**Objective: Covers the usage of Stacks, Deques, Circular Queue and Linked lists. Adding and removing elements using these data structures.**

**NOTE:**

* Do not hard code any values.
* Check the sample output to know how the results need to be printed.
* Read every instruction carefully and follow them strictly.
* Do not change the name of the attributes, and methods given below.
* @author annotation must contain your full name for all the classes in this project.
* Use @override annotation for every relevant method in all the classes of your project.

**Overview:** There are three exercises for this week’s lab. Exercise 1 is relevant to **Stacks And Deques**. Exercise 2 is relevant to **LinkedLists**. Exercise 3 is relevant to Circular Queue. Three of them need to be done in the same project but different packages. The details for the same can be found below:

# Exercise 01: StacksDeques

All the classes related to StacksDeques must be placed in a package named “**stacksanddeques**”.

1. Classes: This package has one generic class named **AStack<E>**

Private instance variable

private ArrayDeque<E> myStack;

**Constructors and methods:**

* One no-arg constructor: initializes myStack to a new empty ArrayDeque of type E
* **push(E element)** – adds an element to the top of the stack; no return value
* **pop()** – removes and returns the top element
* **peek()** – returns, but does not remove, the top element
* **size()** – returns the current size of the stack
* **isEmpty()** – returns true if the stack is empty; false otherwise

1. Classes: This package has one main class named **BalancedParens**

* Create a new **AStack** containing objects of type Character, named parenStack.
* Read input from **expressions.txt**, which contains arithmetical expressions – one per line. **expressions.txt** is provided to you.
* For each expression that you read, determine whether the parentheses are correctly balanced.
* Parse the expression, character by character.
* If you encounter a left parenthesis, push it onto the stack.
* If you encounter a right parenthesis, pop the stack. If the parentheses are balanced correctly, there will always be something to pop.
* When the expression has been completely parsed, the stack should be empty.
* For each expression, write VALID if the parentheses are correctly balanced; otherwise, write INVALID, followed by an appropriate error message describing the specific problem, as shown in the sample output.

**Sample Output:**

|  |
| --- |
| (((( x + y - z: INVALID:  (x + y / c - d - 3)) + 14 \* (a + c) \* d): INVALID:  Trying to pop, but the stack is empty!  a + b - c + d: VALID  )a -b(: INVALID:  Trying to pop, but the stack is empty!  (a + b): VALID  a + (((b -c) -3) \* (4 \* a) / 3): VALID  (a + b: INVALID:  Parsing complete, but the stack is not empty!  a + b): INVALID:  Trying to pop, but the stack is empty!  (x + y / (c - d) - 3 + 14 \* (a + c) \* d): VALID  (x + y / (c - d - 3 + 14 \* (a + c) \* d) : INVALID:  Parsing complete, but the stack is not empty! |

**Exercise 02: LinkedList Implementation**

All the classes related to **LinkedList Implementation** must be placed in a package named “**linkedlists**”.

1. Unzip the “LinkedList.zip” to work on Exercise 02.
2. Create a linked list class. To distinguish it from the built-in **LinkedList** class, we will name this class ***Lastname*\_ALinkedList**.**java**
3. Place the given **Node.java** file in this package and will need to start the linked list implementation from this class.
4. Copy the stubs of **ALinkedList.java** to “***Lastname****\_***ALinkedList.java”** and add the working code.
5. By end of this you must complete the linked list implementation.
6. Place the courses.txt outside “**src**” folder of the project.
7. Unzip the **testerclasses.zip** which contains two main classes “**Tester.java**” and “**CoursesTester.java**”.
8. Run the “**Tester.java**” and check the respective sample output.
9. Place the “**Course.java**” file in the respective package and test the “**CoursesTester.java**”.

**Sample Output for Tester.java:**

|  |
| --- |
| Contents of linked list  55  25  17  Deleting 55  Deleting 25  Deleting 17  Contents of linked list  list is empty |

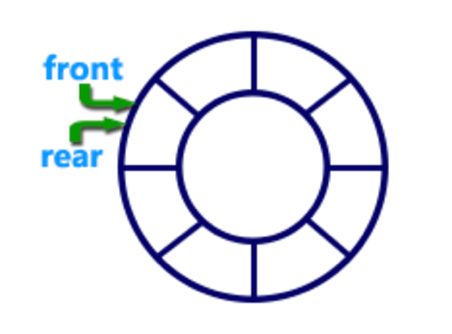
Sample Output for CoursesTester.java

|  |
| --- |
| 17120 4 Calculus I  44543 3 Mobile Computing  44525 3 Theory & Implementation of Programming Languages  44542 3 Object-Oriented Programming  44242 3 Data Structures  44241 3 Computer Programming II  44296 3 Professional Development Seminar  44141 3 Computer Programming I  8 courses in the list  deleting 17120 4 Calculus I  7 courses in the list |

**Exercise 03: Circular Queue**

**Scenario:**

Use an array of 20 integer values to represent a circular queue. Values are added to the queue at the rear and removed at the front of the queue. The index of the head and tail elements are stored in appropriate variables. You can think of a circular queue as an array that bent into a circle so that the beginning and end of the array are joined.



When a value is added to the circular queue, it is inserted at the rear and the rear pointer is increased by one. If the rear pointer was referencing the last element in the array, then increasing its value by one results in it wrapping to the beginning of the array.

When a value is removed from the circular queue, it is removed from the front of the queue and the front pointer is increased by one (also wrapping back to the beginning of the array if it currently references the end of the array.

A circular queue can hold as many values as the length of the array (20 int values in our case).

A queue is full when the array holds 20 elements.

The queue is empty with the array holds 20 elements

You can read about a circular queue (also called a circular buffer) at <https://en.wikipedia.org/wiki/Circular_buffer>.

In this part of the lab you need to design, build a test a circular queue of 50 integer values. It needs to have the following methods (you need to determine the parameters needed):

* Insert: add an element to the circular queue if it is not already full; throw an exception otherwise.
* Remove: remove the element at the front of the queue if it is not empty; throw an exception otherwise.
* Retrieve: return the value at the front of the queue if it is not empty, but do not remove it; throw an exception otherwise.
* Length: Returns the number of elements in the circular queue.
* Print: Print the contents of the queue with the front of the queue shown first. [THIS MUST BE DONE RECURSIVELY].
* IsFull: Returns a Boolean value indicating if the queue is full or not.
* IsEmpty: Returns a Boolean value indicating if the queue is empty or not.

Make sure you develop a main method that demonstrates that each of the methods above work properly and that the queue properly handles attempts to insert an element into a full queue and remove an element from an empty queue.

**Sample Output:**

|  |
| --- |
| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Circular Queue  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Length of the circular queue is:  The length of the queue is 20  Queue Elements:  20  23  22  1  3  4  6  8  5  15  14  9  12  87  13  34  83  17  18  19  Removing element from circular queue  Length of queue after removal  The length of the queue is 19  Element removed |

**Submit your solution by following the steps below:**

* Save your files in NetBeans.
* Zip your entire Project. (It should be called **Lastname\_lab12DataStructures** where Last name is your last name.)
* Submit the zip file to the **lab12DataStructures** dropbox.
* Download the zip file which you have submitted.
* Look into the zip file and verify the class files, and Javadoc in the zip folder are updated. If not, resave your project in NetBeans and resubmit.