Lab: 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 **.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. ArrayList

Your task is to implement the **ADS List<E>** inside the **ArrayList<E>** class provided. You can see that this class implements the **List<E>** interface you have to implement all the methods to solve the problem, however, you are free to add more methods with any access modifier you want.

- **Boolean add (E element)** – **adds** an element at the end of the **sequence** and **returns true** if **successful** (always returns true). This method should in addition **increase** the **size** of the structure and **ensure** that there is **enough space** for the addition to **work**. If needed you will have to **resize the array**.

- **Boolean add (int index, E element)** – the only **difference** from the **above one** is that now we have a **specified index** at which to **add (insert)** an element. This time you have to **validate** the **index** then add the element and **shift** the **remaining** elements if any from the **index + 1** to the **right** (from the index + 1 to the last index + 1).
- **E get (int index)** – **returns** the **element** at the given **index** and **does not remove** it from the collection. If the **index** is **invalid** throw **IndexOutOfBoundsException** with a proper message of your choice (the message itself is not subjected to testing).
- **E set (int index, E element)** – **sets** the element at given **index** and **returns the previously** stored at that **index element**, again you should **validate** the **index** and throw **IndexOutOfBoundsException** if the validation fails.
- **E remove (int index)** – **removes** the element at the **specified index** and **returns** it – **again the same validation**, here you should already **have some way to reuse the index validation**.
- **Int size ()** – returns the **number** of **elements**.
- **Int indexOf (E element)** – **returns** the **index** of an **element** if the element **is not present** in the structure then **returns -1** as an invalid array index.
- **Boolean contains (E element)** – returns **true** or **false** if the element **is present** inside the structure.
- **Boolean isEmpty ()** – **returns** if there are **elements** stored or **not**.

Solution:

- Boolean add (E element):

```
@Override
public boolean add(E element) {
    if (this.size == this.elements.length) {
        this.elements = grow();
    }
    this.elements[this.size++] = element;
    return true;
}
```

- The – grow () helper method:

```
private Object[] grow() {
    return Arrays.copyOf(this.elements, newLength: this.elements.length * 2);
}
```

- Boolean add (int index, E element):

```
@Override
public boolean add(int index, E element) {
    checkIndex(index);
    insert(index, element);
    return true;
}
```

- The insert (int index, E element) method:

```
private void insert(int index, E element) {
    if (this.size == this.elements.length) {
        this.elements = grow();
    }
    E lastElement = this.getElement( index: this.size - 1);
    for (int i = this.size - 1; i > index; i--) {
        this.elements[i] = this.elements[i - 1];
    }
    this.elements[this.size] = lastElement;
    this.elements[index] = element;
    this.size++;
}
```

- E get (int index):

```
@Override
public E get(int index) {
    checkIndex(index);
    return this.getElement(index);
}
```

- E set (int index, E element):

```
@Override
public E set(int index, E element) {
    checkIndex(index);
    E oldElement = this.getElement(index);
    this.elements[index] = element;
    return oldElement;
}
```

- E remove (int index):

```
@Override
public E remove(int index) {
    this.checkIndex(index);
    E element = this.getElement(index);
    this.elements[index] = null;
    this.size--;
    shift(index);
    ensureCapacity();
    return element;
}
```

- Take a look at those additional helper methods, you can reuse them whenever needed: First, ensure the capacity of the array if we have less than one-third of the elements we can shrink the array.

```
private void ensureCapacity() {
    if (this.size < this.elements.length / 3) {
        this.elements = shrink();
    }
}
```

- The shrinking method looks a lot like grow with one major difference – we reduce the space:

```
private Object[] shrink() {
    return Arrays.copyOf(this.elements, newLength: this.elements.length / 2);
}
```

- And last but not least the check index method, feel free to modify the message.

```
private void checkIndex(int index) {
    if (index < 0 || index >= this.size) {
        throw new IndexOutOfBoundsException(String.format("Index out of bounds: %d for size: %d", index, this.size));
    }
}
```

- Iterator<E>

```
@Nonnull
@Override
public Iterator<E> iterator() {
    return new Iterator<E>() {
        private int index = 0;
        @Override
        public boolean hasNext() {
            return this.index < size();
        }

        @Override
        public E next() {
            return get(index++);
        }
    };
}
```

All of the **other methods** are **easy** and **straightforward** to be **implemented** so you won't need any help. If it doesn't work the first time **simply try a different approach**.

2. Stack

Your task is to implement the **ADS AbstractStack<E>** inside the **Stack<E>** class provided. You have to **implement** all the **methods** to solve the problem, however, you are free to add more methods with any access modifier you want.

- **Push (E element)** – adds an element at the **top** of the stack and **increases** the **size**.
- **E pop ()** – **removes** an element at the **current top** of the stack and **returns it** if there is an element if the stack is **empty** throw **IllegalStateException** with an appropriate message.
- **E peek () – return** the element at the **current top** of the stack if the stack is **empty** throw **IllegalStateException** with an appropriate message.
- **Int size ()** – returns the **number** of elements inside the stack.
- **Boolean isEmpty ()** – returns if the stack **contains** any elements or **not**.

Solution:

- Push (E element)

```
@Override
public void push(E element) {
    Node<E> newNode = new Node<>(element);
    newNode.previous = top;
    top = newNode;
    this.size++;
}
```

- E pop ()

```
@Override
public E pop() {
    ensureNonEmpty();
    E element = this.top.element;
    Node<E> temp = this.top.previous;
    this.top.previous = null;
    this.top = temp;
    this.size--;
    return element;
}
```

- E peek ()

```
@Override
public E peek() {
    ensureNonEmpty();
    return this.top.element;
}
```

- Iterator<E>

```
@Nonnull
@Override
public Iterator<E> iterator() {
    return new Iterator<E>() {
        private Node<E> current = top;

        @Override
        public boolean hasNext() { return current != null; }

        @Override
        public E next() {
            E element = current.element;
            this.current = this.current.previous;
            return element;
        }
    };
}
```

All of the **other methods** are **easy** and **straightforward** to be **implemented**. If it doesn't work the first time **simply try a different approach**.

3. Queue

Your task is to implement the **ADS AbstractQueue<E>** inside the **Queue<E>** class provided. You have to implement all the methods to solve the problem, however, you are free to add more methods with any access modifier you want.

- **Offer (E element)** – **adds** an **element** at the **end** of the **queue** and increases the size.
- **E poll ()** – **removes** and **returns** the **first element** at the **queue** also **decreases** the **size** and **performs** a **check** if this **method** is called **upon empty collection** if so throw **IllegalStateException** with a message of your chose the message itself will not be tested.
- **E peek () – return** the element at the **current front** of the queue if the collection is **empty** throw **IllegalStateException** with an appropriate message.
- **Int size ()** – returns the **number** of elements inside the stack.
- **Boolean isEmpty ()** – returns if the stack **contains** any elements or **not**.

Solution:

As you can see a lot of the operations **described above** are a lot like those we did on the **Stack** problem so think about **how** you can **reuse** and **modify** those. Now you can see a **slightly different** way of adding the elements in the **stack implementation** we had a **pointer** to the **top element** here we have to **the first** pointer so you need to **find** the **last element** so you can **offer** the **new node**. The only specific problem here is the **poll ()** method:

```
@Override
public E poll() {
    ensureNonEmpty();
    E element = this.head.element;
    if (this.size == 1) {
        this.head = null;
    } else {
        Node<E> next = this.head.next;
        this.head.next = null;
        this.head = next;
    }
    this.size--;
    return element;
}
```

4. SinglyLinkedList

Your task is to implement the ADS List<E> inside the ArrayList<E> class provided. You can see that this class implements the List<E> interface you have to implement all the methods to solve the problem, however, you are free to add more methods with any access modifier you want.

- **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 the appropriate message.
- **E removeLast ()** – **removes and returns** the **last element** of the collection if **it** is **such** if **no** then **throw IllegalStateException** with the 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 the 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 the appropriate message.
- **Int size ()** – **returns** the **number** of **elements** inside the collection.
- **Boolean isEmpty ()** – returns if the collection **contains** any elements or **not**.

Solution:

Here comes the **tricky part**, **all** of the **operations** above **are alike** the **previous ones** you have implemented combined and modified of course. But in really small and simple matters so try to solve those on your own. **Good Luck!** And **remember** if something gets **too complicated** or **unclear** and **does not work** you can **always start again by choosing a different approach.**

"The presence of those seeking the truth is infinitely to be preferred to the presence of those who think they've found it." — Terry Pratchett, Monstrous Regiment