Prof. Martin Hofmann, PhD
Dr. Steffen Jost

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

Lösungsvorschlag zur 13. Übung zur Vorlesung
Einführung in die Programmierung

Aktuelle Hinweise:

- Dies ist das letzte reguläre Übungsblatt dieser Veranstaltung. In der nächsten Übungsstunde werden wir eine Probeklausur besprechen, welche etwas früher als üblich herausgegeben wird, damit Sie die Probeklausur vor der Übung zur Probe bearbeiten können.
- Die letzten beiden Vorlesungen am 6.2. und 8.2. sind derzeit als Fragestunde und zur Wiederholung eingeplant. Bitte schicken Sie uns zur Vorbereitung Ihre Fragen und Themen-Vorschläge zur Wiederholung/Vertiefung per eMail an: `jost@tcs.ifi.lmu.de`

A13-1 ***Doppelt verkettete Listen I*** In der Vorlesung am 25.01.18 wurde eine Implementierung von doppelt verketteten Listen vorgestellt, welche Sie von der Vorlesungshomepage herunterladen können.

a)
```
MyList2<String> ls = new MyList2<>();
ls.addLast("O");  ls.addLast("V");
ls.addFirst("T"); ls.addLast("I");
iter = ls.iterator();
iter.add("L"); iter.next();
iter.remove(); iter.next();
iter.set("U"); iter.next();
iter.add("E");
ls.addLast("P");
```

Schnell mit Papier & Bleistift ausrechnen: Welchen Inhalt hat die liste `ls`?

LÖSUNGSVORSCHLAG:

`LUVEIP`

b) Erweitern Sie Klasse `MyList2` um eine Methode `public void reverse()`, welche die Reihenfolge aller Elemente in der Liste umdreht. Verwenden Sie dazu keinen Iterator!

LÖSUNGSVORSCHLAG:

```
public void reverse() {
  Link<E> prev = null;
  Link<E> pos  = first;
  Link<E> next = null;
  first = last;
  last = pos;
  while (pos != null) {
    next = pos.getNext();
    pos.setNext(prev);
    pos.setPrev(next);
    prev = pos;
    pos = next;
} }
```

Man kann streiten, ob folgende Alternative lesbarer ist oder nicht, aber es spart etwas Stack-Speicher:

```
public void reverse2() {
  Link<E> pos  = first;
  first = last;
  last = pos;
  while (pos != null) {
    Link<E> swap = pos.getNext();
    pos.setNext(pos.getPrev());
    pos.setPrev(swap);
    pos = swap;
} }
```

c) Dieser Test offenbart, dass Dr. Jost einen Fehler bei der Implementierung gemacht hat:

```
MyList2<Integer> l1 = new MyList2<>();
l1.addLast(3);
l1.addFirst(2);
l1.addLast(4);
l1.addFirst(1);
ListIterator<Integer> it = l1.iterator();
while(it.hasPrevious()){ System.out.print(it.previous() + " "); }
while(it.hasNext()    ){ System.out.print(it.next()     + " "); }
while(it.hasPrevious()){ System.out.print(it.previous() + " "); }
while(it.hasNext()    ){ System.out.print(it.next()     + " "); }
```

Helfen Sie Dr. Jost! Finden, beschreiben und beheben Sie den Fehler fix!

LÖSUNGSVORSCHLAG:

Wenn man den Code ausführt, dann terminiert vorletzte Schleife nicht!

Der Fehler liegt in der Methode `previous` der statischen inneren Klasse `MyList2.MyIterator`: In der regulären Situation, dass der `positionsZeiger` nicht am Ende der Liste steht, wird dieser gar nicht verschoben! Damit liefert `previous` immer wieder das gleiche Element und es kommt zu einer Endlosschleife. Weiterhin wird auch noch `zuletztGelesenesPos` nicht verändert.

Hier ein Lösungsvorschlag:

```
@Override
public E previous() {
  if (positionsZeiger == null) { // Wir sind total am Ende!
    if (letztGelesenesPos == null) { return null;
    } else { positionsZeiger = letztGelesenesPos; }
  } else {
    positionsZeiger = positionsZeiger.getPrev();
    letztGelesenesPos = positionsZeiger;
  }
  index--;
  if (letztGelesenesPos == null) { return null; }
  else { return letztGelesenesPos.getData(); }
}
```

A13-2 ***Effiziente Datenstrukturen*** Betrachten Sie folgende Anwendungsszenarien. Würden Sie jeweils dazu eher ein Array oder eine verkettete Liste verwenden? Begründen Sie Ihre Antwort jeweils kurz mit 2–5 Sätzen!

a) Angenommen Sie müssten eine Applikation zur Verwaltung von Besprechungsterminen schreiben. Mit welcher Datenstruktur verwalten Sie die Objekte, welche die Besprechungen repräsentieren?

LÖSUNGSVORSCHLAG:

Hier bittet sich eine verkettete Liste an, da neue Termine ungeordnet eintreffen und an alle möglichen Positionen einsortiert werden müssen. Bei einem Array müsste man dann jedes Mal alle folgenden Termine umkopieren, während das Einfügen in eine verkettete Liste effizient möglich ist.

Der Zugriff auf die vorhandenen Termine erfolgt dagegen meist in einer geordneten Reihenfolge vom frühesten zum spätesten Termin.

Anmerkung: Eine weitere Alternative wäre auch ein Binärer Suchbaum oder eine endliche Abbildung von Datum zur Terminbeschreibung, falls öfters Terminen an bestimmten Zeitpunkt gesucht wird.

b) Angenommen Sie müssten ein Applikation für eine Telefonzentrale eines großen Konzerns schreiben, mehrmals pro Sekunde zu jeder Telefonnummern den Namen des Teilnehmers ermitteln muss. An die Hardware können maximal 70000 Telefonapparate angeschlossen werden.

LÖSUNGSVORSCHLAG:

Der klassische Fall für ein klassisches Array: Viele wahlfreie Index-Zugriffe. Da die Hardware eine maximale Anzahl an Apparaten unterstützt, reicht es aus, wenn die Software ein Array in genau dieser Größe anlegt und verwendet (allerdings sollte die Zahl ohne Kompilation konfigurierbar sein, z.B. per Kommandozeilen-Argument, damit die Software nach einem Hardware-Update noch einsetzbar bleibt).

Ein Umsortieren der Einträge ist damit auch nicht notwendig. Momentan nicht angeschlossene Apparate können mit `null` modelliert werden.

Anmerkung: Wenn an die Größe nicht eingrenzen kann, bietet sich aber auch hier eine endliche Abbildung an.

c) Sie möchten eine Programmiersprache mit expliziter Speicherverwaltung schreiben. Der Programmierer kann Speicherblöcke in beliebiger Größe anfordern, nach belieben deren Bits verändern (für die Aufgabe irrelevant) und anschließend zur Wiederverwendung freigeben.

LÖSUNGSVORSCHLAG:

Dieses Beispiel einer primitiven Heap-Verwaltung wird üblicherweise mit verketteten Listen realisiert, da auch hier die Reihenfolge der Allokierung und Freigabe nicht vorhersehbar ist und kein sinnvoller Index existiert.

Jedes Kettenglied merkt sich als Datum Start- und Endadresse eines freien zusammenhängenden Blocks. Diese Liste wir gelegentlich sortiert und benachbarte Blöcke werden zu größeren verschmolzen. Bei Allokierung werden größere Blöcke gemäß der angeforderten Größe aufgeteilt falls nötig.

Anmerkung: Je nach Anforderung könnte man auch hier eine Datenstruktur einsetzen, welche die Liste sortiert hält um das Verschmelzen zu beschleunigen, oder um Aufteilungen zu verhindern (z.B. HashMap nach Größe, darin dann verkettete Listen).

A13-3 *Knobelaufgabe** EiP-Teilnehmerin Birke Ballantz hat die Aufgabe A12-2 „Einfügen in Verkettete Listen“ einfach gut gefallen. Als sie dann in der Vorlesung doppelt verkettete Listen kennenlernte, hatte sie eine Idee und implementierte eine Datenstruktur, welche das sortierte Einfügen effizienter erledigen kann! Birke's Datenstruktur hat zwei Verweise pro Kettenglied, so wie eine doppelt verkettete Liste auch. Allerdings merkt sich die Klasse beim Erstellen einen `Comparator` und kann alle enthaltenen Elemente damit vergleichen. Beim Einfügen werden kleinere Elemente mit dem Verweisen nach vorne eingefügt und größere Elemente mit den Verweisen nach hinten!

Birke's Datenstruktur mit Klassennamen `Birke` sollte diesem Übungsblatt beliegen.

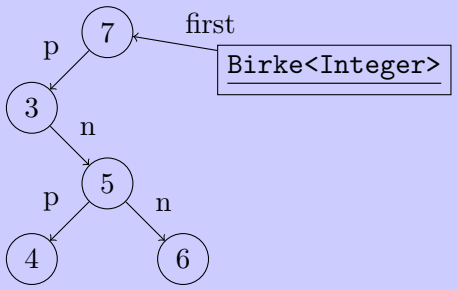
a) Was gibt das folgendes Hauptprogramm aus? Skizzieren Sie auch grob das Speicherbild!

```
Comparator<Integer> cmp = Comparator.naturalOrder();
Birke<Integer> birke = new Birke<>(cmp);
birke.insert(7);
birke.insert(3);
birke.insert(5);
birke.insert(4);
birke.insert(6);
for(Integer i : birke) { System.out.print(i+", "); }
```

LÖSUNGSVORSCHLAG:

Die Ausgabe ist `7, 3, 5, 4, 6`. Um dies zu erkennen, muss man natürlich die `insert`-Methode und den Iterator genauer angeschaut haben, oder es einfach mal ausführen.

Zur Vereinfachung der Darstellung zeichnen wir anstatt wie üblich je eine Box für `Link`- und `Integer`-Objekt in unsere Skizze nur einen runder Kreis mit dem Wert des Integers ein. Die `next`-Verweise haben wir mit n markiert, die `prev`-Verweise mit p:

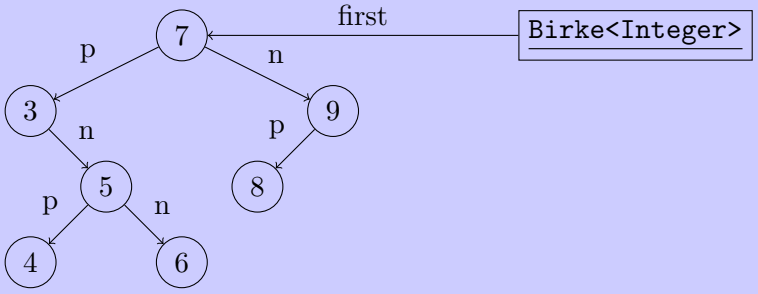


b) Was ändert sich an Ausgabe und Speicherskizze, wenn vor der Schleife noch die Befehle `birke.insert(9);` und `birke.insert(8);` einfügt werden?

LÖSUNGSVORSCHLAG:

Ausgabe: `7, 3, 9, 5, 8, 4, 6`

Wir sehen jetzt also, das die Reihenfolge der Ausgabe nicht der Einfüge-Reihenfolge entsprechen muss!



c) Birke hat tatsächlich eine nützliche Datenstruktur gefunden; doch leider ist diese als Liste denkbar schlecht geeignet. Diskutieren Sie, warum Birke's Datenstruktur keine gute Liste ist!

LÖSUNGSVORSCHLAG:

Die Datenstruktur implementiert nicht das `List`-Interface!

Das Problem dabei ist, dass fehlende Operationen wie etwa `addFirst` oder `addLast` gar nicht sinnvoll hinzugefügt werden können: Eine Liste ist eine vom Benutzer geordnete Folge von Werten. In Birke's Datenstruktur ist die Reihenfolge jedoch durch die Ordnung der Elemente fest vorgegeben. Um eine Damit kann man gar nicht eine Liste wie etwa $[4, 7, 6, 3]$ speichern.

d) Implementieren Sie die Methode `public Iterator<E> sortedIterator()` welche mit `//TODO Ihre Aufgabe!` markiert ist. Diese Methode soll einen Iterator zurückliefern, mit dem man alle Elemente in Birke's Datenstruktur vom kleinsten bis zum größten durchgehen kann.

Die Anweisung `while(iter.hasNext()) System.out.print(iter.next()+", ");` sollte dann für das Beispiel aus der ersten Teilaufgabe `3, 4, 5, 6, 7,` ausgegeben.

Sie können dem Muster des vorhanden Iterators folgen. Eventuell finden Sie die Aufgabe aber leichter, wenn Sie nach dem Muster der Methode `size` vorgehen, wobei Sie anstatt einem `int` eine frische `LinkedList<E>` einsetzen, deren Iterator Sie dann einfach zurückgeben.

LÖSUNGSVORSCHLAG:

Lösung mit geklautem Iterator (Okay, solange wir mit dem Iterator nichts verändern wollen, was hier ja auch keinen Sinn macht, wie wir in der vorherigen Teilaufgabe beobachtet haben.):

```
public Iterator<E> sortedIterator() {
  List<E> l = new LinkedList<> ();
  addOrdered(l,first);
  return l.iterator();
}
private static <E> void addOrdered(List<E> list, Link<E> link){
  if (link == null) return;
  addOrdered(list,link.prev);
  list.add(link.data);
  addOrdered(list,link.next);
}
```

Lösung mit Iterator:

```
private class SortedIterator implements Iterator<E> {
  LinkedList<Link<E>> position;
  // Klasseninvariante: Elemente in position sind nie null!

  public SortedIterator(Link<E> positionsZeiger) {
    this.position = new LinkedList<>(); addPrevs(positionsZeiger);
  }
  @Override
  public boolean hasNext() { return !position.isEmpty(); }
  @Override
  public E next() {
    if (position.isEmpty()) { throw new NoSuchElementException(); }
    Link<E> l = position.remove();
    E ergebnis = l.data;
    addPrevs(l.next);
    return ergebnis;
  }
  private void addPrevs(Link<E> l){
    if (l != null) { position.addFirst(l);
                     addPrevs(l.prev);
} }                }
```

e) Diskutieren Sie, was diese Datenstruktur gut kann! Betrachten Sie zuerst die Methoden `contains` und `insert`: Was ist deren Laufzeitkomplexität?

LÖSUNGSVORSCHLAG:

Ähnlich wie bei der binären Suche verwerfen die Methoden `insert` und `contains` in jedem Schritt einen ganzen Teil der Datenstruktur. Die Laufzeitkomplexität entspricht nicht der Anzahl der gespeicherten Elemente, sondern der Länge der längsten Kette welche man ausgehend von `first` mit `prev` oder `next` bis zu einem Nullpointer bilden kann.

Bemerkung: In der Vorlesung am 30.02.18 werden wir Datenstruktur mit effizienten Operationen zum Einfügen und zum Testen auf enthalten-sein als „Mengen“ bezeichnen.

Leider kann dies dennoch schlecht sein, wie man an folgendem Beispiel sieht, welches im Speicher eine lange Kette mit allen Elementen bildet, d.h. um das größte Element zu finden, müssen wir alle Elemente der Datenstruktur vergleichen:

```
Comparator<Integer> cmp = Comparator.naturalOrder();
Birke<Integer> birke = new Birke<>(cmp);
birke.insert(1); birke.insert(2); birke.insert(3); birke.insert(4);
```

Um wirklich effizient Mengen zu implementieren, fehlt Birke **Ballantz**'s Datenstruktur nur noch eins: die **Balance**!

Ausblick
Wie wir später in der Vorlesung lernen werden **handelt es sich bei `Birke`-Objekten um Bäume**(!), genauer um unbalancierte binäre Suchbäume.

Üblicherweise bezeichnet man die `Link`-Glieder dann als *Knoten* und deren gespeicherte Pointer anstatt mit `prev` und `next` dann mit `left` und `right`. In unseren Speicherskizzen oben haben wir das auch schon grafisch so angeordnet – doch solche Konventionen konnten Sie ja nicht erraten!

Diese Aufgabe sollte lediglich einen ersten Ausblick auf Themen bieten, welche in den nächsten Semestern noch intensiv behandelt werden. Alle dazu notwendigen Java-Konstrukte sollten Ihnen nun bekannt sein, doch wie sich diese dann zu Algorithmen wie etwa „Breitensuche auf Binärbäumen“ zusammenpuzzeln lassen, ist natürlich eine ganz andere Frage!

H13-1 ***Generisches Mapping*** (5 Punkte; Abgabe: `Multipliziere.java`, `Map.java`)
Gegeben ist folgendes Interface: `public interface Function<X,Y> { public Y apply(X x); }`

a) Schreiben Sie eine Klasse `Multipliziere`, welche das Interface `Function<Integer,Integer>` implementiert. Die Methode `apply` soll ein gegebenes Argument mit einem festen Faktor multiplizieren. *Beispiel:*

```
Multipliziere maldrei = new Multipliziere(3);
Multipliziere malvier = new Multipliziere(4);
System.out.print(maldrei.apply(malvier.apply(5))); // druckt 60
```

LÖSUNGSVORSCHLAG:

```
public class Multipliziere implements Function<Integer,Integer> {
  private final Integer faktor;

  public Multipliziere(Integer faktor) { this.faktor = faktor; }
  @Override
  public Integer apply(Integer x) { return x * faktor;  }
}
```

b) Schreiben Sie eine statische, generische Methode `map`, welche zwei Argumente bekommt: Das erste Argument ist Objekt, welche das Interface `Function<X,Y>` implementiert; das zweite Argumente hat den Typ `LinkedList<X>`. Als Ergebnis soll die Methode eine verkette Liste mit Elementen aus `Y` zurückgeben. Jedes Element der Ergebnisliste wurde durch Anwendung des ersten Arguments auf ein Element der Argumentliste berechnet; die Reihenfolge entspricht der Argumentliste. *Beispiel:*

```
Function<Integer,Boolean> even = new Even();
System.out.println(map(even,xs));
```

Für die Liste `xs = [72, 9, 21, 174, 6, 93]` wird in diesem Beispiel `[true, false, false, true, true, false]` gedruckt, wobei die Klasse `Even` eine Funktion implementiert, welche `true` nur für gerade Argumente zurück gibt.

LÖSUNGSVORSCHLAG:

```
static <X,Y> LinkedList<Y> map(Function<X,Y> fun, LinkedList<X> xs){
  LinkedList<Y> result = new LinkedList<>();
  for (X x : xs) result.addLast(fun.apply(x));
  return result;
}
```

Hinweis für Interessierte: Dank dem seit Java 1.8 vorhandenen Stream-Interface kann man solche Dinge nun auch ohne langwierige Definitionen ad hoc durchführen:

```
List<Boolean> result = xs.stream()
                         .map(x -> 3*x)
                         .map(x -> x%2==0)
                         .collect(Collectors.toList());
System.out.println(result.toString());
System.out.println(xs.stream().map(x->4*x).collect(Collectors.toList()));
xs.stream().map(x -> x%2==0).forEach(y -> System.out.print(y+", "));
```

Abgabe: Lösungen zu den Hausaufgaben können bis Sonntag, den 4.2.18, 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/`).