Technische Universität München
Fakultät für Informatik
O. Müller, D. von Oheimb

WS 1998/99
Lösungsvorschläge zu Blatt 2
27. Oktober 1998

Übungen zum Programmieren in Java

Aufgabe 1 Einfach verkettete Listen

a) Der Typ `Object` macht die Liste *generisch*, d.h. die Elemente einer Liste dürfen von beliebigem Referenztyp sein. Der Grund liegt darin, dass `Object` der allgemeinste Referenztyp ist. Die so definierten Listen sind *heterogen*, d.h. verschiedene Elemente können verschiedene Typen haben.

   Grundtypen können allerdings nicht Elemente für die Liste stellen, so dass beispielsweise `l.addHead(4)` nicht möglich ist. Statt dem Grundtyp `int` kann man aber den Referenztyp `Integer` verwenden und `l.addHead(new Integer(4))` schreiben.

   Beim Lesen eines Elements mit der Methode `getHead()` muss man beachten, dass ihr Ergebnis (statisch) nur den allgemeinen Typ `Object` besitzt, selbst wenn es (dynamisch) mit einem spezielleren Typ angefügt wurde. Wenn man nach dem Lesen wieder den spezielleren Typ haben möchte, muss ein entsprechender Typ–Cast durchgeführt werden.

b) Mit der Variable `count` wird der Aufruf der Methode `size` sehr effizient, da lediglich die Variable `count` gelesen wird. Ohne diese Variable müsste die Länge der Liste bei jedem Aufruf von `size` neu berechnet werden. Allerdings ist bei dieser Implementierung darauf zu achten, dass der Inhalt der Variable `count` immer konsistent mit der aktuellen Länge der Liste gehalten wird.

c) Der default-Konstruktor bewirkt eine Initialisierung aller Variablen durch default-Werte. Im konkreten Fall entspricht diese Initialisierung der folgenden Definition:

```
List() {
  head  = null;
  count = 0;
}
```

   Man beachte, dass man hier Werte erhält, die eine sinnvolle und konsistente Belegung der beiden Variablen widerspiegelt. Dies ist allgemein bei impliziten Initialisierungen nicht selbstverständlich und sollte stets vom Programmierer überprüft werden.

d) Die Methoden, die am Ende und nicht am Kopf der Liste operieren, sind deutlich komplizierter, da die Liste nur einfach verkettet ist und demnach erst bis zum Ende durchlaufen werden muss, bevor die eigentliche Operation beginnen kann. Darüberhinaus muss eine Fallunterscheidung zunächst überprüfen, ob die Liste leer ist, da in diesem Fall eine Traversierung keinen Sinn macht. (Zumindest diese Fallunterscheidungen könnte man verhindern, indem man trickreichere Implementierungen wählt, die immer mindestens ein Pseudoelement speichern).

e) Die Methode `toString` wandelt einen Datentyp in einen String um. In der Klasse `List` bewirkt die Methode `toString` eine Umwandlung der Elemente in Strings, die dann zu einem gemeinsamen String — eingeschlossen durch die Zeichen `[` und `]` und getrennt durch Kommata — zusammengefügt werden. Voraussetzung ist dabei, dass für jedes Element der Liste (vom Typ `Object`) in der entsprechenden konkreten Klasse eine Methode `toString()` existiert.

f) Durch den Zugriff auf die internen Variablen vereinfacht sich die Methode `toString` wie folgt:

```
public String toString() {
  int i = count;
  StringBuffer buf = new StringBuffer(); // to compose strings dynamically
  buf.append("[");
  for (Node n = head; n != null; n = n.next) {
    buf.append(n.data.toString());
    if (--i > 0) buf.append(", ");
  }
  buf.append("]");
  return buf.toString();
}
```

Aufgabe 2 Anwendung verketteter Listen

Ein mögliches Testprogramm, das die gestellten Forderungen erfüllt, finden Sie in der Datei `ListTest.java`:

```
class ListTest {
    public static void main (String args[]) {
        List      l    = new List();
        Integer[] ints = new Integer[4];

        l.removeHead();
        l.removeTail();
        System.out.println(l);
        l.addTail(new Integer(22));
        l.addTail(new Integer(33));
        l.addHead(new Integer(11));
        System.out.println(l);
        System.out.println(!l.isEmpty());
        System.out.println(l.size()==3);

        System.out.println();
        l.copyInto(ints);
        System.out.println(!l.contains(new Integer(44)));
        System.out.println(l.contains(new Integer(22)));
        System.out.println(l.getHead()==ints[0]);
        System.out.println(l.elementAt(l.indexOf(new Integer(22)))==ints[1]);
        System.out.println(l.getTail()==ints[2]);

        System.out.println();
        rev(l);
        System.out.println(l);
```

```
        l.removeHead();
        l.removeTail();
        System.out.println(l.contains(new Integer(22)));
        l.removeTail();
        System.out.println(l.isEmpty());
        System.out.println(l.size()==0);

        l.addHead(new Integer(44));
        for (ListEnumerator e = l.elements(); e.hasMoreElements();)
            System.out.println(e.nextElement().toString());
        l.removeAllElements();
    }
    static void rev(List l) {
        if (l.isEmpty())
            return;
        else {
            Object x = l.getHead();
            l.removeHead();
            rev(l);
            l.addTail(x);
        }
    }
}
```

Aufgabe 3 Tabellen: Listen mit Schlüssel

Datei `Table.java`:

```
import java.util.Enumeration;

class TNode implements java.io.Serializable {
    protected TNode   next;
    protected TNode   prev;
    protected HasKey data;
    TNode(TNode next,TNode prev, HasKey data) {
        this.next = next;
        this.prev = prev;
        this.data = data;
    }
}


public class Table implements java.io.Serializable {
    protected TNode head,tail;
    protected int size;

    public int size() {
        return size;
    }
    public boolean isEmpty() {
        return head == null;
    }
```

```
    public HasKey first() {
        return head.data;
    }
    public HasKey last() {
        return tail.data;
    }

    private TNode internal_find(int key) {
        TNode curr;
        for (curr = head; curr != null && curr.data.key() > key;
             curr = curr.next)
            ;
        return curr;
    }
    public HasKey find(int key) {
        TNode n = internal_find(key);
        return (n != null && n.data.key() == key) ? n.data : null;
    }
    public void insert(HasKey obj) {
        int key = obj.key();
        TNode next = internal_find(key);
        if (next != null && next.data.key() == key) {
            System.out.println("Table.insert: key "+key+" already in use");
            return;
        }
        TNode n = new TNode(next,next == null ? tail: next.prev,obj);
        if (next == head)
            head = n;
        else
            n.prev.next = n;
        if (next == null)
            tail = n;
        else
            n.next.prev = n;
        size++;
    }
    public void remove(int key) {
        TNode n = internal_find(key);
        if (n == null || n.data.key() != key) {
            System.out.println("Table.remove: key "+key+" not found");
            return;
        }
        if (n.next == null)
            tail = n.prev;
        else
            n.next.prev = n.prev;
        if (n.prev == null)
            head = n.next;
        else
            n.prev.next = n.next;
        size--;
    }
    public void removeAllElements() {
        head = tail = null;
```

```
        size = 0;
    }

    public String toString() {
        int i = this.size();
        StringBuffer buf = new StringBuffer();
        buf.append("[");
        for (TNode curr = head; curr != null; curr = curr.next) {
            buf.append(curr.data.toString());
            if (--i > 0) buf.append(", ");
        }
        buf.append("]");
        return buf.toString();
    }

    public boolean testConsistency() {
        int count = 0;
        int prev_key = 0;
        for (TNode curr = tail; curr != null; curr = curr.prev) {
            int key = curr.data.key();
            if (curr != tail) {
                if (key <= prev_key)
                    return false;
            }
            prev_key = key;
            count++;
        }
        return count == size;
    }

    public Enumeration elements() {
        return new TableEnumerator(this);
    }
}

class TableEnumerator implements Enumeration {
    TNode curr;

    TableEnumerator(Table l) {
        curr = l.head;
    }

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

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

In dieser Lösung werden Schlüssel in *absteigender* Reihenfolge eingfügt. Dies hat den Vorteil, dass das Einfügen von Elementen, was typischerweise mit aufsteigenden Schlüsselwerten erfolgt, dann am Anfang der Liste geschehen kann und damit sehr effizient wird.

Die Musterlösung enthält zusätzliche Feinheiten, nämlich Import-Deklarationen, Implementierung des Interfaces `java.io.Serializable`, sowie die Modifier `protected` und `private`. Diese werden erst an den folgenden Tagen benötigt und erklärt.)

Aufgabe 4 Erzeugung eindeutiger Schlüssel

Datei `KeyGen.java`:

```
public class KeyGen {
    private static int recent_key = 1000;

    public static int next() {
        return ++recent_key;
    }
    public static void used(int key) {
        if(key > recent_key)
            recent_key = key;
    }
}
```

Die Idee dabei ist der Gebrauch einer *statischen* Variable `recent_key` vom Typ `int`, in der der zuletzt ausgegebene Wert gespeichert wird. Dieser wird einfach bei jedem Aufruf der Klassenmethode `next` inkrementiert und zurückgegeben.

Um zu vermeiden, dass ein bestimmer Schlüssel in der Zukunft benutzt wird, verschiebt die Methode `used` den aktuellen Zählerstand einfach auf das Maximum des letzen Zählerstands und des als Parameter übergebenen Schlüssels. Damit werden bei allen späteren Aufrufen garantiert neue Schlüssel generiert.

Aufgabe 5 Anwendung von Tabellen im Personalwesen

Die vervollständigten Programme sind in den Dateien `Mitarbeiter.java`,

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

    public Mitarbeiter(String nn, String vn, int pn, boolean w) {
        nachname       = nn;
        vorname        = vn;
        personalnummer = pn;
        istWeiblich    = w;
    }

    public int key() {
        return personalnummer;
    }

    public String toString() {
        return (istWeiblich ? "Frau" : "Herr") + " " +
               vorname + " " + nachname + " (#" + personalnummer + ")";
    }
}
```

`Abteilung.java`

```
public class Abteilung implements java.io.Serializable {
    protected String name;
    protected Mitarbeiter chef;
    protected Table personal; // includes chef!

    public Abteilung(String n, Mitarbeiter c) {
        name = n;
        chef = c;
        personal = new Table();
        personal.insert(c);
    }

    public String      name    () { return name;     }
    public Mitarbeiter chef    () { return chef;     }
    public Table       personal() { return personal; }

    public String toString() {
        return "Abteilung: " + name     + "\n" +
               "Chef     : " + chef     + "\n" +
               "Personal : " + personal + "\n";
    }
}
```

und `Test.java` zu finden:

```
import java.util.Enumeration;

public class Test {
    public static void main (String args[]) {
        Abteilung a = new Abteilung("Test",
                        new Mitarbeiter("Schmidt","Martin",KeyGen.next(),false));
        Table p = a.personal();

        System.out.println(p.testConsistency());
        p.insert(new Mitarbeiter("Bader","Petra",KeyGen.next(),true));
        p.insert(new Mitarbeiter("Mayer","Hans" ,KeyGen.next(),false));
        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);
    }
}
```

Man beachte, dass es in der Klasse `Abteilung` keine eigenen Methoden zum Bearbeitung der Mitarbeiterliste (z.B. zum Einfügen) gibt und stattdessen auf die entsprechende Operation auf Listen zurückgegriffen wird. Um das Geheimnisprinzip der Objektorientierung konsequenter anzuwenden, könnten alternativ auch eigene Methoden in `Abteilung` definiert werden.