Übung 4 – Datenstrukturen

Arbeiten Sie im Skript das *Kapitel Datenstrukturen* durch. Suchen Sie sich weitere Informationen, wo Sie die Ausführungen nicht verstehen. Sprechen Sie mit Ihren Kommilitoninnen und Kommilitonen. Fragen Sie im Forum.
Lesen Sie die Aufgaben vollständig und markieren Sie sich zentrale Aspekte. Verwenden Sie **keine Klassen der Java API** außer als Test für Ihre Implementationen und beachten Sie konsistent die Zugriffrechte.

Aufgabe 1 (Theorie)

Wie entwickeln sich im Speicher, in bildlicher Darstellung, folgende Datenstrukturen zur Darstellung des **Fifo** Prinzips

1. Array (Feld) mit fester Speichergröße 6
2. dynamisches Array (Verkleinerung bei Viertelfüllung, Verdopplung im Vergrößerungsfall) mit Startspeichergröße 1
3. einfach verkettete Liste

unter folgenden Operationen?

```
insert(1); insert(2); insert(3); remove(); remove(); insert(4); insert(5);
remove(); insert(6); insert(7); insert(8); insert(9); remove();
```

Geben Sie alle verwendeten Variablen/Speicherplätze an und zeigen Sie jeweils deren aktuellen Zustand (lassen Sie keine Felder leer), beginnend mit dem Startzustand. Markieren Sie uninitialisierte Array-Elemente mit x. Wenn bei einer Aktion Feldelemente verschoben werden müssen, lassen Sie nicht mehr gebrauchte Feldelemente unverändert, wenn diese nicht überschrieben werden.

Aufgabe 2

Implementieren Sie eine generische Klasse `DynArray` als Datenstruktur dynamisches Array, die intern auf einem dynamischem Array basiert. Die Startspeichergröße ist 1. Die Größe soll sich verdoppeln wenn alle Stellen des Arrays belegt sind und ein neues Element eingefügt werden soll. Sie soll sich halbieren, wenn das Array nur noch zu einem Viertel gefüllt ist. Die Klasse soll die folgende Methoden bereitstellen:

- `DynArray()` :
  Der Konstruktor legt ein Array entsprechend der Startspeichergröße an.

- `size()` :
  Die Methode gibt die Anzahl der belegten Plätze im Array als ganzzahligen Wert zurück.
- `capacity()`:
  Die Methode gibt die Länge des aktuellen Arrays als ganzzahligen Wert zurück.
- `get(int pos)` :
  Die Methode gibt den Inhalt des Array an `pos` zurück. Es sollen hierbei nur bereits belegte Positionen abgefragt werden können. *Welche Einschränkungen folgen daraus für `pos`?*
- `set(int pos, T e)` :
  Die Methode setzt den Wert `e` an Position`pos` im Array und gibt den alten Wert zurück. Die Methode definiert ein Überschreiben, d.h. es dürfen nur Positionen verändert werden, die bereits Werte beinhhalten (siehe `get(int pos)`).
- `addFirst(T e)` :
  Die Methode fügt den neuen Inhalt vor dem ersten Wert des Arrays ein - Ihre bisherigen Werte sollen hierbei weiterhin übernommen werden.
  Was bedeutet das für alle bestehenden Werte in Ihrem Array?
- `addLast(T e)`:
  Die Methode fügt den neuen Inhalt hinter dem letzten Wert des Arrays ein.
- `removeFirst()`:
  Die Methode löscht den ersten Wert im Array und gibt den Wert zurück.
- `removeLast()`:
  Die Methode löscht den letzten Wert im Array und gibt den Wert zurück.

Legen Sie sich Hilfsmethoden an, wenn Sie welche benötigen. Überlegen Sie wo Exceptionhandling sinnvoll ist und erweitern Sie dementsprechend Ihre Methoden. Testen Sie Ihre Implementierung mit JUnittests.

Aufgabe 3

Implementieren Sie eine generische Klasse `Ringpuffer` als Datenstruktur Ringpuffer (siehe Skript), die intern auf einem Array fester Größe basiert. Die Klasse soll die folgende Methoden bereitstellen: Überlegen Sie welche Operation Sie brauchen wenn Sie am Ende des Rings angekommen sind.

- `Ringpuffer(int capacity)`:
  Der Konstruktor übernimmt eine Variable `capacity` von Typ `int` und legt ein Array entsprechender Länge an.

- `size()`:
  Die Methode gibt die Anzahl der Elemente im Ringpuffer als ganzzahligen Wert zurück.
- `get(int pos)`:
  Gibt den Inhalt an `pos` zurück. Beachten Sie dass Ihr Array nur `capacity` lang ist. Es sollen hierbei nur bereits belegte Positionen abgefragt werden können. **Welche Einschränkungen folgen daraus für `pos`?**
- `set(int pos, T e)`:
  Fügt das übergebene Element an Position `pos` ein und gibt den alten Wert zurück. Die Methode definiert ein Überschreiben, d.h. es dürfen nur Positionen verändert werden, die bereits Werte beinhhalten (siehe `get(int pos)`).
- `addFirst( T e)`:
  Fügt das übergebene Element vorne ein, gibt aber nichts zurück.
- `addLast(T e)`:
  Fügt das übergebene Element hinten ein, gibt aber nichts zurück.
- `removeFirst()`:
  Entfernt das erste Element und gibt den Inhalt zurück.
- `removeLast()`:
  Entfernt das letzte Element und gibt den Inhalt zurück.

Überlegen Sie wo Exceptionhandling sinnvoll ist und erweitern Sie dementsprechend Ihre Methoden. Testen Sie Ihre Implementierung mit JUnittests.

Aufgabe 4

Wir wollen im Folgenden einen Temperatursensor modellieren. Dieser Sensor soll Temperaturwerte für die letzten 24 Stunden speichern können. Diese Werte sollen Gleitkommazahlen sein, die möglichst wenig Speicher verbrauchen. Er kann die aktuelle (zuletzt gemessene) Temperatur liefern und die Durschnittstemperatur der letzten Messungen (also maximal der letzten 24 Stunden). Werden diese Werte vom Sensor abgefragt, wenn der Sensor leer ist, soll `NaN` geliefert werden (siehe bspw. Java-API zur Klasse `Float`). Wenn der Speicher des Sensors voll ist, soll der älteste Wert überschrieben werden. Außerdem soll es die Möglichkeit geben den Sensor zu flushen, also den Speicher zu leeren.

Schreiben Sie hierzu eine Klasse `Temperatursensor`. Diese Klasse soll zur Speicherung der Werte intern einen Ringpuffer verwenden, der 24 `Float`-Werte speichern kann. Schreiben Sie folgenden Methoden:

- `neueMessung(Float wert)`: diese fügt einen neuen Wert im Ringpuffer hinzu. Da wir unseren Ringpuffer in einer vorigen Aufgabe so implementiert haben, dass beim Einfügen in einen vollen Ringpuffer eine Exception geworfen wird, müssen wir diese hier fangen und entsprechend ggf. die älteste Messung löschen und dann einfügen. *Frage: Was könnte man an unserer Implementierung für den Ringpuffer hier anders machen, um das effizienter zu gestalten?*
- `aktuelleTemperatur()`: gibt den aktuellsten Wert zurück. Sollte der Temperatursensor keine Messung gemacht haben muss eine Exception gefangen werden und es soll `Float.NaN` zurückgegeben werden.
- `durchschnittsTemperatur()`: gibt den Durchschnitt aller Messungen zurück. Sollte der Temperatursensor keine Messungen gemacht haben, soll `Float.NaN` zurückgegeben werden.
- `reset()`: soll alle Werte aus dem Ringpuffer entfernen – der Zustand ist nach Aufruf also nach außen so, als wäre es ein leerer Puffer. *Frage: wie könnte man das einfach machen, wenn wir dies direkt in der Klasse Ringpuffer gemacht hätten?*

Zusatzaufgabe (Programmierübung Projekt Euler)

2520 ist die kleinste Zahl, die ohne Rest durch jede Zahl von 1 bis 10 teilbar ist.

Was ist die kleinste positive Zahl, die durch alle Zahlen von 1 bis 20 glatt teilbar ist?

`https://projekteuler.de/problems/5`

- **a.** Implementieren Sie die Aufgabe als statische Methode, die einen `int`-Wert `range` an nimmt (für das obige Beispiel 10 bzw. 20) und das kleinste gemeinsame Vielfache als Ganzzahl zurück gibt.
- **b.** Testen Sie Ihre Methode mit JUnit Tests.