Programmierkurs Java

Dr.-Ing. Dietrich Boles

Aufgaben zu UE 12 – Collections

(Stand 14.04.2021)

Aufgabe 1:

Ergänzen Sie weitere Methoden in den Klassen ArrayList und LinkedList der Vorlesung:

- void clear(): entfernt alle Elemente aus der Liste
- boolean contains(int value): überprüft, ob der als Parameter übergebene Wert in der Liste ist
- boolean isEmpty(): überprüft, ob die Liste leer ist
- void set(int index, int value): ersetzt das Element am angegebenen Index durch den neuen Wert
- int removeAtIndex(int index): Liefert und entfernt das Element an Position index aus der Liste
- ArrayList subList(int fromIndex, int toIndex): Erzeugt eine neue Liste mit den Elementen zwischen den Indizes fromIndex (inklusive) und toIndex (exklusive)

Aufgabe 2:

Die aus der Vorlesung bekannten Stacks (Stapel) arbeiten nach dem LIFO-Prinzip (Last-In-First-Out), d.h. das jüngste Element des Stacks ist als einziges Element zugreifbar und wird via der pop-Methode entfernt und geliefert. Dahingegen arbeiten Queues (Schlangen) nach dem FIFO-Prinzip (First-In-First-Out), d.h. das älteste Element einer Queue ist als einziges Element zugreifbar und wird via einer Methode dequeue entfernt und geliefert.

Implementieren Sie die folgende Klasse *Queue*:

```
class Queue {

      Queue()   // unbegrenzt grosse Queue

      boolean isEmpty()   // ist leer?

      void enqueue(int value) // speichert value in die Queue

      int dequeue()   // liefert und löscht das aelteste Element
                      // der Queue
```

}

Teilaufgabe (a):

Implementieren Sie die Klasse mit einem dynamischen Array

Teilaufgabe (b):

Implementieren Sie die Klasse mit einer verketteten Liste

Aufgabe 3:

n und *m* seinen ganzzahlige Konstanten größer als 2. In einem Kreis stehen *n* Kinder (durchnummeriert von 0 bis *n-1*). Mit Hilfe eines *m*-silbigen Abzählreims wird das jeweils *m*-te unter den noch im Kreis befindlichen Kindern ausgeschieden, bis kein Kind mehr im Kreis steht.

Schreiben Sie ein Java-Programm, das nach Vorgabe von *n* und *m* die Nummern der Kinder in der Reihenfolge ihres Ausscheidens auf die Konsole ausgibt.

Beispiel 1: Für *n*=6 und *m*=5 ergibt sich die Folge 4, 3, 5, 1, 2, 0

Beispiel 2: Für *n*=7 und *m*=9 ergibt sich die Folge 1, 4, 2, 3, 0, 5, 6

Teilaufgabe (a):

Wählen Sie als zugrunde liegende Datenstruktur des Lösungsalgorithmus ein boolean-Array der Größe n. Die Indizes entsprechen den Nummern der jeweiligen Kinder. Der Wert *true* in einem Element des Arrays gibt an, dass sich das Kind noch im Kreis befindet. Der Wert *false* repräsentiert ein ausgeschiedenes Kind.

Beispiel: Für n = 6 und m = 5 enthält das Array anfangs die Werte true, true, true, true, true, true. Nach dem ersten Durchlauf enthält es dann die Werte true, true, true, true, false, true, d.h. das Kind mit der Nummer 4 ist ausgeschieden.

Hinweis: Nutzen Sie den Modulo-Operator (%) für ein zyklisches Durchlaufen des Arrays (Array als Kreis).

Teilaufgabe (b):

Wählen Sie als zugrunde liegende Datenstruktur des Lösungsalgorithmus ein int-Array der Größe n. Die Indizes entsprechen den Nummern der jeweiligen Kinder. Der Wert jedes Element des Arrays ist die Nummer des nachfolgenden Kindes im Kreis.

Beispiel: Für n = 6 und m = 5 enthält das Array anfangs die Werte 1, 2, 3, 4, 5, 0. Nach dem ersten Durchlauf enthält es dann die Werte 1, 2, 3, 5, 5, 0, d.h. das Kind mit der Nummer 3 verweist jetzt auf das Kind mit der Nummer 5, Kind Nummer 4 ist ausgeschieden.

Hinweis: Nutzen Sie den Modulo-Operator (%) für ein zyklisches Durchlaufen des Arrays (Array als Kreis).

Teilaufgabe (c):

Wählen Sie als zugrunde liegende Datenstruktur des Lösungsalgorithmus eine ArrayList. Die ArrayList enthält zu jedem Zeitpunkt die Nummern der sich noch im Kreis befindlichen Kinder in der entsprechenden Reihenfolge.

Beispiel: Für n = 6 und m = 5 enthält die ArrayList anfangs die Werte 0, 1, 2, 3, 4, 5. Nach dem ersten Durchlauf enthält sie dann die Werte 0, 1, 2, 3, 5, d.h. das Kind mit der Nummer 4 ist ausgeschieden.

Hinweis: Nutzen Sie den Modulo-Operator (%) für ein zyklisches Durchlaufen der ArrayList (ArrayList als Kreis). Erweitern Sie die aus der Vorlesung bekannte ArrayList um weitere benötigte Methoden, bspw. zum Entfernen eines Elementes an einem bestimmten Index.

Teilaufgabe (d):

Wählen Sie als zugrunde liegende Datenstruktur des Lösungsalgorithmus eine selbst implementierte verkettete Liste. Definieren Sie dazu eine Klasse *Child*, die die Nummer des Kindes sowie eine Referenz auf das nachfolgende Kind speichert.

Hinweis: Bauen Sie eine zyklische verkettete Liste auf, d.h. jedes Kind verweist auf das nachfolgende Kind und das letzte Kind verweist auf das erste Kind. Realisieren Sie das Ausscheiden eines Kindes durch das Entfernen des entsprechenden Kindes aus der verketteten Liste, wie in der Vorlesung gelernt.

Teilaufgabe (e):

Wählen Sie als zugrunde liegende Datenstruktur des Lösungsalgorithmus eine *Queue* (siehe Aufgabe (1)). Fügen Sie anfangs die Kinder in der entsprechenden Reihenfolge zur Queue hinzu (enqueue). Gehen Sie dann beim Durchzählen so vor, dass das aktuelle Kind jeweils aus der Queue entfernt (dequeue) und direkt wieder eingefügt wird (enqueue). Das jeweils m-te Kind wird allerdings nur entfernt und nicht wieder eingefügt.

Aufgabe 4:

Implementieren Sie analog zur Klasse *ArrayList* bzw. *LinkedList* aus der Vorlesung eine Klasse *Set* bzw. *LinkedSet*. Die beiden Klassen entsprechen einer Collection im Sinne einer mathematischen Menge, d.h. ein Element (int-Wert) darf höchstens ein Mal in der Menge enthalten sein. Berücksichtigen Sie dies bei den Methoden, die Elemente in die Menge einfügen.

Aufgabe 5:

Implementieren Sie eine Klasse *DoubleSideStack* mit folgenden Methoden:

```
class DoubleSideStack {

      // liefert genau dann true, wenn sich kein Element auf dem
      // DoubleSideStack befindet
      public boolean isEmpty()

      // fuegt den uebergebenen Wert auf der rechten Seite des
     // DoubleSideStacks an
      public void pushRight(int value)
```

```
    // entfernt das am weitesten rechts liegende Element aus dem
    // DoubleSideStack und liefert es als Methodenwert
    public int popRight()

    // fuegt den uebergebenen Wert auf der linken Seite des
    // DoubleSideStacks an
    public void pushLeft(int value)

    // entfernt das am weitesten links liegende Element aus dem
    // DoubleSideStack und liefert es als Methodenwert
    public int popLeft()
}
```

Aufgabe 6:

Ein int-Wert ist ein Palindrom, wenn die Folge der Ziffern von links nach rechts gleich der Folge der Ziffern von rechts nach links ist; bspw. 123454321. Schreiben Sie ein Programm, das überprüft, ob ein eingelesener int-Wert ein Palindrom ist. Setzen Sie dabei zur Überprüfung geschickt einen Stack und eine Queue ein, auf die die einzelnen Ziffern gepackt werden.

Aufgabe 7:

Die bisher bekannte Klasse *Stack* ist in der Lage, int-Werte auf einem Stack zu speichern. Implementieren Sie analog dazu zunächst eine Klasse *CharStack*, die keine int-Werte sondern char-Werte speichert:

```
class CharStack {

    CharStack()  // unbegrent grosser Stack

    boolean isEmpty()  // ist leer?

    void push(char value) // speichert value in die Queue

    char pop()  // liefert das aelteste Element der Queue

}
```

Schreiben Sie dann ein Programm, das einen String mit Klammern der Typen (,), [,] daraufhin untersucht, ob die Klammerung in Ordnung ist, d.h. dass zu jeder öffnenden Klammer auch wieder eine schließende Klammer existiert und dass die Klammerung sich nicht überschneidet.

Korrekt: „`()[()][[]]`“

Nicht korrekt: „`(([][])`“ oder „`([)]`“

Setzen Sie dabei einen CharStack ein, der auf folgende Art genutzt wird: Die char-Werte des Strings werden von links nach rechts abgearbeitet. Öffnende Klammern werden auf den Stack gelegt. Bei einer schließenden Klammern ist es ein Fehler, wenn der Stack leer ist oder wenn der pop-Aufruf nicht die passende öffnende Klammer liefert.

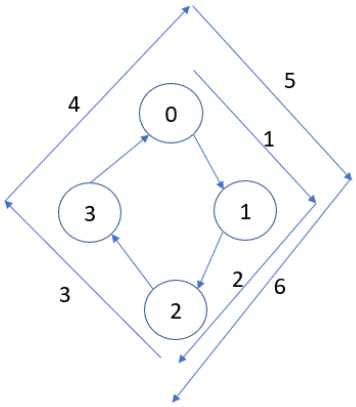
Mit folgendem Aufruf erzeugen Sie aus einem String das dazu gehörende char-Array:

```
String input = "([([][])()])";
char[] inputs = input.toCharArray();
```

Aufgabe 8:

Für positive int-Werte **a** (Dividend) und **b** (Divisor) liefert in Java der Ausdruck **a % b** den Rest der Ganzzahl-Division von a dividiert durch b. Also bspw. 7 % 4 → 3 oder 23 % 6 → 5.

Stellen Sie sich vor, der %-Operator würde in Java nicht existieren, dann ließe er sich mit einer verketteten Liste wie folgt implementieren. Es wird eine verkettete Liste mit den Werten 0 bis b-1 aufgebaut, wobei b-1 wieder auf 0 verweist, d.h. die Kette ist zyklisch. Zur Berechnung a % b wird nun beginnend bei 0 die Verkettung a mal durchlaufen. Der Endwert des Durchlaufs ergibt das Ergebnis.



Beispiel: In der Skizze wird als Beispiel eine Verkettung für die Modulo-Berechnung des Divisors 4 aufgebaut (zyklisch verkettete Knoten 0 bis 3). Die äußeren Pfeile mit den Beschriftungen 1 bis 6 deuten einen Durchlauf der Verkettung zur Berechnung von 6 % 4 an; Ergebnis ist 2.

Implementieren Sie folgende Klasse **Modulo** zur Modulo-Berechnung gemäß dem oben skizzierten Verfahren mit Hilfe einer zyklisch verketteten Liste:

```
class Modulo {

  // baut eine zyklische Verkettung der Zahlen
  // 0 bis number-1 auf
  Modulo(int number)

  // liefert den Wert value % number durch value-malige
  // Verfolgung der Verkettung
  int get(int value)
}
```

Das folgende Testprogramm liefert die Ausgabe „3 1 7 2“:

```
Modulo m = new Modulo(4);
System.out.println(m.get(7)); // 7 % 4 --> 3
System.out.println(m.get(17)); // 17 % 4 --> 1

m = new Modulo(11);
System.out.println(m.get(7)); // 7 % 11 --> 7
System.out.println(m.get(35)); // 35 % 11 --> 2
```