CIS 1100 — Spring 2024 — Exam 2

Full Name: _____

Recitation #: _____

PennID (e.g. 12345678, not PennKey): _____

My signature below certifies that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this examination.

_____ _____

Signature Date

Instructions are below. Not complying will lead to a 0% score on the exam.

- Do not open this exam until told by the proctor.
- You will have exactly 60 minutes to take this exam.
- Make sure your phone is turned OFF (not on vibrate!) before the exam starts. You may not use your phone or open your bag for any reason.
- Food and gum are not permitted—don't be noisy or messy.
- This exam is closed-book, closed-notes, and closed computational devices.
- If you get stuck on a problem, it may be to your benefit to move on to another question and come back later.
- All code must be written in proper Java format, including all curly braces and semicolons.
- Do not separate the exam pages. Do not take any exam pages with you. The entire exam packet must be turned in as is.
- There are two blank pages at the end of the exam if you need extra space for any graded answers.
- If you have any questions, raise your hand and a proctor will come to you.
- When you turn in your exam, you may be required to show your PennCard. If you forgot to bring your ID, talk to an exam proctor immediately.
- We wish you the best of luck!

Q1	Q2	Q3	Q4	Q5	Bonus
Short Answers	Reading & Debugging Code	Writing & Testing Code	Recursion (with Tours)	Nodes	

Q1. Explain Yourself!

All answers for questions in this section should be kept as brief as possible. Sentence fragments are acceptable. Longer answers will be judged more harshly, and mixtures of correct, incorrect, or irrelevant information will not earn full credit. **All answers go on Page 3.**

Q1.1 Nodes & Tours

When building a Tour using heuristics, we repeatedly add a Point to an existing Tour, growing its size by 1 each time. In fact, we don't need to know how many Points will be in the Tour at the end. **Briefly justify why it is easier or simpler to implement a Tour with Nodes instead of arrays of Points.**

Q1.2 Records & Objects

When reading data from an external source, we can represent the data in our programs using Record Types. We could also write a class that replicates all of the behaviors of a record, but this isn't necessarily a good idea. **Briefly explain from either a design perspective or an ease-of-use perspective why you might prefer to define a Record Type instead of a Class when working with data.**

Q1.3 Tests & Correctness

TA Jason's homework didn't earn all points on the autograder despite the fact that it passed all four of the JUnit tests he wrote, and he is very confused. **Briefly explain to Jason why it is possible that his code has faults despite passing his four test cases.**

Q1.4 Conditionals & Recursion

TA Ngaatendwe can often be heard saying that "you can't understand recursion unless you understand conditionals!" And she's right to say it. **Briefly explain why conditionals are important to writing recursive functions.**

Q1.5 Redundant Information

In *Furious Flying Fish,* the Arena class contained an instance variable **ArrayList<Target> targets.** Although the condition for winning is based on the number of elements stored in this list, we did not separately store that information in an instance variable. **Briefly explain why it is unnecessary to separately store the number of targets remaining.**

Q1.6 Redundant Information?

Java provides the **==** operator to check if two values are equal. Separately, we might write an **.equals()** method for an object to check if two objects of a type are equal. **Briefly explain with reference to *references* why we might want to have two different notions of "equals" in Java.**

Q1.1 don't have to resize the array repeatedly/don't know the size of the array to make/inserting in the middle of a tour only requires changing one reference/inserting in the middle of an array requires shifting everything down by one
Q1.2 Correct: shorter to write/records are one-liners/records are immutable but classes are not by default
Q1.3 Correct: the tests are not exhaustive/there is a fault not covered by a test/the tests are incorrectly written/the tests are redundant
Q1.4 Correct: recursion works with cases which are usually expressed with conditionals/need to decide **if** this is a base case or recursive case/need to have the function behave differently on different "sizes" of inputs
Q1.5 Correct: use targets.size()/we can ask the list how many elements it contains/would be tedious to update the number of targets each time we remove one
Q1.6 Correct: objects can contain identical data even if they are stored at different locations/== checks for referential equality whereas .equals() checks for structural equality/two references might not be equal even though their pointees are semantically the same.

Q2. Unit Tests & Debugging

Q2.1

Here is a buggy implementation of binary search inside the file **Exam.java** that is supposed to find the position of a target element in an input array. If the target cannot be found, the function should return -1. You can assume that the input array is sorted and contains no duplicates. Consider the following four unit tests and **mark the names of the ones that fail.**

```
1. public int binarySearch(int[] arr, int target)
{
2.       int left = 0;
3.       int right = arr.length / 2;
4.       while (left < right) {
5.           int mid = (left + right) / 2;
6.           if (arr[mid] == target) {
7.               return mid;
8.           } else if (arr[mid] < target) {
9.               left = mid + 1;
10.          } else {
11.              right = mid - 1;
12.          }
13.      }
14.      return -1;
15. }
```

☑ ~~testOne~~	☐ testTwo	☐ testThree	☑ ~~testFour~~

```
@Test
public void testOne() {
      int[] arr = { 1, 2, 3, 4, 5 };
      int target = 1;
      int expected = 0;
      int actual = binarySearch(arr, target);
      assertEquals(expected, actual);
}
```

```
@Test
public void testThree() {
   int[] arr = { 1, 2, 3, 4, 5 };
   int target = 3;
   int expected = 2;
   int actual = binarySearch(arr, target);
   assertEquals(expected, actual);
}
```

```
@Test
public void testTwo() {
   int[] arr = { 1, 2, 3, 4, 5 };
   int target = 2;
   int expected = 1;
   int actual = binarySearch(arr, target);
   assertEquals(expected, actual);
}
```

```
@Test
public void testFour() {
   int[] arr = { 1, 2, 3, 4, 5 };
   int target = 4;
   int expected = 3;
   int actual = binarySearch(arr, target);
   assertEquals(expected, actual);
}
```

Q2.2 Finally, fix the implementation of binarySearch. Write the numbers of the **two lines** that need to be fixed and then write the fixed versions of the lines.

Line Number	Fixed Line
3	int right = arr.length - 1
4	while (left <= right) {

Q3. Writing Objects

You know how some cool stores calculate your total amount due without the customer needing to scan their items? Your job is to test & write a class that models an automated shopping cart. It will keep track of the items in the cart, allow for adding & removing items, and calculate the total price of all items, factoring in any discounts due to sales. Study **ShoppingCart.java** below and **Item.java** in the appendix.

```
import java.util.ArrayList;
public class ShoppingCart {

    private ArrayList<Item> items; // Item is a record, see appendix

    public ShoppingCart() {
        items = new ArrayList<>(); // Creates an empty cart.
    }
    // Create a new item with name and price, add to end of list.
    public void addItem(String name, double price) {
        items.add(new Item(name, price));
    }
    // Remove the first item in the list with a matching name.
    public void removeItem(String name) {
        for (int i = 0; i < items.size(); i++) {
            if (items.get(i).name().equals(name)) {
                items.remove(i);
                return;
            }
        }
    }
    // Returns an array of all of the item names.
    public String[] listContents() {
        String[] contents = new String[items.size()];
        for (int i = 0; i < items.size(); i++) {
            contents[i] = items.get(i).name();
        }
        return contents;
    }
    // See Q3.3
    public double calculateSubtotal(String saleCategory, double discount) {
        // TODO!
    }
}
```

Writing Tests

Complete the two tests on the **remove** method below. The tests you write should pass, and they should verify the behavior specified by the function names and comments. You can reference other completed test cases in **ShoppingCartTest.java**, included in the appendix.

Q3.1 Test that remove has no effect on the cart if provided item name is not present

```
@Test
public void testRemoveHasNoEffectWhenProvidedNameIsAbsent() {
    ShoppingCart cart = new ShoppingCart();
    cart.addItem("Banana", 1.50);
    cart.addItem("Strawberry", 1.50);
    cart.addItem("Apple", 1.50);
    cart.removeItem("Cherry"); // nothing should change about the cart!
    // TODO: Finish the test to verify that no items were removed.
int expected = 3;
int actual = cart.listContents().length;
assertEquals(expected, actual);

}
```

Q3.2 Test that remove causes the first matching item to be deleted from the list.

```
@Test
public void testRemoveWorksOnFirstMatchingItem() {
    ShoppingCart cart = new ShoppingCart();
    cart.addItem("Banana", 1.50);
    cart.addItem("Strawberry", 1.50);
    cart.addItem("Banana", 1.50);
    cart.removeItem("Banana"); // the banana at index 0 should be removed
    // TODO: Use listContents to verify that the correct item was removed.


String expected = "Strawberry";
String actual = cart.listContents()[0].name();
assertEquals(expected, actual);

}
```

Q3.3 Writing **calculateSubtotal**

Provide an implementation for the **calculateSubtotal** method. This method should sum up the prices of all items in the cart subject to any discounts. The **discount** input should be applied to

any item in the list **items** which has **saleCategory** as a substring of its name. For example, the price of an item with the name **"red beans"** and the price of **1.00** should be discounted by 20% when we call **calculateSubtotal("red", 0.2).** If that were the only item in the cart, the subtotal would consequently be 0.80. There are more examples in **ShoppingCartTest.java** in the appendix.

Hint: you may find it useful to use one or more String methods; a set of these is also listed in the appendix.

```
public double calculateSubtotal(String saleCategory, double discount) {
        double totalCost = 0;
        for (int i = 0; i < items.size(); i++) {
            if (items.get(i).name.contains(saleCategory)) {
                totalCost += items.get(i).fullPrice * (1 - discount);
            } else {
                totalCost += items.get(i).fullPrice;
            }
        }
        return totalCost;
    }
```

Q4. Recursion & Tours

Recall that a Tour is a data structure that allows us to represent an expandable sequence of Nodes, starting and ending in the same location. The Tour consists of two instance variables: **head** and **lastNode**, which contain references to the first and final Nodes. These two Nodes should contain a reference to the same Point; other Nodes also each contain a reference to a Point.

In this section, you will write a method **insertLast()** that calls a recursive helper **insertLastHelper()** that inserts a point into a Tour right before the **lastNode**. This is similar to **insertInOrder** from HW06 in that the relative position of the insertion is always the same each time the method is called. Your solution must be recursive in order to receive credit.

Q4.1 Before writing code, complete/evaluate these statements:

How many Nodes should be constructed when inserting a Point into an empty Tour?	☐ 0	☐ 1 ☑ ~~2~~
Which of the following conditions define an empty Tour?	☑ ~~**head** is null~~	☐ **head** & **lastNode** contain the same Point
How can we identify which Node **n** is the correct one to insert after?	☐ lastNode == n.next.next ☐ lastNode == n	☐ lastNode.next == n ☑ ~~lastNode == n.next~~
If we place a new Node **o** using **insertLast()**, which should be true about **o**?	☐ head.next == o ☐ lastNode == o	☐ lastNode.next == o ☑ ~~lastNode == o.next~~

Q4.2 Now, complete the implementation of the function **insertLastHelper(),** which takes in a Node **n** and a Point **p**. If **n** is the proper Node to insert after, then the method will create a new Node with **p** and insert it. Otherwise, it should make a recursive call to navigate to the proper Node.

```
private void insertLastHelper(Point p, Node n) {
    // Base case: If the Tour is currently empty
    if (n == null) {
        lastNode = new Node(p, null);
        head = new Node(p, lastNode);
        return;
    }

    // Base case: If n is the proper node to insert after
    if (n.next == lastNode) {
        n.next = new Node(p, lastNode);
        return;
    }
    // Recursive case: If n is not the proper node to insert after,
    //                 keep recursing down the Tour towards the end.

    insertLastHelper(p, n.next);

}
```

Q4.3 Finally, complete **insertLast()** by providing the values for the initial call to the recursive helper. You should not need any additional space to write this; it can be done in one line.

```
private void insertLast(Point p) {
  insertLastHelper(p, head);

}
```

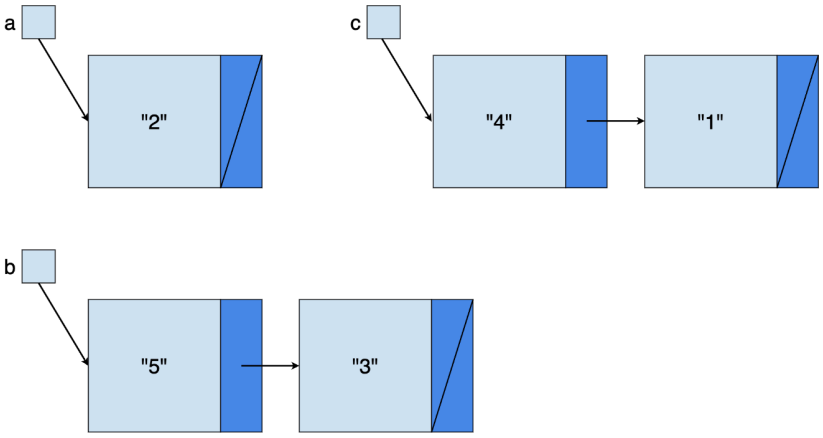
Q5. Nodes

For Question 5, assume we have the following Node class:

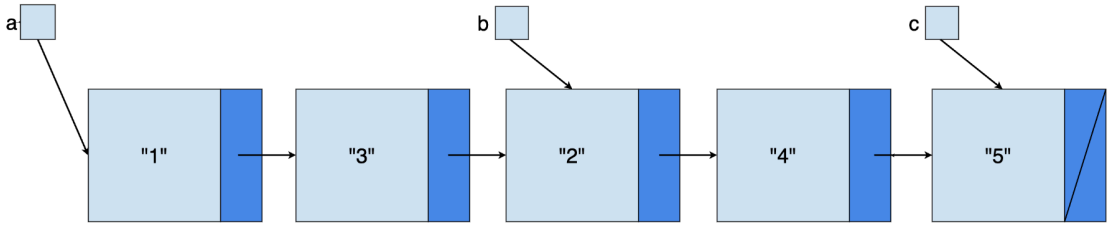
```
/**
 * This node class will store String values.
 */
public class Node {

    // public instance variables
    public String data;
    public Node next;

    public Node(String data, Node next) {
        this.data = data;
        this.next = next;
    }
}
```

Question 5.1:

Given the following state of nodes:



rearrange the references so that the **variables & nodes** look like:



To be clear, you do not have to generalize this to work on any sequence of Nodes. You are given the chains drawn in the first image and you have to transform it into the chain drawn in the second image.

NOTE: you **are not allowed to** access or modify the value in any node's *data* instance variable, and you are not allowed to allocate a new node (e.g. the code you write should not contain *new Node(..., ...)* in it.)

You **are allowed to** create Node reference variables with something like: Node temp = a;

```
public static void main(String[] args) {
      Node a = new Node("2", null);
      Node b = new Node("5", null);
      b.next = new Node("3", null);
      Node c = new Node("4", null);
      c.next = new Node("1", null);

      // your solution here
```

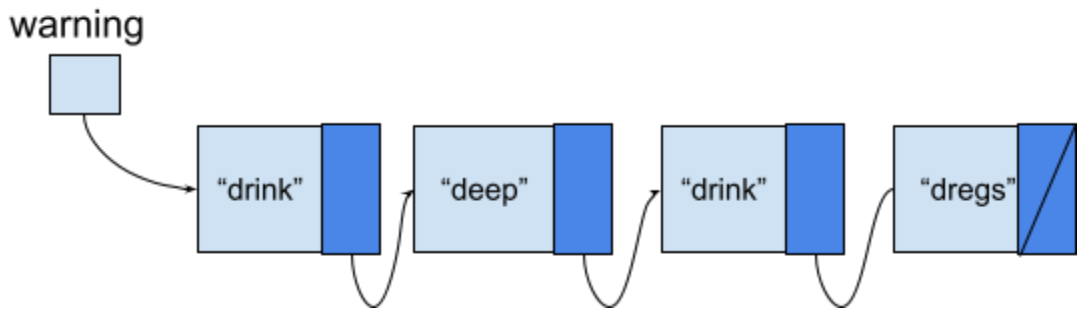
```
c.next.next = b.next;
b.next.next = a;
a.next = c;
a = c.next;
a.next.next.next.next = b;
c = b;
b = a.next.next;
c.next = null;
```

}

Question 5.2:

Complete the following function **hasDuplicateAtOffset()**:

```
/**
* Checks if there is a node with data equal to head.data at the specified
* offset from the head node. (Diagram examples below.)
* Inputs: head, the head node of the linked list
*         idx, the index of the node to compare with the head node
* returns true if there is a node with identical data to head's data
*         at the specified offset, false if head is null,
*         there is no node at the specified offset,
*         or if the data is different in the two nodes
*/
public static boolean hasDuplicateAtOffset(Node head, int idx) {
        int count = 0;
        Node curr = head;
        while (curr != null && count < idx) {
            count++;
            curr = curr.next;
        }

        if (count < idx || curr == null) {
            return false;
        }
        return curr.data.equals(head.data);
    }
```



hasDuplicateAtOffset(warning, 1) → **False,** since "drink" and "deep" are not equal.
hasDuplicateAtOffset(warning, 2) → **True,** since "drink" and "drink" are equal.

hasDuplicateAtOffset(warning, 5) → **False,** since there is no Node at offset of 5.

Bonus:

Draw your TA(s)! Or, if you're not feeling artistic, Recommend something for your grading TAs—movie, book, song, life advice... All exams will get full credit for this question, answer as seriously as you would like!

Room for Extra Answers

You may use this page for additional space for answers; keep it attached to this exam. Clearly note on the original question page that your answer is on this extra page, and clearly note on this page what question you are answering.

Appendix

Lists

method signature	**purpose**
get(int i)	return the value at position i in the List.
set(int i, E e)	set the value at position i in the List to be e.
size()	return the number of values stored in the List.
remove(int i)	remove and return the value stored at position i in the List.
add(E e)	insert the value e at the end of the List.

add(int i, E e)	insert the value e at position i in the List. i must be between 0 and size().

Strings

method signature	**purpose**
char charAt(int i)	return the char at position i in the String
int length()	return the number of characters in the String
int indexOf(String s)	return the index of the first occurrence of the substring s, or -1 if it is not present
boolean contains(String s)	return true if the substring s is present, or false otherwise

Item.java

```
public record Item(String name, double price) {}
```

ShoppingCartTest.java

```
public class ShoppingCartTest {

    @Test
    public void testAddFirstItem() {
        ShoppingCart cart = new ShoppingCart();
        cart.addItem("Apple", 1.00);
        // expected, then actual
        assertEquals(1, cart.listContents().length);
    }

    @Test
    public void testRemoveOnlyItem() {
        ShoppingCart cart = new ShoppingCart();
        cart.addItem("Banana", 1.50);
```

```
        cart.removeItem("Banana");
        assertEquals(0, cart.listContents().length);
    }

    @Test
    public void testCalculateTotalCostWithRedDiscount() {
        ShoppingCart cart = new ShoppingCart();
        cart.addItem("red beans", 1.00); // contains red
        cart.addItem("apple (red delicious)", 1.00); // contains red
        cart.addItem("blue raspberries", 1.00); // doesn't contain red
        assertEquals(2, cart.calculateSubtotal("red", 0.5), 0.01); // 50% discount
    }

    @Test
    public void testCalculateTotalCostWithNoDiscountedItems() {
        ShoppingCart cart = new ShoppingCart();
        cart.addItem("Chocolate", 5.00);
        cart.addItem("Apple", 5.00);
        cart.addItem("Banana", 5.00);
        assertEquals(15, cart.calculateSubtotal("no match", 0.5), 0.01); // 50% discount
    }
}
```