Übungen zu Kapitel 20

20.1 `toString()` *und* `contains()` *in* `AbstractSList`

Grundsätzlich ist es möglich, eine bestehende Liste als Element bei sich selbst aufzunehmen. In diesem Fall löst jedoch der Aufruf von `toString()` in der von uns dargestellten Implementierung eine Ausnahme aus. Erweitern Sie diese Methode in der Klasse `AbstractSList`, so dass sie sich genauso verhält wie im Java Collection Framework.

Falls eine Liste eine `null`-Referenz als Element enthält, wird diese mit `contains()` nicht wiedergefunden, sondern eine `NullPointerException` ausgelöst. Verbessern Sie die Methode `contains()` in `AbstractList` entsprechend.

20.2 Array-basierte Implementierung von `SList`

Erstellen Sie eine Array-basierte Implementierung `ArraySList` von `SList`, in der die Elemente intern in einem Array verwaltet werden. Findet ein neues Objekt im aktuellen Array keinen Platz, wird ein größeres Array angelegt, das die Elemente des alten Array und das neue Objekt aufnimmt.

20.3 Erweiterung von SList-Implementierungen

Überschreiben Sie `contains()` und `toString()` in `LinkedSList` durch effizientere Methoden. Tipp: Schauen Sie sich die Definition von `get()` an. Alternativ ließe sich auch `get()` für sequentielle Zugriffe optimieren, indem man sich die zuletzt ausgegebene Position "merkt".

Grundsätzlich kann man in `LinkedSList` auf das Attribut `size()` verzichten und die Größe stets neu berechnen. Was sind die Nachteile, was die Vorteile?

Erweitern Sie `ArraySList` und `LinkedSList` um eine sinnvolle Implementierung von `remove()`.

20.4 Iteratoren in SList

Ergänzen Sie das Interface `SList` um die Methode

```
public java.util.Iterator iterator();
```

und definieren Sie diese Methode in der Klasse `AbstractSList`. Überschreiben Sie diese Methode in den Klassen `LinkedSList` und `ArraySList`.

Legen Sie anschließend je eine Instanz von `LinkedSList` und `ArraySList` mit 10000 zufälligen Integer-Werten zwischen 0 und 100 an. Ermitteln Sie anschließend für beide Collections den maximalen und minimalen Integer-Wert. Iterieren Sie zunächst mit den überschriebenen Iteratoren, anschließend mit dem von `AbstractSList` geerbten und vergleichen Sie die Laufzeiten.

Definieren Sie für `LinkedSList` einen zusätzlichen Iterator, der die Struktur entgegen ihrer Verkettung durchläuft, und führen Sie entsprechende Laufzeittests durch.

20.5 Iteratoren mit `remove()`

Ergänzen Sie den Iterator in `LinkedSList` und `ArraySList` um die Methode `remove()`, die ihrem Namen auch gerecht wird. Lösen Sie eine `IllegalStateException` aus, falls diese Methode ohne vorheriges `next()` oder mehrmals ohne `next()` dazwischen aufgerufen wird.

20.6 Fast-Fail Iterator

Wird für eine Collection von Objekten ein Iterator erzeugt, ist es kritisch, nachträglich Objekte hinzuzufügen oder zu löschen. Warum?

Implementieren Sie den Iterator in `LinkedSList` als so genannten fast-fail Iterator, dessen Methoden eine `ConcurrentModificationException` auslösen, falls zwischen dem Erzeugen des Iterators und dem Aufruf einer seiner Methoden eine Modifikation durch SList-Methoden `add()`, `remove()` und `addAll()` erfolgt. Eine Modifikation durch die Iterator-Methode `remove()` soll jedoch erlaubt sein.

Tipp: Erweitern Sie `LinkedList` um einen Zähler, der bei jeder kritischen Modifikation erhöht wird. Wird dieser Wert beim Erzeugen im Iterator registriert, kann man ihn zu Beginn einer jeden Iterator-Methode mit dem aktuellen Wert der Klasse vergleichen und entsprechend reagieren.

Anmerkung: Ein korrekter fast-fail Iterator moniert auch alle Änderungen, die nicht durch ihn selbst verursacht wurden, also auch durch `remove()`-Aufrufe eines anderen Iterators. Beachten Sie dies! (Die Implementierung muss jedoch nicht Thread-sicher erfolgen.)

20.7 LinkedList ohne Duplikate

Wie lässt sich zu einer `LinkedList` allein durch Konstruktorenaufrufe eine zweite `LinkedList` erzeugen, die jedes in der ursprünglichen List vorkommende Objekte genau einmal enthält? Formulieren Sie eine entsprechende Code-Strecke und testen Sie diese. Tipp: `new HashSet(...)`

20.8 Iteratoren außerhalb von Collections: Primzahl-Iterator

Formulieren Sie eine Klasse `PrimzahlIterator`, die der Reihe nach Primzahlen in `String`-Darstellung ausgibt. Ein mit `PrimzahlIterator(String z)` erzeugter Iterator gibt Primzahlen bis einschließlich `z` aus, wobei z eine ganze Zahl in der `String`-Darstellung bezeichnet. Wird ein Iterator mit dem parameterlosen Konstruktor erzeugt, so werden der Reihe nach alle Primzahlen (ohne obere Grenze) ausgegeben. Tipp: Verwenden Sie Attribute vom Typ `BigInteger`.

20.9 Iteratoren außerhalb von Collections: Permutations-Iterator

Viele Probleme mit kombinatorischem Hintergrund basieren auf dem Begriff der Permutation. Eine *n-stellige Permutation* lässt sich auffassen als ein Array der Länge *n* mit ganzzahligen, unterschiedlichen Elementen zwischen *0* und *n-1*.

Beispiel: Alle 3-stelligen Permutationen: 0-1-2, 0-2-1, 1-0-2, 1-2-0, 2-0-1, 2-1-0

Durch eine *n*-stellige Permutation wird eine Vertauschung einer sortierten Ausgangsmenge von *n* Elementen beschrieben. Insgesamt gibt es *n!* (Fakultät) *n*-stellige Permutationen.

Implementieren Sie eine Klasse `PermutationIterator`, so dass die mit

```
public PermutationIterator( int n )
```

erzeugten Instanzen alle *n*-stelligen Permutationen (als `int`-Array) liefern.

Tipp: Es wäre fatal, intern eine Liste aller n-stelligen Permutationen aufzubauen und diese der Reihe nach auszugeben. Vermeiden Sie eine explizite Berechnung von *n!*, da dieser Wert für kleine *n* schon sehr groß wird. Studieren Sie an folgendem Beispiel die natürliche Ordnungsrelation und entwickeln Sie einen Algorithmus, der die nächste Permutation (bzgl. dieser Ordnung) liefert:

0-1-2-3-4, 0-1-2-4-3, 0-1-3-2-4, ... 4-3-2-1-0

Die Methode `remove()` macht hier keinen Sinn und wird daher leer implementiert.

Alle 6-stelligen Permutationen lassen sich dann wie folgt erzeugen und ausgeben:

```
public static void main(String ag[]) {
   Iterator p = new PermutationIterator ( 6 );
   while( p.hasNext() ) {
      int a[] = (int[]) p.next();
      System.out.println( Arrays.toString( a ) );
   }
}
```

20.10 Traveling Salesman Problem (Brute-Force Lösung)

Das Problem des Handlungsreisenden ist ein klassisches Optimierungsproblem der Informatik. Zu einer gegebenen Menge von Orten ist eine Reiseroute zu finden, so dass jeder Ort genau einmal besucht wird und die zurückgelegte Wegstrecke minimal ist. Ist die Anzahl der Orte gering, lässt sich eine Lösung ermitteln, indem man ***alle*** möglichen Wege berücksichtigt: (Die Laufzeit wächst jedoch sehr schnell.) Sind *n* Orte gegeben, dann entspricht einer Reiseroute eineindeutig eine *n*-stellige Permutation (die nämlich angibt, in welcher Reihenfolge die Orte bereist werden). Wir gehen davon aus, dass der Reisende von einem ausgezeichneten Ort O_A abreist und dorthin wieder zurückkehrt.

Ist etwa `ort[]` ein Array der Länge *n*, der die Orte beschreibt, und `a[]` eine *n*-stellige Permutation (vgl. 20.9), dann ist dadurch die Reiseroute

O_A - `ort[a[0]]` - `ort[a[1]]` - `ort[a[2]]` - ... - `ort[a[n-1]]` - O_A

eindeutig festgelegt.

Die Klasse `CityMap` modelliert eine Karte mit ganzzahligen nicht-negativen Koordinaten. Der Konstruktor `public CityMap( int n, int max )` soll eine Landkarte mit *n* zufällig gewählten unterschiedlichen Orten erzeugen, deren Koordinaten nicht größer als `max` sind. Zusätzlich sollen die Orte auch vom ausgezeichneten Ort O_A=<*0, 0*> verschieden sein.

Die Klasse `Ort` besitzt die Attribute `int x` und `int y` (Koordinaten), einen passenden Initialisierungskonstruktor und eine Methode

```
double dist( Ort o )
```

zur Berechnung des Abstandes zwischen `this` und `o`.

Erzeugen Sie in der Methode `main()` der ausführbaren Klasse `TSP` eine Instanz von `CityMap` und durchlaufen Sie mit einem `PermutationIterator` alle möglichen Wege (vgl. Übung 20.8). Man ermittelt eine Reiseroute minimaler Länge, indem man in der Schleife die Länge der aktuellen Reiseroute mit der bislang kürzesten Wegstrecke vergleicht. Geben Sie am Ende die minimale Reiseroute und deren Länge aus. Analog kann man auch die Route mit der maximalen Weglänge bestimmen.

Hinweis: Aus Symmetriegründen wird jeder Weg 2-mal berücksichtigt, da er in zwei Richtungen durchlaufen werden kann. Modifizieren Sie den Iterator, so dass dieser keine "Duplikate" mehr liefert. Dennoch kann es mehrere minimale Wege geben. Modifizieren Sie das Programm, so dass alle minimalen Wege ermittelt werden (Beachten Sie, dass sich zwei Wege in der errechneten Weglänge minimal unterscheiden können, obwohl sie - mathematisch betrachtet - gleich lang sind.)

20.11 Mengenoperationen in Collections

Implementieren Sie die folgenden Mengenoperationen in der Klasse `AbstractSList`.

```
boolean containsAll(Collection c);
boolean addAll(Collection c);
boolean removeAll(Collection c);
boolean retainAll(Collection c);
void clear();
```

Erläutern Sie die unterschiedliche Semantik von `removeAll()` im Gegensatz zu `remove()`, dargestellt in folgenden beiden Code-Strecken:

```
LinkedList l = new LinkedList();
l.add( new Integer(17) );
l.add( new Integer(17) );
//Alternative 1                        //Alternative 2
ArrayList a = new ArrayList();
a.add( new Integer(17) );
l.removeAll( a );                      l.remove( new Integer(17) );
System.out.println( l );               System.out.println( l );
```

Die Alternative 1 lässt sich einfacher durch folgenden Aufruf bewerkstelligen.

```
l.removeAll( Collections.singleton( new Integer(17) ) );
```

Studieren Sie die API-Dokumentation der Methode `Collections.singleton()` unter [SUN05C].

20.12 Erweiterte for-Schleife

Mit dem Java 1.5 Feature der erweiterten for-Schleife lassen sich viele Iterationen sprachlich kompakt formulieren, ohne die Methoden `hasNext()` und `next()` explizit zu benutzen. Die Verwendung

```
for( Object o : c ){    // for-each Object o in c
   ...
   ...
}
```

ist erlaubt, falls `c` ein Array bezeichnet oder vom folgenden Interface-Typ ist:

```
public interface Iterable {
   java.util.Iterator iterator();
}
```

Zeigen Sie, dass mit den Klassen und Interfaces aus Übung 20.4 eine Methode mit der Code-Sequenz

```
SList sl = new LinkedSList();
sl.add( new Integer(12) );
sl.add( new Integer(13) );
for( Object o : sl )
   System.out.println( o );
```

nicht übersetzbar ist und beheben Sie das Problem durch eine Änderung in der Definition von `SList`:

```
public interface SList implements Iterable {
   ...
}
```

Weisen Sie nach, dass der vom Compiler erzeugte Byte-Code einer erweiterten for-Schleife auf die Methoden `hasNext()` und `next()` Bezug nimmt, indem Sie diese mit Konsolenausgaben versehen.

20.13 Generische Typen

Nicht-generische Collections erlauben eine Verwendung, wie im Eingangsbeispiel des Kapitels gezeigt:

```
List l = new ArrayList();
l.add( new Integer( 17 ) );
l.add( new Mitarbeiter( 1004, "Rolf" ) );
l.add( "Hallo" );
System.out.println( l );
```

Muss man auf diese Möglichkeit verzichten, falls man sich generische Typen beschränkt? Formulieren Sie das Coding unter Verwendung einer generischen Klasse und eines generischen Interface um, und verifizieren Sie, dass es sich übersetzen und ausführen lässt.