Datenstruktur Liste und Kapselung

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

Wir haben in der Fallstudie zum Code-Tagebuch eine Liste verwendet, um die Einträge zu verwalten. Dabei stand die Verwendung von Listen-Objekten im Vordergrund. Wir haben Listenobjekte erstellt, Elemente hinzugefügt und entfernt, über die Elemente iteriert, etc. Aber wie ist diese Datenstruktur eigentlich implementiert?
Wie wir an dem Beispiel der Liste gesehen haben, stellen Objekte eine bestimmte Funktionalität bzw. Dienste zur Verfügung.

Die ***Anwendungsschnittstelle*** eines Objekts besteht aus

- den Nachrichten, die es für Anwender zur Verfügung stellt;
- den Attributen, die für den direkten Zugriff von Anwendern bestimmt sind.

Das Ziel eines objekt-orientierten Designs ist es die Anwendungsschnittstelle präzise festzulegen. Die Festlegung von Anwendungsschnittstellen ist eine Entwurfsentscheidung, d.h. sie wird bereits beim Entwurf getroffen. Gleichzeitig soll der Zugriff "von außen" auf Teile der Implementierung, die nur für internen Gebrauch bestimmt sind, verhindert werden. Direkter Zugriff auf Attribute beispielsweise muss nicht gewährt werden, sondern kann mit Nachrichten/Methoden realisiert werden (siehe Abschnitt "Zugriffsmethoden").
Wir werden sehen, wie diese Aspekte bei der Implementierung von Listen zur Anwendung kommen.

Lernziele dieses Kapitels:

- Verschiedene Listenarten zu implementieren (einfach-/doppelt-verkettet, Array-Listen).
- Das Konzept Information Hiding zu verstehen.
- Die Semantik von Zugriffsmodifikatoren zu kennen und diese beim Design von Anwendungsschnittstellen einzusetzen.
- getter-/setter-Methoden sinnvoll einzusetzen.
- Die Grenzen des Information Hiding in Java zu erläutern.
- Das Konzept des Iterators im Hinblick auf Information Hiding zu implementieren.

1 Datenstruktur Liste

In diesem Kapitel werden wir verschiedene Implementierungen von Listen kennenlernen und miteinander diese vergleichen.
Wir beginnen dazu mit einer formalen Definition:

Definition Eine ***Liste über einem Typ*** T ist eine total geordnete Multimenge mit Elementen aus T (bzw. eine Folge, d.h. eine Abbildung $\mathbb{N} \to T$).
Eine Liste heißt ***endlich***, wenn sie nur endlich viele Elemente enthält.

Es gibt viele verschiedene Möglichkeiten Listen in Java zu implementieren. Wir betrachten hier zunächst drei Implementierungsvarianten:

1. als einfachverkettete Liste (Abschnitt 1.1)
2. als doppeltverkettete Liste (Abschnitt 4.1)
3. als Array-Liste (Abschnitt 4.2)

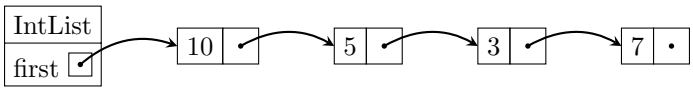
Um die Implementierung zu vereinfachen, betrachten wir hier zunächst nur Listen aus `int`-Werten. Um Listen abstrakt zu repräsentieren, verwenden wir hier die folgende Schreibweise: `[6,-3,84]` Die leere Liste wird dabei als `[]` repräsentiert.

1.1 Einfachverkettete Listen

Bei einfachverketteten Listen wird für jedes Listenelement ein Objekt mit zwei Attributen angelegt:

- zum Speichern des Elements
- zum Speichern der Referenz auf den Rest der Liste.

Die Liste mit den Elementen `[10,5,3,7]` erhält also folgende Repräsentation:



Die Listenknoten können dabei folgendermaßen implementiert werden:

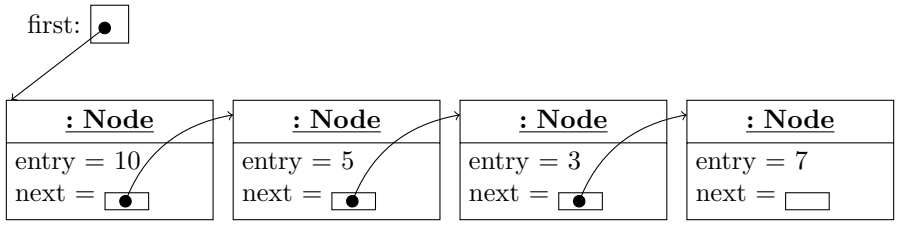
```
class Node {
    int value;
    Node next;

    Node(int value) {
        this.value = value;
        this.next = null;
    }
}
```

Mit dieser Implementierung von `Node` lässt sich die Liste [10,5,3,7] wie folgt erstellen:

```
Node n1 = new Node(10);
Node first = n1;
Node n2 = new Node(5);
n1.next = n2;
Node n3 = new Node(3);
n2.next = n3;
Node n4 = new Node(7);
n3.next = n4;
```

Der Speicherzustand lässt sich danach grafisch über ein Objektdiagramm darstellen:



Eine Menge von Objekten, die sich gegenseitig referenzieren, nennen wir ein ***Objektgeflecht***. Objektgeflechte werden zur Laufzeit aufgebaut und verändert, sind also dynamische Entitäten. Klassendiagramme kann man als vereinfachte statische Approximationen von Objektgeflechten verstehen.
In der Praxis will man nicht direkt mit den `Node`-Objekten arbeiten, sondern eine abstraktere Schnittstelle anbieten. Dazu schreiben wir eine Klasse `IntList`, welche Methoden wie `add`, `size` und `get` anbietet.

```
class IntList {
  Node first;

  IntList() {
    first = null;
  }

  void add(int element) { ... }
  int size()            { ... }
  int get(int index)    { ... }
}
```

Diese Liste kann dann wie folgt verwendet werden:

```
// Neue leere Liste erstellen:
IntList list = new IntList();
// Elemente einfügen:
list.add(10);
list.add(5);
list.add(3);
list.add(7);
// Element an Index 2 ausgeben:
System.out.println(list.get(2));
// Größe der Liste ausgeben:
System.out.println(list.size());
// Alle Zahlen in der Liste ausgeben:
Node n = list.first;
```

```
while (n != null) {
  System.out.println(n.value);
  n = n.next;
}
```

Wir zeigen nun die Implementierung der einzelnen Methoden im Detail:

Hinzufügen von neuen Einträgen am Ende der Liste Um neue Elemente am Ende der Liste einzutragen, gehen wir folgendermaßen vor.

- Erstelle zunächst einen neuen Knoten (ohne Nachfolger!).
- Wir machen eine Fallunterscheidung:
  - Falls die Liste bisher leer war (d.h. `first == null`), füge das Element als erstes Element ein.
  - Andernfalls, iteriere zum Ende der Liste und füge das Element dort an.

```
void add(int element) {
  Node newNode = new Node(element);
  if (first == null) {
    first = newNode;
  } else {
    Node n = first;
    while (n.next != null) {
      n = n.next;
    }
    n.next = newNode;
  }
}
```

Optimierung: `IntList`-Objekte könnten zusätzliche eine Referenz auf den letzten Knoten der Liste verwalten. Dies vereinfacht das Einfügen am Ende der Liste.

Länge der Liste Um die Länger der Liste zu ermitteln, iterieren wir, ausgehend vom ersten Knoten, über die Liste und inkrementieren für jeden Knoten eine Zählvariable (hier `res`). Die liefert die Anzahl der Knoten/Einträge.

```
int size() {
  int res = 0;
  Node n = first;
  while (n != null) {
    res = res + 1;
    n = n.next;
  }
  return res;
}
```

Element an bestimmter Position Um das Element an einer bestimmten Position zu ermitteln, iterieren wir über die Listeneinträge bis zum gewünschten Knoten. Danach wird der Eintrag, der in diesem Knoten abgelegt ist, zurückgeliefert. Die Einträge der Liste sind dazu, analog zu Arrays, ab 0 nummeriert.

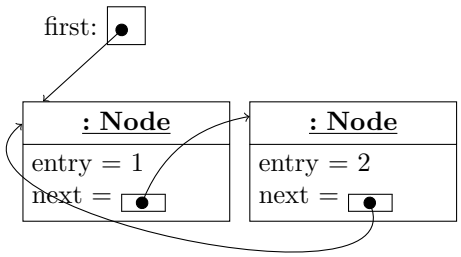
```
/* Liefert das Element an Position index
   requires 0 <= index < Anzahl der Eintraege
 */
int get(int index) {
  Node n = first;
  for (int i = 0; i < index; i++) {
    n = n.next;
  }
  return n.getValue();
}
```

1.1.1 Terminierung und Invarianten

Die Schleifen in den Methoden `add` und `size` terminieren nur dann, wenn wir beim Folgen der `next`-Referenzen immer zu einem anderem Knoten gelangen und somit irgendwann beim letzten Knoten und somit der `null`-Referenz ankommen. Die Klasse `IntList` stellt aber nicht sicher, dass dies immer gegeben ist. Beispielsweise könnte ein Benutzer von `IntList` diese wie folgt verwenden:

```
IntList list = new IntList();
list.add(1);
list.add(2);
list.first.next.next = list.first;
```

Damit würde ein Kreis in den Referenzen zwischen Knoten entstehen:



Die Klasse `IntList` hat also bestimmte Anforderungen an den Zustand der Liste, die immer gelten sollten. Solche Anforderungen nennt man auch ***Invarianten***.
Um sicherzustellen, dass Benutzer der Klasse `IntList` diese nur so verwenden, wie die Implementierung gedacht war (also insbesondere nicht die Invarianten verletzen), muss verhindert werden, dass von außen auf bestimmte Aspekte des Zustands zugegriffen werden kann. Mit diesem Problem beschäftigen wir uns nun im nächsten Abschnitt.

2 Information Hiding

Objekte stellen Dienste bzw. bestimmte Funktionalität zur Verfügung.

- Aus *Anwendersicht* bedeutet dies, dass Objekte Nachrichten empfangen und Ergebnisse liefern können.
- Aus *Implementierungssicht* müssen Zustände und Funktionalität durch
  → objektlokale Attribute
  → Referenzen auf andere Objekte
  → Implementierung von Methoden
  realisiert werden.

Das Prinzip des ***Information Hiding*** (deutsch meist ***Geheimnisprinzip***) besagt, dass Anwendern nur die Informationen zur Verfügung stehen sollen, die zur Anwendungsschnittstelle gehören, und alle anderen Informationen und Implementierungsdetails für ihn verborgen und möglichst nicht zugreifbar sind.
Die Gründe für Information Hiding sind vielfältig: Zum einen vermeidet man die unsachgemäße Verwendung von Attributen. Es vereinfacht die Struktur von Software durch Reduktion der Abhängigkeiten zwischen ihren Teilen. Außerdem ermöglicht es den Austausch von Implementierungsteilen, ohne dass die Verwendung von Objekten beeinträchtigt wird.

Beispiele Die Darstellung von Adressen ändert sich regelmäßig und passt sich dabei den aktuellen Gegebenheiten an.

- **Postleitzahlen**: In (West-)Deutschland wurden bis 1962 zweistellige, danach vierstellige Postleitzahlen, seit 1993 schließlich fünfstellige Postleitzahlen verwendet. Es ist semantisch nicht sinnvoll mit Postleitzahlen zu rechnen. In vielen Ländern bestehen die Postleitzahlen aus bis zu 10 Zeichen und enthalten nicht nur Ziffern, sondern auch Buchstaben und Bindestriche.
- **IP-Adressen**: Das Internetprotokoll (IP) ist eines der wichtigsten Protokolle des Internets. Es legt fest, wie Geräte eindeutig addressiert werden können. Die Version IPv4 verwendet 32-Bit-Adressen und erlaubt es daher, Netze von bis zu 4.3 Milliarden direkt addressierbaren Geräten zu verwalten. Die Nachfolgeversion IPv6 nutzt 128-Bit-Adressen und erlaubt die direkte Adressierung von etwa $3,4 \cdot 10^{38}$ Geräten.

Programme, die von der tatsächliche Darstellung von Postleitzahlen bzw. IP-Adressen in Klassen abstrahieren und die Implementierungsdetails geheim halten, können deren Implementierung auf einfache Art ändern und an neue Anforderungen anpassen, indem die sie lediglich die String-Repräsentierung der Objekte nach außen sichtbar und verfügbar machen.

Information Hiding in Java Java ermöglicht es für Programmelemente ***Zugriffsbereiche*** zu deklarieren. Vereinfachend gesagt kann ein Programmelement nur innerhalb seines Zugriffsbereichs verwendet werden.
Java unterscheidet vier Arten von Zugriffsbereichen:

- nur innerhalb der eigenen Klasse (Modifikator `private`)
- nur innerhalb des eigenen Pakets[1] (ohne Modifikator)
- nur innerhalb des eigenen Pakets[1] und in Subklassen[2] (Modifikator `protected`)
- ohne Zugriffsbeschränkung (Modifikator `public`)

Generell können Klassen, Attribute, Methoden und Konstruktoren mit diesen Zugriffsmodifikatoren deklariert werden.

Beispiel: IntList Wir können die Klasse `IntList` so anpassen, dass von außen nicht mehr auf die internen Knoten zugegriffen werden kann. Dazu ist es ausreichend, das Attribut `first` mit dem Zugriffsmodifikator `private` zu markieren:

```
public class IntList {
  private Node first;

  IntList() {
    first = null;
  }

  public void add(int element) { ... }
  public int size()            { ... }
  public int get(int index)    { ... }
}
```

Die Methoden `add`, `size`, `get` werden mit `public` annotiert, da sie die Anwendungsschnittstelle des Liste darstellen.

2.1 Zugriffsfunktionen

Der Zugriff auf Attribute wird häufig über spezifische *getter- / setter-* Methoden realisiert. Hier die beiden Varianten im Vergleich:

```
class AttributMitDirektemZugriff {
   public int attr;
}

class AttributZugriffUeberMethoden {
   private int attr;
   public int getAttr() {
    return attr;
   }
```

[1] Pakete fassen Klassen zu einer größeren Einheit zusammen. Wir werden uns später noch näher mit Paketen beschäftigen.
[2] Wir werden uns später näher mit Subklassen beschäftigen.

```
    public void setAttr(int a) {
     if (a > 0) {
       attr = a;
     }
    }
}
```

Die Verwendung von getter-/setter-Methoden erlaubt es insbesondere Modifikationen zu kontrollieren, beispielsweise

- das Überprüfen der Gültigkeit von neuen Attributwerten
- das Benachrichtigen von anderen Objekten bei Änderungen

Man kann so ausserdem erzwingen, dass Attributwerte nicht direkt verändert werden können, in dem z.B. keine setter-Methode zur Verfügung gestellt wird.

Frage 1:

1. Schreiben Sie die Klasse `Date` so um, dass die Felder nicht direkt zugreifbar sind.

```
class Date {
    int day;
    int month;
    int year;

    Date(int day, int month, int year) {
      this.day = day;
      this.month = month;
      this.year = year;
    }
}
```

2. Fügen Sie nun Getter-Methoden für die einzelnen Attribute hinzu.
3. Wie kann man die Erzeugung von ungültigen Datumswerten (z.B. 31.02.2018) verhindern?

Beispiel: Web-Seiten Die Klasse `W3Seite` implementiert eine einfache Repräsentierung von Web-Seiten mit Titelzeile und Inhalt.

```
private String titel;
private String inhalt;

public W3Seite(String t, String i) {
   titel  = t;
   inhalt = i;
}
public String getTitel() {
 return titel;
}
```

```
   public String getInhalt(){
    return inhalt;
   }
}
```

Die obige Klasse kann ersetzt werden durch die folgende Implementierung, ohne dass Anwender der Klasse davon betroffen werden. Diese alternative Klassendeklaration verwendet nur ein Attribut; der Titel wird dem Inhalt vorangestellt und mit dem TITLE-Tag markiert.

```
  private String seite;
  public W3Seite(String t, String i) {
    seite =  "<TITLE>" + t + "</TITLE>" + i ;
  }
  public String getTitel() {
    int ix = seite.indexOf("</TITLE>") - 7;
    return new String(seite.toCharArray(), 7, ix);
  }
  public String getInhalt() {
    int ix = seite.indexOf("</TITLE>") + 8;
    return new String(seite.toCharArray(), ix,
                          seite.length() - ix );
  }
}
```

Wie an den Beispielen erläutert, erlaubt Information Hiding konsistente Namensänderungen in versteckten Implementierungsteilen und das Verändern versteckter Implementierungsteile, soweit sie keine Auswirkungen auf die öffentliche Funktionalität haben.
Diese Auswirkungen können mitunter subtil, aber dennoch wichtig sein: Die zweite Implementierung von W3Seite kann z.B. nur dann anstelle der ersten benutzt werden, wenn Titel den Substring </TITLE> nicht enthalten.
Es gilt für einen guten Programmierstil in Java die Regel:

> Attribute sollten privat sein und nur in Ausnahmefällen öffentlich.

2.2 Grenzen der Zugriffskontrolle

Die Verwendung von Zugriffsmodifikatoren schließt in Java nicht automatisch aus, dass interne Eigenschaften von außen verändert werden können.

Frage 2: Die Verwendung von `private` führt **nicht** automatisch zur Kapselung.

```
public class A {
   private int[] werte;

   public void setWerte(int[] ar){
      for (int i = 0; i < ar.length; i++) {
        if (ar[i] < 0) {
          // Repariere Eintrag
          ar[i] = 0;
        }
      }
      this.werte = ar;
   }
}
```

In obigem Beispiel ist nicht sichergestellt, dass das Array `werte` keine negative Einträge enthält.
Zeigen Sie an einem Beispiel, wie man negative Zahl in das `werte`-Array einträgt, ohne die Implementierung der Klasse `A` zu verändern.

3 Iteratoren

Nun zurück zu unserer Klasse `IntList`!
Nachdem wir das Attribut `first` in der Klasse `IntList` als `private` markiert haben, können wir als Nutzer der Klasse nicht mehr über alle Einträge in der Liste iterieren, wie wir es im ersten Beispiel gesehen haben:

```
Node n = list.first; // Fehler: kann nicht auf first zugreifen
while (n != null) {
  System.out.println(n.value);
  n = n.next;
}
```

Nun könnte man statt dessen die `get`- und `size`-Methode verwenden, um über die Liste zu iterieren:

```
for (int i = 0; i < list.size(); i++) {
  System.out.println(list.get(i));
}
```

Frage 3: Warum ist das Iterieren mit `get` und `size` nicht optimal?

Um trotz korrekter Kapselung effizient über die einzelnen Elemente einer Liste iterieren zu können, kann das Konzept ***Iterator*** verwendet werden.
Iteratoren erlauben es, schrittweise über sogenannte Datenstrukturen für Datenansammlungen (engl. *collections*) wie Listen zu laufen, so dass alle Elemente der Reihe nach besucht werden. Im Zusammenhang mit Kapselung sind sie unverzichtbar.

Ein Iterator stellt dabei zwei Methoden zur Verfügung:

- `hasNext()` prüft, ob es weitere Einträge gibt.
- `next()` liefert den nächsten Eintrag in der Datenansammlung.

Hier das Code-Gerüst für einen Iterator über eine Liste von `int`s:

```
public class IntListIterator {
    //prueft, ob es noch weitere Eintraege gibt
    public boolean hasNext(){ ... }

    //liefert den naechsten Eintrag
    public int next() {...}
}
```

Die Implementierung des Iterators hängt essentiell von der Implementierung der entsprechenden Datenansammlung ab. Im folgenden zeigen wir als Beispiel die Implementierung eines Iterators für einfachverkettete Listen.

```
public class IntListIterator {
  private Node current;
  public IntListIterator(Node e) {
    this.current = e;
  }
  //prueft, ob es noch weitere Eintraege gibt
  public boolean hasNext(){
    return current != null;
  }
  //liefert den naechsten Eintrag
  public int next() {
    int res = current.getValue();
    current = current.getNext();
    return res;
  }
}
```

Wir reichern nun die Klasse `IntList` mit Iteratoren an:

```
public class IntList {
  // Erweiterung um Iteratoren
  private Node first;
  ...
  public IntListIterator iterator() {
     return new IntListIterator(first);
  }
}
```

Jeder Aufruf von `iterator()` liefert einen neuen Iterator, mit dem über die Liste iteriert werden kann.

```
public class IntListTest {
  public static void main(String[] args) {
    IntList l = new IntList();
    l.add(10);
    l.add(5);
    l.add(3);
```

```
    l.add(7);

    IntListIterator iter = l.iterator();
    while (iter.hasNext()) {
      StdOut.println(iter.next());
    }
  }
}
```

Der Iterator muss Zugriff auf die interne Repräsentation der Datenstruktur haben, über die er iteriert. Der `IntListIterator` erhält daher die Referenz auf das erste Listenelement, `first`.

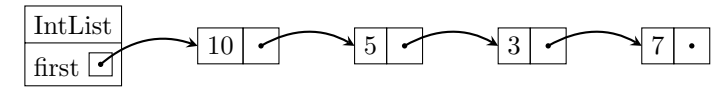
Bemerkung Eine alternative Möglichkeit, die Kapselung zu gewährleisten, sind innere Klassen, die wir im weiteren Verlauf der Vorlesung noch behandeln werden.

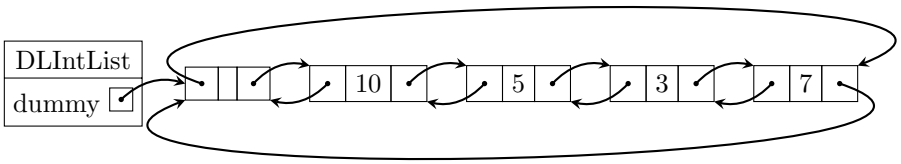
4 Weitere Listen-Implementierungen

Neben der einfach verketteten Listen aus Abschnitt 1.1 gibt es noch weitere Möglichkeiten, Listen zu implementieren.

4.1 Doppeltverkettete Listen

Einfachverkettete Listen kann man nur effizient in eine Richtung durchlaufen, nämlich vom ersten Element zum letzten. Um die Liste auch "rückwärts" einfach zu durchlaufen, fügen wir eine weitere, doppelte Verkettung ein: Jeder Knoten zeigt nicht nur auf den Nachfolgerknoten (`Node`-Attribute `next`), sondern auch auf den Vorgängerknoten. Wir ergänzen dazu die Listenknoten um ein weiteres `Node`-Attribute `prev`.
Hier sehen Sie die beiden Listentypen skizziert:





Effizientes Einfügen am Ende der Liste Das Einfügen am Ende der Liste erfordert zuerst das vollständige Durchlaufen der Liste. Dies ist bei Listen mit vielen Elementen nicht effizient. Wir skizzieren hier zwei Möglichkeiten, dieses Problem zu lösen:

- Variante 1: Wir verwalteten nicht nur eine Referenz auf das erste, sondern auch auf das letzte Element der Liste.
- Variante 2: Wir fügen ein Hilfselement ("Dummy") bei der doppeltverketteten Liste ein.
  - Dessen `next`-Referenz zeigt auf den ersten Knoten der Liste.
  - Seine `prev`-Referenz zeigt auf das letzte Element der Liste.

  Diese Lösung vermeidet Spezialfälle bei der Implementierung der Methoden (z.B. beim Einfügen am Anfang der Liste).

4.2 Array-Listen

Statt mit Verkettungen zu arbeiten, kann man Listen auch mit Hilfe von Arrays implementieren. Die Elemente werden dabei intern in einem Array gespeichert. Dazu wird zunächst ein Array mit einer bestimmten Anfangsgröße initialisiert. Das Array wird dann sukkzessive mit neuen Elementen gefüllt. Das Attribut `size` verwaltet dabei, wie viele Elemente bereits hinzugefügt wurden. Wenn das Array zu klein für neue Elemente ist, wird ein größeres erstellt und die alten Elemente werden in das neue Array kopiert.

```
public class IntArrayList {
    private int[] elementData;
    private int size;

    public IntArrayList(int initialArrayLength) {
      elementData = new int[initialArrayLength];
      size = 0;
    }

    // Element am Ende der Liste einfuegen
    public void add(int element) {
        ensureCapacity(size + 1);
        elementData[size] = element;
        size++;
    }

    // stellt sicher, dass Array genug Platz hat
    private void ensureCapacity(int minCapacity) {
        if (elementData.length < minCapacity) {
            // Groesse verdoppeln, mindestens auf minCapacity
            int newSize = Math.max(minCapacity,
                                   2 * elementData.length);
            elementData = Arrays.copyOf(elementData, newSize);
        }
    }
```

Die Operationen müssen dabei sicherstellen, dass nur der Teil des Arrays benutzt wird, der gültig ist. Um beispielsweise ein Element an einer bestimmten Indexposition zu erhalten, ist die Vorbedingung, dass dieser Index ein gültiges Listenelement enthält.

```
  /**
```

```
    *  Liefert das Element an Position index
    *
    *  requires 0 <= index < Anzahl der Eintraege
    */
public int get(int index) {
    Node n = first;
    for (int i = 0; i < index; i++) {
        n = n.getNext();
    }
    return n.getValue();
}
```

Wenn man sichergehen will, dass diese Vorbedingung eingehalten wird, kann auch ein Fehler ausgelöst werden, falls die Vorbedingung verletzt wird (Details zur Fehlerauslösung im Kapitel zu Exceptions):

```
    // liefert das Element an der gegebenen Position
    public int get(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException();
        }
        return elementData[index];
    }
```

Beim Suchen von Elementen wird nur über den gültigen Indexbereich iteriert:

```
    // gibt die erste Position eines Elements in der Liste
    //  oder -1, wenn das Element nicht in der Liste ist
    public int indexOf(int element) {
        for (int i = 0; i < size; i++) {
            if (elementData[i] == element) {
                return i;
            }
        }
        return -1;
    }
```

Zum Entfernen eines Elements aus der Array-Liste, wird zunächst das erste Vorkommen dieses Elements gesucht. Alle folgenden Elemente werden dann um eins nach links verschoben:

```
    // entfernt das 1. Vorkommen eines Elements aus der Liste
    public void remove(int element) {
      int indexOf = indexOf(element);
      if (indexOf < 0) {
          // Element nicht vorhanden
          return;
      }
      // alle Elemente nach dem ersten Vorkommen
      // nach links verschieben:
      for (int i = indexOf; i < size - 1; i++) {
          elementData[i] = elementData[i+1];
      }
      size--;
    }
```

Das Löschen von Einträgen ist im Vergleich zu verketteten Listen aufwendig, insbesondere das Löschen am Anfang der Liste.

Iterator für die Array-Liste Wir zeigen als weiteres Beispiel für die Implementierung von Iteratoren eine Implementierung für die Array-Liste. Diese erhält eine Referenz auf das interne Array der Array-Liste sowie die aktuelle Anzahl der Elemente. Das Attribut `position` gibt den Index des nächsten Elements an.

```
public class IntArrayListIterator {
    private int[] elementData;
    private int size;
    private int position;

    IntArrayListIterator(int[] elementData, int size) {
        this.elementData = elementData;
        this.size = size;
        this.position = 0;
    }

    public boolean hasNext() {
        return position < size;
    }

    public int next() {
        int elem = elementData[position];
        position++;
        return elem;
    }

}
```

Beim Erstellen des Iterators (in der Klasse `IntArrayList`) wird das interne Array an den Iterator übergeben:

```
    // liefert einen Iterator fuer die Liste
    public IntArrayListIterator iterator() {
        return new IntArrayListIterator(elementData, size);
    }
```

Frage 4: Was passiert, wenn der Array-Liste Elemente hinzugefügt bzw. entfernt werden, während der Iterator noch über die Liste iteriert?