (für Vorgänger, Nachfolger) gesetzt werden. Der Aufwand hierfür ist unabhängig von der Größe des Containers, also *konstant*.

Laufzeitverhalten (Forts.)

- Beim Einfügen oder Löschen eines Wertes in der Mitte müssen bei Ansatz 1 (Array) wiederum alle nachfolgenden Elemente verschoben werden. Der Aufwand hierfür wächst somit als ebenfalls *linear* mit der Größe des Containers.
- Bei Ansatz 2 (verkettete Liste) ist der Aufwand zum Finden des Elementes in der Mitte linear, der Aufwand zum Einfügen oder Löschen an dieser Stelle dann konstant. Bei großen Containern überwiegt der Aufwand zum Finden der Mitte, somit ist wächst der Aufwand also *linear* zur Größe des Containers.

Übersicht

Operation	Array-basiert `ArrayList`	verkettete Liste `LinkedList`
Zugriff . . .		
. . . am Anfang	konstant	konstant
. . . in der Mitte	konstant	linear
. . . am Ende	konstant	konstant
Einfügen/Löschen . . .		
. . . am Anfang	linear	konstant
. . . in der Mitte	linear	linear*
. . . am Ende	konstant	konstant

* Finden linear; eigentliches Einf./Löschen konstant

Collections

In `java.util` existieren fertige Klassen für Datencontainer, die das Interface `Collection` implementieren, und daher auch oft *Collections* genannt werden. Hierbei existieren drei wesentliche Arten von Containern:

- **sequenzielle Container** bei denen die Werte innerhalb des Containers durch die Position identifiziert werden,
- **Mengen** bei denen es nur relevant ist, ob ein Objekt in der Menge enthalten ist oder nicht, und die Reihenfolge irrelevant ist,
- **assoziative Container** bei denen die Werte mit einem Schlüssel assoziiert werden, analog zu einem Wörter- oder Telefonbuch. Diese implementieren das Interface `Map` statt `Collection`, werden aber trotzdem häufig zu den Collections gezählt.

Das Interface `List`

Für sequenzielle Container mit einem wahlfreien Zugriff auf die Objekt-Positionen existiert das Interface `List`, das von zwei Klassen implementiert wird:

- **ArrayList** als Datencontainer, der intern einen Array verwaltet, analog zu Ansatz 1 aus dem letzten Abschnitt,
- **LinkedList** als doppelt verkettete Liste, analog zu Ansatz 2 aus dem letzten Abschnitt.

Beide liegen als parametrisierte Klassen vor, können also auch direkt für einen bestimmten Typ neben der (alten) Roh-Klasse verwendet werden.

Methoden von List<E>

boolean add(E e) fügt ein Objekt e hinten an.

void add(int index, E e) fügt das Objekt e an Position index ein.

void clear() löscht alle Werte aus der Liste.

boolean contains(Object o) gibt **true** zurück, wenn es in der Liste ein Element e gibt, für das der Ausdruck e.equals(o) wahr ist, sonst **false**. Es wird also nicht auf Identität der Referenzen, sondern Gleichheit (Äquivalenz) der Objekte geprüft.

E get(int index) gibt den Wert an dem angegebenen Index zurück. Wirft bei ungültigem Index eine IndexOutOfBoundsException.

Methoden von List<E> (Forts.)

int indexOf(Object o) gibt den Index des *ersten* Auftretens eines Listenelements an, das zu o äquivalent ist, sonst -1.

boolean isEmpty() ist wahr, wenn die Liste leer ist.

lastIndexOf gibt den Index des *letzten* Auftretens eines Listenelements an, das zu o äquivalent ist, sonst -1.

E remove(int index) entfernt das Element an Position index und gibt das entfernte Element zurück.

boolean remove(Object o) entfernt das erste Element, das zu o äquivalent ist (so vorhanden). Gibt **true** zurück, falls ein Element entfernt wurde.

Methoden von List<E> (Forts.)

E set(int index, E e) setzt den Listenwert an Position index aus e, gibt dann den *alten* Wert zurück.

int size() Anzahl der Listenelemente.

List<E> sublist(int start, int end) gibt eine List mit den Elementen von start (inkl.) bis end (exkl.) zurück.

Object[] toArray() gibt alle Elemente der Liste als Array vom Typ Object[] zurück.

T[] toArray(T[] a) füllt den übergebenen Array vom Typ T[] mit allen Elementen der Liste oder erzeugt einen neuen, wenn die Größe des Array nicht ausreicht. Rückgabewert ist der gefüllte Array.

Beispiele für Listenoperationen

```
Integer[] a = {5, 1, 4, 3, 1, -5, 1, 0};

List<Integer> l = new ArrayList(Arrays.asList(a));
l.add(23);

System.out.println(l.get(5)); // -5
System.out.println(l.indexOf(1)); // 1
System.out.println(l.lastIndexOf(1)); // 6
System.out.println(l.indexOf(2)); // -1

Integer alt = l.set(1, 2);
System.out.println("Alter Wert: "+alt);
System.out.println(l.indexOf(2)); // 1

l.remove(7); // Element an Position 7 entfernt
boolean result = l.remove(new Integer(3));
System.out.println(result); // true
result = l.remove(new Integer(-9));
System.out.println(result); // false, da kein Wert entfernt
```


`ArrayList` oder `LinkedList`

Die beiden Klassen `ArrayList` und `LinkedList` implementieren beide das Interface `List` und sind daher einfach gegeneinander austauschbar.

Welche der beiden Implementierungen von `List` verwendet wird, hängt von den Anforderungen der Anwendung ab:

- `ArrayList` hat Vorteile, wenn häufig ein Zugriff auf Elemente über den Index statt findet. Einfügen und Löschen am Ende ist schnell, das Einfügen und Löschen an einer beliebigen Position kann allerdings aufwendig sein (im Mittel linearer Aufwand).
- `LinkedList` ist bei einem Zugriff auf die Elemente über einen Index im Nachteil. Dafür ist das eigentliche Einfügen bzw. Löschen schnell und im Aufwand konstant, unabhängig von Position und Größe der Liste.
- I. A. ist der Speicherbedarf für `LinkedList` größer.

Von `List` zu Array ...

Mit der Methode `Object[] toArray()` wird ein Array vom Typ `Object[]` erzeugt, d. h. beim Zugriff auf die Array-Elemente ist meist ein expliziter Cast notwendig.

Hier hilft die zweite Variante `T[] toArray(T[])`. Diese wirkt etwas seltsam, weil sie einen Array vom Typ `T[]` als Parameter erwartet. Der so übergebene Array wird mit den Listenelementen gefüllt, sofern der Array hinreichend groß ist. Ansonsten wird stattdessen ein neuer Array angelegt. Rückgabewert ist in jedem Fall ein Array vom Typ `T[]`.
Häufig sieht man Ausdrücke der Art

```
Double[] a = l.toArray(new Double[0]);
```

. . . und zurück

Für den Weg von einem Array zu einer Liste existiert in `java.util.Arrays` die statische Methode `asList(...)`. Diese kann wahlweise mit einem Array oder mit mehreren Parametern aufgerufen werden und legt darüber die Implementierung einer `List`. Diese kann dann z. B. dem Konstruktor von `ArrayList` oder `LinkedList` übergeben werden.

Beispiel:

```
Integer[] array = {4, 7, 13, -5};
ArrayList<Integer> list1 =
         new ArrayList<Integer>(Arrays.asList(array));
LinkedList<String> list2 =
     new LinkedList<String>(Arrays.asList("Hallo","Echo"));
```