Semesterendprüfung ADS FS 2011

IT

24.6.11

Vorname	Name	Klasse	Punkte	Note
		IT10		

90 Minuten, 59 Punkte

Regeln

- erlaubt: Zusammenfassung auf max. 8 A4-Seiten
- nicht erlaubt: alte Prüfungen (sind keine Zusammenfassung des Unterrichtsstoffes)
- Prüfungsblätter bitte zusammengeheftet lassen
- Rot ist Korrekturfarbe, bitte nicht verwenden!

1 Bäume, Iterator [10 Pkte]

Die Klasse *BinaryTree* enthält als innere Klasse eine Klasse *BinaryTreeIterator*, die einen Iteraor für einen Binärbaum implementiert, wobei die Elemente in der Reihenfolge **levelorder** (entsprechend dem Levelorder-Traversal) zurückgegeben werden.

Implementieren Sie die fett markierten Methoden.

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.Queue;

class BinaryTree<T> {

      private T element;
      private BinaryTree<T> left;
      private BinaryTree<T> right;

      class BinaryTreeIterator<T> implements Iterator<T> {
                  // Hier muessen Sie wahrscheinlich noch etwas
                  // deklarieren


            ........................................

            ........................................

            public BinaryTreeIterator(BinaryTree<T> root) {

                  ..................................

                  ..................................
            }

            boolean hasNext() {

                  ..................................

                  ..................................
            }

            T next() {

                  ..................................

                  ..................................

                  ..................................

                  ..................................

                  ..................................
            }

      }     // end class BinaryTreeIterator
```

```
	public Iterator<T> getIterator() {
		return new BinaryTreeIterator<T>(this);
	}

	// weitere Methoden, nicht von Interesse
}
```

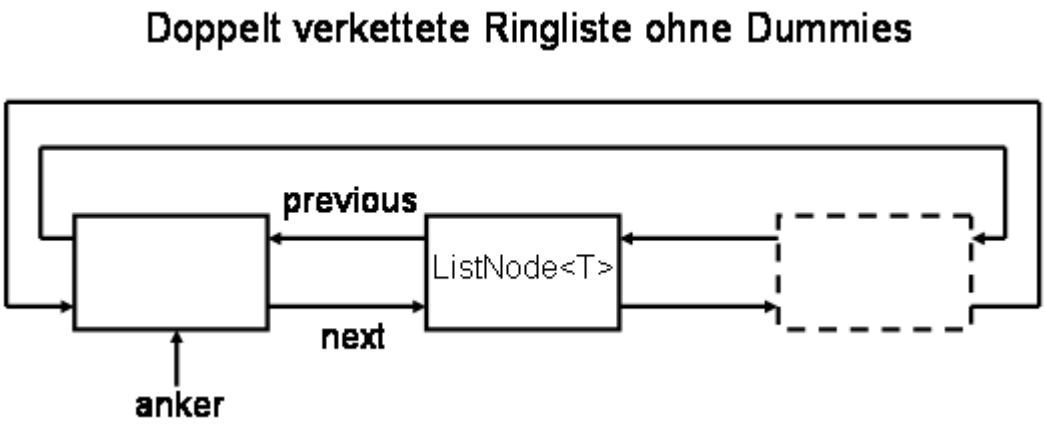
Was müssten Sie in der Klasse BinaryTree ändern, falls BinaryTreeIterator nicht eine innere Klasse von BinaryTree wäre?

.....

.....

2 Doppelt verkettete Ringliste ohne Dummies [10 Pkte]

Gegeben ist folgende Liste:



Der ListNode anker enthält das erste Datenelement.

Aufgabe

Schreiben Sie die Methode `public T removeLast()`. Der Rückgabewert ist der Wert des letzten Elements bzw. null, falls die Liste leer ist.

Hinweis

Beachten Sie die Codevorlage im Anhang und halten Sie sich strikte daran.

2 Lösung

3 Sortieren [10 Pkte]

Gegeben sei eine Klasse `Person` (Fragment):

```
public class Person {
      private String name;
      private String wohnOrt;
      ...
      public String getName() { return name; }

      public String getWohnOrt() { return wohnOrt; }
      ...
}
```

Instanzen dieser Klasse sollen (wie in einem gedruckten Telefonbuch) nach Wohnort und innerhalb des Wohnorts nach Namen sortiert werden. Zu diesem Zweck werden zwei Klassen `PersonNameComparator` und `PersonWohnOrtComparator` geschrieben, die beide das Interface `java.util.Comparator<Person>` implementieren. Wie die Klassennamen sagen, definiert PersonNameComparator eine Sortierreihenfolge der Personen aufgrund des Namens und PersonWohnOrtComparator eine aufgrund des Wohnortes. Die Namen wie auch die Wohnorte sollen einfachheitshalber gemäss der natürlichen Reihenfolge für Strings sortiert werden, d.h. wie es die Methode `public int compareTo(...)` der Klasse String definiert.

3a

Schreiben Sie die Klasse PersonNameComparator.

3b

Wie gehen Sie vor, falls Sie nur diese beiden Komparatoren benützen dürfen, um Instanzen der Klasse Person (z.B. in einer Liste) wie gefordert nach Wohnort und innerhalb des Wohnorts nach Namen zu sortieren?

3c

Welchen Sortieralgorithmus benützen Sie, falls die Anzahl Personen-Datensätze gross (z.B. über eine Million) ist?

3d

Die Klasse `Person` wird derart erweitert, dass sie das Interface `java.lang.Comparable<Person>` implementiert. Die dadurch definierte natürliche Reihenfolge der Personen soll die oben beschriebene Telefonbuch-Sortierung sein, also nach Wohnort und innerhalb des Wohnorts nach Namen. Schreiben Sie die für diese Erweiterung notwendige(n) Methode(n).

Lösung

3a

3b

3c

3d

4 Hashing [10 Pkte]

Die zehn nach der Einwohnerzahl grössten Städte der Schweiz werden in einer Hash-Tabelle der Grösse 13 abgelegt. Für die Kollisionsauflösung ist „Quadratic Probing“ festgelegt.
Die Städte und die für sie berechneten Hash-Codes sind:

Zürich	12
Genf	5
Basel	5
Bern	10
Lausanne	10
Winterthur	5
St. Gallen	12
Luzern	6
Lugano	3
Biel	6

4a

Wie sieht die Hash-Tabelle aus, nachdem alle zehn Städte in der Reihenfolge abnehmender Einwohnerzahlen wie oben eingefügt wurden?

4b

Warum kann in einer Hash-Tabelle ein zu löschender Eintrag nicht einfach gelöscht, d.h. in diesem Beispiel auf `null` gesetzt werden? (Stichworte oder einer bis zwei kurze Sätze)

Lösung

4a

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	

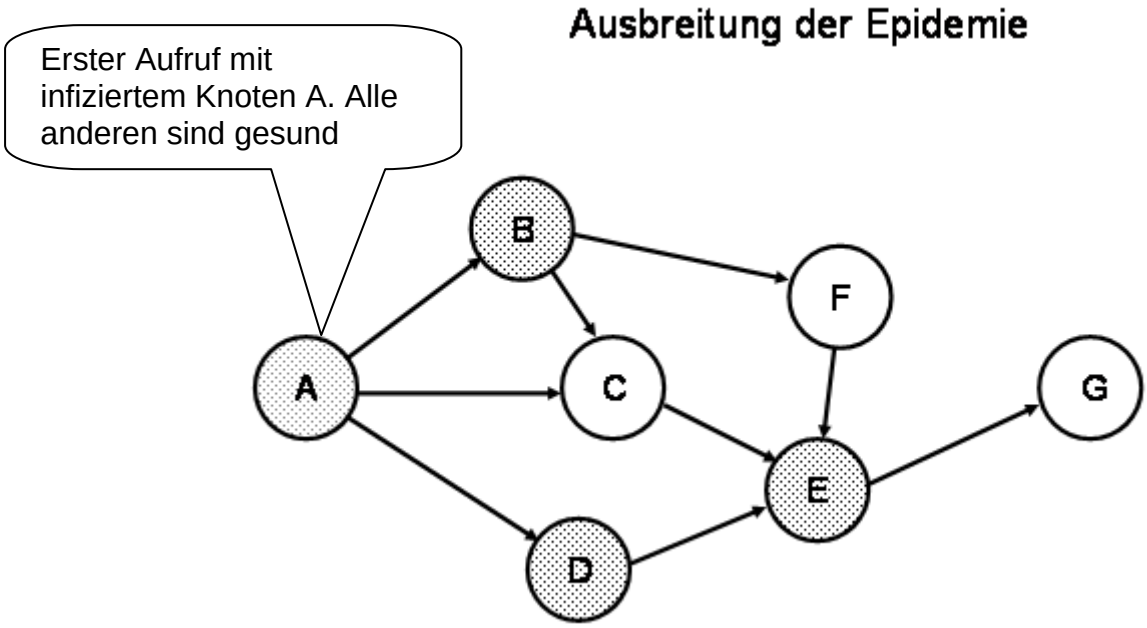
4b

---

5 Graphen: Ausbreitung einer Epidemie [10 Pkte]

Gegeben ist ein gerichteter Graph mit Adjazenzlisten. Ein Knoten (Vertex) ist zu Beginn von einer ansteckenden Krankheit befallen. Kranke Knoten stecken benachbarte Knoten entlang der Beziehungen (Edge) mit 50%-iger Wahrscheinlichkeit an. Gesunde Knoten machen kranke nicht wieder gesund. Bleibt ein Knoten gesund, so ist er immun, d.h. ein weiterer Knoten soll nicht mehr versuchen, ihn anzustecken.

Beispiel:



Zu Beginn wird nur Knoten A angesteckt. Er steckt seinerseits B und D an, C hingegen bleibt gesund. B darf nicht mehr versuchen, C anzustecken, und steckt F nicht an. D steckt E an. Hier endet die Ausbreitung der Krankheit.
Gesunde Knoten stecken keine anderen an („Gesundheit ist nicht ansteckend“).

Aufgabe

Schreiben Sie die Methode `public void spreadDisease(Vertex<T> vert)`. Sie berechnet die Ausbreitung der Epidemie.

Hinweis

Beachten Sie die Codevorlage im Anhang und halten Sie sich strikte daran.
`Math.random()` liefert eine `double`-Zufallszahl im Bereich [0..1[

Lösung 5

6 Gemischte Fragen [9 Pkte]

```
boolean richtig = kreuzWoNötig && !kreuzWoNichtNötig
```

Behauptung	wahr	falsch
Ein bestimmtes Element in einer sortierten doppelt verketteten sequentiellen (=linearen) Liste mit *n* Elementen zu finden, erfordert Aufwand O(log *n*).		
Ein bestimmtes Element in einem sortierten Array mit *n* Elementen zu finden, erfordert Aufwand O(log *n*).		
Ein bestimmtes Element korrekt in einen binären AVL-Baum mit *n* Elementen einzufügen, erfordert höchstens O(log *n*) Rotationen.		
Einen vollständig balancierten Binärbaum mit *n* Elementen Inorder zu traversieren, erfordert Aufwand O(log *n*).		
Ein bestimmtes Element in einer Hashtable mit *n* Elementen zu finden, erfordert nur dann Aufwand O(log *n*), falls keine Kollisionen auftreten.		
In einem Binärbaum, wie er typischerweise / klassischerweise implementiert wird, sind immer mehr als die Hälfte aller links- und rechts-Referenzen (Pointers) null**.**		
Es sei s ein Set von Zeichen. Die Aussage "*s enthält einmal das Zeichen 'a', einmal das Zeichen 'b' und zweimal das Zeichen 'c*'" ist unsinnig.		
Die statische Methode *sort* der Klasse *Collections* hat folgende Signatur: void sort(List<E> list, Comparator<? super E> comp); Dieser Methode kann eine List<Person> zusammen mit einem *Comparator<Student>* übergeben werden (*Student* sein abgeleitet von *Person*).		
Die Aussage, ein Sortieralgorithmus ist stabil, bedeutet, dass er auch für Elemente funktioniert, bei denen keine vollständige Ordnung definiert ist.		
Ein Rot-Schwarz-Baum ist ein Binärbaum.		
Graphen, deren Kanten Gewichte haben, sind nicht ausgeglichen		
Der Selection-Sort Algorithmus implementiert das Java Interface *Invariant*.		
Ein Suchalgorithmus wird „greedy“ genannt, wenn er die beste Lösung findet		
Der Aufwand, um ein Element in einer Hashtabelle (load factor < 0.1) mit n Elementen zu finden, ist O(1).		
Eine Liste verhält sich bei Breitensuche wie ein Stack		
BinarySearch kann bei Tiefensuche im Graphen angewendet werden, falls kein Backtracking auftritt		
Indirekte Rekursion liegt vor, wenn Methode a Methode b aufruft und Methode b wiederum Methode a		
Listen können als Bäume mit nur einem Nachfolger betrachtet werden		

Anhang

Code zu Ringliste

RingList

```
public class RingList<T>{
	private ListNode<T> anker;

	public T removeLast() {
		// to do
	}

	class ListNode<T>{
		T data;
		ListNode<T> previous, next;

		ListNode(T t) {data = t;}
	}
}
```

Code zu Epidemie-Graphen

```
public class Vertex<T>{
	private T data;
	private boolean infected = false, visited = false;
	private LinkedList<Edge> adj;

	public boolean isInfected()		{return infected;}

	public void setInfected(boolean b)	{infected = b;}

	public boolean isVisited()		{return visited;}

	public void setVisited()		{visited = true;}

	public T getData()			{return data;}

	public Iterator<Edges> getEdges()	{return adj.iterator();}
}

public class Edge<T> {
	private Vertex<T> target;

	public Vertex<T> getTarget()		{return target;}
}

public class EpiGraph<T>{
	private LinkedList<Vertex<T>>		vertices = new LinkedList();

	public Vertex<T> findVertex(T t) {
		// findet den Knoten und gibt ihn zurück
	}

	public void spreadDisease(Vertex<T> vert) {
		// to do
	}
}
```