Assignment 2 – Stacks and Queues

CSDS 233 – Introduction to Data Structures

General instruction: This assignment includes two parts, written and programming. Please write/type your answers neatly so they can be readable. Please submit a single PDF file for the written part and a zip file for the programming part before **11:59 P.M. on February 23th, 2024.** Please ensure that your written assignment is outside the zip file.

Office Hours + Additional Office Hours

Emma Chio (exc513@case.edu): Monday, 3:30 - 4:30pm
Tammy Lin (txl619@case.edu): Wednesday, 10:30am - 11:30am
Additional Office Hours:

- Friday February 16th, 11:00am-12:00pm → ZOOM
- Wednesday February 21st, 1pm - 2pm → ZOOM

Best of luck!

Written Assignment 30 pts

1. Big-O Notation (18pts)

Find tightest $O(f(n))$ for each of the following functions: The tightest big-O bound is the narrowest upper bound within the big-O category.

Example: n = O(n^2), n = O(n^4).. and so on, but the tightest bound would be n = O(n)

Example: $f(n) = 5n^2 + 2n + 1 = O(n^2)$

a. $f(n) = 3n$

b. $f(n) = \frac{log(n)}{n^2}$

c. $f(n) = n \times logn$

d. $f(n) = n + \frac{n}{2} + \frac{n}{4} + \ldots + \frac{n}{2^n}$

e. $f(n) = (log(n))^n + n^4$

f. $f(n) = \frac{n! + n^n}{3n}$

True/False.

Show you work by using the definition of big-O and finding values for c and N.

Reminder $f(n)$ is $O(g(n))$ — if a positive real number c and positive integer N exist such that $f(n) \leq c \times g(n)$ for all $n \geq N$

g. $2^{n-1} = O(n)$

h. $n(logn)^3 = O(n^{4/3})$

i. $\frac{n^4+1}{n^2} = O(n)$

2. Algorithmic Analysis (12pts)

Given the following code, analyze and give the tightest big-Θ bound. Show how you came to your answer by indicating what the big-Θ is for each line.

a.
```
public static int sum1() {
      int sum = 0;
      for(int i = 0; i < n; i++) {
           if(sum < n) {
                for(int j = 0; j < n; j++) {
                     sum++;
                }
           }
      }
      return sum;
}
```

b.
```
public static int sum2() {
      int sum = 0;
      for(int i = n; i > 1; i = i/3) {
           sum = sum + 2;
      }
      return sum;
}
```

c.
```
public static int sum3() {
      int sum = 0;
      for(int i = 0; i < n; i++) {
           for(int j = 0; j < n; j++) {
                if(i < j) {
                     for(int k = i; k < j; k++) {
                          sum++;
                     }
                }
           }
      }
      return sum;
}
```

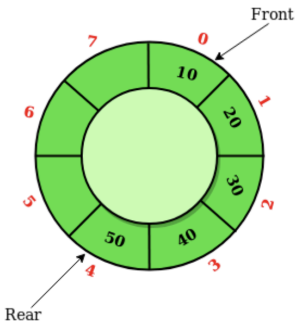
Programming Assignment 70 pts

public class SuperDeque<E>

Write a class called **SuperDeque** where it has the properties of both a stack (first in, last out) and a queue (first in, first out) depending on which method is being called. Implement **SuperDeque** using an array.

Brief Overview

The implementation should work similarly to a circular queue, essentially this means that there should be two global fields keeping track of the front/top and back/bottom index rather than shifting the array elements.



When an element is inserted at the front, the front index should go back to 7 in this example. On the other hand, if an element is removed from the front, the front index should be incremented and be 1. You should use the mod (%) operator to loop around the data structure. In the event that the data structure fills up, you should double the size of the array and maintain the elements stored in the data structure.

Required Constructor

```
public SuperDeque(){
      this.dq =  (E[]) new Object[DEFAULT_CAP];
      this.front = 0;
      this.back = 0;
}
```

Required fields:

```
private int front;
```

- The index of the array that represents the front/top of the data structure.

```
private int back;
```

- The index of the array that represents the back/bottom of the data structure.

`private E[] dq;`

- The array storing the data of the **SuperDeque**

`private static final int DEFAULT_CAP = 1;`

- The initial size of the **SuperDeque** to be set in the constructor

Required methods:

`public void push(E element)`

- In this case, the **SuperDeque** should work as a stack and the method works to add an element to the top of the **SuperDeque**.

`public E pop()`

- In this case, the **SuperDeque** should work as a stack and the method works to remove the element at the top of the **SuperDeque** and returns the element that was removed. If the **SuperDeque** is empty, return null.

`public E peek()`

- Returns the front of the **SuperDeque**, but does not remove it from the data structure. If the **SuperDeque** is empty, return null.

`public void enqueue(E element)`

- In this case, the **SuperDeque** should work as a queue and the method works to add an element at the back of the **SuperDeque**

`public E dequeue()`

- In this case, the **SuperDeque** should work as a queue and the method works to remove the element at the front of the **SuperDeque.** If the SuperDeque is empty, return null.

`public boolean isEmpty()`

- Checks if there are any elements in the **SuperDeque**

`private void doubleSize()`

- Double the size of the array. This function should be called when the array is full and the size needs to increase to accommodate adding more elements

`public String toString()`

- Prints the deque as a string in the format of "Element1, Element2, Element3…" with a comma and a space separating each element.
- Example: `toString()` returns "`1, 2, 3, 4, 5`"

QueueStackProblems Class

`public static int evaluatePostFix(String postfix)`

- **Without using** `enqueue` and `dequeue` from **SuperDeque**, evaluate the postfix expression and return the value.
- Example: `evaluatePostFix(`"27 3 9× + 1−"`)` returns 53

`public static String reverseWords(String s)`

- **Without using** `enqueue` and `dequeue` from **SuperDeque**, reverse the contents of the queue. This function should reverse the word in a string.
- Example: `reverseWords(queue, "This is a string. Hello World!")` returns "`sihT si a .gnirts olleH !dlroW`"

`public static <E> SuperDeque<E> reverseK(SuperDeque<E> dq, int k)`

- Reverse the first K elements of the `SuperDeque dq` using any of the functions in the **SuperDeque**. If k >= size of dq, reverse all of the elements.
- Example:
  `SuperDeque<String> dq = new SuperDeque<>();`
  // add elements to dq → `{1, 2, 3, 4, 5, 6, 7}`
  `reverseK(SuperDeque<E> dq, 4)` returns dq containing `{4,3,2,1,5,6,7}`

`public static int playGame(int n, int offset)`

- **Without using** `pop` and `push` from **SuperDeque**, return the winner of the game. The game is as follows there are `n` number of players. For each iteration, the player at index `current position + offset` is removed. After `n - 1` iterations, there should be one player remaining. Return the id of the player. The parameter offset will always be positive. If `n <= 0`, return -1. Always start the game at index 0.

Example: `playGame(6, 3)` returns `5`

- Iteration 1: 1 | 2 | 3 | 4 | 5 | 6
  - Current position is the first player; thus, we start at player 1, count 3 spaces forward and end at player 4. Remove player 4.
- Iteration 2: 1 | 2 | 3 | 5 | 6
  - Current position is where player 4 was, then count 3 places. We would end at player 7, but since there are not 7 players, we will have to loop back around to the first player. Thus, we went to player 6, 1, and 2. Remove player 2.
- Iteration 3: 1 | 3 | 5 | 6
  - Current position is where player 2 was, and count 3 places. We will go to player 5, 6, and 1. We end at player 1. Remove player 1.
- Iteration 4: 3 | 5 | 6
  - Current position is where player 1 was, and count 3 places. We will go to player 5, 6, and 3. We end at player 3, thus remove player 3.
- Iteration 5: 5 | 6
  - Current position is where player 3 was, and count 3 places. We will go to player 6, 5, and back to 6. Since we ended on player 6, we will remove player 6.
- Iteration 6: 5
  - Since player 5 is the only player remaining, return player 5.

Note: When an index doesn't exist anymore you want to use the mod operator to loop back to the first player.

JUnit Tests

For each function provide at least 3 test cases for each function in QueueStackProblems (small, medium, large input sizes) that test the correctness of your implementation. This should not be done in the main function, but rather a test file should be created. For the SuperDeque, you can just test general functionality within 1-2 test functions.

Additional Notes

- You are **NOT** allowed to use Java defined `Lists`, `LinkedList`, `ArrayList`, `Queues`, etc. while implementing the `SuperDeque`. You should be using an array.

- If your `SuperDeque` is not working fully, you may use built-in Java `Stack` and `Queue` to implement the `QueueStackProblems` Class, but you may only use the functions listed in the assignment.

- There may be different ways to implement each function, but you must use a SuperDeque in some way for each problem in the `QueueStackProblems` Class for practice.

- You are not allowed to change the function headers, class headers, etc. But feel free to add private helper functions and fields.

- `this.dq = (E[]) new Object[DEFAULT_CAP];` If you are curious why we need to set up the constructor in this way, Java does not allow us to make generic arrays as the space in memory for the array needs to be allocated during compile time. However, we will not know what type `E` is until runtime, thus creating an Object array then casting it is the only option. This is just a quirk of Java, you will not need to do anything else weird in your code to create the SuperDeque.

Rubric

[30 pts] SuperDeque
[30 pts] StackQueueProblems
[5 pts] jUnit
[5 pts] Code cleanliness, proper encapsulation and abstraction

- Fields and helper functions should be private, and all complex blocks of code (if, for, while, etc) should have a comment above them.