Institut für Informatik
Heinrich-Heine-Universität Düsseldorf
Prof. Dr. Alice McHardy
Janine Haas
Sebastian Konietzny
Aaron Weimann

WS 2013/2014
21.01.2014
Übungsblatt 10

Einführung in die Informatik I
Übungen zur Vorlesung
Musterlösung

Dieses Übungsblatt wird nicht mehr von Ihren Tutoren korrigiert. Für Ihre Zulassung zählen nur die Übungsblätter bis einschließlich Blatt 9. Weitere Übungsblätter dienen nur Ihrer Klausurvorbereitung. Die Musterlösung für dieses Blatt wird am 28.1.2014 veröffentlicht. Falls Sie Fragen haben, wenden Sie sich bitte an Ihren Übungsleiter oder vereinbaren Sie einen Termin bei Janine Haas (haas@cs.uni-duesseldorf.de, Büro: 25.02.O1.31).

10.1 Einfach verkettete Listen

Gegeben Sei folgendes Gerüst:

```
public class MyLinkedList{
  private Node head;
  private void invert(){
    //TODO
  }
  public static void main(String[] args){
    //TODO
  }
}
```

a) Implementieren Sie die Methode `invert`, durch welche die Reihenfolge der Elemente einer einfach verketteten Liste, die bei `head` beginnt, umgekehrt wird. Verwenden Sie die Klasse `Node` aus der Vorlesung.

b) Ergänzen Sie die `main`-Funktion so, dass Sie eine einfach verkettete Liste mit zehn Knoten erstellen. Diese Knoten sollen die Werte eins bis zehn (in dieser Reihenfolge eingefügt) enthalten und ein neuer Knoten soll immer am Anfang der Liste eingefügt werden. Geben Sie die Listenelemente anschließend aus. Wenden Sie nun Ihre Methode `invert` auf die Liste an und geben Sie die Liste erneut aus.
Hinweis: Es ist unter Umständen sinnvoll Teile dieser Aufgabe in einzelne Funktionen auszulagern.

Musterlösung

```
public class MyLinkedList{
  private Node head;

  private void invert(){
/*Zwei Nodes um eine temporäre Referenz
* bzw. den noch nicht invertierten Teil der Liste zu speichern
* rem speichert dann den Anfang des noch nicht invertierten Teils
*/
    Node tmp, rem;
    //tmp zeigt auf das erste Element der noch nicht invertierten Liste
    tmp = head;
    //head ist zunächst eine leere Liste
    head=null;
    while (tmp!=null) {
  /*Solange noch nicht die ganze Liste invertiert ist:
  * Füge das erste Element aus der noch nicht invertierten
  * Liste (tmp) vor dem Anfang der invertierten Liste (head) ein.
  * Damit head weiterhin auf den Anfang der Liste zeigt muss head
  * angepasst werden. Wir müssen eine Referenz auf den Rest der noch
  * nicht invertierten Liste zwischenspeichern (rem).
  * Zum Schluss soll tmp für den nächsten Schleifendurchlauf wieder auf
  * den Anfang der restlichen Liste zeigen.
  * Hinweis: Führen Sie dies mit Bleistift und Papier durch um es zu
  * verstehen!
  */
      rem = tmp.getNext();
      tmp.setNext(head);
      head = tmp;
      tmp=rem;
    }
  }
  private void print(){
    Node tmp;
    tmp = head;
    while (tmp != null){
      System.out.print(tmp.getElement() + " ");
      tmp = tmp.getNext();
    }
    System.out.print("\n");
  }

  private void insert(int val){
    Node node = new Node(val, head);
    head = node;
  }
```

```
  public static void main(String[] args){
    MyLinkedList list = new MyLinkedList();
    for (int i = 1; i < 11; i++){
      list.insert(i);
    }
    list.print();
    list.invert();
    list.print();
  }
}
```

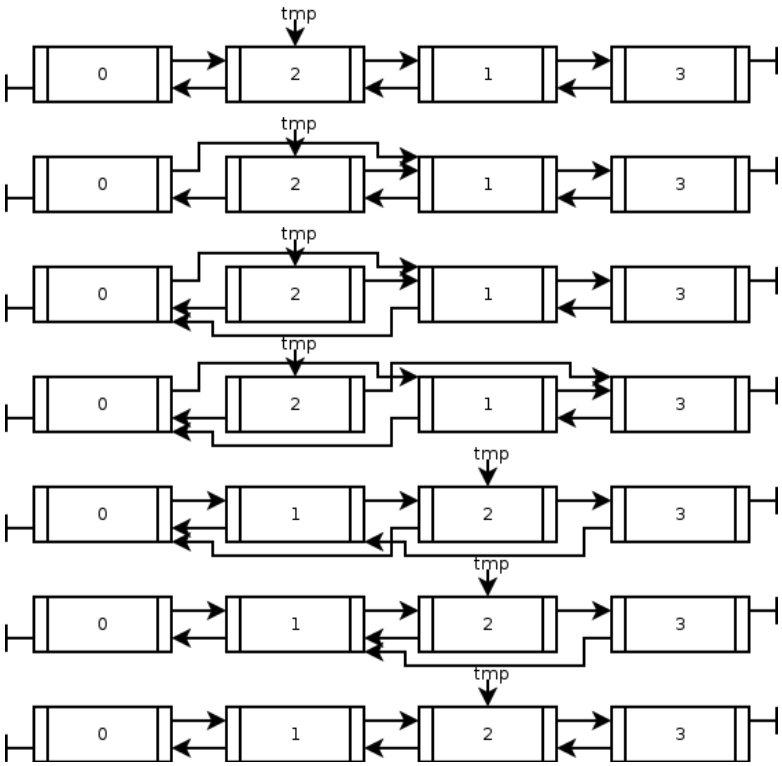
10.2 Doppelt verkettete Listen

a) Schreiben Sie einen Algorithmus, um eine doppelt verkettete Liste mittels Bubblesort zu sortieren.

b) In der Vorlesung haben Sie gelernt, dass binäre Suche (bei geeigneter) Implementierung in $O(n \log(n))$ läuft. Weshalb könnten Sie diese Laufzeit nicht ohne weiteres annehmen, wenn Sie binäre Suche auf einer doppelt verketteten Liste (anstelle eines Arrays) implementieren?

Musterlösung

a)
```
//Gegeben Sei eine doppelt verkettete Liste, mit dem ersten Element first
//und dem letzten Element last
Boolean sorted = false;
DoubleNode tmp;
while (!sorted){
  sorted = true;
  tmp = first;
  while(tmp != last){
    if(tmp.getElement() > tmp.getNext().getElement()){
      tmp.getPrevious().setNext(tmp.getNext());
      tmp.getNext().setPrevious(tmp.getPrevious());
      tmp.setNext(tmp.getNext().getNext());
      tmp.getNext().getPrevious().setNext(tmp);
      tmp.setPrevious(tmp.getNext().getPrevious());
      tmp.getNext().setPrevious(tmp);
      sorted = false;
    }
    tmp = tmp.getNext();
  }
}
```



b) Für die Laufzeitabschätzung der binären Suche ist wichtig, dass Sie in konstanter Zeit auf das mittlere Element der zu durchsuchenden Liste zugreifen können. Bei Doppelt verketteten Listen können Sie genau das aber nicht. Eine Möglichkeit das mittlere Element zu finden ist, die Liste von vorne und hinten zu durchlaufen - beim mittleren Element treffen sich die Zeiger. Durch dieses Suchen des mittleren Elements benötigen Sie aber zusätzliche Zeit und verlieren daher die Vorteile der binären Suche.

10.3 Stacks und Queues

a) Führen Sie folgende Operationen auf einem zu Beginn leeren Stack aus und geben Sie an, wie der Stack nach der Operation jeweils aussieht: `push(1)`, `push(5)`, `push(3)`, `pop()`, `top()`, `push(9)`, `push(9)`, `top()`, `pop()`, `pop()`

b) Führen Sie folgende Operationen auf einer zu Beginn leeren Queue aus und geben Sie an, wie die Queue nach der Operation jeweils aussieht: `enqueue(1)`, `enqueue(5)`, `enqueue(3)`, `dequeue()`, `dequeue()`, `enqueue(9)`, `enqueue(9)`, `isEmpty()`, `dequeue()`, `dequeue()`

Musterlösung

a)

Operation	Stack	Bemerkung
push(1)	1	
push(5)	5 1	
push(3)	3 5 1	
pop()	5 1	3 wird zurückgegeben und entfernt
top()	5 1	5 wird zurückgegeben
push(9)	9 5 1	
push(9)	9 9 5 1	
top()	9 9 5 1	9 wird zurückgegeben
pop()	9 5 1	9 wird zurückgegeben und entfernt
pop()	5 1	5 wird zurückgegegben und entfernt

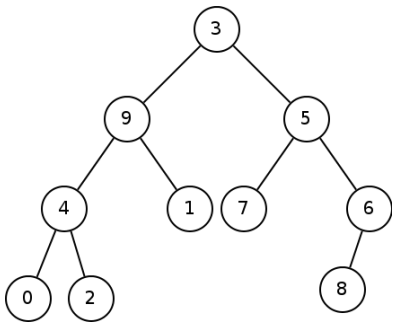
b)

Operation	Queue	Bemerkung
enqueue(1)	1	
enqueue(5)	1, 5	
enqueue(3)	1, 5, 3	
dequeue()	5, 3	
dequeue()	3	
enqueue(9)	3, 9	
enqueue(9)	3, 9, 9	
isEmpty()	3, 9, 9	ändert nichts an der Queue, gibt `false` zurück
dequeue()	9, 9	
dequeue()	9	

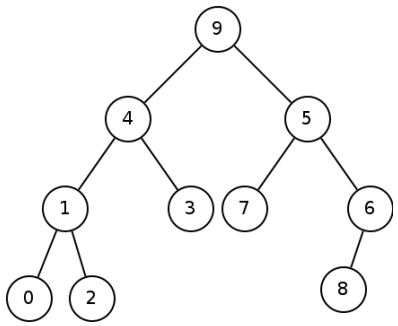
10.4 Binärbäume

a) Zu welchem der folgenden Binärbäume passt sowohl die Infix-Folge 0 1 2 4 3 9 7 5 8 6 als auch die Postfix-Folge 0 2 4 1 7 9 8 6 5 3

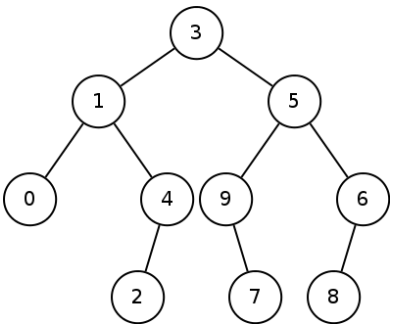
1)



2)



3)



b) Geben Sie die Präfix Reihenfolge des Baums aus a) an. Falls Sie Aufgabenteil a) nicht gelöst haben, geben Sie die Präfix-Reihenfolge des ersten Baums an.

Musterlösung

a) Baum 3 ist der passende Baum.

b) Die Präfix Reihenfolge von Baum 3 ist 3, 1, 0, 4, 2, 5, 9, 7, 6, 8.
Die Präfix Reihenfolge des ersten Baums ist 3, 9, 4, 0, 2, 1, 5, 7, 6, 8.

10.5 Suchbäume

a) Erweitern Sie die Klasse `SearchTreeNode` aus der Vorlesung um eine iterative Funktion `min`, die das kleinste Element des (Teil)baums, dessen Wurzel angesprochen wird, zurückgibt. *Hinweis: Es soll der Inhalt des Knotens, nicht der Knoten selber zurückgegeben werden.*

b) Erweitern Sie die Klasse `SerachTreeNode` aus der Vorlesung um eine rekursive Funktion `max`, die das größte Element des (Teil)baums, dessem Wurzel angesprochen wird, zurückgibt.

c) Erweitern Sie die Klasse `SearchTree` aus der Vorlesung, um Funktionen `min` und `max`, die das kleinste bzw. größte Element des gesamten Baums zurück geben.

Musterlösung

a)
```
public Comparable min(){
  SearchTreeNode tmp = left;
  if(left == null) return this.element;

  while (tmp.left != null){
    tmp = tmp.left;
  }
  return tmp.element;
}
```

b)
```
public Comparable max(){
  if (right == null) return this.element;
  return right.max();
}
```

c)
```
  public Comparable min(){
    return root.min();
  }
  public Comparable max(){
    return root.max();
  }
```