Übungen zu Kapitel 18

18.1 Stackimplementierung mittels Array – "semidynamischer" Stack:

Das folgende Coding realisiert einen Stack auf Grundlage eines Arrays. Da Arrays statische Datenstrukturen sind ist die Zahl der ablegbaren Stackelemente gleich der Grösse `size` des Arrays und fest vorgegeben – ein deutlicher Nachteil im Vergleich zur Realisation als verkettete Liste. Somit kann nicht nur der `underflow`-Fall sondern auch der `overflow`-Fall (Stack komplett belegt) eintreten. Der Zugriff auf Elemente des Stacks erfolgt nur über die öffentlichen Schnittstellenmethoden `push()` und `pop()` – es gibt keinen direkten wahlfreien Zugriff auf die Plätze des Arrays, so dass die Stack-Funktionalität gewahrt bleibt. Das Attribut `counter` ist das "Gedächtnis" des Stacks: Es bezeichnet die Anzahl momentan auf dem Stack abgelegter Datenelemente. Zugleich ist `counter` jedoch auch der Index der freien Arrayposition, auf den das nächste einzufügende Datenelement zu platzieren ist. Wenn `counter==0` ist der Stack leer und der `underflow`-Fall tritt ein:

```
class Stack {
   private int[] s;      // Behälter für Integer
   private int counter;
   private boolean underflow, overflow ;
   public Stack( int size ) {   // Array initialisieren
       s = new int[size];      counter = 0;
   }
   public void push( int x ) {
      overflow = ( counter >= s.length );  // Stack voll?
      if( !overflow ) { s[counter] = x;  counter++; }
      else throw new RuntimeException( "Stack overflow" );
   }
   public int pop( ) {
      underflow = ( counter == 0 );        // Stack leer?
      if( !underflow ) { counter--;   return s[counter]; }
      else throw new RuntimeException( "Stack underflow" );
   }
}
```

a) Machen Sie sich die Wirkungsweise der Implementierung klar: Erläutern Sie die Wirkung der Anweisung `counter++` der Methode `push()` und `counter--` der Methode `pop()`. Warum müssen in der Methode `pop()` die zurückgelieferten Datenelemente nicht aus dem Array "entfernt" werden?

b) Verwenden Sie das `Stack`-Coding: Erstellen Sie eine ausführbare Klasse, in der Sie ein `Stack`-Objekt erzeugen. Legen Sie Daten auf den Stack und holen Sie diese wieder ab. Lassen Sie sich den momentanen Wert von `counter` ausgeben. Lösen Sie den `overflow`- und `underflow`-Fall aus.

c) Entwickeln Sie aus der Klasse `Stack` eine Klasse `BetterStack`: Diese Klasse soll einen "quasidynamischen Stack" (mittels Integer-Array `s`) darstellen: In der Methode `push()` soll der `overflow`-Fall vom Benutzer verborgen bleiben, indem in diesem Fall ein ***doppelt*** so großes Array neu angelegt und der Inhalt des bisherigen Arrays mittels `System.arraycopy()` hineinkopiert wird. Es muß die bisherige Arrayvariable `s` am Ende der Operation auf das neue, vergrößerte Array zeigen. In der Methode `pop()` soll geprüft werden, ob das bisherige Array nur noch ***halb*** gefüllt ist. In diesem Fall soll ein nur noch halb so großes Array neu angelegt und der Inhalt des bisherigen Arrays hineinkopiert werden. Die bisherige Arrayvariable `s` muß am Ende der Operation auf das neue, verkleinerte Array zeigen. Verwenden Sie diese Klasse in einer ausführbaren Klasse, in der Sie eine Instanz von Betterstack erzeugen und deren Methoden testen. Überprüfen Sie, ob der Vergrößerungs- und Verkleinerungsvorgang korrekt arbeiteten.

18.2 Queue-Implementierung mittels Array – zyklisches Array:

Das folgende Coding realisiert eine Queue auf Grundlage eines Arrays. Somit ist die Zahl der Elemente der Queue gleich der Grösse `size` des Arrays und fest vorgegeben – ein deutlicher Nachteil im Ver-

gleich zur Realisation als verketteter Liste. Es kann nicht nur der `underflow`-Fall sondern auch der `overflow`-Fall (Queue komplett belegt) eintreten. Der Zugriff auf Elemente der Queue erfolgt nur über die öffentlichen Schnittstellenmethoden `put()` und `get()` – es gibt keinen direkten wahlfreien Zugriff auf die Plätze des Arrays, so dass die Queue-Funktionalität gewahrt bleibt.

Der Index `head` zeigt auf das erste ("älteste") Element der Queue . Die Entnahme alter Elemente geschieht bei `q[head]`. Der Index `tail` zeigt auf den nächsten freien Platz am Ende der Queue. Das Einfügen neuer Elemente geschieht bei `q[tail]`. Zur Vereinfachung wird nach Entnahme eines Elements der gesamte Arrayinhalt ***nicht*** zu niedrigeren Indices verschoben. Stattdessen wird `head` einfach hochgezählt. Der belegte Teil wandert somit zu höheren Indices. Der Nachteil ist: Der belegte Teil erreicht das Arrayende - aber das Array ist eventuell noch nicht voll; am Anfang befinden sich eventuell noch leere Plätze. Zur Nutzung der leeren Plätze am Anfang wird ein ***zyklisches Array*** realisiert - d.h. der belegte Teil wandert wieder in den Anfang des Array hinein: Die Indices `head` und `tail` laufen "im Kreis".

Dies geschieht durch Neuberechnung des Index `tail` nach Einfügen zu `tail=(tail+1)%size` und durch Neuberechnung des Index `head` nach Datenentnahme zu `head=(head+1)%size.` Fehlersituationen sind die leere Schlange (`underflow`-Fall), wobei `head==tail` und die volle Schlange (`overflow`-Fall) wobei der Index `tail` "rundherum gelaufen" ist, also wiederum `head==tail` gilt. Zur Unterscheidung vom `underflow`-Fall wird deshalb als `overflow`-Fall definiert, dass `tail` direkt vor `head` steht, d.h. `(tail+1)%size==head`. Auf diese Weise lässt man zwar einen Array-Platz ungenutzt, jedoch bleibt das Coding einfach:

```
class Queue {
   private int[] q;
   private int head, tail;     // Anfang + Ende der Queue
   private int size;           // Größe der Queue
   private boolean underflow, overflow ;
   public Queue( int n ) {
       q = new int[n];     size = n;     tail = 0;     head = 0;
   }
   public void put( int x ) {
      overflow = ( (tail + 1)%size == head );
      if( !overflow ) {
         q[tail] = x;     tail = (tail + 1)%size ;
      }
      throw new RuntimeException( "Queue overflow" );
   }
   public int get() {
      underflow = ( head == tail );
      if( underflow )
         throw new RuntimeException( "Queue underflow" );
      else {
         int x = q[head];     head = (head + 1) % size;
         return x;
      }
   }
}
```

Machen Sie sich die Wirkungsweise des zyklischen Arrays klar. Skizzieren Sie die Wirkung aufeinanderfolgender `put()`- und `get()`-Operationen hinsichtlich der Array-Belegung. Verwenden Sie das `Queue`-Coding: Erstellen Sie eine ausführbare Klasse, in der Sie ein `Queue`-Objekt erzeugen. Legen Sie Daten in der Queue ab und entfernen Sie diese wieder. Lassen Sie sich den momentanen Wert von `head` und `tail` ausgeben. Lösen Sie den `overflow`- und `underflow`-Fall aus.

Anmerkung zu Übung 18.1 und 18.2: Die Implementierungen von Stack und Queue als verkettete Listen oder aber (wie hier) arraybasiert verdeutlichen nochmals das Grundprinzip des ***Information Hi-***

dings: Die Schnittstelle der arraybasierten Klassen Stack und Queue ist dieselbe wie bei der Realisation mittels verketteter Listen - an der Außensicht und Verwendbarkeit hat sich somit nicht geändert. Die interne Umsetzung (Implementierung, verwendete Datenstrukturen) ist jedoch völlig anders. Die Konstanz der Methodenschnittstellen und ihrer wahrnehmbaren Funktionalität bei gleichzeitigem Verbergen interner Abläufe und Datenstrukturen ist ein Beispiel für die Grundprinzipien ***Kapselung*** und ***Information Hiding***:

Implementierungsdetails sollten soweit wie möglich "versteckt" werden. Datenstukturen erlauben den Zugriff auf ihre Datenelemente nur über wenige öffentliche, einfach handhabbare Methoden (eine einfache Schnittstelle) ohne die interne Datenstruktur und deren Implementierungsdetails offenzulegen. Der Vorteil ist: Interne Implementierungsdetails können jederzeit geändert werden, solange die Schnittstelle konstant bleibt. Das "Wie" der Implementierung wird versteckt, nur das "Was" der Datenhaltung wird in der öffentlichen Schnittstelle exponiert.

So haben unsere Stack-Klassen eine konstante Schnittstelle in Form der Methoden `push()` und `pop()`, unabhängig davon, ob intern als Array oder verkettete Liste implementiert. Implementierungsdetails sind von "Außen" nicht sichtbar und können jederzeit verändert werden.

18.3 Abzählen mittels zyklischer verketteter Liste:

Es stellen sich `n` Kinder im Kreis auf. Durch zyklisches Abzählen wird jedes `m`-te Kind abgezählt und scheidet aus dem Kreis aus. Die Kinder werden zu Anfang im Geiste von 1 bis n durchnummeriert. Das Abzählen beginnt vor dem Kind Nr. 1, dh Kind Nr. m scheidet als erstes aus. Man ermittle, in welcher Reihenfolge die Kinder ausscheiden und gebe dies aus, zB für den Fall n=8 und m=4 wäre dies: 4,8,5,2,1,3,7,6. Der Benutzer kann n und m vorgeben.

Man repräsentiere jedes Kind durch den Knoten einer verketteten zyklischen Liste. Es soll eine zyklischen verkettete Liste verwendet und der Abzähl- und Ausscheidevorgang durch Verschieben von Referenzen und Löschen von Knoten dargestellt werden. Jeder Knoten enthält die Nummer des Kindes und zeigt auf das ***nächste*** Kind in der Kette. Wird ein Kind beim Abzählen ausgewählt, so ist der entsprechende Knoten zu löschen. Beim Löschen eines Knotens soll dessen Nummer ausgegeben werden. Der nächste Abzählvorgang beginnt vor dem ***Nachfolger*** des ausgeschiedenen Kindes. Man kann das Coding der unsortierten verketteten Liste verwenden, muß aber die Methoden `add()` und `remove()` anpassen.

18.4 Stack als verkettete Liste + Erweiterungen:

Realisieren Sie in einer nichtausführbaren Klasse `Stack` sowie einer zugehörigen Knotenklasse `Node` einen Stack in Form einer verketteten Liste. Verwenden Sie das Coding dieses Kapitels für die Methoden `push()` und `pop()`. Erzeugen Sie in einer ausführbaren Klasse eine Instanz der Klasse `Stack`. Legen Sie mit `push()` Daten auf den Stack und entnehmen Sie mittels `pop()` Daten. Erweitern Sie Ihre Klasse `Stack` um folgende Methoden und testen Sie diese:

a) `void clear()`: Durch diesen Aufruf soll der gesamte Stack "auf einen Schlag" geleert werden.

b) `boolean isEmpty()`: Wenn der Stack leer ist, soll `true` zurückgegeben werden, andernfalls `false`.

c) `int size()`: Die Methode gibt die Anzahl der abgelegten Elemente des Stacks zurück.

d) `int peekTop()`: Die Methode liefert den Wert des obersten Datenelements, ***ohne*** dieses vom Stack zu enfernen. Wenn der Stack leer ist, soll eine `RuntimeException` ausgelöst werden.

e) `int peek( int index )`: Die Methode soll den Wert des Elements an der Stelle `index` zurückliefern. Die Elementzählung soll bei 1 beginnen: Das oberste Stack-Element habe die Position 1 usw. Wird eine nicht vorhandene Index-Position vorgegeben, soll eine `RuntimeException` ausgelöst werden.

f) `int search( int x )`: Die Methode soll den Stack nach einem Wert `x` durchsuchen und (wenn im Stack enthalten) dessen Position im Stack zurückliefern, andernfalls soll eine `RuntimeException` ausgelöst werden.

18.5 Unsortierte verkettete Liste:

Verwenden Sie die Klassen `Node` und `List`, durch die eine unsortierte verkettete Liste dargestellt wird: Es sollen Integer-Werte gespeichert werden. Erweitern Sie die Klasse `List` um die Methoden `add()` und `contains()` dieses Kapitels. Eine ausführbare Klasse diene zum Testen: Erzeugen Sie ein `List`-Objekt. Lassen Sie vom Benutzer vorgeben, wieviele Zahlen `n` in der Liste zu speichern sind. Dann sol-

len n ganzzahlige Zufallszahlen zwischen 0 und 100 aufgenommen werden. Erweitern Sie die Klasse List um folgende Methoden:

a) String toString(): Die Methode soll den Listeninhalt als String aufbereiten.

b) boolean removeLast(): Die Methode soll den letzten Knoten aus der Liste entfernen und true zurückliefern. Ist die Liste leer so wird false zurückgeliefert.

c) int sum(): Die Methode soll die Summe der Listenwerte zurückliefern.

d) int max(): Die Methode soll den maximalen Listenwert zurückliefern.

e) int size(): Die Methode soll die Anzahl der Listenelemente zurückliefern.

f) double mean(): Die Methode soll das Mittel der Listenwerte zurückliefern.

g) void clear(): Die Methode soll die Liste mit einem Schlag leeren.

h) int get(int index): Die Methode liefert den Wert im Knoten an der Indexposition index zurück. Falls die Position nicht existiert wird eine Ausnahme ausgelöst.

i) int indexOf(int x): Die Methode liefert die Indexposition des ersten Vorkommens von x in der Liste zurück – bzw. den Wert –1, falls die Liste den Wert x nicht enthält.

18.6 Operation auf verketteter Liste: Analysieren Sie das folgende Coding. Gegeben sei eine unsortierte verkettete Liste mit ihrer Knotenklasse Node. Was bewirkt die Methode reverse()?

```
class List {
   private Node head = null;      private Node tail = null;
   public List reverse() {
      List res = new List();
      Node p = head;
      while( p!=null ) {
         Node r = new Node( p.data );
         r.next = res.head;
         res.head = r;
         p = p.next;
      }
      return res;
   }
}
```

Stellen Sie die Wirkungsweise der Methode reverse() mittels der im Kapitel verwendeten "Kästchen-Pfeil"-Symbolik dar!

18.6 Zirkuläre Referenzen:

Verwenden Sie die Knoten Klasse Node dieses Kapitels und analysieren Sie mittels Debugging folgendes Testcoding:

```
class Zirkulation {
   public static void main( String[] args ) {
      Node n1 = new Node( 1 );
      Node n2 = new Node( 2 );
      Node n3 = new Node( 3 );
      n1.next = n2;
      n2.next = n3;
      n3.next = n1;   // Breakpoint setzen und Debuggen!
      n1 = null;
      n2 = null;
      n3 = null;     // Der Spuk verschwindet!
   }
}
```

Es wird eine sogenannte ***zirkuläre Referenz*** aufgebaut. Erläutern Sie die Bedeutung dieses Ausdrucks, indem Sie in "Kästchen-Pfeil"-Symbolik die Struktur darstellen, die durch die ersten sechs Zeilen von `main()` aufgebaut wird. Warum stellen zirkuläre Referenzen für die Garbage Collection ein gewisses Problem dar?

Anmerkung: Zirkuläre Referenzen werden vom Java Garbage Collector erkannt: Nachdem die letzte ***externe*** Referenz auf die "Insel" sich selbst referenzierender Objekte entfernt wurde, wird die sich selbst referenzierende Objektkette abgebaut.