CS 61B Midterm 1

Problem 2

a) countChar (16 points). Fill in the `countChar` method below so that it counts the number of instances of the given character. For example, `countChar("exxon", 'x')` should return 2, and `countChar("zebraZzz", 'Z')` should return 1. **You may use a maximum of 10 blank lines for this problem.** Note: These 10 lines do not include the method signature nor the closing brace, i.e. these lines are not initially blank.

```
/** Counts the number of occurrences of c in a.
    For example countChar("moo", 'o') would return 2.
    You may assume the String is not null.
    */
public static int countChar(String a, char c) {









}
// Reminder: s.charAt(0) gets the 0th character of s
// Our solution uses the 7 blank lines provided.
// Yours may use more or less (up to the maximum of 10).
```

b) XComparator (18 points). Suppose we want to build a comparator `XComparator` that compares two strings based on the number of times lowercase 'x' appears in the string. For example an `XComparator` would consider "exxon" to be greater than "some axons". Below, implement the `XComparator`. For this and the remaining questions, assume the `countChar(String a, char c)` method from part a has been imported and works correctly, i.e. you can call `countChar("exxon", 'x')` and expect to get back 2.

Recall from lecture that the compare method of the Comparator interface in Java returns a negative number if the first argument is less than the second, 0 if they are equal, and a positive number if the first argument is greater, e.g. `compare("exxon", "some axons")` would return a positive number. You may assume none of the parameters to `compare` are null.

```
public class XComparator implements Comparator<_________> {
    /** Compares two strings based on the number of xs in
       the strings. For example exxon > axon, because 2
       occurrences of x is more than 1 occurrence of x. */
    public int compare(____________, _____________) {
```

```
    }   // Our solution uses 1 line. Yours may use more.
}
```

c) CharCountComparator (22 points) Now suppose we want to build a more general character counting comparator whose constructor takes a character as an argument, and whose compare method compares two strings based on the count of that character. For example, if we create `CharCountComparator c = new CharCountComparator('z')`, then `c.compare("mazzy", "lazer")` would consider mazzy larger since it has two 'z's vs. lazer's single 'z'. You may assume none of the parameters to `compare` are null.

```
public class CharCountComparator implements Comparator<_________> {
    // our solution has one instance variable, yours doesn't have to.


    public CharCountComparator(char givenC) {
        // our constructor has one line, yours doesn't have to.

    }

    /** Compares two strings based on the number of occurrences
       of the given character. */
    public int compare(_____________, _____________) {



    } // our compare method is 1 line long. yours may use more.
}
```

d) WordFinder (36 points). Now complete the following code that will read in all the words from a words.txt file and return the word with the most 'z's. **You may use a maximum of 11 blank lines for this problem.**

```
public class WordFinder {
    public static void main(String[] args) {
        In in = new In("words.txt");
        String[] words = in.readAllStrings();
        String wordWithMostZs = findMax(words, ____________________________);
        System.out.println(wordWithMostZs);
    }

    /** Finds the maximum string in the array of strings using the
        given comparator. You may assume the array is of length
        at least 1, and that none of the strings are null. If there is
        a tie, then break the tie arbitrarily. For example, if all the
```

```
        strings are equal according to the comparator, you can return
        any of the strings. */
    private static String findMax(String[] strings,
                                  Comparator<______________> cmp) {









    } // our solution uses 8 blank lines. yours may use up to 11.
}     // findMax must work for any provided comparator! That is,
      // it cannot be hard coded to look for strings with the most zs.
```

Problem 3

a) (50 points) Suppose we define the following strange class. Note that the `equals` method has the signature `equals(Student other)`. Recall from lecture 11 that the `Object` class's `equals` method has the signature `equals(Object other)`. That is, this class is not overriding the `equals` method!

```
public class Student {
    public String name;
    public int SID;

    public Student(String n, int id) {
        name = n;
        SID = id;
    }

    public boolean equals(Student other) {
        if (name.equals(other.name) &&
            SID == other.SID) {
            return true;
        }
        return false;
    }
}
```

Suppose we define five Java variables as shown:

```
Student sB = new Student("Borf", 123);
Student sG = new Student("Gorf", 123);
```

```
Student sB2 = new Student("Borf", 123);
Object oG = sG;
Object oB2  = sB2;
```

For each of the lines below, fill in the bubble for the return of the expression. Or if the line has a compile or runtime error, fill in the "runtime error" (RE) or "compile error" (CE) bubble instead. Not all answers may be used.

1. `sB.equals(sG)`

- [ ] true
- [ ] false
- [ ] compile error
- [ ] runtime error

2. `sB.equals(sB2)`

- [ ] true
- [ ] false
- [ ] compile error
- [ ] runtime error

3. `sB == sB2`

- [ ] true
- [ ] false
- [ ] compile error
- [ ] runtime error

4. `sG.equals(oG)`

- [ ] true
- [ ] false
- [ ] compile error
- [ ] runtime error

5. `sG == oG`

- [ ] true
- [ ] false
- [ ] compile error
- [ ] runtime error

6. `sG.equals((Student) oG)`

- [ ] true
- [ ] false
- [ ] compile error
- [ ] runtime error

7. `sB.equals(oB2)`

- [ ] true

- [ ] false
- [ ] compile error
- [ ] runtime error

8. `sB == oB2`

- [ ] true
- [ ] false
- [ ] compile error
- [ ] runtime error

9. `sB.equals((Student) oB2)`

- [ ] true
- [ ] false
- [ ] compile error
- [ ] runtime error

10. `((Object) sB2).equals((Student) oB2)`

- [ ] true
- [ ] false
- [ ] compile error
- [ ] runtime error

b) (8 points) Suppose we fix the `equals` method so that it takes an `Object` as an argument instead of a `Student`. Once we've done that, we are now properly overriding the `equals` method.

```
@Override
public boolean equals(Object o) {
    Student other = (Student) o;
    if (name.equals(other.name) && SID == other.SID) {
        return true;
    }
    return false;
}
```

After making this change, which of the following returns true that did not return true before? Only one is correct.

- [ ] `sB.equals(sG)`
- [ ] `sB.equals(sB2)`
- [ ] `sB == sB2`
- [ ] `sG.equals(oG)`
- [ ] `sG == oG`
- [ ] `sG.equals((Student) oG)`
- [ ] `sB.equals(oB2)`
- [ ] `sB == oB2`
- [ ] `sB.equals((Student) oB2)`
- [ ] `((Object) sB2).equals((Student) oB2)`

c) (8 points) The `equals` method above does not do all of the things that we said that an `equals` method should do in lecture. Describe at least one improvement we could make below:

Problem 4

DMSDogs (28 points) Suppose we define the classes below:

```
public class Dog {
    public void bark() {
        System.out.println("bark");
    }
}

public class Puppy extends Dog {
    public void bark() {
        System.out.println("lil bark");
    }
}

public class DogLover {
    public void pet(Dog d) {
        System.out.print("pet dog");
        d.bark();
    }
    public void pet(Puppy p) {
        System.out.print("pet puppy");
        p.bark();
    }
}

public class PuppyLover extends DogLover {
    public void pet(Puppy p) {
        System.out.print("pup love");
        p.bark();
    }
}
```

What will be the output of the lines of code below?

```
public static void main(String[] args) {
    PuppyLover pl = new PuppyLover();
    Puppy p = new Puppy();
    pl.pet(p);
    ((DogLover) pl).pet(p);
    pl.pet((Dog) p);
```

```
    ((DogLover) pl).pet((Dog) p);
}
```

For each call to pet above, notice that multiple things can be printed. Check the boxes for all that is printed. The order in which you check the checkboxes doesn't matter.

1. `pl.pet(p)`

- [ ] bark
- [ ] lil bark
- [ ] pet dog
- [ ] pet puppy
- [ ] pup love

2. `((DogLover) pl).pet(p)`

- [ ] bark
- [ ] lil bark
- [ ] pet dog
- [ ] pet puppy
- [ ] pup love

3. `pl.pet((Dog) p)`

- [ ] bark
- [ ] lil bark
- [ ] pet dog
- [ ] pet puppy
- [ ] pup love

4. `((DogLover) pl).pet((Dog) p)`

- [ ] bark
- [ ] lil bark
- [ ] pet dog
- [ ] pet puppy
- [ ] pup love

Problem 5

AddOnlyList (118 points). Before starting this problem, be aware that **for all parts of this problem, you may assume the earlier parts were done correctly and can be used, e.g. feel free to use `indexOfI` in part b, even if you didn't get part a correct. You may not need earlier methods for later problems, e.g. `get(int i)` (part b) may not be useful for `addLast` (part c).**

Suppose we define a new `AddOnlyList` class as follows:

```
public class AddOnlyList<T> {
    private T[] items;
    private int size;
    public AddOnlyList() {
        items = (T[]) new Object[8];
        size = 0;
    }
    private int indexOfI(int i, int M) { /* you'll write this */ }
    public T get(int index) { /* you'll write this */ }
    public void addLast(T item) { /* you'll write this */ }
    private void resize(int newlen) { /* you'll write this */ }
}
```

Instead of treating the array like a circle like you did in `ArrayDeque`, the `AddOnlyList` uses an outward-in pattern to store its items, as shown in the figure below:

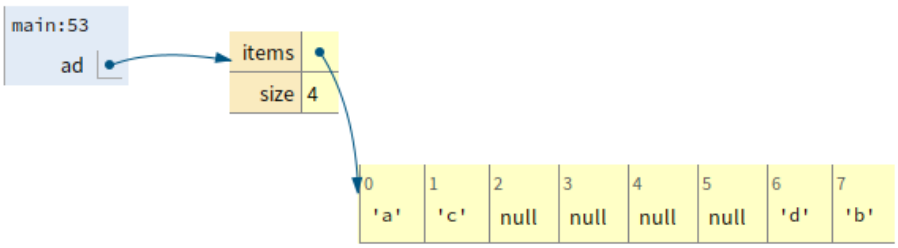

Figure 1: MidtermArrayDeque Visualization

For an `items` array of size 8, item 0 in the list goes in index 0, item 1 goes in index 7, item 2 goes in index 1, item 3 goes in index 6, item 4 goes in index 2, and so forth. The figure above corresponds to a list whose items are [a, b, c, d] in that order. If we were to add e to the end of the list, it would go in index 2. If we were to add f after that, it would go in index 5.

Fill in the functions `indexOfI`, `get`, `addLast`, and `resize`.

`indexOfI` gives the position of the ith element if the size array is of length M. For example if i = 3, and M = 8, then `indexOfI(3, 8)` is 6. Note that item 3 ('d') is in position 6 in the figure above. You may assume that M is a power of 2 that is greater than or equal to 8.

These functions are worth 40, 32, 32, and 14 points, respectively. **For `indexOfI` and `get`, your solution must fit in the skeleton provided. For `addLast` and `resize` you may usea maximum of 5 and 8 lines respectively. The lines we provide in `addLast` and `resize`do not add to your line count.**

```
public class AddOnlyList<T> {
    private T[] items;
    private int size;
    public AddOnlyList() {
        items = (T[]) new Object[8];
        size = 0;
    }

    /* DO NOT CHANGE ANYTHING ABOVE THIS LINE. */

    private int indexOfI(int i, int M) {
        if (_______________) {
            return ___________;
        } else {
            return ____________________;
        }
    }

    public T get(int i) {
       return ____________________________;
    }

    public void addLast(T x) {
        if (size _________________) {
            resize(_________ * 2);
        }

    } // Our addLast code uses 3 blank lines.
      // Yours may use less or more up to a max of 5.

    private void resize(int newlen) {
        T[] newItems = (T[]) new Object[_______];




        items = newItems;
    } // Our addLast code uses 5 blank lines.
      // Yours may use less or more up to a max of 8.
}
```

Problem 6

a) Testing contains (20 points). Suppose we add a method contains to our `List61B`interface with the signature below. This method returns true if the given `List61B` contains the given item. The midterm 1 reference page might be useful for this problem.

```
default public boolean contains(T x)
```

Write a JUnit test that verifies that this method works correctly for a list of length 3. Your test should verify that it works for all three items in the list, and that it works correctly for an input which is not in the list. Assume all JUnit classes needed have been imported.**You may use a maximum of 8 lines for this problem.**

```
@Test
public void testContains() {
    AList<Integer> L1 = new AList<>();

} // Our test uses the 7 lines provided above,
```

b) contains (30 points) Write the method `contains`. It should work for all possible inputs, not just your JUnit test above. **Your method must be non-destructive.** You may not need all lines. **You may use a maximum of 8 lines for this problem.**

```
public interface List61B<T> {
    public void addLast(T x);
    public T getLast();
    public T get(int i);
    public int size();
    public T removeLast();
    public void insert(T x, int position);
    public void addFirst(T x);
    public T getFirst();
    default boolean contains(T x) {


    } // Our code uses 6 lines.
}
```

c) Testing itemsNotIn (35 points). Suppose we add a method `itemsNotIn` to our **List61B** interface with the signature below. This method returns a list containing only the items that are in the current list, but not in the **List61B** provided as an argument. The items in the returned list should be in the same order. Your midterm 1 reference sheet might be useful for this problem. This method should be non-destructive.

```
default List61B<T> itemsNotIn(List61B<T> other)
```

For example, if you have a list `x` that contains the numbers 1 through 100 and a list `y`which contains the numbers 80 through 120, then `x.itemsNotIn(y)` would create a new list that contains the numbers 1 through 79, in that order. After this function call, `x` and `y`should remain unchanged.

Write a JUnit test that verifies that this method works correctly on the lists given in the starter code. You do not need to test that `L1` and `L2` are unchanged. **You may use a maximum of 8 lines for this problem.**

```
@Test
/** Your test does not need to verify that itemsNotIn is
    non-destructive! In real life, you would, probably as
    a separate JUnit test.*/
public void testItemsNotIn() {
        AList<Integer> L1 = new AList<>();
        L1.addLast(0);
        L1.addLast(1);
        L1.addLast(2);

        AList<Integer> L2 = new AList<>();
        L2.addLast(1);
        L2.addLast(4);
        L2.addLast(5);

} // Our test uses 4 lines above
```

d) itemsNotIn (35 points) Write the method `itemsNotIn`. **It should work for all possible inputs**, not just your JUnit test above. **Your method must be non-destructive.** You may not need all lines. You may assume the `contains` method from part b is available and works correctly, even if you did not complete part b. **You may use a maximum of 10 lines for this problem.**

```
public interface List61B<T> {
    public void addLast(T x);
    public T getLast();
    public T get(int i);
    public int size();
    public T removeLast();
    public void insert(T x, int position);
    public void addFirst(T x);
    public T getFirst();
    default boolean contains(T x) { /* not shown */ }
    default List61B<T> itemsNotIn(List61B<T> other) {
```

```
    } // Our code uses 8 lines.
}
```

Problem 7

a) removeNode (25 points). Suppose we have a version of the `LinkedListDeque` class from project 1, but hard-coded to use integers instead of a generic type. This new version is called `IntLinkedListDeque`. Its instance variable and `Node` class are as shown:

```
public class IntLinkedListDeque {
    private Node sentinel;

    private class Node {
        private int value; // Note: Value for a node is an integer, not a generic type!
        private Node next, prev;
        // constructor not shown
    }
    ... // constructor and methods not shown
}
```

Suppose we add a private `removeNode` method to `IntLinkedListDeque` that removes the specified node and keeps the list contiguous. The signature for this method is:

```
private void removeNode(Node t)
```

Assume that we're using the circular sentinel topology described in lecture and recommended for project 1A. Write the `removeNode` method below. You may assume that `t` is not null and that `t` is a valid node in the list. There's no need to update the size, because `IntLinkedListDeque` does not have a size variable.

As an example, suppose we have a `IntLinkedListDeque` containing three `Nodes` with values 1, 2, and 3. Now suppose we call `removeNode(sentinel.next)`, the front item of the list would be removed, and the list would have two `Nodes` with values 2 and 3. **You may use a maximum of 4 lines for this problem**.

```
/** Removes the specified node from the list.
    You may assume that t is not null.
    You may assume that the node is a valid node in the list. */
private void removeNode(Node t) {
```

```
} // Our solution is 2 lines.
  // Yours may use less or more, up to the maximum of 4.
```

b) removeOneIterative (40 points). Suppose we want to provide a public facing `remove`method with the signature shown below:

```
public boolean removeOneIterative(int x)
```

This method should return true if the item was located and removed, and false if the item was not in the list. Fill in the `removeOneIterative` move method below. Your method should only delete the first instance of x. For example, if the list contains

```
3, 4, 5, 6, 4, 5, 6
```

and you call

```
removeOneIterative(4)
```

on that list, the list would now contain

```
3, 5, 6, 4, 5, 6
```

You may use the `removeNode` method from part a, even if you did not complete part a correctly.

You may use a maximum of 10 lines for this problem, which does not include the lines of skeleton code already provided

```
public boolean removeOneIterative(int x) {
    Node p = _________________________________;
    while (_________________________________) {









    }


} // Our removeOneIterative method uses 8 blank lines,
  // not including the two starter code lines.
  // Yours may use less or more up to a max of 10.
```

c) removeOneRecursive (40 points). Repeat the exercise from part b, but this time with a recursive approach:

```
public boolean removeOneRecursive(int x)
```

We've provided a private recursive helper method that you must use. If you can't figure out how to write the private recursive helper method, you can still use it in the public method. You may not call `removeOneIterative`. You may use the `removeNode` method from part a, even if you did not complete part a correctly. **You may use a maximum of 7 lines for this problem, which does not include the lines of skeleton code already provided**.

```
private boolean removeOneRecursive(__________, __________) {
    if (_____________________________________) {
        return ______________________________;
    }
```

```
} // Our solution uses 5 lines, not including the
  // starter code lines.
  // Yours may use less or more up to a max of 7.

public boolean removeOneRecursive(int x) {
    return removeOneRecursive(__________, __________);
}
```

d) Using RemoveOne (50 points). Suppose that the `IntLinkedListDeque` implements the `Fa20MidtermList` interface, given below:

```
public interface Fa20MidtermList {
    public boolean removeOne(int x); // equivalent to removeOneIterative and removeOneRecurs
    public void addFirst(int x);
    public void addLast(int x);
    public int getFirst();
    public int getLast();
    public int removeFirst();
    public int removeLast();
    public int get(int i);
    public int size();
}
```

Suppose we want to add a default method `moveToFront` with the signature below:

```
default public void moveToFront(int x)
```

This method should move all instances of `x` to the front of the list. For example,

if the list initially contains

`1, 2, 10, 10, 4, 10, 7, 10`

and you call

`moveToFront(10)`

the list should change so that it now contains

`10, 10, 10, 10, 1, 2, 4, 7`

As you can see, the items have all moved to the front. You may assume all of the methods of `Fa20MidtermList` ****have been implemented properly.

Note that this is a default interface method, i.e. it should work for any type of `Fa20MidtermList`. You may use a maximum of 10 lines for this problem.

```
default public void moveToFront(int x) {




} // Our first solution used 7 blank lines total.
  // There is a shorter, more clever solution.
  // Yours may be more or less, up to a maximum of 10.
  // Note: THIS MUST WORK FOR ANY TYPE OF Fa20MidtermList
  // NOT JUST AN IntLinkedListDeque!
```

e) Using RemoveOne II (50 points). Suppose we want to add a default method `rippleRemove` with the signature below:

```
default public int rippleRemove(int x)
```

This method has strange behavior (hey, it's an exam!). It attempts to remove ALL instances of `x`. If it deletes at least one copy of `x`, then it will also attempt to delete all copies of `x - 1`. If it deletes at least one `x - 1` from the list, it will attempt to delete `x - 2`, and so forth until it can find no more numbers to delete. For example, suppose we have a `Fa20MidtermList` containing the values

`10, 6, 5, 6, 6, 4, 9, 8, 9, 10, 8, 6, 12`

Suppose we call `rippleRemove(10)` on that list. Now the list will contain

`6, 5, 6, 6, 4, 6, 12`

That is, all of the 10s, 9s, and 8s have been deleted. This function returns the total number of items it removed. In the example above, `rippleRemove(10)` should return 6, since it deleted 2 of each of the numbers 8, 9, and 10. For partial credit, write a function that only deletes all instances of `x` and returns the number of `x`s removed. You may assume that all of the methods from the `Fa20MidtermList` listed above have been implemented correctly. **You may use a maximum of 14 lines for this problem**.

```
default public int rippleRemove(int x) {




} // Our solution uses 4 blank lines.
  // Yours may use less or more up to a max of 14.
  // Note: THIS MUST WORK FOR ANY TYPE OF Fa20MidtermList
  // NOT JUST AN IntLinkedListDeque!
```