Die Datenstruktur ***Queue***

Die Datenstruktur *Schlange* (engl. `Queue`) entspricht einer Warteschlange mit höflichen Menschen: Jeder Neuankömmling stellt sich hinten an und wartet geduldig, bis er ganz vorne steht und an der Reihe ist. Wer also zuerst kommt, ist auch zuerst dran. Entsprechend spricht man bei Schlangen in Anlehnung an die englische Kurzform *`first in`*, *`first out`* vom **FIFO-Prinzip**.
Wie beim Stapel soll zunächst auch hier das Verkettungsprinzip betrachtet werden:

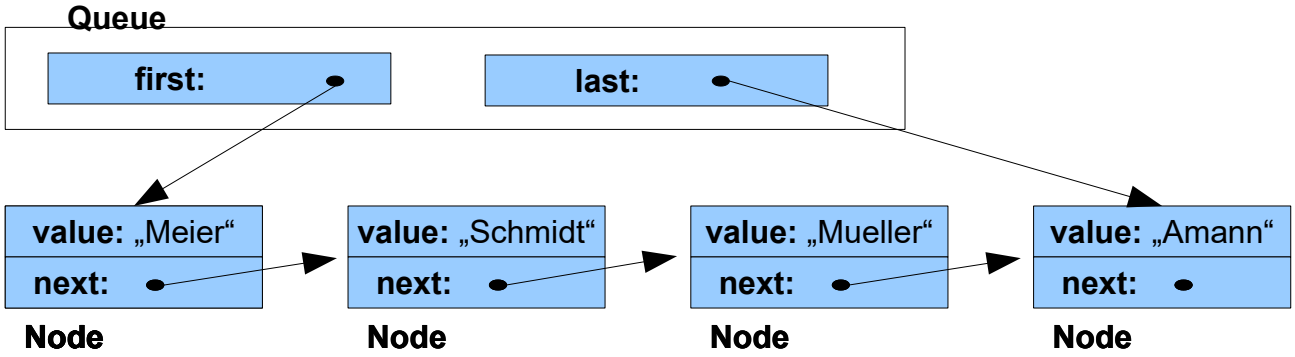

Abbildung: *Queue als Verkettung von Nodes*

Ein Überlauf ist bei dieser Sichtweise einer Schlange sozusagen ausgeschlossen, da beliebig viele neue Knoten dynamisch hinzugefügt werden können.
Die Schlange hat einen Anfang und ein Ende, die durch zwei Zeiger ***`first`*** und ***`last`*** markiert sind. Bei einer leeren Schlange sind beide Zeiger ohne Wert, d.h. sie haben die Werte ***`null`***. Das Anstellen kann immer erfolgen, das Entnehmen von Elementen allerdings nicht bei leerer Schlange.

Dokumentation der Methoden der Klasse Queue

Konstruktor **`Queue<ContentType>()`**
Eine leere Schlange wird erzeugt. Objekte, die in dieser Schlange verwaltet werden, müssen vom Typ `ContentType` sein.

Anfrage **`boolean isEmpty()`**
Die Anfrage liefert den Wert `true`, wenn die Schlange keine Objekte enthält.
Sonst liefert sie den Wert `false`.

Auftrag **`enqueue(ContentType pContent)`**
Das Objekt `pContent` wird an die Schlange angehängt.
Falls `pContent` gleich `null` ist, bleibt die Schlange unverändert.

Auftrag **`dequeue()`**
Das erste Objekt wird aus der Schlange entfernt.
Falls die Schlange leer ist, wird sie nicht verändert.

Anfrage **`ContentType front()`**
Die Anfrage liefert das erste Objekt der Schlange. Die Schlange bleibt unverändert.
Falls die Schlange leer ist, wird `null` zurückgegeben.

Aufgaben

1) Die folgende Methode soll in `celebritiesQueue` eine `Celebrity` mit Namen `pName` suchen:

```
public Celebrity suche(String pName)
{
  Celebrity ergebnis = _____________________
  while(celebritiesQueue._________________ == _______________){
    Celebrity c = celebritiesQueue._____________________________;
    if(c.getName().equals(pName)){
      ergebnis = c;
    }
    celebritiesQueue._________________
  }
  return ergebnis;
}
```

2) Implementiere die Methode
`public int anzahl()`
Die Methode soll die Anzahl der Elemente des `celebritiesQueue` zurückgeben, ohne diesen dabei zu zerstören.
Hinweis: Dafür braucht man einen zweiten Queue, auf den man alle Elemente überträgt und am Ende wieder zurück. Den zweiten Queue kann man z.B. so erzeugen:
`Queue<Celebrity> hilfsQueue = new Queue<Celebrity>();`

3) Implementiere die Methode
`public Celebrity suche(String pName)`
Die Methode soll überprüfen, ob `celebritiesQueue` eine Celebrity mit pName enthält. Dabei soll dieser nicht zerstört werden.
Hinweis: Strings vergleicht man so:
`if(c.getName().equals(pName))`

4) Implementiere die Methode
`public Queue<Celebrity> reicherAls(String pName)`
Die Methode soll alle Celebrities zurückgeben, die reicher sind als `pName`.
Hinweis: Man ruft erst die Methode suche auf!
Dann erzeugt man einen Ergebnis-Queue, den man am Ende zurückgibt:
`Queue<Celebrity> ergebnis = new QueueWithViewer<Celebrity>();`

5) Finde eine Strategie für die Methode
`public Queue umkehren()`
Die Rückgabe der Methode soll dem `celebritiesQueue` entsprechen, aber genau in umgekehrter Reihenfolge. Implementiere die Methode.

6) Programmiere eine Methode
`public Queue<Celebrity> sortierenNachVermoegen()`
Die Methode soll die Celebrities nach Vermögen sortieren und zurückgeben.
Der `celebritiesQueue` darf dabei zerstört werden.
Hinweis: Du brauchst zwei geeignete Hilfsmethoden! Welche?