Aufgabe Verkette Listen: Definieren Sie eine Klasse zum Speichern von Integer-Werten in einer **einfach**

verketteten Liste. Diese Klasse soll mindestens folgende Methoden bereitstellen:

a) addFirst: Fügt einen Integer-Wert am Anfang ein b) clear:EntferntalleWerteausderListe
c) sum: Berechnet die Summe alle Werte in der Liste

Aufgabe Verkette Listen: Implementieren Sie das Interface intern als verkettete Liste:

```
public class IntLinkedList implements IntList {

     private class DataNode {
          int value;
          DataNode next;
          DataNode (int value , DataNode next) {
_____________

}

}
public int get(int index) throws NoSuchElementException {
}
public boolean isEmpty() {
}
public boolean isFull() {

}

          this.value = value;
public void addFirst(int value) {
}
public void addLast(int value) {
this.next = next;

} }
```

a) Implementieren Sie das Interface intern als Array:

```
public class IntArrayList implements IntList { final private int data [];
public IntArrayList (int size) {

          data = new int [size];
     }

// Ggf. weitere interne Eigenschaften deklarieren, z.B. für Index: public
void addFirst(int value) {

     }
     public void addLast(int value) {
```

```
}
public int get(int index) throws NoSuchElementException {

	}
	public boolean isEmpty() {
	}
	public boolean isFull() {

}

}
```

Aufgabe **Verkette Listen**: Definieren Sie eine Klasse zum Speichern von String-Werten in einer **einfach**

verketteten Liste. Diese Klasse soll mindestens folgende Methoden bereitstellen:

a) addFirst: Fügt einen String am Anfang ein
b) isEmpty:lieferttrue,wenndieListeleerist
c) contains: liefert true, wenn ein String in der Liste vorhanden ist d) remove: enfernt das erste Vorkommen eines Strings aus der Liste

Aufgabe Verkette Listen: Definieren Sie eine Warteschlange mit `int` **Werten als verkettete Liste.**

Eine Warteschlange funktioniert nach dem **First-In-First-Out** Prinzip. Werte, die als erstes hinzugefügt werden, werden auch als erstes entnommen. Die Methoden dazu werden auf der nächsten Seite definiert.

a) Definieren Sie zunächst die benötigten internen Eigenschaften und Datenstrukturen, d.h.

- **- Knoten (Node) zum Speichern der Integerwerte in einer verketten Liste**
- **- Eine Referenz auf das erste Element der Liste (bzw. der Warteschlange)**
- **- Eine Referenz auf das letzte Element der Liste (bzw. der Warteschlange)**
- `public class IntegerWarteschlange {`

```
}
```

Definieren Sie nun für die `IntegerWarteschlange` **die folgenden Methoden. Beachten Sie, dass sowohl die Referenz auf das erste und das letzte Element angepasst werden müssen!!!!**

b) Diese Methode hängt einen int Wert ans Ende der Warteschlange:

```
public void put(int value){

}
```

c) Dieses Methode liefert den `int` **Wert am Anfang der Warteschlange (erstes Element) und**

entfernt dieses aus der Warteschlange.

```
public int take(){


}
```

d) Diese Methode berechnet die Summe aller Werte, die sich in der Warteschlange befinden:

```
public int sum(){

}
```

Aufgabe Verkette Listen: Gegeben sei die folgende Definition einer einfach verketteten Liste zum Speichern von `String` **Werten:**

```
public class StringList {
      private StringNode first = null;
      class StringNode {

String text;

          StringNode nextNode;
      }

}
```

Die interne Klasse `StringNode` repräsentiert einen einzelnen Knoten, die Referenzvariable `first`

verweist auf den ersten Knoten. Das Listenende wird durch `null` repräsentiert. **5a) Definieren Sie eine Methode isEmpty:**

```
      public boolean isEmpty () {

}
```

a) Definieren Sie eine Methode `void addFirst (String s)`**. Die Methode fügt einen Knoten**

mit dem Textinhalt s am Anfang der Liste ein.

```
     public void addFirst(String s) {

}
```

b) Definieren Sie eine Methode `void reverse()`**, mit der die Reihenfolge der Liste**

umgedreht wird (Bsp: aus „Eins“ – „Zwei“ – „Drei“ wird „Drei“ – „Zwei“ - „Eins“).

```
     public void reverse() {

}
```

Aufgabe Verkette Listen: Gegeben sei die folgende Definition einer einfach verketteten Liste zum Speichern von `String` **Werten:**

```
public class StringList {
      private StringNode first = null;
      private class StringNode {

String text;
StringNode nextNode;
StringNode (String t, StringNode nd) { text = t; nextNode = nd;}

} }
```

Die interne Klasse `StringNode` repräsentiert einen einzelnen Knoten, die Referenzvariable `first` verweist auf den ersten Knoten. Das Listenende wird durch `null` repräsentiert.

a) Definieren Sie eine Methode `void addFirst (String s)`**. Die Methode fügt einen Knoten mit dem Textinhalt** `s` **am Anfang der Liste ein.**

```
public void addFirst(String s) {

}
```

b) Definieren Sie eine Methode `int count(String s)`**. Diese Methode zählt, wie häufig die Zeichenkette** `s` **als Text in den Knoten gespeichert ist.**
Bsp: Für die Liste „Hamburg“-„Berlin“-„Köln“-„München“-„Köln“ liefert `count ("Köln")` **den Wert 2, da „Köln“ zweimal in der Liste auftaucht.**

```
public int count(String s) {

}
```

c) Definieren Sie eine Methode `boolean contains(String s)`**. Diese Methode liefert** `true`

wenn die Zeichenkette `s` **in mindestens einem der Knoten gespeichert ist, sonst** `false`**.**

```
public boolean contains(String s) {

}
```

Aufgabe Verkette Listen: Gegeben sei die folgende Definition einer einfach verketteten Liste zum Speichern von `String` **Werten:**

```
public class StringList {
      private StringNode first = null;
      private class StringNode {
          String text;
          StringNode nextNode;
          // Konstruktor:

StringNode (String t, StringNode nd) { text = t; nextNode = nd;} }

}
```

Die interne Klasse `StringNode` repräsentiert einen einzelnen Knoten, die Referenzvariable `first`

verweist auf den ersten Knoten. Das Listenende wird durch `null` repräsentiert.

5a) Definieren Sie eine Methode `void addFirst (String s)`**. Die Methode fügt einen Knoten**

mit dem Textinhalt `s` **am Anfang der Liste ein.**

```
public void addFirst(String s) {

}
```

5b) Definieren Sie eine Methode `String removeFirst()`**. Die Methode entfernt den ersten Knoten und liefert den darin enthaltenden String zurück.**
Falls die Liste leer ist, wird eine Exception ausgelöst.

```
public String removeFirst() throws NoSuchElementException {

}
```

5c)DieseMethodeliefert`true`**wenndieListeleerist, sonst**`false`**.**

```
public boolean isEmpty() {

}
```

5d) Lässt sich mit dieser Listenimplementierung eine Warteschlange oder ein Stapelspeicher (Stack) implementieren? Oder beides?

Aufgabe Verkette Listen: Gegeben sei die folgende Definition einer einfach verketteten Liste zum Speichern von `int` **Werten:**

```
public class IntList {
      private IntNode first = null;
      class IntNode {

int value;

          IntNode nextNode;
      }
```

```
}
```
Die interne Klasse `IntNode` repräsentiert einen einzelnen Knoten, die Referenzvariable `first`

verweist auf den ersten Knoten. Das Listenende wird durch `null` repräsentiert. **5a) Definieren Sie eine Methode isEmpty:**

```
      public boolean isEmpty () {

}
```

5b) Definieren Sie eine Methode `zaehleVorkommen (int zahl)`**. Die Methode zählt, wie oft die** `zahl` **in der verketteten List vorkommt (Beispiel: In der Liste „3 – 5 – 5 – 6 – 5" kommt die Zahl 5 dreimal vor).**

```
     public int zaehleVorkommen(int zahl) {

}
```

5c) Definieren Sie eine Methode `contains(int zahl)`**, die** `true` **zurück gibt, wenn** `zahl` **in der**

Liste vorhanden ist:

```
     public boolean contains (int zahl) {

}
```

Aufgabe Verkette Listen: Gegeben sei die folgende Definition einer einfach verketteten Liste zum Speichern von `int` **Werten:**

```
public class IntegerList {
      private IntegerNode first = null;
      private class IntegerNode {
          int value;
          IntegerNode nextNode;
          // Konstruktor:

IntegerNode (int v , IntegerNode nd) { value = v; nextNode = nd;} }

}
```
Die interne Klasse `IntegerNode` repräsentiert einen einzelnen Knoten, die Referenzvariable `first`

verweist auf den ersten Knoten. Das Listenende wird durch `null` repräsentiert.
Implementieren Sie die folgenden Methoden.

5a) Diese Methode fügt einen Knoten mit dem Wert `i` **am Anfang der Liste ein:** `public void addFirst(int i) {`

```
}
```

5b) Diese Methode liefert den kleinsten Datenwert der Liste. Die Methode wirft eine Exception wenn die Liste leer ist.

```
public int getSmallestElement () throws NoSuchElementException {

}
```

5c) Diese Methode entfernt den Wert w aus der Liste:

```
public void remove (int w) {

}
```

5d) Diese Methode sortiert die Liste mit dem Selection Sort Algorithmus:

```
public void sort() {

}
```

Hinweis: Sie dürfen Methoden auf andere Methoden abbilden.

Aufgabe Verkette Listen: Gegeben sei die folgende Schnittstelle für den Knoten einer einfach verketteten Liste mit int-Werten:

```
public interface IntegerNodeInterface {
/**
 * Liefert true zurück wenn der Wert n
 * in der Liste vorhanden ist:
 */
public boolean contains (int n);
     /**
      * Gibt an wie viele Elemente in der Liste sind
      */
     /**
      * Liefert die Summe aller Werte in der Liste zurück
      */

}
```

4a) Das Ende der Liste soll durch einen leeren Knoten definiert sein, d.h. durch ein Objekt einer Klasse `EmptyNode`**. Definieren Sie die Methoden für die Klasse** `EmptyNode`**:**

```
public class EmptyNode implements IntegerNodeInterface {
public boolean contains(int n) {
// TODO: Ihr Code:
```

```
}
public int size() { // TODO: Ihr Code:

}
public int sum() { // TODO: Ihr Code:

} }
```

```
public int size ();
public int sum ();
```

4b) Implementieren Sie nun einen Knoten, der einen int-Wert enthält. Definieren Sie die Methoden `contains(...), size()` **und** `sum()` **rekursiv.**

```
public class IntNode implements IntegerNodeInterface {
     // TODO: Ihr Code für Eigenschaften:
_____

} }

public IntNode(int value , IntegerNodeInterface next ) {

}

// TODO: Ihr Code:
}
public boolean contains(int n) {
// TODO: Rekursive Implementierung:
@Override
public int size() {

}

// TODO: Rekursive Implementierung:
@Override
public int sum() {
// TODO: Rekursive Implementierung:
```

4c) Ergänzen Sie die Klasse `IntegerList`**, die zur Datenspeicherung** `IntNode` **Objekte verwendet.**

```
public class IntegerList {
     private IntegerNodeInterface first;
     private final static IntegerNodeInterface EMPTY =
                                             new EmptyNode();
     // Konstruktor für die leere Liste
     public IntegerList() {
          //TODO: Worauf zeigt first bei der leeren Liste?

first =
```

```
}
public boolean contains (int n) {
// TODO: Prüfen, ob einer der Knoten n enthält

}

public int size () {
     return first.size();
}
public int sum () {
// TODO: Die Summe aller Knoten berechnen



     public void addFirst (int n) {
          // TODO: Wert n an den Anfang der Liste hängen

} }
```