Advanced Object-Oriented Programming

CPT204 – Lab 8
Erick Purwanto

Xi'an Jiaotong-Liverpool University
西交利物浦大学

CPT204 Advanced Object-Oriented Programming

Lab 8

Linked List 4, Deque 3, Exception 2

Welcome !

- Welcome to Lab 8 !
  - We are going to implement deque using array: the ARDeque

- You will find in this lab
  1. Lab Exercise 8.1 - 8.4, and their hints
  2. Exercise 8.1 - 8.4

- Download **lab8** zip files from Learning Mall
- Don't forget to import the **lab8** files and the library into an IntelliJ project
  - Read **lab1** again for reference

ARDeque

- In this lab, we are going to implement **deque** using ***an array***
  - previously in Lab 5, 6, we implement deque using linked-list
- We will also use **generic types**, so that the deque can store *any type* of objects
- Here are the additional specifications:
  - The starting size/length of your array must be 4
  - Use the resizing : array doubling and array halving discussed in the lecture
    - before adding, double the size if it's full
    - after deleting, halve the size if it's less than or equal to a quarter full
  - Use ***circular array*** which defined in the next slides, followed by examples

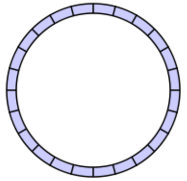
ARDeque

- In this lab, we are going to implement **deque** using ***an array***
  - previously in Lab 5, 6, we implement deque using linked-list
- We will also use **generic types**, so that the deque can store *any type* of objects

- Here are the additional specifications:
  - The starting size/length of your array must be 4
  - Use the resizing : array doubling and array halving discussed in the lecture
    - before adding, double the size if it's full
    - after deleting, halve the size if it's less than or equal to a quarter full
  - Use ***circular array*** which defined in the next slides, followed by examples
  - You may define and submit your own private helper method(s)
    - for example, private void **resize** (int capacity)

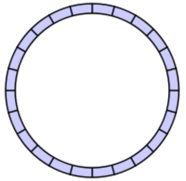
include in your LMO submission

Circular Array



- As the name suggests, think of the array a circular object
  - we keep two indices, **nextFirst** and **nextLast**
  - in the beginning, nextFirst + 1 *is* nextLast
  - when we addLast, we put the new item in index nextLast, then shift it to right circularly
  - when we addFirst, we put the new item in index nextFirst, then shift it to left circularly
  - delFirst and delLast is also set accordingly

Circular Array



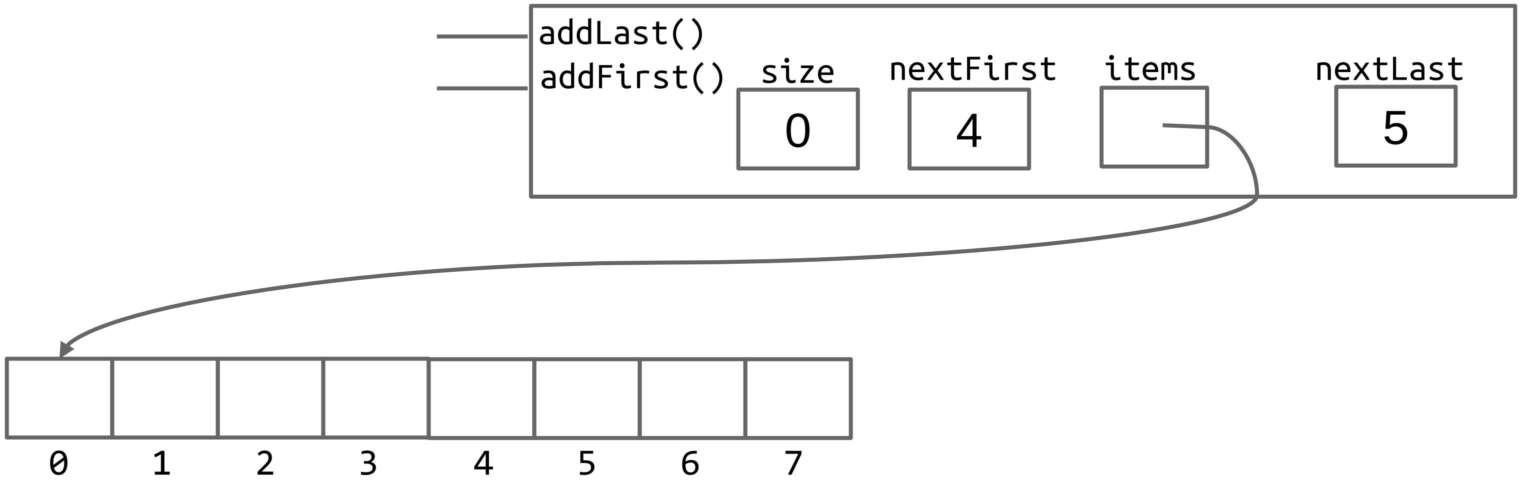
- As the name suggests, think of the array a circular object
  - we keep two indices, **nextFirst** and **nextLast**
  - in the beginning, nextFirst + 1 *is* nextLast
  - when we addLast, we put the new item in index nextLast, then shift it to right circularly
  - when we addFirst, we put the new item in index nextFirst, then shift it to left circularly
  - delFirst and delLast is also set accordingly

- In the example on the next slides,
  - we start with an empty array of length 8
  - nextFirst is 4
  - nextLast is 5

picked arbitrarily as long as following the rules above

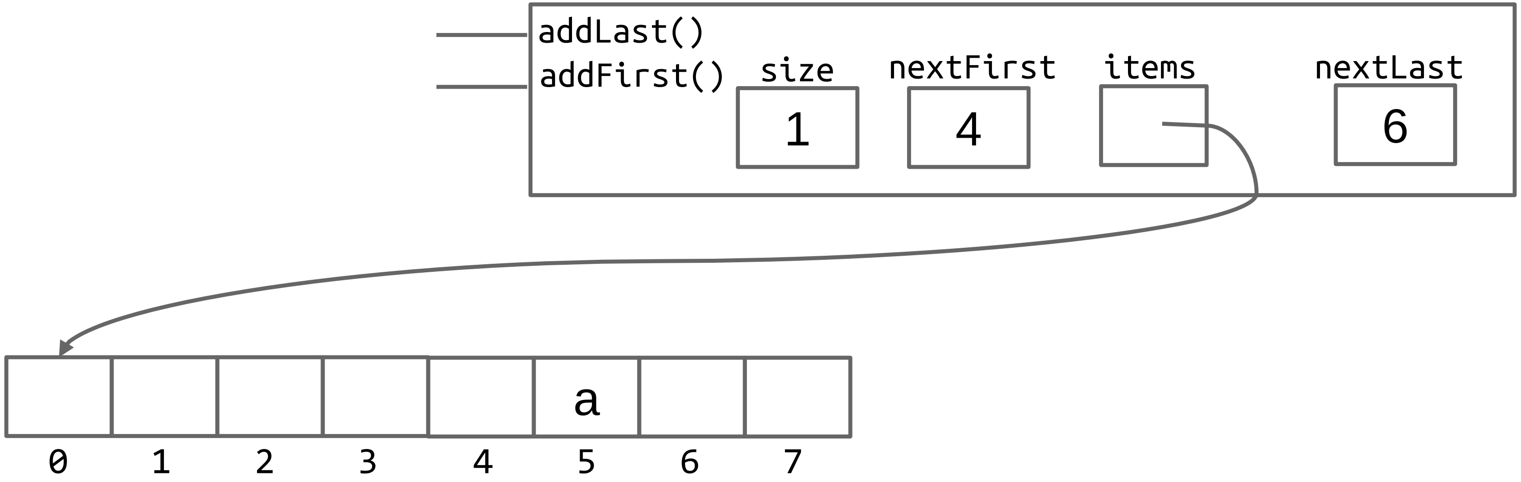
Circular Array Example

- In this example, the ARDeque<String> starts with an empty array items of length 8
  - addLast("a")



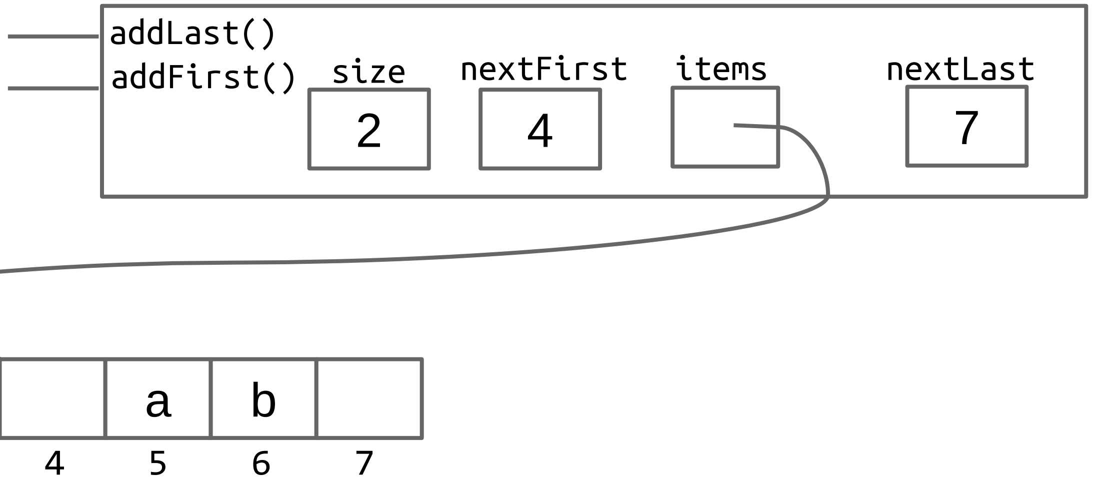
Circular Array Example

- In this example, the ARDeque<String> starts with an empty array items of length 8
  - addLast("a")
  - addLast("b")



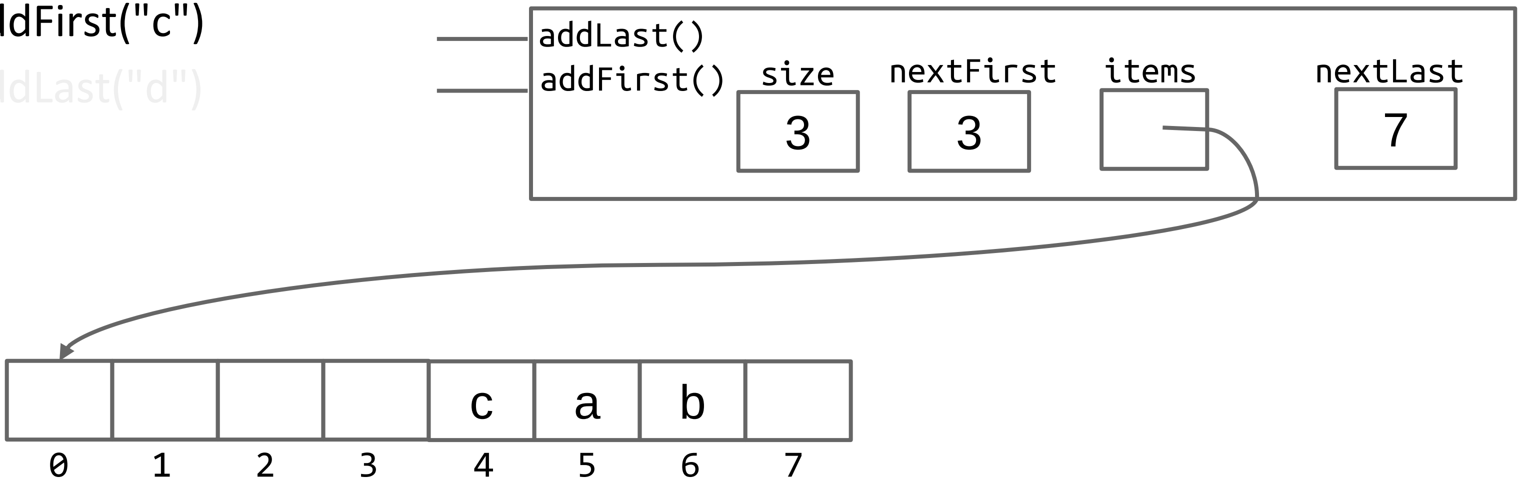
Circular Array Example

- In this example, the ARDeque<String> starts with an empty array items of length 8
  - addLast("a")
  - addLast("b")
  - addFirst("c")



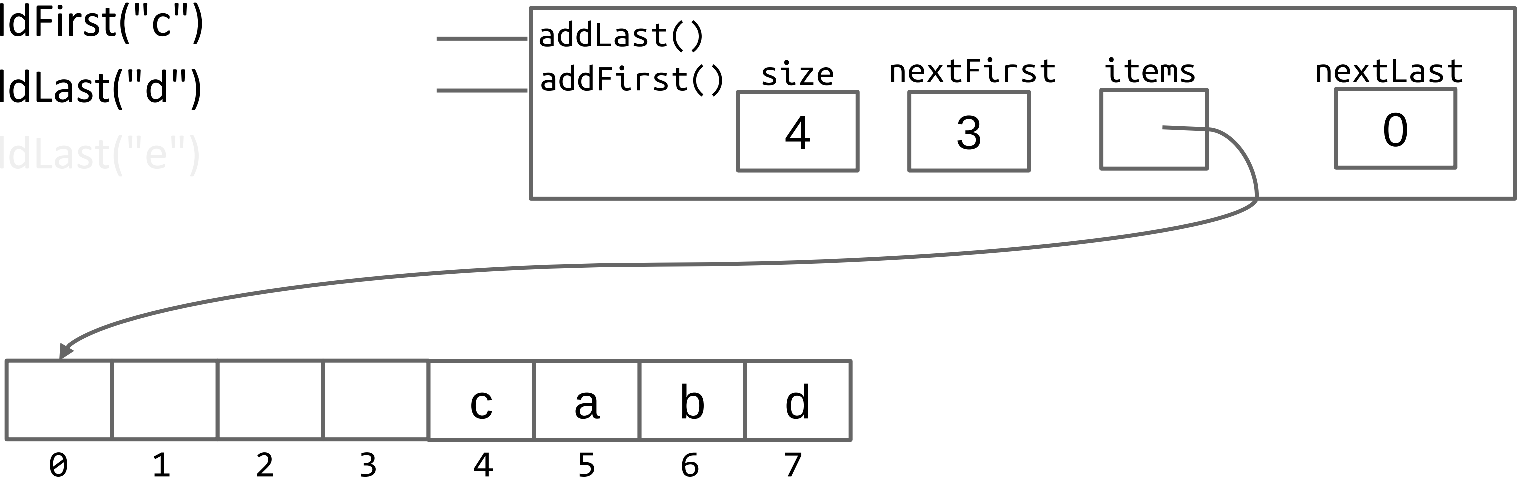
Circular Array Example

- In this example, the ARDeque<String> starts with an empty array items of length 8
  - addLast("a")
  - addLast("b")
  - addFirst("c")
  - addLast("d")



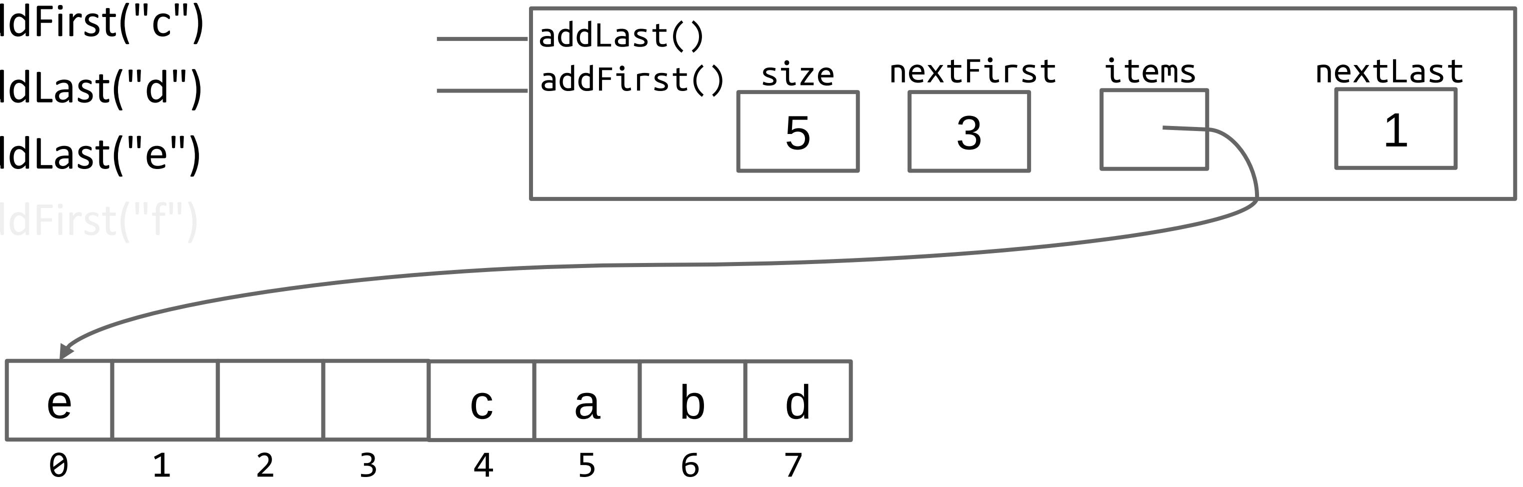
Circular Array Example

- In this example, the ARDeque<String> starts with an empty array items of length 8
  - addLast("a")
  - addLast("b")
  - addFirst("c")
  - addLast("d")
  - addLast("e")



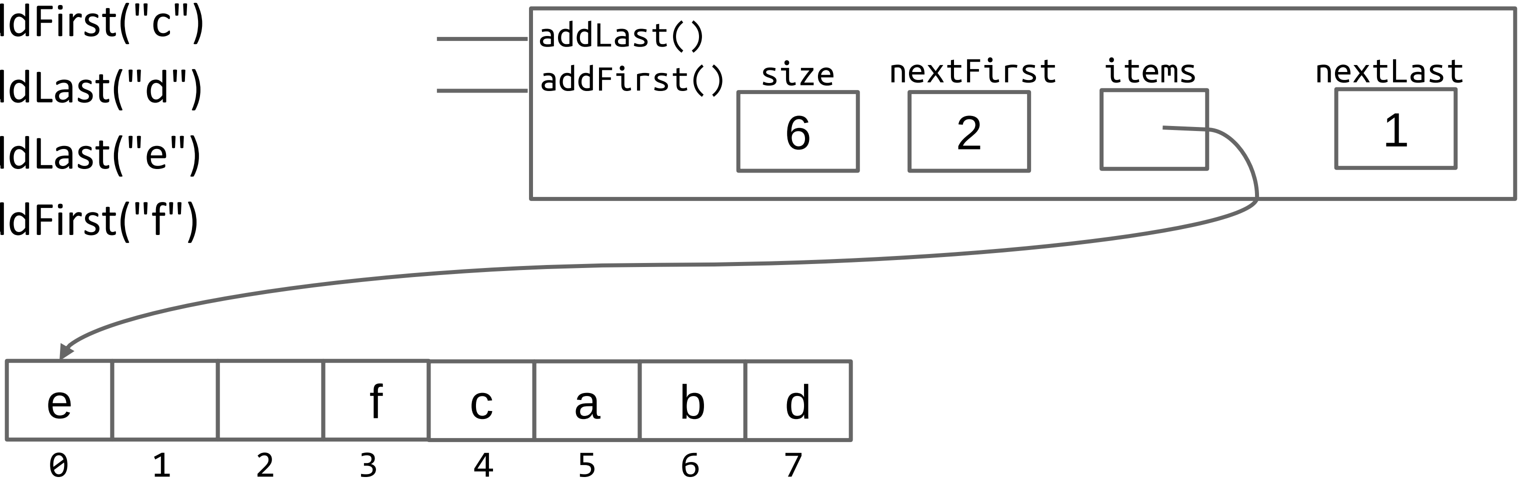
Circular Array Example

- In this example, the ARDeque<String> starts with an empty array items of length 8
  - addLast("a")
  - addLast("b")
  - addFirst("c")
  - addLast("d")
  - addLast("e")
  - addFirst("f")



Circular Array Example

- In this example, the ARDeque<String> starts with an empty array items of length 8
    - addLast("a")
    - addLast("b")
    - addFirst("c")
    - addLast("d")
    - addLast("e")
    - addFirst("f")



Lab Exercise 8.1 - 8.4

- Lab Exercise 8.1 ARDeque EMPTY CONSTRUCTOR
- Lab Exercise 8.2 ARDeque ADD TO BACK
- Lab Exercise 8.3 ARDeque PRINT ITEMS
- Lab Exercise 8.4 ARDeque GET ITEM

- Hint: Start without resizing/generic first, draw and design your code in paper, unit-test each method *separately* using JUnit, and debug using Java Visualizer

Test Case for Lab Exercise 8.1 - 8.4

- Test case 1:

```
ARDeque<String> deque = new ARDeque<>();
deque.isEmpty();                         →        true
deque.size();                            →        0
deque.itemsLength();                     →        4
deque.addLast("a");  deque.addLast("b");  deque.addLast("c");  deque.addLast("d");
deque.size();                            →        4
deque.itemsLength();                     →        4
deque.get(0);                            →        "a"
deque.get(1);                            →        "b"
deque.get(2);                            →        "c"
deque.get(3);                            →        "d"
deque.printDeque();                      →        "a b c d↵"
deque.addLast("e");
deque.size();                            →        5
deque.itemsLength();                     →        8
deque.get(0);                            →        "a"
deque.get(3);                            →        "d"
deque.get(4);                            →        "e"
deque.printDeque();                      →        "a b c d e↵"
```

Lab Exercise 8.1 ARDeque EMPTY CONSTRUCTOR

- Complete the empty deque constructor public **ARDeque**().
- It creates an empty deque.
- You have to start with an array of length 4.

WARNING: Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

Lab Exercise 8.1 ARDeque EMPTY CONSTRUCTOR Hints

- An empty deque is just an array of length 4, so let us code to create that!

- Initialize items with a new Object array of length 4, that is cast into array of T
- Set nextFirst and nextLast to valid indices following the setting of a circular array
    - Read page 6 of this lab notes
- Set size to 0

Lab Exercise 8.2 ARDeque ADD TO BACK

- Complete the method void **addLast**(T item).
- It adds an item of type T to the back of the deque.
- It must **not** use any loops or recursion, and
  each operation must take **constant time**, that is,
  it does not depend on the deque's size, *except* when resizing.

WARNING: Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

Lab Exercise 8.2 ARDeque ADD TO BACK Hints

- We need to place the item in the correct index in the array, but we may need to do resizing beforehand!
- If the array is full, we have to resize first, and do array doubling
  - it is better to define private helper method resize with input parameter the new capacity
  - it can be reused by other methods in the exercises/assignments
- Set the item to array items index nextLast
- Increment nextLast circularly
  - it can also be done using a private helper method, useful for others
- Increment the size

Lab Exercise 8.3 ARDeque PRINT ITEMS

- Complete the method void **printDeque**().
- It prints the items in the deque from first to last, separated by a space, ended with a new line.

WARNING: Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

Lab Exercise 8.3 ARDeque PRINT ITEMS Hints

- We need to go through every item in array items and print it
    - thus, we need to compute the real indices first

- The item starts after nextFirst
- The item ends before nextLast and there are size items
- Use while/for and print to display the items separated by a space
- Add a new line with println

Lab Exercise 8.4 ARDeque GET ITEM

- Complete the method T **get**(int index).
- It returns the item at the given index, where index 0 is the front.
- If no such item exists, throw an **IndexOutOfBoundsException** with message as in the test case 2 below.
- It must **not** use any loops or recursion, and it must **not** mutate the deque.
  Each operation must take **constant time**, that is, it does not depend on the deque's size.
- Test case 2:

```
ARDeque<String> deque = new ARDeque<>();
deque.addFirst("a");
try {
    deque.get(1);
} catch (IndexOutOfBoundsException e) {
    System.out.println(e.getMessage());    →  "Index 1 is not valid"
}
```

WARNING: Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

Lab Exercise 8.4 ARDeque GET ITEM Hints

- If the deque is empty, or if index is invalid (negative, greater or equal size), then throws an object of IndexOutOfBoundsException
    - pass the message into the constructor, including the invalid index
- Compute the real index in the array items
    - relative to the nextFirst, nextLast, length of items
- Return the item at the real index

Exercise 8.1 - 8.4

- Exercise 8.1 ARDeque ADD TO FRONT
- Exercise 8.2 ARDeque DELETE FRONT
- Exercise 8.3 ARDeque DELETE BACK
- Exercise 8.4 ARDeque COPY CONSTRUCTOR

- Hint: Start without resizing/generic first, draw and design your code in paper, unit-test each method *separately* using JUnit, and debug using Java Visualizer

Test Case for Exercise 8.1 - 8.4

- Test case 1:

```
ARDeque<String> deque = new ARDeque<>();
for (int i=0; i<8; i++) { deque.addFirst("test"); }
deque.size();                             →           8
deque.itemsLength();                      →           8
deque.addLast("test");
deque.size();                             →           9
deque.itemsLength();                      →           16
deque.addFirst("test");
deque.size();                             →           10
deque.itemsLength();                      →           16
for (int i=0; i<5; i++) { deque.delFirst(); }
deque.size();                             →           5
deque.itemsLength();                      →           16
deque.delLast();
deque.size();                             →           4
deque.itemsLength();                      →           8
deque.delFirst();
deque.size();                             →           3
deque.itemsLength();                      →           8
deque.delLast();
deque.size();                             →           2
deque.itemsLength();                      →           4
```

Exercise 8.1 ARDeque ADD TO FRONT

- Complete the method void **addFirst**(T item).
- It adds an item of type T to the front of the deque.
- It must **not** use any loops or recursion, and
  each operation must take **constant time**, that is,
  it does not depend on the deque's size, *except* when resizing.

Exercise 8.2 ARDeque DELETE FRONT

- Complete the method T **delFirst**().
- It deletes and returns the item at the front of the deque.
  If no such item exists, returns null.
- It must **not** use any loops or recursion, and
  each operation must take **constant time**, that is,
  it does not depend on the deque's size, *except* when resizing.

Exercise 8.3 ARDeque DELETE BACK

- Complete the method T **delLast**().
- It deletes and returns the item at the back of the deque.
  If no such item exists, returns null.
- It must **not** use any loops or recursion, and
  each operation must take **constant time**, that is,
  it does not depend on the deque's size, *except* when resizing.

Exercise 8.4 ARDeque COPY CONSTRUCTOR

- Complete the copy constructor public **ARDeque**(ARDeque<T> other).
- It creates a deep copy of other.
- Test case 1:

```
ARDeque<String> deque = new ARDeque<>();
deque.addFirst("a");
ARDeque<String> copyDeque = new ARDeque<>(deque);
deque.addFirst("x");
copyDeque.addFirst("y");
deque.get(0);                →       "x"
deque.get(1);                →       "a"
copyDeque.get(0);            →       "y"
copyDeque.get(1);            →       "a"
```

Thank you for your attention !

- In this lab, you have learned:
    - To create a data structure called deque using a circular array, complete with
        - resizing array technique: dynamically expanding and shrinking
        - fast constant-(amortized)-time methods
        - deep copy, and
        - unchecked exception