Technische Universität München
Fakultät für Informatik
Programmieren in Java

WS 1998/99
Übungsblatt 2
27. Oktober 1998

Übungen zum Programmieren in Java

Aufgabe 1 Einfach verkettete Listen

Betrachten Sie folgende Implementierung des Datentyps einer generischen einfach verketteten Liste[1].

a) Warum hat die Instanzvariable `data` in `Node` den Typ `Object`? Was muss man deshalb beim Lesen von Elementen (z.B. mit `getHead()`) zusätzlich tun?

b) Die Speicherung der Elementzahl in `count` ist redundant. Welchen Vorteil bietet diese Technik? Was ist allerdings bei der Implementierung zu beachten?

c) Die Klasse `List` hat nur den default-Konstruktor. Was bewirkt er in dieser Klasse, d.h. wie könnte man ihn explizit definieren?

d) Warum sind die Methoden `getTail()`, `addTail()` und `removeTail()` deutlich komplizierter (und weniger effizient) als die entsprechenden `Head`-Varianten?

e) In jeder Klasse gibt es (implizit oder explizit) eine Methode `toString`. Wozu dient diese im allgemeinen, und was ist also speziell in der Klasse `List` zu tun?

f) Mit einem sog. Enumeratoren, hier `ListEnumerator`, kann man die Elemente eines Container-Datentyps der Reihe nach durchgehen, und zwar unabhängig von seiner Implementierung. Ein Anwendungsbeispiel hierfür ist die Methode `toString`. Nachdem diese Methode in der Klasse `List` selbst definiert ist, hat sie Zugriff auf deren interne Variablen, so dass die abstrakte Zugriffsweise hier nicht unbedingt nötig ist. Wie sähe die direkte Implementierung (ohne `ListEnumerator`) aus?

```
class Node {
  Object data; // reference to user data
  Node   next; // reference to next element
  Node(Object d, Node n) {
    data = d;
    next = n;
  }
}
```

[1] Die Methoden der Klasse `List` sind die üblichen Standard-Methoden für verkettete Listen. Sie sind in ihrer Funktionalität – aber natürlich nicht in ihrer Implementierung – weitgehend identisch mit den Methoden der Klasse `Vector` aus der Java-Utility-Bibliothek, die als Feld implementiert ist. Diese Klasse wird heute abend im Detail vorgestellt.

```
class List {
  Node head; // reference to first element, or null
  int count; // redundant (could be computed)

// we use the default constructor

  int size() {
    return count;
  }

  boolean isEmpty() {
    return (head == null);
  }

  Object getHead() {
    return head.data;
  }

  Object getTail() {
    Node curr;
    for (curr = head; curr.next != null; curr = curr.next)
      ;
    return curr.data;
  }

  void addHead(Object obj) {
    head = new Node(obj,head);
    count++;
  }

  void addTail(Object obj) {
    Node n = new Node(obj,null);
    if (head == null)
      head = n;
    else {
      Node curr;
      for (curr = head; curr.next != null; curr = curr.next)
        ;
      curr.next = n;
    }
    count++;
  }


  void removeHead() {
    if (head == null)
      System.out.println("List is empty");
    else {
      head = head.next;
      count--;
    }
  }
```

```
void removeTail() {
  if (head == null)
    System.out.println("List is empty");
  else {
    if (head.next == null)
      head = null;
    else {
      Node prev = head;
      for (Node curr = head.next; curr.next != null; curr = curr.next)
        prev = curr;
      prev.next = null;
    }
    count--;
  }
}

void removeAllElements() {
  head = null;
  count = 0;
}

boolean contains(Object elem) {
  return indexOf(elem) >= 0;
}

int indexOf(Object elem) {
  int i = 0;
  for (Node curr = head; curr != null; curr = curr.next) {
    if (elem.equals(curr.data))
      return i;
    i++;
  }
  return -1;  // not found
}

Object elementAt(int index) {
  Node curr;
  for (curr = head; index-- > 0; curr = curr.next)
    ;
  return curr.data;
}

void copyInto(Object anArray[]) {
  int i = 0;
  for (Node curr = head; curr != null; curr = curr.next) {
    anArray[i++] = curr.data;
  }
}

ListEnumerator elements() { // give an enumerator, see below.
  return new ListEnumerator(this);
}
```

```
  public String toString() {
    int i = this.size();
    StringBuffer buf = new StringBuffer(); // to compose strings dynamically
    buf.append("[");
    for (ListEnumerator e = this.elements(); e.hasMoreElements();) {
      buf.append(e.nextElement().toString());
      if (--i > 0) buf.append(", ");
    }
    buf.append("]");
    return buf.toString();
  }
}


class ListEnumerator {// used to visit each element in turn, in an abstract way
  Node curr;

  ListEnumerator(List l) {
    curr = l.head;
  }

  boolean hasMoreElements() {
    return curr != null;
  }

  Object nextElement() {
    Object result = curr.data;
    curr = curr.next;
    return result;
  }
}
```

Aufgabe 2 (P) Anwendung verketteter Listen

Schreiben Sie ein Hauptprogramm `ListTest`, das jede Methode von `List` mindestens einmal zu Testzwecken – sinnvoll – verwendet. Achten Sie dabei auf eine vollständige Pfadüberdeckung, d.h. jedes Stück Programmcode der Methoden (insbesondere alle Fälle bei einer Fallunterscheidung) sollte mindestens einmal durchlaufen werden. Das Programm soll eine Prozedur `static void rev(List l)` enthalten und verwenden, die eine Liste revertiert, indem sie sie rekursiv durchläuft, elementweise abbaut und wieder (umgekehrt) aufbaut.

Aufgabe 3 (P) Tabellen: Listen mit Schlüssel

Die Klasse `List` von Aufgabe 1 soll nun in zweifacher Weise erweitert werden. Zum einen wird sie zur Effizienzsteigerung als doppelt verkettete Liste (d.h. mit zusätzlichen Rückwärtsreferenzen) implementiert.

Zum anderen soll die Liste als *Tabelle* (im Sinne einer Datenbank-Anwendung) verwendbar sein, d.h. die gespeicherten Elemente müssen nun mit Schlüsseln versehen sein, womit sie eindeutig geordnet gespeichert und gezielt wieder gelesen werden können. Um dies zu ermöglichen, wird ein *Interface* namens `HasKey` definiert, das den Typ für Elemente der Tabelle angibt:

```
public interface HasKey {
    public int key();
}
```

Interfaces werden erst morgen ausführlich behandelt. In erster Näherung können sie als (abstrakte) Klassen aufgefasst werden, in denen nur Methoden durch ihren Namen sowie durch Anzahl und Typ der Parameter deklariert werden. Eine Klasse *implementiert* ein Interface, wenn es eine Realisierung für die verlangten Methoden anbietet. Im Fall der Tabellenelemente bedeutet dies, dass sie eine Methode `public int key()` anbieten müssen, die den jeweils zugehörigen Schlüssel liefert. Ein Beispiel dafür ist die Klasse `Mitarbeiter` aus dem Skript, die durch ihre Personalnummer identifiziert werden können (siehe Aufgabe 5).

Entwickeln Sie den Datentyp `Table` mit seinen Hilfsklassen, möglichst analog zu `List`, gemäß `Table.skelett`:

```
class TNode //...

public class Table {
    //...
    public int size() //...
    public boolean isEmpty() //...
    public HasKey first() //...
    public HasKey last() //...
    public HasKey find(int key) //...
    public void insert(HasKey obj)
        //...
        // mit einfacher Fehlerbehandlung:
        if (/*...*/) {
            System.out.println("Table.insert: key "+key+" already in use");
            return;
        }
        //...
    public void remove(int key)
        //...
        // mit einfacher Fehlerbehandlung:
        if (/*...*/) {
            System.out.println("Table.remove: key "+key+" not found");
            return;
        }
        //...
    public void removeAllElements() //...
```

```
    public String toString() //...

    public boolean testConsistency() //... die Liste _rueckwaerts_ durchlaufen
        // und dabei pruefen, ob die Elementzahl stimmt und die Elemente in
        // der richtigen Ordnung vorliegen.

    public Enumeration elements() //...
}

class TableEnumerator implements Enumeration //...
```

Aufgabe 4 (P) Erzeugung eindeutiger Schlüssel

Für Anwendungen von `Table` ist es sinnvoll, ein Verfahren zu haben, mit dem eindeutige Schlüssel automatisch erzeugt werden können. Zum Beispiel wird in der nächsten Aufgabe eine Tabelle von Mitarbeitern mit eindeutigen Personalnummern erstellt, ohne dass der Anwender sich um die Details der Schlüsselerzeugung kümmern muss.

Füllen Sie das Skelett der Klasse `KeyGen`,

```
public class KeyGen {
    //...
    public static int next() //...
    public static void used(int key) //...
}
```

so dass sie folgender Spezifikation genügt: Bei jedem Aufruf von `next` wird ein neuer (d.h. bisher noch nicht verwendeter) Schlüssel zurückgegeben. Mittels `used` kann `KeyGen` zusätzlich angewiesen werden, den als Parameter übergebenen Schlüssel nicht zu erzeugen.

Hinweis: Diese Aufgabe ist mit etwas Geschick in wenigen Zeilen lösbar!

Aufgabe 5 (P) Anwendung von Tabellen im Personalwesen

Die in den beiden vorigen Aufgaben programmierten Teile werden nun zu folgender kleinen Beispiel-Anwendung zusammengesetzt: Für das Personalwesen einer Firma sollen Abteilungen modelliert werden, die im wesentlichen eine Tabelle von Mitarbeitern darstellen.

Erweitern Sie dazu zunächst das folgende Skelett der Klasse `Mitarbeiter`:

```
public class Mitarbeiter implements HasKey {
    public    String  nachname;
    public    String  vorname;
    int     personalnummer; // used as (non-modifiable) key
    public    boolean istWeiblich;

    public Mitarbeiter(String nn, String vn, int pn, boolean w) //...

    public int key() //...

    public String toString() //...
}
```

Modellieren Sie damit eine Klasse `Abteilung` mit Name, Chef-Eintrag und einer Tabelle von Mitarbeitern, wobei der Chef/die Chefin aus Uniformitätsgründen als Mitarbeiter eingetragen wird, gemäß dem folgenden Skelett:

```
public class Abteilung {
    //...
    Table personal; // includes chef!

    public Abteilung(String n, Mitarbeiter c) // ..

    public String      name    () //... Lesefunktion
    public Mitarbeiter chef    () //... Lesefunktion
    public Table       personal() //... Lesefunktion

    public String toString() //...
}
```

Vervollständigen Sie zuletzt folgendes Hauptprogramm `Test.skelett` zum Testen:

```
public class Test {
    public static void main (String args[]) {
        Abteilung a = //... nur mit ChefIn, sonst keine Mitarbeiter
        Table p = a.personal();

        System.out.println(p.testConsistency());
        //... zwei weitere MitarbeiterInnen einfügen
        System.out.println("");
        System.out.println(p.testConsistency());
        System.out.println(a);
        System.out.println("");

        // we assume now that the keys are 1001, 1002, and 1003
        for (Enumeration e = p.elements(); e.hasMoreElements();) {
            System.out.println(e.nextElement().toString());
        }
        System.out.println(!p.isEmpty());
        System.out.println(p.size()==3);
        System.out.println((Mitarbeiter)p.first());
        System.out.println((Mitarbeiter)p.find(1002));
        System.out.println((Mitarbeiter)p.last());

        p.remove(1002);
        p.remove(1003);
        System.out.println("");
        System.out.println(p.testConsistency());
        System.out.println(a);
        System.out.println("");

        p.remove(p.first().key());
        System.out.println(p.testConsistency());
        System.out.println(p);
    }
}
```