built on 2023/05/03 at 20:40:55 ***due:*** *wed may 17 @ 11:59pm*

This assignment will give you practice with class design, references, and object-oriented programming. You will implement a few Java programs, test them thoroughly, and hand them in. There is a starter package, which you must download to begin working on this assignment.

For each task, write helper functions as you see fit. Strive to write code that promotes clarity.
When you hand in your work, your zip file will contain the following files:

```
MinMax.java
dance/Competitor.java
dance/DanceCompetition.java
dance/ScoreKeeper.java
mydeque/LinkedListDeque.java.
mydeque/ArrayDeque.java
```

Collaboration

We interpret collaboration very liberally. You may work with other students. However, each student ***must*** write up and hand in his or her assignment separately. Let us repeat: You need to write your own code. You must not look at or copy someone else's code. You need to write up answers to written problems individually. The fact that you can recreate the solution from memory will be taken as proof that you actually understood it, and you may actually be interviewed about your answers.

Be sure to indicate who you have worked with (refer to the hand-in instructions).

Logistics

We're using a script to grade your submission before any human being looks at it. Sadly, the script is not as forgiving as we are. *So, make sure you follow the instructions strictly.* It's a bad omen when the course staff has to manually recover your file because the script doesn't like it. Hence:

- Save your work in a file as described in the task description. This will be different for each task. **Do not save your file(s) with names other than specified.**
- You'll zip these files into a single file called `a2.zip` and you will upload this one zip file to Canvas before the due date.
- Attempt to solve each task on your own first. After a day, if you still can't solve a task, come to office hours.
- Before handing anything in, you should thoroughly test everything you write.
- The course staff is here to help. We'll steer you toward solutions. Catch us in real-life or online.

Task 1: Min and Max (4 points)

For this task, save your work in `MinMax.java`

Consider the following problem: given an array of n numbers, we want to find both the minimum and the maximum of these numbers. For such a problem, we often measure the cost in terms of the number of comparisons made—that is, if we compare any two numbers from the input, that's one comparison.

As an example, the following algorithm requires $n-1$ comparisons:

```
// assume a.length > 0
int maxArray(int[] a) {
    int maxSoFar = a[0];
    for (int i=1;i<a.length;i++) {
        if (a[i] > maxSoFar)
            maxSoFar = a[i];
    }
    return maxSoFar;
}
```

This is because in an array a of length n, *only* $a[1], a[2], \ldots, a[n-1]$ are compared with our `maxSoFar` in the `if` statement. Notice that $a[0]$ is *not* involved in the `if` statement.

You can use this algorithm to the find the maximum value and an almost-identical algorithm to find the minimum value. However, you'll need $2n-2$ comparisons ($n-1$ for max and another $n-1$ for min). *Only comparisons between input integers (either directly or indirectly) matter here,* which is why we don't count comparisons made by `i<a.length` in the above example. Your goal in this problem is to do better!

Your Task: First, implement a function

```
public static double minMaxAverage(int[] numbers) {
    // your code goes here
    int myMin = ...;
    int myMax = ...;
    return (myMin + myMax)/2.0;
}
```

that takes in an array of integer numbers, finds the minimum and the maximum among these numbers, and returns the average of the minimum and the maximum (as the code above shows). For full credit, if input contains n numbers, your function must use ***fewer*** than $3n/2$ comparisons.

Then: As comments in your code file, make a logical argument for why your code is indeed using fewer than $3n/2$ comparisons.

(Hint: Remember the maximum-number problem from class? What happens after one round in the pairing-up algorithm?)

Task 2: Dance Competition (8 points)

For this task, save your work in `dance/`

You have been hired by DSA to implement a system for a dance competition. This season everything will happen on Zoom, so they want an online system.

For a brief overview, there will be multiple competitors (players) competing this time. Each will have to complete two styles of dances: popping dance and hip-hop dance. They will be scored by a panel of

judges. Our system will keep track of scores and aggregate them to determine the gold medalists (i.e., winners) of the competition. Full detail appears in the program specification in the starter code.

The ~~evil~~ dance sorcerer association (DSA) wants you to implement the following *three* classes, which they have carefully designed and scaffolded (You're advised to complete them in this order):

- The `ScoreKeeper` class stores scores and answers basic statistical average about the scores. It is intended to be use as part of another class for score storage.
- The `Competitor` class represents a competitor in a dance competition. It keeps attributes related to the competitor, including the competitor's scores from dances.
- The `DanceCompetition` class represents a dance competition that involves multiple competitors. As mentioned before, two types of dances have been planned: popping dance and hip-hop dance. The final score for each competitor is computed using a weighted average. This class stores the weights as set in the constructors.

Multiple Files? For command-line folks: to compile interdependent Java classes, go into the corresponding folder and issue `javac *.java`, which will compile *all* Java files in that folder. You can still selectively run a class by running `java className`, where `className` is the name of the class you wish to run its `main`.

IntelliJ/VSCode peeps should make a brand-new project for this task to avoid your code inadvertently getting swept into a package.

Step 0: Implement the `ScoreKeeper` Class

The `ScoreKeeper` class stores scores and answers basic statistical average about the scores. Specifically, it internally keeps the scores that have been set. To set the scores, call the `setScores` method. To retrieve the scores, use the `getScores` method.

To find the calibrated average of the score, call the `getCalibratedAverage` method. The calibrated average is the average of the scores after excluding the minimum and maximum value.

You will look into `ScoreKeeper.java` and complete the code. We have provided the `main` method that demonstrates basic use of the class and below is the expected output:

```
=== ScoreKeeper ===
scoreKeeper0: [2.5, 1.0, 9.8, 5.4, 3.3, 0.25, 4.25]
Calibrated Average: 3.2900000000000005

scoreKeeper1: [0.5, 9.0]
Calibrated Average: NaN
```

Step 1: Implement the `Competitor` Class

The class `Competitor` keeps the scores that a competitor is given by the judges. Aside from setter- and getter- methods, the class provides methods for printing out raw scores from each dance type, as well as methods to compute averages.

Most notably, the method for computing the total dance score of a competitor, `getTotalDanceScore`, returns the weighted average of the dance scores. The weights are specified in `competition` object as follows.

Let

```
p = competition.getPoppingDanceFraction();
h = competition.getHipHopFraction();
```

If, however, the `competition` object is **`null`**, use `p = 0.6` and `h = 0.4`. Then, the weighted average is

```
p * (popping dance calibrated average) + h * (hip-hop dance calibrated average)
```

Remember that the calibrated average of a dance is the average computed after excluding the min and the max. This is conveniently what is computed by the `ScoreKeeper` class.

You will look into `Competitor.java` and fill in the missing code. We have provided the `main` method that demonstrates how to use the class, and below is the expected output:

```
== Peter C: Scores ===
[Popping]
[9.0, 8.0, 8.5, 9.5, 8.0, 7.5]
AVG Popping Dance Score: 8.375

[Hip Hop]
[10.0, 9.0, 9.5, 8.0, 8.5, 9.0]
AVG Hip Hop Dance Score: 9.0

Total Score: 8.625
========================

=== Monkey D: Scores ===
[Popping]
[10.0, 9.0, 9.5, 10.0, 9.0, 9.5]
AVG Popping Dance Score: 9.5
[Hip Hop]
[10.0, 9.0, 9.0, 9.0, 9.5, 9.5]
AVG Hip Hop Dance Score: 9.25

Total Score: 9.4
========================
```

Step 2: Implement the `DanceCompetition` Class

Finally, you will work on `DanceCompetition`, which represents a dance competition involving multiple competitors. This class keeps track of the weights for different dance types and maintains a list of competitors. Additionally, it provides a method `getGoldMedal`, which returns the list of gold medalists (defined as every competitor whose total score is at least 8.0).

You will look into `DanceCompetition.java` and complete it. We have provided the `main` method that demonstrates how to use the class, and below is the expected output:

```
== ALL Competition's Scores ==
'Locking J': 8.612499999999999
'Breaking B': 4.8374999999999995
'Popping C': 9.424999999999999
==============================
'Locking J' get a gold medal with score = 8.612499999999999
'Popping C' get a gold medal with score = 9.424999999999999
```

Task 3: Deque Using Doubly Linked Lists and Arrays (8 points)

For this task, save your work in `mydeque/`

In this problem, you will build implementations of a "double-ended queue" using both lists and arrays. This extends the discussion of our linked list and array list from class. You can look at a more detailed explanation in Chapter 5 of the book[1].

(The next assignment will deal with repackaging it in a more Java proper way.)

The Deque API

The *double ended queue* is similar to the linked list and array list data structures that you have seen in class. For a more authoritative definition (from cplusplus.com), a *deque* (usually pronounced like "deck") is an irregular acronym of double-ended queue. Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends (either its front or its back).

For our needs, any deque implementation must have exactly the following operations:

```
// Adds an item of type T to the front of the deque.
public void addFirst(T item)

// Adds an item of type T to the back of the deque.
public void addLast(T item)

// Returns true if deque is empty, false otherwise.
public boolean isEmpty()

// Returns the number of items in the deque.
public int size()

// Returns a string showing the items in the deque from first to last,
// separated by a space.
public String toString()

// Removes and returns the item at the front of the deque.
// If no such item exists, returns null.
public T removeFirst()

// Removes and returns the item at the back of the deque.
// If no such item exists, returns null.
public T removeLast()

// Gets the item at the given index, where 0 is the front, 1 is the next item,
// and so forth. If no such item exists, returns null. Must not alter the deque!
public T get(int index)
```

Your class should accept any generic type (not just integers).

Linked List Deque

For this part, save your work in a file called `LinkedListDeque.java`.

Your task is to build a `LinkedListDeque` class, which will be (doubly) linked list based. Your operations are subject to the following rules:

- `.add` and `.remove` family of operations must not involve any looping or recursion. Hence, a single such operation must take "constant time." That is to say, its execution time should not depend on the size of the deque.

[1] https://introds.philinelabs.net

- `.get` must use iteration, not recursion.
- `.size` must take constant time.
- You must not have extraneous/dangling nodes. Specifically, the amount of memory that your program uses at any given time must be proportional to the number of items. For example, if you add 1,000 items to the deque, and then remove 999 items, the resulting size should be more like a deque with 1 item than 1,000. This means that you must **not** maintain references to items that are no longer in the deque.

You will implement all the methods listed above in "The Deque API" section (above), together the following constructors:

```
// Creates an empty linked list deque.
public LinkedListDeque()
// Creates a deep copy of other.
public LinkedListDeque(LinkedListDeque<T> other)
```

Note that creating a deep copy means that you create an entirely new `LinkedListDeque`, with the exact same items. However, they are copies so they should be different objects. A good litmus test is, if you change other, the "copied" `LinkedListDeque` should not change.

NOTE: You are not allowed to use Java's built-in `LinkedList` data structure (or any data structure from `java.util.*`) in your implementation.

NOTE #2: We're providing a very simple simple sanity check in `LinkedListDequeTest.java`. For your benefit, you must write more comprehensive tests. Passing the given tests does not necessarily mean that you will pass our test or receive full credit.

NOTE #3: You may wish to implement a `printDeque()` method, which unlike `toString`, will print a detailed view of your internal representation—make it print whatever you wish to see when implementing/debugging the code. This is to help you debug and make sense of your deque structure. You are *not* required to hand this in, but we recommend that you write one to help you work through the task.

Array Deque

For this part, save your work in a file called `ArrayDeque.java`.

As another deque implementation, you'll build an `ArrayDeque` class. This deque must use fixed-size arrays as the core data structure. You'll implement all the methods listed above in the Deque API. Other than that, your operations are subject to the following rules:

- The `.add` and `.remove` family of operations must take constant time, except during resizing (grow and perhaps shrink) operations.
- `.get` and `.size` must take constant time.
- The starting size of your array should be 8. The amount of memory that your program uses at any given time must be proportional to the number of items. For example, if you add 10,000 items to the deque, and then remove 9,999 items, you shouldn't still be using an array of length 10,000ish.
- In addition, for arrays of length 16 or more, your array utilization (the ratio between array cells that are used compared to the total array capacity) always be at least 25%. For smaller arrays, your usage factor can be arbitrarily low.

You will also implement the following constructors:

```
// Creates an empty array deque.
public ArrayDeque()

// Creates a deep copy of other.
public ArrayDeque(ArrayDeque<T> other)
```

Like before, creating a deep copy means that creating an entirely new `ArrayDeque`, with the exact same items as `other`. However, they should be different objects, i.e. if you change other, the new `ArrayDeque` you created should not change as well. You may add any private helper classes or methods in the same file as you see fit.

TIPS #1: The biggest challege for this part is, how to support the add and remove operations in constant time (independent of the size)? We strongly recommend that you learn about the circular buffer. Chapter 5.3 of the book explains this for queues and has some code examples. You might also find more inspirations from Wikipedia (`https://en.wikipedia.org/wiki/Circular_buffer`). That is, you'll treat your array as circular. This means, for example, if your front pointer is at position zero, and you `addFirst`, the front pointer should loop back around to the end of the array (so the new front item in the deque will be the last item in the underlying array). Simiarly, if the rear end of the deque is at the last slot of the array and you `addLast`, it should wrap around and stores the item at position 0 (unless already full, in which case you'd resize).

TIPS #2: Consider not doing resizing at all until you know your code works without it. Resizing is a performance optimization (but it is required for full credit). And when you do resizing, make sure you think carefully about what happens if the data structure goes from empty, to some non-zero size (e.g. 4 items) back down to zero again, and then back to some non-zero size. Pro tip: $0 \times 2 = 0$, but it might not be what you want.

TIPS #3: Chapter 4.3 of the book discusses some idea(s) for resizing both for growing and shrinking the underlying array.

TIPS #4: Like in the linked list version, you may wish to implement a `printDeque()` method, which will print a detailed view of your internal representation—make it print whatever you wish to see when implementing/debugging the code. This is to help you debug and make sense of your deque structure. You are *not* required to hand this in, but we recommend that you write one to help you work through the task.