Fakultät IV
Elektrotechnik/Informatik

Einführung in die Informatik - Vertiefung
Probeklausur

Sommersemester 2016

Hinweis: Diese Probeklausur enthält eine kleine Sammlung an Aufgaben, deren Schwierigkeitsgrad etwa dem der schriftlichen Prüfung des Moduls Einführung in die Informatik Vertiefung entspricht. Die Aufgaben decken nicht alle behandelten Themenbereiche ab und der Umfang der Probeklausur entspricht nicht dem der echten Prüfung.

Musterlösung
Stand: 29. Juli 2016

In dieser Aufgabe ist jeweils genau eine Antwort richtig, welche Sie ankreuzen sollen. Kreuzen Sie pro Teilaufgabe nur ein Kästchen an. Eine richtige Antwort ergibt einen Punkt, eine falsche 0 Punkte. Es gibt keine Minuspunkte.

Aufgabe 1 Allgemeine Fragen.

1. **Teilaufgabe:** Womit kann man einen Booleschen Ausdruck im Allgemeinen nicht in seine minimale Form umwandeln?

   ☑ Ablesen aus Wahrheitstabellen.
   ☐ Verfahren von Quine und McCluskey.
   ☐ KV-Diagramme.
   ☐ Anwendung Boolescher Axiome.

2. **Teilaufgabe:** Welche der folgenden Komplexitätsklassen ist so groß, dass die anderen drei angegebenen Klassen darin enthalten sind?

   ☑ $O(n^2)$
   ☐ $O(k)$
   ☐ $O(n \log(n))$
   ☐ $O(n)$

3. **Teilaufgabe:** Wonach werden Objekte von allgemeinen Datentypen üblicherweise sortiert?

   ☐ Objekte allgemeiner Datentypen können nicht sortiert werden.
   ☐ Nach der Reihenfolge des Operators <=, welcher auch für allgemeine Datentypen definiert ist.
   ☑ Nach einem zu definierenden Schlüssel.
   ☐ Nach dem ersten Ganzzahl- oder Fließkomma-Attribut.

4. **Teilaufgabe:** Was ist eine generische Klasse?

   ☐ Eine Klasse, von der nicht geerbt werden kann.
   ☐ Eine Klasse, die einen Typen als Parameter besitzt, der zur Laufzeit verändert werden kann.
   ☑ Eine Klasse, die einen Typen als Parameter besitzt, der bei der Instanziierung dieser Klasse festgelegt wird.
   ☐ Ein primitiver Datentyp.

5. **Teilaufgabe:** Welche Interface(s) braucht man, um eine nicht-abstrakte iterierbare Klasse zu implementieren?

   ☑ `Iterator<E>`, `Iterable<T>`
   ☐ `Comparable<T>`, `Iterator<E>`
   ☐ `Iterable<T>`
   ☐ `Iterator<E>`

6. **Teilaufgabe:** Worin unterscheidet sich eine doppelt verkettete Liste im Vergleich zur einfach verketteten Liste?

   ☐ Jeder Knoten zeigt auf seine zwei nachfolgenden Knoten.
   ☐ Jeder Knoten speichert zusätzlich zu seinem eigenen Wert auch den Wert des Vorgängers.
   ☐ Sie unterscheidet sich lediglich in einer Tail-Referenz, die auf das letzte Element der Liste zeigt.
   ☑ Jeder Knoten zeigt auf seinen Vorgänger und Nachfolger.

7. **Teilaufgabe:** Wo steht in einem Max-Heap das größte Element?

   ☐ Im linken äußersten Blatt des Baumes.
   ☐ Im rechten äußersten Blatt des Baumes.
   ☐ Das ist in einem Max-Heap nicht genau definiert.
   ☑ In der Wurzel des Baumes.

8. **Teilaufgabe:** Wenn x der linke Nachfolger von y in einem binären Suchbaum ist, dann gilt:

   ☑ key(x) $\leq$ key(y)
   ☐ key(x) $<$ key(y)
   ☐ key(x) $\geq$ key(y)
   ☐ key(x) $>$ key(y)

9. **Teilaufgabe:** Welche der folgenden Datenstrukturen ist linear?

   ☐ Graph.
   ☑ Queue.
   ☐ AVL-Baum.
   ☐ Heap.

10. **Teilaufgabe:** Was gilt für AVL-Bäume?

    ☐ Es werden immer Rotationen beim Einfügen oder Löschen von Elementen benötigt.
    ☐ Sie sind linksvoll.
    ☑ Sie verhindern eine Degeneration zu einer Liste.
    ☐ Sie verhalten sich wie Listen.

11. **Teilaufgabe:** Wie viele Einsen werden für den Ausdruck $\overline{x} \cdot y \cdot z$ in eine KV-Tafel mit 5 Eingangsvariablen eingetragen?

    ☐ 8
    ☐ 1
    ☑ 4
    ☐ 2

12. **Teilaufgabe:** Welche Aussage zu Heapsort ist richtig?

    ☐ Heapsort hat im Average-Case die Komplexität O(n).
    ☐ Bei Heapsort werden Sift-Down und Sift-Up benötigt.
    ☑ Heapsort führt abwechselnd nacheinander Swaps und Heapifys aus.
    ☐ Heapsort kann manche Arrays nicht sortieren, da diese als Heap vorliegen müssen.

13. **Teilaufgabe:** Wozu dient die Methode iterator() des Interfaces Iterable?

- ☐ Sie ruft die for-each Schleife auf und sorgt somit dafür, dass die Iteration einmal vollständig durchgeführt wird.
- ☑ Sie erzeugt ein neues Iteratorobjekt und gibt dessen Referenz zurück.
- ☐ Sie gibt true zurück, falls ein Iterator noch Elemente enthält.
- ☐ Sie überprüft, welche der implementierten Iteratorklassen verwendet werden soll und gibt einen entsprechenden String zurück.

14. **Teilaufgabe:** Welche Wege findet der Dijkstra-Algorithmus ?

- ☑ die kürzesten Wege
- ☐ die schönsten Wege
- ☐ die längsten Wege
- ☐ einen optimalen Weg über alle Knoten im Graph

Aufgabe 2 Boolesche Algebra.

1. **Teilaufgabe:** Wandeln Sie den Booleschen Ausdruck der Funktion:

   $$f(x,y,z) = \overline{(x \equiv y)} + z$$

   mit Hilfe der algebraischen Umformung in eine aKNF um. Die Zwischenschritte müssen erkennbar sein.

   Hinweis: Es gilt $x \equiv y := (x \cdot y) + (\overline{x} \cdot \overline{y})$.

   Lösung:

   $$\begin{aligned} f(x,y,z) &= \overline{(x \equiv y)} + z \\ &= (\overline{(x \cdot y) + (\overline{x} \cdot \overline{y})}) + z \\ &= (\overline{(x \cdot y)} \cdot \overline{(\overline{x} \cdot \overline{y})}) + z \\ &= ((\overline{x} + \overline{y}) \cdot (x + y)) + z \\ &= (\overline{x} + \overline{y} + z) \cdot (x + y + z) \end{aligned}$$

2. **Teilaufgabe:** Wandeln Sie die DNF:

   $$f(x,y,z,w) = w \cdot y + z \cdot \overline{y} + \overline{w} \cdot z \cdot y + w \cdot \overline{z} \cdot \overline{y}$$

   mit Hilfe einer KV-Tafel in eine minimale DNF um. Die Zwischenschritte müssen erkennbar sein.

   Lösung:
   Für: $f(x,y,z,w) = w \cdot y + z \cdot \overline{y} + \overline{w} \cdot z \cdot y + w \cdot \overline{z} \cdot \overline{y}$ ergibt sich folgendes KV-Diagramm:

   $f(x,y,z,w)$:

w \ z \ x \ y				
	0	0	0	0
	1	1	1	1
	1	1	1	1
	1	1	1	1

   Vereinfacht ergibt sich:

   DNF: $f(x,y,z,w) = w + z$

Aufgabe 3 Komplexität.

Bestimmen Sie eine Formel für den Aufwand $T_g(n)$ der folgenden Methode $g(n)$. Dabei soll für die Berechnung des Zeitaufwands nur in Zeile 4 der Funktionsaufruf `funA(n)` und in Zeile 8 der Funktionsaufruf `funB(n)` berücksichtigt werden. Die Funktion `funA(n)` hat einen Aufwand von $T_{\text{funA}}(n) = log(n)$ und die Funkion `funB(n)` hat einen Aufwand von $T_{\text{funB}}(n) = n$.

```
1  public void g(int n) {
2      int i = 0;
3      while (i < n) {
4          funA(n);
5          if(n % 4 == 0){
6              int j = 0;
7              while(j < n){
8                  funB(n);
9                  j++;
10             }
11         }
12         i++;
13     }
14 }
```

- Beantworten Sie für welche Werte von n der Worst-Case und wann der Best-Case eintritt.

- Bestimmen Sie eine Formel für den Laufzeitaufwand $T^{worst}(n)$ der Methode `g(n)` im *Worst-Case*. Geben Sie außerdem an, in welcher kleinsten Komplexitätsklasse sich $T^{worst}(n)$ gerade noch befindet.
  Hinweis: Ein Beweis ist nicht gefordert.

- Bestimmen Sie eine Formel für den Laufzeitaufwand $T^{best}(n)$ der Methode `g(n)` im *Best-Case*. Geben Sie außerdem an, in welcher kleinsten Komplexitätsklasse sich $T^{best}(n)$ gerade noch befindet.
  Hinweis: Ein Beweis ist nicht gefordert.

  Lösung:
  Worst-Case: $\forall n : n \bmod 4 = 0$
  Best-Case: $\forall n : n \bmod 4 \neq 0$
  $T_g^{Worst}(n) = n \cdot (log(n) + n \cdot n) = n^3 + n \cdot log(n) \in \mathcal{O}(n^3)$
  $T_g^{Best}(n) = n \cdot log(n) \in \mathcal{O}(n \cdot \log(n))$

Aufgabe 4 Heapsort.

1. **Teilaufgabe:** Gegeben sei die Zahlenfolge

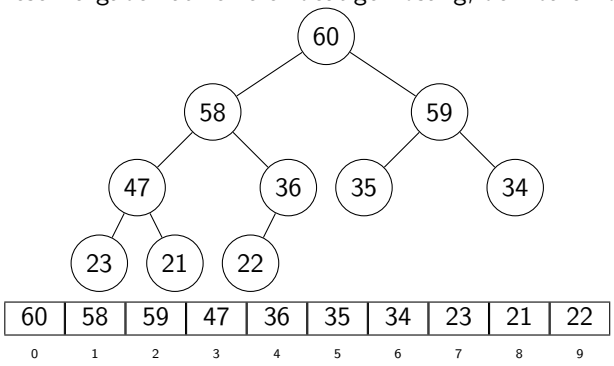
   $$F_1 = 47, 21, 35, 34, 59, 22, 36, 58, 60, 23$$

   Geben sie einen Binärbaum an, der die Elemente der Folge F_1 enthält und die (Max-)Heap Eigenschaft erfüllt. Geben Sie zusätzlich dazu den zum Heap gehörigen Array an.

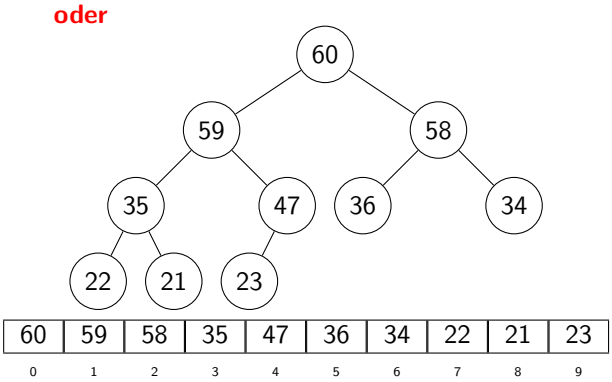
   Lösung:
   Wir lösen diese Aufgabe unter Beachtung der Heapbedingung für Binärheaps. Sie lautet für Maxheaps:

   $$\text{B ist Kindknoten von A} \Leftrightarrow \text{Key(A)} \geq \text{Key(B)}$$

   Wir suchen uns also die größte Zahl der Zahlenfolge F_1 (hier: 60) als erstes Element und bauen den Heap dann von oben nach unten (und dann in den einzelnen Ebenen von links nach rechts) auf. Gleichzeitig füllen wir das Array auf. Das erste Element hat den Index 0. Wenn wir immer die nächstkleinere noch nicht benutzte Zahl der Zahlenfolge F_1 als nächstes Element fortlaufend in das Array (bzw. in den Heap) einfügen, wird die Heapbedingung nie verletzt. Es ist außerdem darauf zu achten, dass der fertige Heap ein linksvoller Baum sein sollte, damit die Arraydarstellung auch Sinn macht.
   Diese Aufgabe hat keine eindeutige Lösung, denkbare Lösungen sind aber z.B:

   oder

   Hinweis: Wenn die Zahlenfolge aufsteigend sortiert ist, dann ist der zugehörige Binärbaum ein Heap.

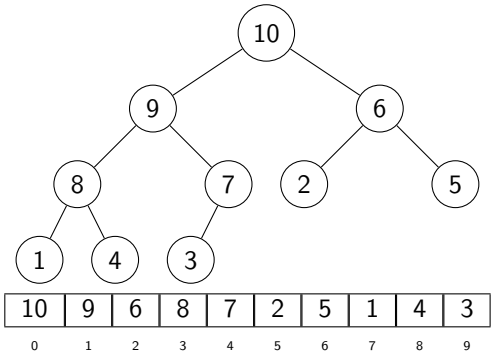
2. **Teilaufgabe:** Gegeben sei die Zahlenfolge

   $$F_2 = 10, 9, 6, 8, 7, 2, 5, 1, 4, 3.$$

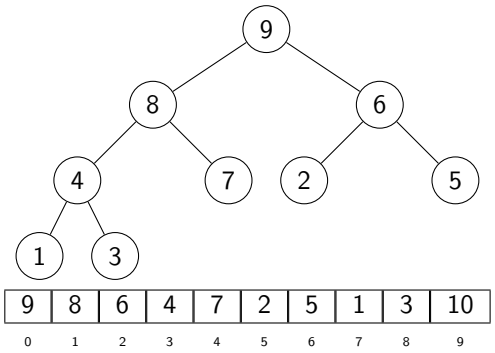
   Sortieren Sie die Folge F_2 mit Heapsort. Stellen Sie nach jedem Sift-Down den Restheap als Baum und die gesamte Zahlenfolge als Array dar.

   Hinweis: Die Zahlenfolge F_2 ist ein Heap.

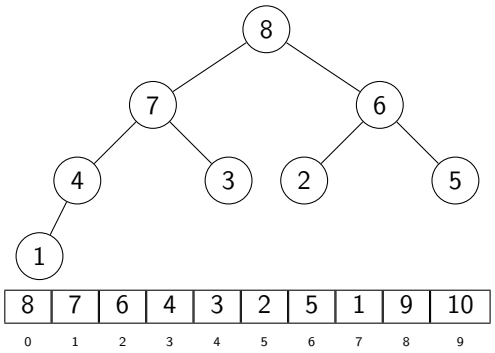
Lösung:
Heapsort kann nur sinnvoll auf einen Heap angewendet werden. Der Hinweis versichert, dass F_2 einer ist. Der Heapsort-Algorithmus ist ausführlich in Tutoriumsvorbereitung # 8 zu finden.
Ausgangsheap:



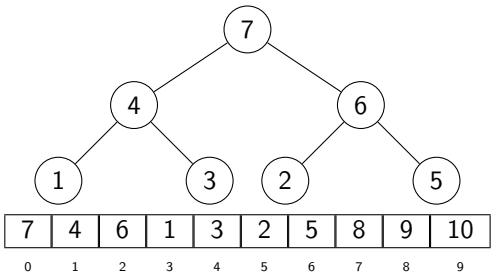
Tausche 3 mit 10 und führe Sift-Down auf die neue Wurzel durch (tausche sie also mit 9,8,4).



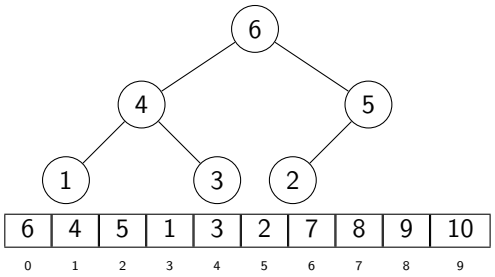
Tausche 3 mit 9 und führe Sift-Down auf die neue Wurzel durch (tausche sie also mit 8,7).



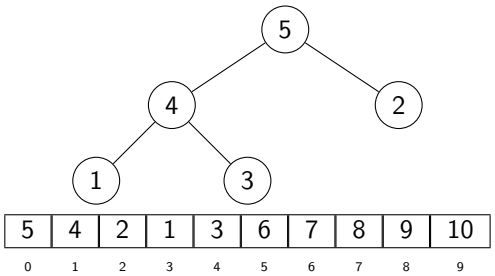
Tausche 1 mit 8 und führe Sift-Down auf die neue Wurzel durch (tausche sie also mit 7,4).



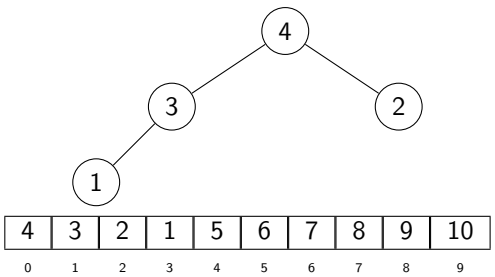
Tausche 5 mit 7 und führe Sift-Down auf die neue Wurzel durch (tausche sie also mit 6).



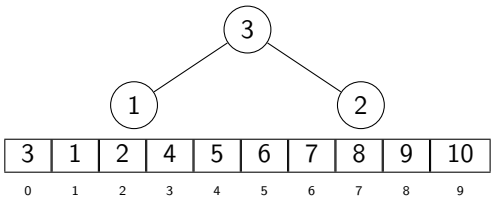
Tausche 2 mit 6 und führe Sift-Down auf die neue Wurzel durch (tausche sie also mit 5).



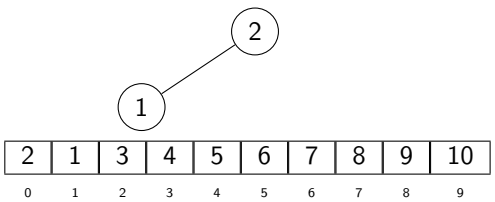
Tausche 3 mit 5 und führe Sift-Down auf die neue Wurzel durch (tausche sie also mit 4).



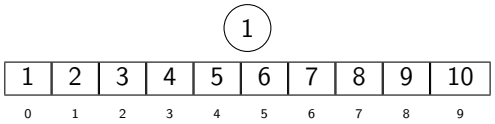
Tausche 1 mit 4 und führe Sift-Down auf die neue Wurzel durch (tausche sie also mit 3).



Tausche 2 mit 3 und führe Sift-Down auf die neue Wurzel durch (kein Tausch notwendig).



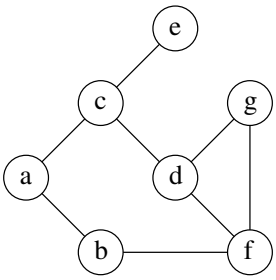
Tausche 1 mit 2 und führe Sift-Down auf die neue Wurzel durch (kein Tausch notwendig).



Fertig!

Aufgabe 5 Tiefensuche.

Betrachten Sie den folgenden Graphen:



1. **Teilaufgabe:** Welche Datenstruktur verwendet die Breitensuche? Wie lautet das Speicherprinzip dieses Datentyps?

   Lösung:
   Queue, FIFO (first in, first out)

2. **Teilaufgabe:** Traversieren Sie den Graphen G mit Tiefensuche. Führen Sie dazu eine Handsimulation mit Hilfe der untenstehenden Tabelle durch. Dabei bezeichne *Schritt* die Nummer des aktuellen Schleifendurchlaufs und AK den aktuellen Knoten. Beachten Sie bei der Handsimulation Folgendes:

   - Startknoten ist der Knoten mit Bezeichner a, welcher sich nach Initialisierung (Schritt 0) im Stack befindet.
   - Geben Sie für $Schritt > 0$ den Inhalt des Stacks jeweils am Ende des aktuellen Schleifendurchlaufs an.
   - Fügen Sie pro Schleifendurchlauf jeweils alle weißen Nachfolger von AK stets in *alphabetisch aufsteigender* Reihenfolge in den Stack ein.
   - Die schwarze Liste enthält alle Knoten, die schon abgearbeitet worden sind. Fügen Sie einen Knoten in dem selben Schleifendurchlauf in die Schwarze Liste ein, in welchem alle seine Nachfolger-Knoten in den Stack eingefügt wurden.

Schritt	AK	Stack	schwarze Liste
0	-	a	-

Lösung:

Schritt	AK	Stack	schwarze Liste
0		a	
1	a	c,b	a
2	c	e,d,b	a,c
3	e	d,b	a,c,e
4	d	g,f,b	a,c,e,d
5	g	f,b	a,c,e,d,g
6	f	b	a,c,e,d,g,f
7	b		a,c,e,d,g,f,b

Aufgabe 6 Heapsort.

Implementieren Sie eine Java-Methode `private void heapify(int currIdx, int endIndex)` die für das Sortierverfahren Heapsort bereitgestellt wird. Vervollständigen Sie dazu den unten vorgegebenen Quellcode.

Lösung:

```
1  public class HeapSort{
2
3      private Integer[] knoten;
4
5      public HeapSort(Integer[] knoten){
6          this.knoten = knoten;
7      }
8
9      // Hilfsmethode getLeftChild liefert den Index des linken Kindknotens zurueck
10     private int getLeftChild(int i){
11         return 2*i+1;
12     }
13
14     // Hilfsmethode getRightChild liefert den Index des rechten Kindknotens zurueck
15     private int getRightChild(int i){
16         return 2*i+2;
17     }
18
19     // Hilfsmethode swap tauscht zwei Elemente des Heaps
20     private void swap(int a, int b){
21         Integer temp = knoten[a];
22         knoten[a] = knoten[b];
23         knoten[b] = temp;
24     }
25
26     // Methode buildHeap wandelt das Array in einen Heap um
27     public void buildHeap(){
28         for(int i = ((knoten.length/2)-1); i>=0; i--){
29             heapify(i, knoten.length-1);
30         }
31     }
32
33     // Methode heapSort sortiert den Heap
34     public void heapSort(){
35         buildHeap();   // zunaechst muss das Array in Heapform gebracht werden
36         for(int size = 0; size < knoten.length; size++){ // size = Groesse des
               sortierten Bereichs
37             swap(0, knoten.length-1-size);
38             heapify(0, knoten.length-2-size); // zu sortierenden Heap reparieren
39         }
40     }
41
42     // Hilfsmethdoe heapify
43     private void heapify(int currIdx, int endIndex){
44         int leftChild = getLeftChild(currIdx);
45         int rightChild = getRightChild(currIdx);
46         if(leftChild<=endIndex){ // false, falls aktueller Knoten keinen linken
               Kindknoten hat und somit das Ende des Heaps erreicht wurde
```

```
47              if(rightChild>endIndex){ // aktueller Knoten hat keinen rechten Kindknoten,
                    es muessen nur der aktuelle Knoten und der linke Kindknoten verglichen
                    werden, danach ist das Ende des Heaps erreicht
48                  if(knoten[currIdx].compareTo(knoten[leftChild])<0){
49                      swap(currIdx, leftChild);
50                  }
51              } else { // aktueller Knoten hat linken und rechten Kindknoten
52                  if((knoten[leftChild].compareTo(knoten[rightChild]))<0){ // linker
                        Kindknoten ist kleiner als rechter Kindknoten, aktueller Knoten muss
                        also mit rechtem Kindknoten verglichen werden
53                      if(knoten[currIdx].compareTo(knoten[rightChild])<0){
54                          swap(currIdx, rightChild);
55                          heapify(rightChild, endIndex);
56                      }
57                  } else { // rechter Kindknoten ist kleiner als der linke Kindknoten (oder
                         gleich), aktueller Knoten muss also mit linkem Kindknoten verglichen
                         werden
58                      if(knoten[currIdx].compareTo(knoten[leftChild])<0){
59                          swap(currIdx, leftChild);
60                          heapify(leftChild, endIndex);
61                      }
62                  }
63              }
64          }
65      }
66  }
```

Aufgabe 7 Iterator.

Betrachten Sie das folgende unvollständige Java-Programm für eine Liste von Listen mit Elementen vom Typ `T`:

```
1
2  import java.util.ArrayList;
3  import java.util.Iterator;
4
5  public class ListOfLists<T> implements Iterable<T> {
6
7      private ArrayList<ArrayList<T>> list;
8
9      public ListOfLists (ArrayList<ArrayList<T>> a){
10       list = a;
11     }
12
13     public Iterator<T> iterator(){
14         return new LIterator();
15     }
16
17     private class LIterator implements Iterator<T>{
18
19         //Ihr Code
20
21     }
22
23 }
```

1. **Teilaufgabe:**

   Implementieren Sie die Methoden `hasNext` und `next` der Klasse `LIterator` sowie ihren Konstruktor. Beachten Sie dabei folgendes:

   - Sie können die Attribute in die Klasse `LIterator` hinzufügen, um die aktuelle Position zu speichern.
   - Der `LIterator` soll die Elemente der Liste `list` traversieren. Die Liste {{1,4,7},{2,3}} wird dabei in der Reihenfolge 1,4,7,2,3 traversiert.
   - Die Methode `hasNext` liefert `true` genau dann zurück, wenn es ein nächstes Element gibt. Der Sonderfall einer nicht-initialisierten oder leeren Liste muss nicht geprüft werden.
   - Die Methode `next` liefert das nächste Element zurück.
   - Die Methode `remove` müssen Sie nicht bearbeiten.

   Hinweis: `ArrayList<T>` hat unter anderem folgende öffentliche Methoden:

   - `int size()` - gibt die Länge der Liste zurück.
   - `T get (int index)` - liefert das Element an der spezifizierten Position zurück.

Lösung:

```
1  private class LIterator implements Iterator<T>{
2      private int a, b;
3
4      public LIterator(){
5
6      }
7
8      public boolean hasNext(){
9          return a < list.size();
10     }
11
12     public T next(){
13
14         T res = list.get(a).get(b);
15
16         if (b < list.get(a).size()-1 ){
17             b++;
18         }else{
19             b = 0;
20             a++;
21         }
22         return res;
23     }
24
25     public void remove(){};
26
27 }
```

2. **Teilaufgabe:** Vervollständigen Sie die untenstehende `test(...)`-Methode. Erzeugen Sie dazu innerhalb der Methode ein Objekt vom Typ `ListofLists<Integer>` (die vor Teilaufgabe 1 definierte Klasse). Initialisieren Sie das Objekt mit dem übergebenen Parameter. Geben Sie nacheinander die Elemente der Liste unter Verwendung einer for-each-Schleife oder der Methoden des Iterator-Objekts auf dem Bildschirm aus.

```
import java.util.ArrayList;
import java.util.Iterator;

public class Test{

public static void test(ArrayList<ArrayList<Integer>> a){









    }
}
```

Lösung:

```
1  import java.util.ArrayList;
2  import java.util.Iterator;
3
4  public class Test{
5
6      public static void test(ArrayList<ArrayList<Integer>> a){
7
8          ListOfLists<Integer> list = new ListOfLists<Integer>(a);
9
10         for (Integer i : list){
11             System.out.println(i);
12     }
13
14 }
```

Aufgabe 8 Listen.

Betrachten Sie das folgende unvollständige Java-Programm für die doppelt verkettete Liste:

```
1  public class DoppeltVerketteteListe<T> {
2
3      private class ListElem {
4
5          T data;
6
7          ListElem(T data) {
8              this.data = data;
9          }
10     }
11
12 }
```

1. **Teilaufgabe:**

   Ergänzen Sie die Klasse `DoppeltVerketteteListe` um Referenzen auf das erste und letzte Element der Liste (head und tail). Ergänzen Sie weiter die innere Klasse ListElem um die benötigten Referenzen auf Vorgänger- und Nachfolge-Elemente.

2. **Teilaufgabe:**

   Implementieren Sie eine Methode `public T get(int i)`, die das Datenobjekt `data` des Listelements an der i-ten Stelle zurückgibt.

   Hinweise:
   1. Es sei n die Anzahl der Listenelemente. Der Kopf der Liste befindet sich an der 0-ten Stelle. Das letzte Listenelement befindet sich an der (n-1)-ten Stelle.

   2. Gehen Sie davon aus, dass stets $0 \leq i < n$ gilt.

3. **Teilaufgabe:**

   Implementieren Sie eine Methode `public T addFirst(T data)`, die an der ersten Stelle der Liste das übergebene Datenobjekt einfügt.

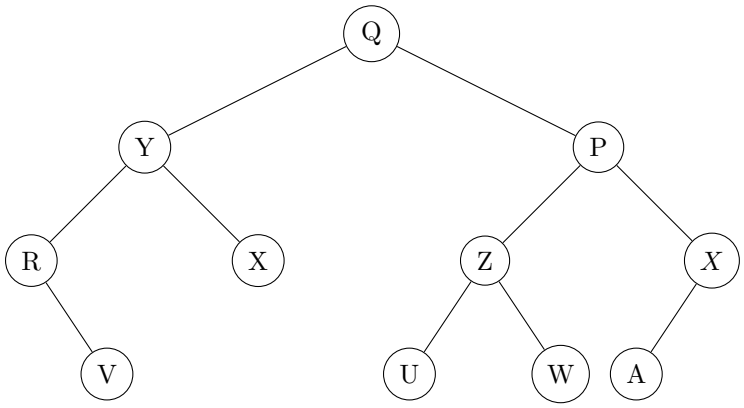
   Lösung:

```
1  public class DoppeltVerketteteListe<T> {
2
3      private class ListElem {
4
5          T data;
6          ListElem next;
7          ListElem prev;
8
9          public ListElem(T data) {
10             this.data = data;
11         }
12
13     }
14
```

```
15      private ListElem head;
16      private ListElem tail;
17
18      public Liste() {
19          head = null;
20          tail = null;
21      }
22
23      public T get(int i) {
24          //leere Liste, kann nach Aufgabenstellung auch weggelassen werden
25          if(head == null) {
26              return null;
27          }
28          //Liste hat mind. 1 Element
29          else {
30              ListElem current = head;
31              int currentIdx = 0;
32              while(currentIdx < i) {
33                  current = current.next;
34                  currentIdx++;
35              }
36              return current.data;
37          }
38      }
39
40      public void addFirst(T data) {
41          Listelem neu = new Listelem(data);
42          if(head == null) {
43              head = neu;
44              tail = head;
45          }
46          //Liste hat mind. 1 Element
47          else {
48              neu.next = head;
49              neu.next.prev = neu;
50              head = neu;
51          }
52      }
53
54  }
```

Aufgabe 9 Traversierung von Bäumen.

1. **Teilaufgabe:** Geben Sie die entstehende Buchstabenfolge aus, wenn Sie den folgenden Binärbaum in postorder-Reihenfolge traversieren.



Lösung:

V, R, X, Y, U, W, Z, A, X, P, Q

2. **Teilaufgabe:** Implementieren Sie eine Java-Klasse `Tree` mit einer Unterklasse `Node` für einen Baum und für Baumknoten. Dabei sollen folgende Bedingungen erfüllt sein:

   - Die in den Knoten abgespeicherten Nutzdaten sind Elemente eines generischen Datentyps.
   - Die Klasse `Node` ist außerhalb der Klasse `Tree` unsichtbar.
   - Jeder Knoten des Baums hat beliebig viele Nachfolger.

   Es sollen nur die Attribute (und keine Methoden) der Klassen angegeben werden.

Lösung:

```
1 public class Tree<T> {
2     private class Node{
3         public Node[] children;
4         public T data;
5     }
6     Node root;
7 }
```

3. **Teilaufgabe:** Implementieren Sie für die Klasse `Tree` aus Aufgabenteil 2 eine *rekursive* Methode `int countNodes()`, welche die Anzahl der Knoten dieses Baumes zurückgibt.

Lösung:

```
1 int countNodes() {
2     return countNodes(root);
3 }
4
5 int countNodes(Node pos){
6     if (root == null){
7         return 0;
8     }
```

```
 9     int count = 0;
10     if (pos.children != null){
11         for (int i=0;i<pos.children.length;i++){
12             count += countNodes(pos.children[i]);
13         }
14     }
15     return count+1;
16 }
```

Aufgabe 10 AVL-Bäume.

Gegeben sei eine Klasse `AVLBaum`, die die spezifischen Unterklassen für innere Knoten (Fork) und Blätter (Leaf) sowie eine Referenz auf das Wurzelelement enthält.

```
1  public class AVLBaum<T>{
2
3      private abstract class Node{
4          public int key;
5          public int hoehe;
6
7          public Node(int key){
8              this.key = key;
9              hoehe = 0;
10         }
11
12         public abstract boolean checkAVLCondition();
13
14     }
15
16     private class Fork extends Node{
17         public Node links;
18         public Node rechts;
19
20         public Fork(int key, Node links, Node rechts){
21             super(key);
22             this.links = links;
23             this.rechts = rechts;
24             hoehe = Math.max(links.hoehe, rechts.hoehe)+1;
25         }
26
27         public void setRechts(Node rechts){
28             this.rechts = rechts;
29             hoehe = Math.max(links.hoehe, rechts.hoehe)+1;
30         }
31
32         public void setLinks(Node links){
33             this.links = links;
34             hoehe = Math.max(links.hoehe, rechts.hoehe)+1;
35         }
36
37     }
38
39     private class Leaf extends Node{
40         public T daten;
41
42         public Leaf(int schluessel, T daten){
43             super(schluessel);
44             this.daten = daten;
45         }
46
47     }
48
49     // Wurzel des AVL-Baums
50     private Node root;
```

```
51
52 }
```

1. **Teilaufgabe:** Erweitern Sie die Klasse Fork um eine Methode `public Fork rotateLeft()`, die eine Linksrotation am aufrufenden Knoten durchführt. Gehen Sie davon aus, dass die Implementierung in der Klasse Fork stattfindet.

2. **Teilaufgabe:** Erweitern Sie die Klasse AVLTree um eine Methode `public boolean checkAVLCondition()`, die die AVL-Eigenschaft des gesamten Baumes testet.
Hinweis: Für Ihre Implementierung müssen Sie die Methode `public boolean checkAVLCondition()` innerhalb der Fork- und Leaf-Klasse implementieren. Erwähnen Sie vor jeder Implementation in welcher Klasse die Implementierung erfolgt.

Lösung:

```
1  public class AVLBaum<T>{
2
3      private abstract class Node{
4          public int key;
5          public int hoehe;
6
7          public Node(int key){
8              this.key = key;
9              hoehe = 0;
10         }
11
12         public abstract boolean checkAVLCondition();
13
14     }
15
16     private class Fork extends Node{
17         public Node links;
18         public Node rechts;
19
20         public Fork(int key, Node links, Node rechts){
21             super(key);
22             this.links = links;
23             this.rechts = rechts;
24             hoehe = Math.max(links.hoehe, rechts.hoehe)+1;
25         }
26
27         public void setRechts(Node rechts){
28             this.rechts = rechts;
29             hoehe = Math.max(links.hoehe, rechts.hoehe)+1;
30         }
31
32         public void setLinks(Node links){
33             this.links = links;
34             hoehe = Math.max(links.hoehe, rechts.hoehe)+1;
35         }
36
37         private Fork rotateLeft() {
38             Fork b = (Fork) this.rechts;
39             this.setRechts(b.links);
```

```
40              b.setLinks(this);
41              return b;
42          }
43
44          public boolean checkAVLCondition(){
45              if((Math.abs(links.hoehe-rechts.hoehe))>1){
46                  return false;
47              } else{
48                  return links.checkAVLCondition()&& rechts.checkAVLCondition();
49              }
50          }
51
52      }
53
54      private class Leaf extends Node{
55          public T daten;
56
57          public Leaf(int schluessel, T daten){
58              super(schluessel);
59              this.daten = daten;
60          }
61
62          public boolean checkAVLCondition(){
63              return true;
64          }
65      }
66
67      // Wurzel des AVL-Baums
68      private Node root;
69
70      //pruefe auf AVL-Bedingung
71      public boolean checkAVLCondition(){
72          return root.checkAVLCondition();
73      }
74
75  }
```