Prof. Martin Hofmann, PhD
Dr. Steffen Jost

Ludwig-Maximilians-Universität München
Institut für Informatik
22. Januar 2018

Lösungsvorschlag zur 12. Übung zur Vorlesung
Einführung in die Programmierung

Hinweis Bearbeiten Sie zuerst Präsenzaufgaben von Blatt 11, falls Sie damit noch nicht fertig geworden sind!

Wie in der Vorlesung besprochen wurde die Abgabefrist für Blatt 11 um 7 Tage verlängert. Bis zur Abgabefrist können Sie Ihre Abgabe zu Blatt 11 erneut hochladen, eine etwaige vorherige Abgabe wird dadurch komplett überschrieben.

A12-1 ***Verkettete Liste vs Array*** Dieser Übung sollten die Dateien `LinkedListDemo.java`, `SimpleList.java`, `SimpleIterator.java`, `MyArray.java` und `MyList.java` beiliegen. Bei Ausführung von `main` aus `LinkedListDemo` sollten Sie folgende Ausgabe erhalten:

```
List length=3: 1,2,3,
List length=3: 1,2,3,
```

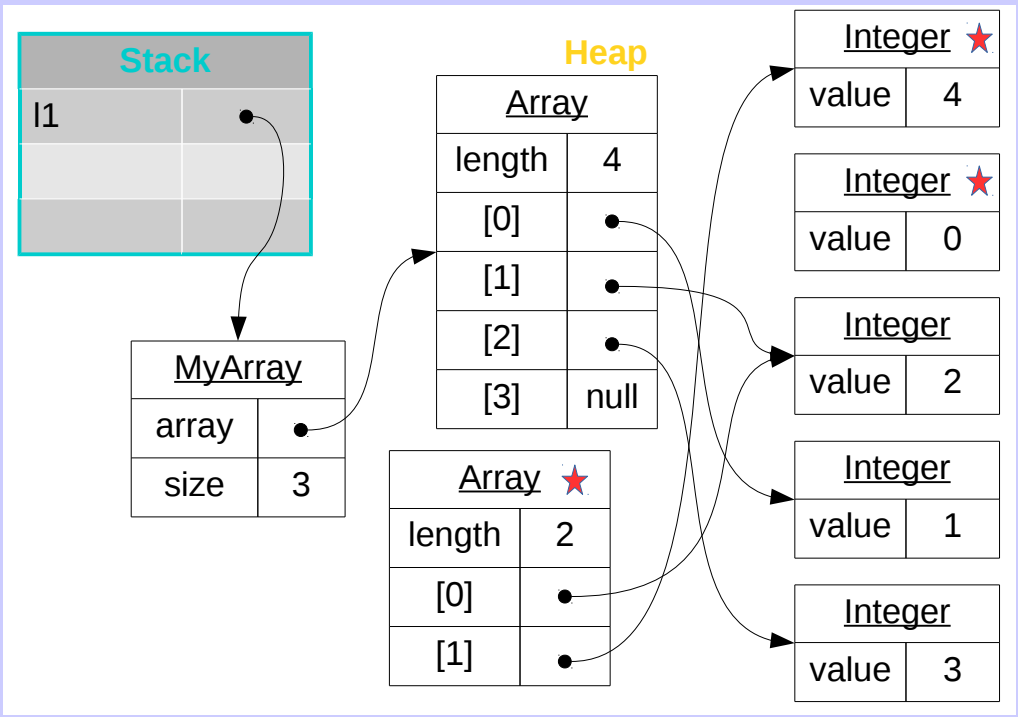
Aufgabe Skizzieren Sie Stack und Heap jeweils an den markierten Stellen in Methode `main`!

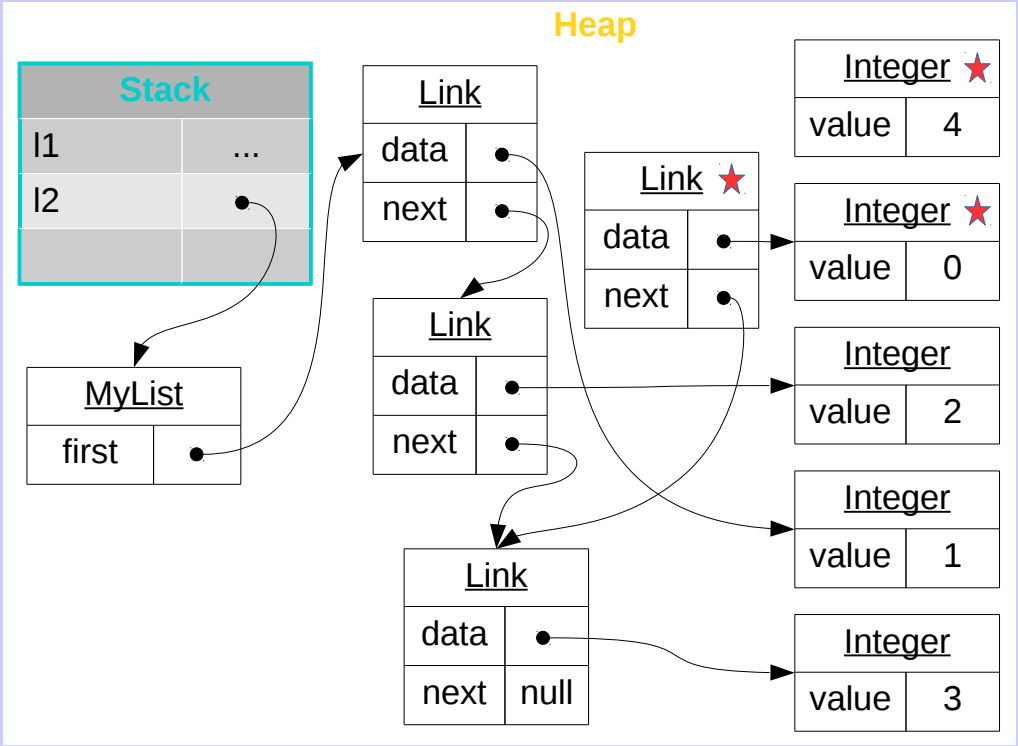
- Zeichnen Sie die unveränderlichen Objekte des Typs `Integer` als Objekte in eigene Boxen ein.
- Zeichnen Sie dabei auch Objekte in den Heap ein, auf welche keine Verweise mehr existieren (d.h. Objekte welche der Garbage Collector eigentlich deallokieren würde).
- Lassen Sie zur Übersichtlichkeit in Skizze 2 alles weg, was bereits in Skizze 1 eingezeichnet war und sich seitdem nicht verändert hat.

Hinweis: In der Vorlesung am 23.01.18 werden wir diese Klassen schrittweise entwickeln.

LÖSUNGSVORSCHLAG:

Integer sind immutable, weshalb Additionen neue Integer-Objekte erzeugen.
Objekte mit einem roten Stern werden vom Garbage Collector gelöscht.





A12-2 ***Einfügen in Verkettete Listen***

Schreiben Sie eine statische Methode `einfuegen` erweitern, welche ein Objekt in eine bereits aufsteigend *sortierte* verkette Liste an die richtige Stelle einfügt.

Das einzufügende Objekt soll der erster Parameter, und die sortierte Liste vom Typ `LinkedList` der zweite Parameter der Methode `einfuegen` sein. Das Einfügen soll ein Seiteneffekt sein, d.h. die Methode `einfuegen` liefert kein Ergebnis zurück.[1]

Die Methode `einfuegen` soll generisch sein, d.h. Objekte eines beliebigen Typs können in eine `LinkedList` eingefügt werden, falls diese Liste Objekte des gleichen Typs (oder eines Supertyps) speichert. Allerdings müssen wir zum Einfügen in der Lage sein, zwei solcher Objekte zu vergleichen. Nutzen Sie dafür erneut das Interface `Comparable` (oder auch `Comparator`).

a) Welche Signatur hat die Methode `einfuegen`?

Hinweis: Gesucht ist lediglich die Deklaration von `einfuegen` ohne den Methodenrumpf. Wenn Sie keine Ahnung haben, wie Sie an diese Aufgabe herangehen sollen, könnte es vielleicht hilfreich sein, wenn Sie sich zuerst eine nicht-generische Signatur ausdenken. Wie lautet etwa die Signatur diese Methode um einen String in eine String-Liste einzufügen? Ersetzen Sie dann den Typ `String` durch eine Typvariable und überlegen Sie, wie Sie sicherstellen, dass Sie die Listeneinträge noch vergleichen können – der Aufgabentext gibt ja schon das Stichwort `Comparable` vor, doch wie erzwingen Sie das ein beliebiger Typ dieses Interface implementiert?

LÖSUNGSVORSCHLAG:

```
public static <E extends Comparable<? super E>>
  void einfuegen(E element, LinkedList<E> list)
```

Wer lieber Comparator verwendet, benötigt natürlich ein drittes Argument:

```
public static void einfuegen(E element, LinkedList<E> list, Comparator<E> cmp)
```

Bemerkung: Eine Signatur wie etwa

```
public static <E extends Comparable<? super E>>
  void einfuegen(E element, LinkedList<? super E> list)
```

erlaubt auch nicht mehr als die oben angegebene Signatur, da wir aufgrund des regulären Subtypings für Argument `element` immer auch einen Erben von `E` übergeben dürfen.

[1] *Hinweis:* `LinkedList` funktioniert sehr ähnlich wie `MyList` in Aufgabe A12-1, bietet aber etwas mehr Komfort, z.B. einen `ListIterator` welcher auch über eine Methode `previous` verfügt. Wir will kann aber auch `MyList` verwenden, doch dazu muss entweder der Iterator erweitert werden oder man muss direkt Objekte der inneren Klasse `Link` manipulieren.

b) Implementieren Sie nun die Methode `einfuegen`!
Beispiel:

```
LinkedList<Integer> sorted = new LinkedList<>();
einfuegen(23,sorted); einfuegen(3,sorted); einfuegen(7,sorted);
einfuegen(57,sorted); einfuegen(1,sorted); einfuegen(31,sorted);
System.out.println(sorted.toString());
// Gibt aus: [1, 3, 7, 23, 31, 57]
```

Hinweis: Es ist unspezifiziert was passiert, wenn die Eingabeliste nicht sortiert war.

LÖSUNGSVORSCHLAG:

Verschiedene Lösungsvarianten sind möglich. Hier ist einer als Vorschlag:

```
public static <E extends Comparable<? super E>>
  void einfuegen(E element, LinkedList<E> list) {
    ListIterator<E> iter = list.listIterator();
    boolean notfound = true;
    while (iter.hasNext() && notfound) {
      notfound = element.compareTo(iter.next()) >= 0;
    }
    if (!notfound) iter.previous(); // Ein Schritt zu weit!
    iter.add(element);
    }
}
```

Achtung: wenn wir ein "zu großes" Element gefunden haben, dann müssen wir auch wieder einen Schritt zurückgehen, bevor wir einfügen können.

c) *Für Fortgeschrittene, welche die vorangegangenen Teilaufgaben schnell gelöst haben:* Kreieren Sie einen Code-Beispiel, welches nicht mehr kompiliert, sobald Sie probeweise alle Wildcards `?` aus der Signatur der Methode `einfuegen` entfernen.

Hinweis: Kreieren Sie dazu am besten zwei einfache Klassen.

LÖSUNGSVORSCHLAG:

Wir schreiben erst einmal eine Klasse `Ahne`, welche lediglich eine neue Umverpackung von `String` ist. Was diese Klasse macht, ist unerheblich. Hauptsache `Comparable` wird implementiert:

```
public class Ahne implements Comparable<Ahne> {
  private final String key;
  public Ahne(String key) { this.key = key; }

  @Override
  public String toString() { return key; }

  @Override
  public int compareTo(Ahne ahne) { return key.compareTo(ahne.key); }
}
```

Nun definieren wir noch einen Erben von `Ahne`, der für unser Beispiel keine weiteren Eigenschaften benötigt.

```
public class Erbe extends Ahne {
  public Erbe(String key) { super(key); }
}
```

Zuletzt noch Test-Code für die `main`-Methode:

```
LinkedList<Erbe> erben = new LinkedList<>();
einfuegen(new Erbe("D"),erben);
einfuegen(new Erbe("B"),erben);
einfuegen(new Erbe("A"),erben);
einfuegen(new Erbe("C"),erben);
einfuegen(new Erbe("E"),erben);
System.out.println(erben.toString());
```

Dieser Code wird abgelehnt, wenn wir die Wildcard aus der Signatur von `einfuegen` entfernen würden. Achtung, auch ohne Wildcard würde `einfuegen` von `Erbe`-Objekten in eine `LinkedList<Ahne>` noch funktionieren!

d) Wer noch 5 Minuten übrig hat, kann zur Vollständigkeit auch noch den einfachen Algorithmus "Sortieren durch Einfügen" in 3–4 Zeilen implementieren: Bei diesem Algorithmus werden alle Elemente der Eingabeliste in eine anfangs leere Ausgabeliste "eingefuegt". Es wird dabei also eine neue, sortierte Liste erschaffen, während die ursprüngliche Eingabeliste unverändert bleibt.

LÖSUNGSVORSCHLAG:

```
public static <E extends Comparable<? super E>>
  LinkedList<E> sortByInsertion(LinkedList<E> tosort) {
    LinkedList<E> result = new LinkedList<>();
    for (E elem : tosort)  einfuegen(elem,result);
    return result;
}
```

H12-1 ***Warteschlange*** (6 Punkte; Alle `.java`-Dateien Ihrer Lösung abgeben)
Schreiben Sie Klassen, welches das nachfolgende Interface `Warteschlange` Implementieren. Sie dürfen dabei keine Datenstrukturen aus der Standardbibliothek verwenden! Sie können sich gerne an den Klassen aus Aufgabe A12-1 orientieren, geben Sie aber alle verwendeten Klassen ab!

```
public interface Warteschlange<A> {
  /**
  * @param a Element welches hinten angestellt wird
  * @return Ob das Elment hinten angestellt werden konnte
  */
  public boolean push(A a);

  /**
  * @return Element, welches am längsten gewartet hat
  */
  public A pop();

  /**
  * @return true, falls die Liste leer ist.
  */
  boolean isEmpty();

  /**
  * @return Anzahl Elemente in der Liste
  */
  int size();
}
```

a) Schreiben Sie eine Klasse `MeineSchlangeList`, welche als interne Datenstruktur eine Art verkettete Liste verwendet.

b) Schreiben Sie eine Klasse `MeineSchlangeArray`, welche als interne Datenstruktur ein klassisches Array (keine ArrayList) verwendet.

In diesem Fall hat die Warteschlange eine feste Größe von 7 und kann nur noch neue Elemente aufnehmen, wenn vorher mit `pop` Elemente abgefragt werden.[2] Um dies zu realisieren muss sich Ihre Datenstruktur jeweils einen Index zum Einfügen und zum Abfragen merken.

LÖSUNGSVORSCHLAG:

[2] Insbesondere im Embedded Bereich werden gerne solche „Ringpuffer“ fester Größe eingesetzt, da diese ohne zusätzlichen Speicherverbrauch auskommen.

Hinweis: In der Standardbibliothek gibt es auch das umfassendere Interface Queue, dort haben die Methoden die sinnvolleren Namen `add` und `remove`.

Die Namen der Methoden `push`/`pop` sind hier leider irreführend, da diese traditionell für die Datenstruktur „Stack“ verwendet werden, welche Last-In-First-Out (LIFO) Funktionalität bietet (wie eine einfach verkettete Liste).

Für eine Warteschlange mit First-In-First-Out (FIFO) sollten die Methoden besser `enqueue`/`dequeue` oder eben wie im Queue Interface heissen.

Lösung Für die erste Teilaufgabe können wir die Vorlagen aus A12-1 gut wiederverwenden, entweder man kopiert `MyList` oder man nutzt Vererbung:

```
public class MeinSchlangeList<E> extends MyList<E> implements Warteschlange<E> {
  @Override
  public boolean push(E e) { super.addFirst(e); return true; }
  @Override
  public E pop() { return removeLast(); }
}
```

In beiden Fällen muss man jedoch in `MyList` noch die Methode `removeLast` implementieren, welche direkten Zugriff auf die Kettenglieder benötigt:

```
public E removeLast() {
  if (first==null) throw new NoSuchElementException();
  Link<E> pos  = first;
  Link<E> prev = null;
  while (pos.next != null) { prev= pos;  pos= pos.next; }
  E res = pos.data;
  if (prev == null) first = null;
  else          prev.next = null;
  return res;
}
```

Wir bemerken, dass die `pop` Operation eine lineare Laufzeit hat, was für eine Warteschlange einfach zu schlecht ist. Abhilfe schafft hier die Verwendung einer doppelt verketteten Liste, wie in der Vorlesung gezeigt.

Für die zweite Teilaufgabe ändern wir die Klasse `MyArray` leicht ab:

```
public class MeinSchlangeArray<E> implements Warteschlange<E> {
  E[] array;
  int size;   // Tatsächlich vorhandene Elemente
  int oldest; // ältester Index

  public MeinSchlangeArray(int maxsize) {
    // this.array = new E[size]; Leider nicht erlaubt in Java
    this.array = (E[]) new Object[maxsize]; // Typecast muss gut durchdacht sein!
    this.size = 0; this.oldest = 0;
  }

  @Override
  public boolean isEmpty() { return size == 0; } // unverändert
  @Override
  public int size() { return this.size; } // unverändert
  @Override
  public boolean push(E e) {
    if (size >= array.length) { return false; }
    int newPos = (oldest + size) % array.length; // erste leere Position im Array
    array[newPos] = e;
    size++;
    return true;
  }
  @Override
  public E pop() {
    if (size <= 0) { return null; }
    E res = array[oldest];
    array[oldest] = null;  // verhindert Memory Leak
    oldest = (oldest + 1) % array.length;
    size--;
    return res;
} }
```

Abgabe: Lösungen zu den Hausaufgaben können bis Sonntag, den 28.1.18, mit UniWorX nur als `.zip` abgegeben werden. Aufgrund des Klausurbonus müssen die Hausaufgaben von Ihnen alleine gelöst werden. Abschreiben bei den Hausaufgaben gilt als Betrug und kann zum Ausschluss von der Klausur zur Vorlesung führen. Bitte beachten Sie auch die Hinweise zum Übungsbetrieb auf der Vorlesungshomepage (`www.tcs.ifi.lmu.de/lehre/ws-2017-18/eip/`).