Koç University
COMP202
Data Structures and Algorithms
Assignment 2

Instructor: Barış Akgün
Responsible TA: Buket Yüksel
Due Date: March 18 2018, 23:59
Submission Trough: Blackboard
Relevant Programming Session: March 8 2018 17:30-18:45 at SOS180/CASEZ48

This programming assignment will test your knowledge and your implementation abilities of what you have learned in the Lists, Stacks, and Queues parts of the class.

This homework must be completed individually. Discussion about algorithms and data structures are allowed but group work is not. Any academic dishonesty, which includes taking someone else's code or having another person doing the assignment for you will not be tolerated. **By submitting your assignment, you agree to abide by the Koç University codes of conduct.**

Description

This assignment is designed to assess your understanding of Arrays, Linked Lists, Stacks, and Queues and your implementation abilities. This assignment requires you to implement the *Double Ended Queue - Deque* data structure. Deque, which is pronounced as "deck", is a generalization of the queue data structure that supports inserting and removing elements from either the front or the back of the data structure. The assignment will have two phases; (1) implementation of the data structures and (2) application of the data structures.

In the first part of the assignment, you will implement the deque ADT both using **Arrays** and **Doubly Linked Lists**. You will be given a deque interface and two starter files. All of these three files have comments for your convenience. You are also given another interface for a simple container. This interface specifies three operations; (1) push, (2) pop and (3) peek. The details are in the comments but you can figure out that these methods resemble stack and queue operations. You are going to implement a stack and a queue that implement this interface. However, you will be required to implement these by using a deque; by wrapping the dequeue operations to achieve the desired input-output behavior. The type of deque will be given as a generic. See the relevant files and the main file for how they are used.

The code has comments about the assignment. Majority of what you need to do is actually in these comments so make sure you read them carefully. You are provided with a java project which contains following files. Bold ones are the ones you need to modify and they are placed under the *code* folder, with the rest under *given*.

- *iDeque.java*: This is the interface that shows you the operations of the Deque ADT. You do not need to modify this file. The ArrayDeque and LLDeque classes implements this interface. Make sure you pay attention to the comments. **Additions coming with part 2:** Now `iDeque` interface extends the `Iterable` interface and defines a `clear` method that clears up the storage.
- ***ArrayDeque.java***: You need to code the `ArrayDeque` class that implements the `iDeque` interface using a *Dynamic Array* in this file. Follow the instructions given in the code. **Additions coming with part 2:** See the additions for `iDeque`. **Bonus:** The comments had the statement that if you were able to return the contents of the deque from front to back, than you would get a bonus. This still stands but it is trivial if you successfully implement the iterator!
- ***LLDeque.java***: You need to code the `LLDeque` class that implements the `iDeque` interface using a *Doubly Linked Lists* in this file. Follow the instructions given in the code. **Additions coming with part 2:** See the additions for `iDeque`.

- *iSimpleContainer.java*: This interface has three simple methods of a container that allow manipulation from a "single point". You do not need to modify this file but make sure to read its comments. **Additions coming with part 2:** Three new trivial to implement methods are added: `size, isEmpty, clear/`
- ***Stack.java***: You need to code the `Stack` class that implements the `iSimpleContainer` interface using a generic deque in this file. Look at the code to see how this is setup. This must have the last-in first-out behavior. **Additions coming with part 2:** See the additions for `iSimpleContainer`.
- ***Queue.java***: You need to code the `Queue` class that implements the `iSimpleContainer` interface using a generic deque in this file. Look at the code to see how this is setup. This must have the first-in first-out behavior. **Additions coming with part 2:** See the additions for `iSimpleContainer`.
- *Main.java*: Gives printouts for you to check your implementation. Will change after the release of the second phase and the release of the autograder. You can compare the outputs with the *example outputs.txt* file. **Additions coming with part 2:** Updated.
- *Util.java*: This file includes utility functions. Do not worry about it, you will not need to use anything from this file in this assignment.

In the second part of the assignment, you are going implement an algorithm, given below, that finds the exit in a maze given a start coordinate. This algorithm uses containers which changes its behavior. If a stack is used the algorithm acts as a depth-first search and if a queue is used, it acts as a breadth-first search. The idea is to start from the start state and systematically search the maze. The containers hold the next states to be checked. The mazes are given to as txt files where characters define the whether a cell is a wall, empty, a start state or an end state. The algorithm is as below:

Algorithm 1: solveMaze

```
Input: Maze (maze), initially empty container (sc), initially empty visited node storage
  (deque)
Output: Wheter a path to an exit cell is found, the filled visited node storage
sc.push(maze startState)
while sc is not empty do
    currentState = sc.pop()
    if currentState is an exit state then
        return true
    else if currentState is not visited then
        mark currentState as visited
        deque.addBehind(currentState) foreach empty neighbor of currentState do
            sc.push(neighbor)
        end
end
return false
```

In addition to changes to the interfaces and the *Main.java*, you have a new file that you need to implement:

- ***Maze.java***: This file holds the maze information. Go over its comments carefully and understand what it does. You are going to fill in the `solveMaze` method. We defined additional methods to be helpful but they are not going to be checked.

Grading

Your assignment will be graded through an autograder. Make sure to implement the code as instructed, use the same variable and method names.

A version of the autograder is also released to you. Our version will more or less be similar, potentially including more tests or tests with additional reference implementations of Stack and Queue. Since this release follows an initial upload, make sure you are working on the correct version. Suggestions on how to integrate the autograder is included as a text file. **Important:** Make sure to change the visibility of the constructor of the nested `Coordinate` class inside *Maze.java* to `public`. Update package names if they give you trouble. Run the new main program in the *Autograde.java* to get the autograder output and your grade.

In case the autograder fails or gives you 0 when you think you should get more credit, do not panic. Let us know. We can go over everything even after your submission.

Submission

You are going to submit a compressed archive through the blackboard site. The file can have *zip*, *tar*, *rar*, *tar.gz* or *7z* format. This should extract to a folder with your student ID without the leading zeros. This folder should contain *ArrayDeque.java*, *LLDeque.java*, *Stack.java*, *Queue.java* and the other designated files (after the release of the second phase). Other files, which you should not have modified anyways, will be deleted.

Submission Instructions

- You are going to submit a compressed archive through the blackboard site.
- The file can have *zip*, *tar*, *tar.gz* or *7z* format.
- This compressed file should extract to a folder with your student identification number with the two leading zeros removed which should have 5 digits.
- The previous point is very important, I do not want to see multiple folders (apart from operating system ones such as MACOSX or DS Store). I do not want to play inception with your code.
- Inside the folder, you should only have *ArrayDeque.java*, *LLDeque.java*, *Stack.java*, *Queue.java* and the other designated files (after the release of the second phase). Anything else will be deleted. Make sure your code runs with the other provided files that should not have been modified.
- Code that does not compile will receive 0 credits!
- Do not trust the way that your operating system extracts your file! They will mostly put the contents inside a folder with the same name as the compressed file. We are going to call a program (based on your file extension) from the command line. The safest way is to put everyting inside a folder with your ID, then compress the folder and give it your ID as its name.
- One advice is after creating the compressed file, move it to your desktop and extract it. Then check if all the above criteria is met.
- Once you are sure about your assignment and the compressed file, submit it through Blackboard.
- **DO NOT SUBMIT CODE THAT DOES NOT TERMINATE OR THAT BLOWS UP THE MEMORY.** I will take these as malicious acts and will proceed accordingly.

Let us know if you need any help with setting up your compressed file. This is very important. We will put all of your compressed files into a folder and run multiple scripts to extract, cleanup, grade and do plagiarism checking. If you do not follow the above instructions, then scripts might fail. This will lead you to get a 0.