Exercises: Linear-Data-Structures

This document defines the lab for "Data Structures – Fundamentals (Java)" course @ Software University.

Please submit your solutions (source code) of all below-described problems in Judge.

Write Java code for solving the tasks on the following pages. Code should compile under the Java 8 and above standards, you can write and locally test your solution with the Java 13 standard, however, **Judge will run the submission with Java 10 JRE**. Avoid submissions with **features included after Java 10** release doing **otherwise** will result in **compile time error**.

Any code files that are part of the task are provided as **Skeleton**. In the beginning import the project skeleton, do not change any of the interfaces or classes provided. You are free to add additional logic in form of methods in both interfaces and implementations you are not allowed to delete or remove any of the code provided. Do not change the names of the files as they are part of the tests logic. **Do not change the packages** or move any of the files provided inside the skeleton if you have to add a new file add it in the same package of usage.

Some **tests may be provided** within the skeleton – use those for local **testing and debugging**, however, there **is no guarantee that there are no hidden tests added inside Judge**.

Please follow the exact instructions on uploading the solutions for each task. Submit as a **.zip archive** the files contained inside the **"...\src\main\java"** folder this should work for all tasks regardless of current DS implementation.

For the solution to compile the tests **successfully** the project **must** have a **single Main.java** file containing single **public static void main(String[] args)** method even an empty one within the **Main class**.

Some of the problems will have simple **Benchmark tests** inside the skeleton. You can try to run those with **different values** and **different implementations** to **observe** behavior. However, **keep** in mind that the result comes **only as numbers** and this data may be **misleading** in some situations. Also, the tests are not started from the command prompt which may **influence** the **accuracy** of the results. Those tests are only added as an **example** of **different data structures performance** on their **common** operations.

The Benchmark tool we are using is **JMH** (Java Microbenchmark Harness) and that is a Java harness for building, running, and analyzing, **nano/micro/milli/macro** benchmarks written in Java and other languages targeting the JVM.

Additional information can be found here: JMH and also there are other examples over the **internet**.

Important: when importing the skeleton **select import project** and then **select from** the **maven module**, this way any following **dependencies** will be **automatically resolved**. The project has **NO default version** of **JDK so after the import you may (depending on some configurations) need to specify the SDK, you can download JDK 13** from **HERE.**

1. Faster Queue

You have the basic implementation of the Queue<E> data structure from the lecture lab. The task is simple you have to modify the structure so now we can reduce the complexity when adding to a **constant factor**.

- **Offer (E element)** – modify this operation so you can **perform the offer in constant time**, also modify anything required to achieve that.

Hint: you can add a node that points to the end of the queue. But now you have to modify everything that somehow relates to the node chaining.

Here the tests are hidden so you have to figure out how to solve the problem above. Remember you can use the **benchmark tests** to observe the **performance**.

2. DoublyLinkedList

Your task is to take the implementation of the SinglyLinkedList<E> from the lab and make it a doubly linked list. This means that you have to add two things:

1. Add additional field **Node<E> tail** that will always **point to the last** element of the linked list.
2. Add field **Node<E> previous** to the **Node class** this should point to the **previous node**.

Do the changes above the methods below should remain with unchanged erasure, use the tests provided to ensure that.

- **AddFirst (E element)** – **adds** an **element** in **front** of the collection and **increases the size**.
- **AddLast (E element)** – **adds** an **element** after **the last element** of the collection and **increases the size**.
- **E removeFirst ()** – **removes and returns** the **first element** of the collection if **it** is **such** if **no** then **throw IllegalStateException** with an appropriate message.
- **E removeLast ()** – **removes and returns** the **last element** of the collection if **it** is **such** if **no** then **throw IllegalStateException** with an appropriate message.
- **E getFirst ()** – **returns** but **does not remove** the **first element** of the collection if **it** is **such** if **no** then **throw IllegalStateException** with an appropriate message.
- **E getLast ()** – **returns** but **does not remove** the **last element** of the collection if **it** is **such** if **no** then **throw IllegalStateException** with an appropriate message.
- **Int size ()** – **returns** the **number** of **elements** inside the collection.
- **Boolean isEmpty ()** – returns if the collection **contains** any elements or **not**.

Going on with the changes you will notice that **we do similar operations** every time **we do chaining**. **Change the Node constructor** so its **call does that work** instead of you.

3. ArrayDeque – Circular Queue

Implement a data structure **ArrayDeque<E>** that holds a sequence of elements of generic type **E**. The structure should have some **capacity** that **grows twice** when it is filled, **always starting with an odd number**. This data structure should be usable as **Stack**, **Queue,** and **ArrayList** in some manners. Should Support the following operations:

- **Add (E element)** – **adds** an element at the **end**.
- **Offer (E element)** – **adds** an element the same way a **Queue does**.
- **AddFirst (E element)** – **adds** an element in **front** of all other elements.
- **AddLast (E element)** – **adds** an element after the **last one**.
- **Push (E element)** – **adds** an element the same way a **Stack does**.
- **Insert (int index, E element)** – **inserts** an element at given **index** if **valid** if **not** throw **IndexOutOfBounds** exception.
- **Set (int index, E element)** – **sets** an element at given **index** if **valid** if **not** throw **IndexOutOfBounds** exception.

- **Peek ()** – **peeks** an element the same way a **Queue** and a **Stack** do make it **work** for **both usages**. If there are **no elements** return **null**.
- **Poll ()** – **removes** the element in **front** and **returns** it, if **no** elements are stored return **null**.
- **Pop ()** – **removes** the element at the **end** and **returns** it, if **no** elements are stored return **null**.
- **Get (int index)** – **gets** an element at given **index** if **valid** if not throw **IndexOutOfBounds** exception.
- **Get (Object object)** – **gets** the **first occurrence** of an element and **returns** it if there is **no** such element return **null**.
- **Remove (int index)** – **removes** the element at given **index** and **returns** it valid if **no** throw **IndexOutOfBounds** exception.
- **Remove (Object object)** – **removes** the **first occurrence** of an element is **present** if not returns **null**.
- **RemoveFirst ()** – **removes** the element in **front** and **returns** it, if **no** elements are stored return **null**.
- **RemoveLast ()** – **removes** the element at the **end** and **returns** it, if **no** elements are stored return **null**.
- **Size ()** – **returns** the **number** of elements stored.
- **Capacity ()** – **returns** the **capacity** of the structure.
- **TrimToSize ()** – **shrinks** the **capacity** to the **number** of elements so the two values become **equal**.
- **IsEmpty ()** – **returns** if there **are** elements present or **not**.

As you can see some methods do pretty much the same thing. So why do we need them? We can make the usage of the data structure much more clear when we read the code that uses its operations if they are well defined. For example, if you want to use it as a Stack add by calling **push () or something else** which one of the above methods can cover that case? Think about reusing some parts of the code and mostly think in such a way that it is clear which operation does what without the need to look at the implementation details.

Try to figure out the **similar** operations between this DS and Stack or Queue etc…

Hints: **Constructor and fields** made easy so you can start from somewhere:

```
public class ArrayDeque<E> implements Deque<E> {
    private final int DEFAULT_CAPACITY = 7;
    private int head;
    private int tail;
    private int size;

    private Object[] elements;

    public ArrayDeque() {
        this.elements = new Object[DEFAULT_CAPACITY];
        this.head = this.elements.length / 2;
        this.tail = this.head;
    }
```

That is all the help you need the rest is on you and of course as **always** you are going to **make it**.

4. ReversedList

Implement a data structure **ReversedList<E>** that holds a sequence of elements of generic type **E**. It should hold a **sequence of items in reversed order**. The structure should have some **capacity** that **grows twice** when it is filled, **always starting at 2**. The reversed list should support the following operations:

- **`Add(E element)`** – adds an element to the sequence (grow twice the underlying array to extend its capacity in case the capacity is full)
- **`Size()`** – returns the number of elements in the structure
- **`Capacity()`** – returns the capacity of the underlying array holding the elements of the structure
- **`Get(index)`** – the indexer should access the elements by **index** (in range **`0`** … **`size-1`**) in the reverse order of adding
- **`RemoveAt(index)`** – removes an element by **index** (in range **`0`** … **`size-1`**) in the reverse order of adding
- **`Iterator<E>`** – implement an iterator to allow iterating over the elements in a **`foreach`** loop in a reversed order of their addition

Hint: you can keep the elements in the order of their adding, by access them in reversed order (from end to start).

5. Balanced Parentheses

Inside the **skeleton,** you are given class **BalancedParentheses** and **BalancedParenthesesTest**. Your task is to **implement** the **method solve ()** – which **performs analysis** of the **parentheses** filed and returns **true** or **false** whether the **parentheses** are **balanced** or **not**.

A sequence of parentheses **is balanced if** every open parenthesis can be paired uniquely with a closed parenthesis that occurs after the former. Also, **the interval between them must be balanced**.
You will be given three types of parentheses: **`(`**, **`{`**, and **`[`**.

{[()]} - This is a balanced parenthesis.

{[(])} - This is not a balanced parenthesis.

"Wisdom comes from experience. Experience is often a result of lack of wisdom." — Terry Pratchett