Midterm

CS 245: Summer 2024

Due: July 2 by 11:59pm PST

Instructions

Retrieve the skeleton files for the Midterm by running the command **git pull main** while inside the local version of Assignments repository. Make a copy of the Midterm folder and move it to your own private repository. Also download this document. You will need it for the written portion of the exam.

This exam has a mix of coding and non-coding questions. For the coding portions of the exam, please modify the relevant files that are specified in each question. Feel free to define any helper methods or private nested classes as you see fit. Any written non-coding answers should be included in **this document** in the appropriate section.

This is an individual assignment. You are not allowed to collaborate with any of your fellow classmates. We will not be answering any conceptual questions related to midterm either on Piazza, during lecture or office hours. We will only respond to clarifying questions or comments that point out potential bugs in the problems listed below.

You are allowed to reference any class slides, code examples, past assignments and lecture recordings to help you formulate your answers. However, this is an individual assignment so you cannot collaborate with your classmates. You cannot use the internet or AI tools to crowd-source your solution in any way. **Any** form of cheating will get you a 0 and referral to the Dean's office with no exceptions.

Late Policy

The midterm is due on July 2 by 11:59 PM PST. **You cannot use any slip hours on this assignment**. Any late submissions will get an automatic 0.

Grading

We have setup autograders on Gradescope that we will be using to verify the correctness of your code. All of the test cases we are running on Gradescope are exactly the same as the ones we have released. Although we have made most of the test case visible, a couple questions have hidden tests that we will not be releasing.

The reason why we are not releasing all test cases is that we want to encourage you to write your own test cases. In academia and industry, you will be responsible for robustly showing your code works for various inputs and input sizes. You can find the value of each individual test case in the testing suites that we have provided.

Please see the Autograder section for more details on how to use the Gradescope autograders.

Questions

1. Your friend builds a robot that can only move forward and backward. The robot can accept a sequence of instructions represented as a string where a character of “F” tells the robot to move forward and “R” tells the robot to move backward. The robot has a particular behavior where it doubles its step size for every consecutive instruction that is the same. If it receives an instruction to go in the opposite direction, it switches course and takes a step size of 1 in the specified direction.

   The robot starts at position 0 with a step size 1 and is initially set to move in the forward direction. If the robot receives the instructions of “FFF”, the robot will take 1 + 2 + 4 steps forward and up at position 7. If the sequence of instructions is “FFFRR” the robot will take 1 + 2 + 4 = 7 steps forward and then 1 + 2 = 3 steps backward. The final position would then be 7 - 3 = 4.

   In **MT.java** in the **misc** package implement the method ***boolean canReach(String s, int p)*** that returns a boolean indicating whether or not the robot will end up at position p when executing the instruction set specified by s. You can assume the only characters in the sequence of instructions will be “F” and “R” **(4 points)**

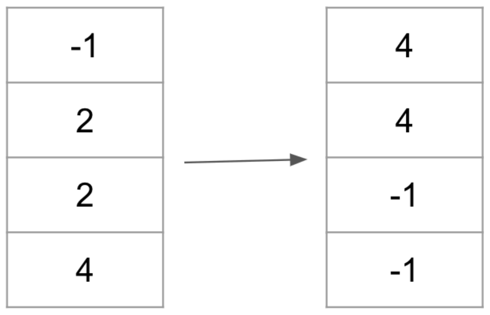
2. The game 2048 is played on a 4x4 grid of squares, each of which can either be empty or contain a tile bearing an integer–a power of 2 greater than or equal to 2. The player then chooses a direction to tilt the board: up, down, left or right. All tiles slide in that direction until there is no empty space left in the direction of motion (there might not be any to start with). A tile can possibly merge with another tile which earns the player points. The game ends when either the player forms a tile of the value 2048 or there are no moves for the player to make.

   For this question, you will be responsible for implementing the logic that updates the game board only when the board is tilted in the up direction. We will not handle tilting in the other directions for this question, though doing so is trivial once you get one direction working.

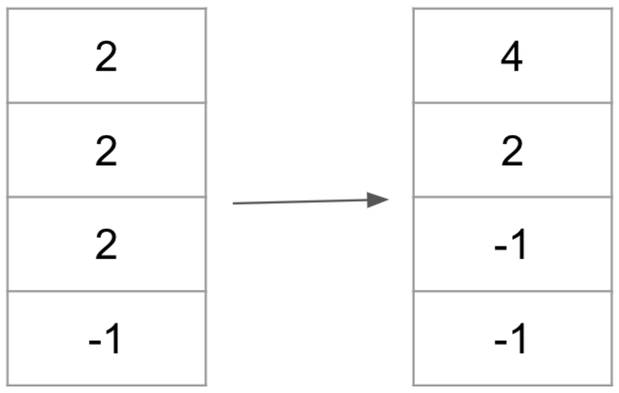
   Here are the full rules for when merges occur.

   a. Two tiles of the same value merge into one tile containing double the initial number.

b. A tile can only merge one time when we tilt the board in a certain direction. In the example below when we tilt the board up, the two 2 tiles merge into a 4 tile. This 4-tile that was created as a result of the merge **does not** merge with the existing 4 tile and become an 8 tile. We represent an empty space with the value of -1.



c. When three adjacent tiles in the direction of motion have the same number, then the leading two tiles in the direction of motion merge, and the trailing tile does not (see example below)



In **MT.java** in the **misc** package, implement the method ***void tilt(int[][] gameBoard)*** that implements the tilting logic described above. Your method must modify the gameBoard array directly. Even though 2048 is traditionally played on a 4x4 grid, you cannot make any assumptions about the board dimensions for this question.

We have provided a couple visible test cases for this question that are worth 9 points, but there is one hidden test case worth 6 points that we will not be releasing. **(15 points total) (Note:** This is probably one of the harder questions on the midterm**)**

3. Given a list of strings, we define the interleaving of the strings to be another string where we initially append first characters of all the strings to the output, then the second

characters and so on. For example, if we have two strings ["cat", "dog", "bee"], the interleaving would be another string of "cdbaoetge". For this exercise, we want to retrieve individual characters of the interleaved string. Implement the iterator defined in ***InterLeaveIterator.java*** in the **misc** package so that the iterator returns characters of the interleaved result one at a time. For example, if we initialize the iterator with the list of strings ["cat", "dog", "bee"], and we call the .next() method three times, the returned characters should "c", "d", "b". The iterator cannot make any assumptions about the number of strings in the list or the the length of the individual strings (i.e the strings don't have to be the same length). For example, the interleaved result of the strings ["abcdef", "xy"] is "axbycdef".

Your implementation cannot compute the resulting interleaved string all at once. You will get no credit if it does. If we initialize your iterator with the list of strings ["cat", "dog", "bee"], your iterator cannot pre-compute the result of "cdbaoetge" and store this value in a variable.

We have provided a couple visible test cases for this question that are worth 7 points, but there is one hidden test case worth 3 points that we will not be releasing. **(10 points total).** See the test cases to understand how your iterator will be used. You can define any helper methods or attributes as you see fit in your implementation.

4. Suppose each index in an integer array holds a digit of an integer, with the most significant digit being at index 0. For example, an array containing the values [4, 0, 5, 6] would represent the number 4056. Implement the function **boolean correctSum(int[] A, int[] B, int[] C)** in **MT.java** in the **misc** package that returns true if C represents the correct addition of the numbers represented by A and B.

   Example: A = [1, 3, 4], B = [5, 7], C = [1, 9, 1] → true since 134 + 57 = 191
   Example: A = [0, 4, 5], B = [3, 6], C = [8, 0] → false since 45 + 36 != 80

   You can assume that A, B and C represent positive integers, but you cannot assume that there won't be leading 0s. For example, we could represent the number 91 as [0, 0, 9, 1] or [9, 1]. We have provided a couple visible test cases for this question that are worth 6 points, but there is one hidden test case worth 2 points that we will not be releasing. **(8 points total)**

5. Queues follow a First in First Out (FIFO) policy. However, sometimes we want to promote or demote values. Promoting a value moves it one step closer to the front of the queue while demoting a value pushes it one step towards the back. For this problem, we are going to implement the class **PromotionQueue** defined in the **lists** package. Each entry in our PromotionQueue is a QueueNode that has a prev and next pointer just like nodes in Doubly Linked Lists do. Implement the ***void promote(int indexToPromote)*** so that the node at the specified index moves up one spot in the queue. Similarly,

implement the ***void demote(int indexToDemote)*** so that the value at the index goes back 1.

For example, if our PromotionQueue had the values of 4 <-> 10 <-> 3 <-> 8, calling promote(2) would make the PromotionQueue look like 4 <-> 3 <-> 10 <-> 8. Calling demote(1) would then revert the PromotionQueue back to the original state of 4 <-> 10 <-> 3 <-> 8

We have provided a couple visible test cases for this question that are worth 8 points, but there is one hidden test case worth 2 points that we will not be releasing. **(10 points total)**

6. Assume our Singly Linked List contains integers. In this Linked List, there can be runs of integers, or a series of consecutive nodes that have the same value. In **SList.java** in the **lists** package, implement the ***void compressNodes()*** method so that consecutive nodes with the same values are merged into a single node whose new value is the total sum of the merged nodes.

   If our Linked List was 4 → 3 → 3 → 3 → 2, the modified Linked List should be 4 → 9 → 2. Merging should not be recursive. If our list was 1 → 2 → 2 → 2 -> 6, the output should be 1 → 6 → 6.

   We have provided a couple visible test cases for this question that are worth 6 points, but there is one hidden test case worth 2 points that we will not be releasing. **(8 points total)**

7. When we introduced lists in this class, we mentioned that Java does not support negative indices while others do. For example, in Python if you had a variable x that stored the values [10, 20, 30, 40, 50], x[-1] would return 50, x[-2] would give 40 and so on. Such logic would throw an error in Java.

   However, we can add functionality to our classes to support negative indexing. For this exercise, you will be responsible for adding this behavior to our Doubly Linked List data structure. In **DList.java** in the **lists** package, implement the ***T get(int index)*** that returns the value stored at the specified index. This function should work for both positive and negative indices. If we called .get(2) and .get(-2) on a Doubly Linked List that had the values of 4 <-> 10 <-> 3 <-> 8, both calls should return the value 3. Your function should throw an IndexOutOfBoundsException for indices that are out of bounds. **(4 points)**

8. In class we mentioned that Stacks follow a Last in First Out (LIFO) Policy. Recall a Stack has two primary methods:
   a. push(T value): push value onto stack
   b. T pop(): Remove and return the top value of the stack

   We are going to create a new data structure called **MaxStack** where calls to pop() returns the maximum value in the stack. For example if we have an empty MaxStack and then push the values of 45, 34, 57, 14, the first call to .pop() should return 57. If we call .pop() again, the returned value should be 45. The stack should throw an EmptyStackException if we call the .pop() method on an empty stack.

   Complete the **MaxStack** class skeleton provided in the **lists** package. The .pop() Method should run in O(1) time while the .push(T value) method should run in O(n) time. Implementations that don't meet these time bounds will get no credit. You can use the built-in Java Stack data structure in your solution. Hint: You might find the the.peek() Stack function useful. (**4 points**)

9. In many applications where users edit an object, for example like Google Docs, the application keeps track of historical states. This gives users the ability to undo a revision. They also have the ability to **revert an undo** operation (i.e. undo the undo). Suppose a user starts with a blank document, and the state of the document looks like the following after each of the following revisions:

   Revision 1: Hello world.

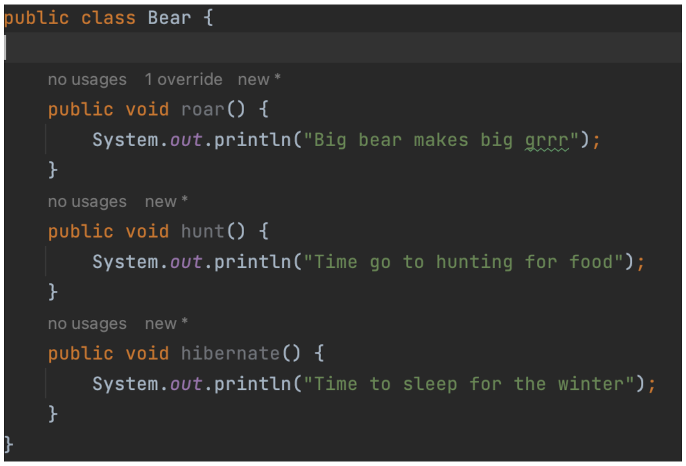
   Revision 2: Hello world. It is a sunny day.

   Revision 3: Hello world. It is a really really sunny day today.

   If after Revision 3, a user performs an **undo** operation, the document should go back to the state in Revision 2 and say "Hello world. It is a sunny day." . Another undo operation would put the document in the same state as Revision 1. However starting at Revision 3 if a user did an **undo** operation followed by a **revert undo**, the document would stay in the original state and show "Hello world. It is a really really sunny day today.".

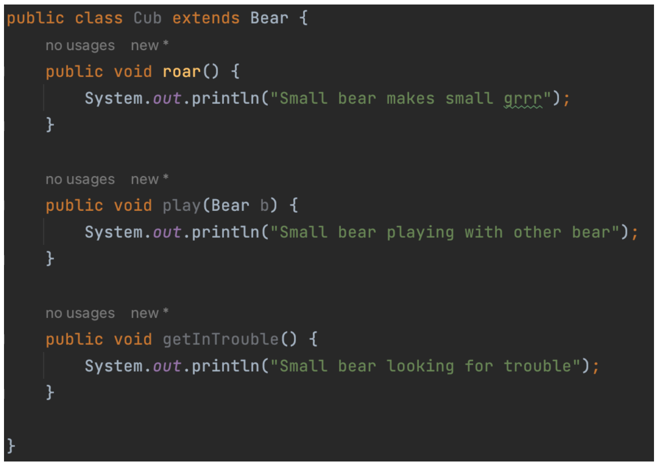
   A friend of yours asks you to help him design such a system. Below, explain what data structure(s) you would use (you may need multiple), what methods/attributes you would define, and how the data structures would interact with each other to support the desired behavior mentioned above. You are not required to write any code for this question, but your explanation should make it clear what any functions you define would do, what the purpose of certain attributes are etc. You will be graded based on how reasonable and coherent your explanation is (**10 points**).

10. Suppose we have the following class definitions and variable declarations as shown below:



```java
public class Bear {

    no usages   1 override   new *
    public void roar() {
        System.out.println("Big bear makes big grrr");
    }
    no usages   new *
    public void hunt() {
        System.out.println("Time go to hunting for food");
    }
    no usages   new *
    public void hibernate() {
        System.out.println("Time to sleep for the winter");
    }
}
```

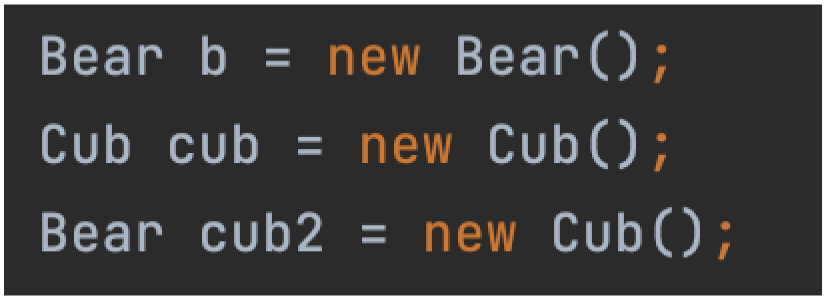


```java
public class Cub extends Bear {
    no usages   new *
    public void roar() {
        System.out.println("Small bear makes small grrr");
    }

    no usages   new *
    public void play(Bear b) {
        System.out.println("Small bear playing with other bear");
    }

    no usages   new *
    public void getInTrouble() {
        System.out.println("Small bear looking for trouble");
    }

}
```



```java
Bear b = new Bear();
Cub cub = new Cub();
Bear cub2 = new Cub();
```

For this next question, write what gets printed out when each of the following lines is executed. Be sure to justify your answer by mentioning principles of inheritance and dynamic method selection. If the line of code throws an error, explain what the error is and how we can fix it (**7 points**)

a. b.hibernate() (1 point)
   Output: Time to sleep for the winter.

   Explanation: The variable b is a Bear instance, so the hibernate method associated with the Bear class is called.

b. cub.hunt() (1 point)
   Output: TIme to go hunting for food

   Explanation: The Cub class inherits the hunt method of the Bear class.

c. cub.roar() (1 points)
Output: Small bear makes small grr

Explanation: Both the static and dynamic type of the variable are Cub. The roar method defined in the Cub class gets invoked.

d. cub2.roar() (1 points)
Output: Small bear makes small grr

Explanation: The static type of cub2 is Bear and the dynamic type is Cub. The Cub class overrides the roar() method. By the rules of Dynamic Method selection, the roar() method defined in the Cub class gets invoked.

e. cub2.play(cub) (3 points)

The static type of the variable cub2 is Bear and the dynamic type is Cub. However, dynamic method selection does not apply here because the *play* method is not an overridden method. Since the Bear class does not implement the *play* method, the compiler throws an error.

Both fixes below are valid:

Fix 1: Since the dynamic type of cub2 is Cub, we can safely cast cub2 and re-write the line as ((Cub) cub2).play(cub).

Fix 2: Change the definition of cub2 to Bear cub2 = new Cub().

Autograders

We have setup autograders on Gradescope that we will be using to verify the correctness of your Deque implementations. You can find the instructions on how to use each of the autograders below. The grade you get on the assignment will be strongly correlated with the scores the autograders give you. It is your responsibility that the autograders can process your submission. If you are facing problems using them, please reach out to the course staff or come to office hours so that we can help you resolve any problems you are facing.

Midterm: Misc - MT

Compress the **misc** directory and upload the zip file to this assignment on Gradescope. Your final submission to Gradescope should only contain the file **MT.java**. This autograder grades questions 1, 2 and 4.

Midterm: Misc - Interleave

Compress the **misc** directory and upload the zip file to this assignment on Gradescope. Your final submission to Gradescope should only contain the file **InterleaveIterator.java**. This autograder grades questions 3.

Midterm: lists - PromotionQueue

Compress the **lists** directory and upload the zip file to this assignment on Gradescope. Your final submission to GradeScope should only contain the file **PromotionQueue.java**. This autograder grades questions 5.

Midterm: lists - SList

Compress the **lists** directory and upload the zip file to this assignment on Gradescope. Your final submission to Gradescope should only contain the file **SList.java**. This autograder grades questions 6.

Midterm: lists - DList

Compress the **lists** directory and upload the zip file to this assignment on Gradescope. Your final submission to Gradescope should only contain the file **DList.java**. This autograder grades questions 7.

Midterm: lists - MaxStack

Compress the **lists** directory and upload the zip file to this assignment on Gradescope. Your final submission to Gradescope should only contain the file **MaxStack.java.** This autograder grades questions 8.

Final Point Breakdown

Coding Section

- Midterm: Misc - MT: (27 points)
- Midterm: Misc - Interleave (10 points)
- Midterm: lists - PromotionQueue (10 points)
- Midterm: lists - SList (8 points)
- Midterm: lists - DList (4 points)

- Midterm: lists - MaxStack (4 points)

Section Total: 63

Written Section

- Question 9: 10 points
- Question 10: 7 points

Section Total: 17

Total Possible: 80 points

Submission

Push all your Midterm code to your private repository on Github.

Your answers to the non-coding questions should be on this document. Rename this document as {LastName}_{FirstName}_MidtermWritten. Submit this document either as a pdf or word document to the assignment called **Midterm:Written on Canvas**

Congratulations, you are officially done with the midterm!