Homeworks

Homework 1 - Description

Title	Date Posted	Date Due
Homework 1: Spinning Some Yarn	9 / 12 / 17	9 / 29 / 17

By now, you're probably used to seeing **arrays** of different data types, which are simply ordered, contiguous lists.

An array of Strings, for example, is simply an array of references to Strings where each index holds a (not necessarily unique) String.

Arrays hold ordered data where each item may not necessarily be unique from the others.

Sets, by contrast, hold *unordered* data where each item is unique.

Example For example, a set of 4 strings might look like: `{ "hi", "I", "am", "Andrew" }`

Things to note about :

- Each String in the set is unique (i.e., no duplicates).
- Because order doesn't matter, the set above is considered equivalent to the set: `{ "Andrew", "hi", "I", "am" }`

Yarns

In this assignment, we're going to develop a new data structure called a "Yarn", which is a happy medium between arrays and sets of Strings.

A **Yarn** is an unordered collection of Strings in which duplicates are allowed. A Yarn maps Strings to the number of occurrences of each String in the Yarn.

heh... get it? Because a ball of yarn has a bunch of strings in it? (take my word for it, it's funny)

Example For each example that follows, the first entry is a sentence that has then been converted into a Yarn, which follows.

```
Sentence 1: "I think I like this homework"
Yarn 1:     { "I": 2, "think": 1, "like": 1, "this": 1, "homework": 1 }

Sentence 2: "YARNS YARNS YARNS"
Yarn 2:     { "YARNS": 3 }
```

Note:

- The `size` of Yarn 2 is 3, but its `uniqueSize` is 1. In other words, it contains 3 occurrences of 1 unique String.
- Remember that order does not matter for entries of a Yarn, so Yarn 1 is equivalent to: `{ "think": 1, "I": 2, "this": 1, "homework": 1, "like": 1 }` (among many other equivalent Yarns).
- The above Yarn syntax (i.e., the `{ ... }` notation) is for illustrative purposes alone; the following section describes how to programmatically implement a Yarn.
-

Specifications

Problem 1[90 points]

The following section instructs you on how to implement the Yarn class, as well as expectations for each method. Read through this section and the "Restrictions" section that follow before you begin any coding!

Fields

Your Yarn objects must be capable of accommodating 100 unique Strings.

For HW1, we will be implementing Yarns using an array of Strand objects. Each Strand (see class definition below) holds a unique String in the Yarn, as well as the number of occurrences of that String.

```
class Strand {
    // [!] Intentionally not private fields, since
    // the Strand class will be used internally
    String text;
    int count;

    Strand (String s, int c) {
        text = s;
        count = c;
    }
}
```

Note: the user of your Yarn class will never know about Strand objects -- these are merely convenient record-keeping entities to be used in the Yarn class' private fields.

As such, your Yarn class should maintain the following fields:

- `private Strand[] items;` the array of entries containing each individual String in each Yarn, as well as the number of occurrences of each String.
- `private int size;` the number of Strings currently in your Yarn (counting duplicates separately).
- `private int uniqueSize;` the number of unique Strings currently in your Yarn (counting duplicates as 1).

The above are the only mandatory fields for the Yarn class. You may add any additional fields that help you implement the rest of the project (though my solution adds none!).

Constructor

You will define two constructors for the Yarn class:

1. A **default constructor** that merely reserves space for an array of 100 Strand objects, and instantiates the size and uniqueSize to 0.
2. A **copy constructor** parameterized by another Yarn object `Yarn (Yarn other) {...}`, that reserves space for an array of 100 Strand objects, but then copies all Strands (i.e., a deep copy that does not simply copy references to Strands) from `other` into the Yarn being constructed.

If you have any additional fields that you have defined, you should initialize them in the above as well.

Methods

Your Yarn class will implement the following interface, with individual method descriptions to follow.

YarnInterface.java

```
package yarn;

public interface YarnInterface {

    boolean isEmpty ();
    int getSize ();
    int getUniqueSize ();
    boolean insert (String toAdd);
    int remove (String toRemove);
    int count (String toCount);
    void removeAll (String toNuke);
    boolean contains (String toCheck);
    String getNth (int n);
    String getMostCommon ();
    void swap (Yarn other);
    String toString ();

}
```

Methods
`boolean isEmpty();` Returns true if the Yarn has no Strings inside, false otherwise.
`int getSize();` Returns the current size of the Yarn (i.e., the number of Strings inside, counting duplicates separately).
`int getUniqueSize();` Returns the number of unique Strings in the Yarn (counting duplicates as 1).
`boolean insert (String toAdd);` Adds the String toAdd to the Yarn, and returns true if successful.If the Yarn is at capacity (already at 100 unique Strings), then does nothing and returns false.
`int remove (String toRemove);` Removes 1 occurrence of the String toRemove from the Yarn, and returns the number of occurrences remaining after removal.If toRemove does not exist in the Yarn, simply return 0 and do nothing.
`void removeAll (String toNuke);` Removes ALL occurrences of the String toNuke from the Yarn.If toNuke does not exist in the Yarn, do nothing.
`int count (String toCount);` Return the number of occurrences of String toCount found in the Yarn.
`boolean contains (String toCheck);` Returns true if the String toCheck appears at least once inside of the Yarn.

`String getNth (int n);` Even though Yarns are order-independent, we may at times want a way to iterate through the occurrences stored within. We'll use the getNth method for this purpose.getNth is defined for 0 <= n < size, such that iterating n as many times as there are String occurrences in the Yarn will return each occurrence, but in *any* order.Because of this "any order" constraint, you may assume that any operations that would affect the contents of a Yarn (e.g., insert or remove) may invalidate an existing iteration using getNth.In other words, the requirement of getNth to produce each String occurrence in the Yarn if iterated through from the beginning is removed whenever the Yarn's contents have been modified.In other, other words, if you modify the Yarn in any way, then getNth need not adhere to any prescribed behavior. The user is assumed to know this risk.For example if your Yarn contains: { "dup":2, "unique":1}, then getNth(i) for i = 0, 1, 2 could be: i=0: "dup", i=1: "dup", i=2: "unique" OR i=0: "unique", i=1: "dup", i=2: "dup" OR i=0: "dup", i=1: "unique", i=2: "dup" (though this last one would not likely come from our implementation).Think that constraint is too contrived? Well, certain Java collections that we'll learn about make the same warning...

`String getMostCommon ();` Returns the String that occurs most frequently in the Yarn. In the event of a tie, you may return *either* of the most frequent. If the Yarn is empty, return null.

`void swap (Yarn other);` Swaps the contents of the calling Yarn and the other specified. Restriction: you may NOT use iteration/recursion to solve this problem! Hint: use fields that have references!

`String toString ();` Returns a String representation of the calling Yarn (useful for debugging too!).The String returned should have the following format:Begins and ends with curly-brackets `{ }`, plus a space after the starting curly-bracket and a space before the closing one.Every Strand in the Yarn is printed as `"StrandText": count`, i.e., quotation marks surrounding the Strand's text, followed by a colon, followed by a space, and finally, the count.Every Strand that is printed which is NOT the last one printed has a comma and space separating it from the next, e.g. `"Strand1": count, "Strand2": count` Hint: recall that, to add a quote character to a String, you use the escape character (backslash), a la: `"\""` (the String containing the quote character).See intro of spec for example Strands in their toString format, as well as the attached sample tests. Note: because Yarns are order-independent, you may print out your Strands in any order that makes sense in the toString result. However, you may NOT duplicate any Strand's text in the output.

Static Methods

In addition to the above methods, you must implement the following 3 static methods:

Methods
`static Yarn knit (Yarn y1, Yarn y2);` Returns a *new* Yarn object consisting of a combination of all String occurrences from y1 and y2.
`static Yarn tear (Yarn y1, Yarn y2);` Returns a *new* Yarn object consisting of all String occurrences from y1 that do NOT appear in y2.
`static boolean sameYarn (Yarn y1, Yarn y2);` Returns true if y1 and y2 contain the exact same unique Strings and String occurrences (i.e., the same Strings and the same counts of each String).Note: because order does not matter for Yarns, the Strings in y1 and y2 can be found in different orders but still be considered equivalent.Refer to the notes in the above Description section for examples of equivalent Yarns.

Problem 2[10 points]

Unit Tests

In addition to the Unit Tests that are found in the following section, add at least 1 Unit Test for each method and static method, or expand each of the ones given in the next section.

Note: you do not have to use the Eclipse JUnit test suite format for this part of the assignment. Simply submit a file called YarnTests.java with your assignment that contains your unit tests, in whatever format they may be.

Add a comment to your test suite indicating each unit test that revealed a bug in your program!

Unit Tests

You may use the following sample unit tests to verify your understanding of the specifications above. Note: these are not an exclusive list of tests that I will use to grade your assignment, so to ensure as many points as possible, you should add many tests to this list (including those required above).

YarnTests.java

Solution Skeleton

The following .zip file contains a solution skeleton that you may use for your submission's starting point. It is highly recommended that you download this as a scaffold and work from there.

Yarn Solution Skeleton

Solution Restrictions

Read the following list of submission restrictions carefully! **Violating any restriction will net you a 0 on this homework!**

- You may NOT use ANY data structure from the Java collections framework in your solution (that includes ArrayLists).
- You may NOT add any methods or fields to the Yarn class' public interface. You may, however, add any private fields or methods that you like.
- Your classes and therefore source files must be named exactly as intimated above (as is in the Solution Skeleton).

Hints

The implementation of this assignment requires you to make some design decisions. However, here are some hints for how you might structure your own.

- Consider making helper methods that are private to the class -- these can reduce complex code to more readable segments that better organize your thoughts, and can be used to keep your code DRY (in the case of behavior that is repeated).
- Remember that Yarns do not care about the order of their contained strings; when removing a unique String from your Yarn, consider replacing its spot in your Strand[] items array with the last unique String, and then maintaining that only the first uniqueSize items in the array will be valid entries (to avoid null pointer exceptions).

Submission

We will be using Brightspace to submit this assignment. See the submission instructions below.

To submit this assignment:

1. Find the assignment's listing on Brightspace.
2. Add Yarn.java, YarnInterface.java, and YarnTests.java to the Homework 1 submission folder. Do NOT zip or enclose these files in a folder.
3. Click "Submit" at the bottom right hand corner of the screen.

Answer and feedback (at the bottom and where you see [AF])

```java
/**
 *  A Yarn is an unordered collection of Strings in which duplicates are allowed.
 *  A Yarn maps Strings to the number of occurrences of each String in the Yarn.
 */

public class Yarn implements YarnInterface {

    // --------------------------------------------------------
    // Fields
    // --------------------------------------------------------
    private Strand[] items;
    private int size;
    private int uniqueSize;
    private final int MAX_SIZE = 100;


    // --------------------------------------------------------
    // Constructors
    // --------------------------------------------------------
    Yarn () {
        this.items = new Strand[MAX_SIZE];
        this.size = 0;
        this.uniqueSize = 0;
    }

    Yarn (Yarn other) {

        this.items = new Strand[MAX_SIZE];
        for (int i = 0; i < other.getUniqueSize(); i++) {
            // >> [AF] Random space here

            this.items[i] = new Strand (other.items[i].text, other.items[i].count);
        }
        this.size = other.getSize();
        this.uniqueSize = other.getUniqueSize();

    }

    // --------------------------------------------------------
    // Methods
    // --------------------------------------------------------
    /**
     * @return true if Yarn is empty
```

```
44      */
45
46     public boolean isEmpty () {
47         return uniqueSize == 0;
48     }
49
50     /**
51      * @return size
52      */
53
54     public int getSize () {
55         return size;
56     }
57
58     /**
59      * @return uniqueSize
60      */
61
62     public int getUniqueSize () {
63         return uniqueSize;
64     }
65
66     /**
67      * @param toAdd String to be added to the Yarn.
68      * @return true if successful insertion, false if Yarn is at capacity
(already at 100 unique Strings).
69      */
70
71     public boolean insert (String toAdd) {
72         // >> [AF] As opposed to doing: if (condition) { BUNCH OF CODE },
73         // consider instead doing: if (!condition) { return x; } BUNCH OF
CODE AFTER
74         if (uniqueSize < MAX_SIZE) {
75             if (findIndex(toAdd) == -1) {
76                 items[uniqueSize] = new Strand(toAdd, 1);
77                 uniqueSize++;
78                 // >> [AF] You have common code between an if and an else
clause
79                 // (stuff that will happen regardless of whether the
condition is
80                 // true or false); pull this logic outside!
81                 size++;
82                 return true;
83             } else {
84                 items[findIndex(toAdd)].count++;
85                 size++;
```

```
                return true;
            }
        } else {
            return false;
        }
    }

    /**
     * @param toRemove String to be removed from the Yarn (only one occurrance)
     * @return the number of occurrences remaining after removal, (0 if toRemove does not exist in Yarn)
     */

    public int remove (String toRemove) {
        int idx = findIndex(toRemove);
        if (idx != -1) {
            if(items[idx].count == 1) {
                removeAll(toRemove);
            } else {
                items[idx].count--;
                size -= 1;
                return items[idx].count;
            }
        }
        return 0;
    }

    /**
     * @param toNuke String to be removed from the Yarn (all occurrances)
     */

     public void removeAll (String toNuke) {
         int idx = findIndex(toNuke);
         if (idx == -1) { return; }

         int toNukeCount = items[idx].count;
         swapStrand(idx, uniqueSize - 1);
         items[uniqueSize] = null;
         uniqueSize--;
         size -= toNukeCount;
         return;
     }

    /**
```

```
129      * @param toCount String to which the number of occurrances we want to
     know
130      * @return the number of occurrences of toCount in Yarn
131      */
132
133     public int count (String toCount) {
134
135         int idx = findIndex(toCount);
136         if (idx == -1) { return 0; }
137
138         return items[findIndex(toCount)].count;
139     }
140
141     /**
142      * @param toCheck String to which we want to check its occurrance in
     the Yarn
143      * @return true if the String toCheck appears at least once inside of
     the Yarn.
144      */
145
146     public boolean contains (String toCheck) {
147         return findIndex(toCheck) != -1;
148     }
149
150     /**
151      * @param n integer position to find in yarn
152      * @return the text found at position n
153      */
154     public String getNth(int n) {
155        if (n < size) {
156            int index = 0;
157            for (int i = 0; i <= n + 1; i ++) {
158                if (index <= n) {
159                    index += items[i].count;
160                } else {
161                    return items[i - 1].text;
162                }
163            }
164
165        }
166        throw new IndexOutOfBoundsException();
167    }
168
169
170     /**
```

```
171      * @return the String that occurs most frequently in the Yarn (if it
     is a tie
172      * return *either* of the most frequent. If the Yarn is empty return
     null.
173      */
174
175     public String getMostCommon () {
176         if (isEmpty()) {
177             return "null";
178         }
179         int mostCommonIdx = 0;
180         int valueMostCommonCount = items[0].count;
181
182         for (int i = 1; i < uniqueSize; i++) {
183             if (items[i].count > valueMostCommonCount ) {
184                 mostCommonIdx = i;
185                 valueMostCommonCount = items[i].count;
186             }
187         }
188         return items[mostCommonIdx].text;
189     }
190
191     /**
192      * @param other Yarn to swap
193     */
194
195     public void swap (Yarn other) {
196         // >> [AF] No need to create a new Yarn for swap; just swap the
     fields directly!
197         Yarn tempYarn = new Yarn(other);
198
199         other.items = this.items;
200         other.size = this.size;
201         other.uniqueSize = uniqueSize;
202
203         this.size = tempYarn.getSize();
204         this.uniqueSize = tempYarn.getUniqueSize();
205         this.items = tempYarn.items;
206         return;
207     }
208
209
210     @Override
211     public String toString () {
212         String toPrint = "{ ";
213         for (int i = 0; i < uniqueSize ; i ++) {
```

```
214             toPrint += "\"";
215             toPrint += items[i].text;
216             toPrint += "\": ";
217             toPrint += items[i].count;
218             if (i != uniqueSize - 1) {
219                 toPrint += ", ";
220             }
221         }
222         return toPrint + " }";
223     }
224 
225 
226     // ------------------------------------------------------------
227     // Static methods
228     // ------------------------------------------------------------
229 
230     /**
231      * @param y1,y2 Yarns that you want to knit (put together into one
    yarn)
232      * @return knitted Yarn (y1 together with y2)
233      */
234 
235     public static Yarn knit (Yarn y1, Yarn y2) {
236         Yarn knitted = new Yarn(y1);
237         for( int i = 0; i < y2.getSize(); i++){
238             knitted.insert(y2.getNth(i));
239         }
240         return knitted;
241     }
242 
243     /**
244      * @param y1,y2 Yarns that you want to tear (put y1 together with y2
    except for elements of y2 that are in y1 alredy)
245      * @return teared Yarn
246      */
247 
248     public static Yarn tear (Yarn y1, Yarn y2) {
249         Yarn diff = new Yarn(y1);
250         // >> [AF] Fairly inefficient if size of y2 is large (could be in
    the millions!);
251         // see solution for how to operate on the items array itself
252         for( int i = 0; i < y2.getSize(); i++){
253             diff.remove(y2.getNth(i));
254         }
255         return diff;
256     }
```

```
257
258     /**
259      * @param y1,y2 two Yarns
260      * @return true if y1 and y2 contain the exact same unique Strings and
    String occurrences
261      */
262
263     // >> [AF] sameYarn can be done in 1 line! See solution to experience
    the
264     // zen for yourself!
265     public static boolean sameYarn (Yarn y1, Yarn y2) {
266
267         if (y1.getUniqueSize() != y2.getUniqueSize()) { return false; }
268         if (y1.getSize() != y2.getSize()) { return false; }
269
270         for (int i = 0; i < y1.getUniqueSize(); i++) {
271             if (y2.count(y1.items[i].text) != y1.items[i].count ) {return
    false;}
272         }
273         return true;
274     }
275
276
277     // ------------------------------------------------------------
278     // Private helper methods
279     // ------------------------------------------------------------
280     /*finds the current index of the Entry with text "toFind"
281     * returns -1 if not found*/
282     private int findIndex(String toFind) {
283         for (int i = 0; i < uniqueSize; i++){
284             if (items[i].text.equals(toFind)) {
285                 return i;
286             }
287         }
288         return -1;
289     }
290
291     private void swapStrand (int idx1, int idx2) {
292     if (idx1 > uniqueSize || idx1 < 0 || idx2 > uniqueSize || idx2 < 0 ) {
293         return;
294     }
295     Strand save = this.items[idx1];
296     this.items[idx1] = this.items[idx2];
297     this.items[idx2] = save;
298 // >> [AF] These brackets are weirdly indented -- what do they go to? :O
299 }
```

```
300
301 }
302
303 class Strand {
304     String text;
305     int count;
306
307     Strand (String s, int c) {
308         text = s;
309         count = c;
310     }
311 }
312
313 // >> [AF] >>> Style Quality: Excellent <<<
314 // Legend: [X] = Needs major improvement
315 //         [~] = Needs some improvement
316 //         [ ] = You done good!
317 // Your Checklist:
318 // [ ] Variables named to clearly indicate purpose
319 // [ ] Adequate documentation of code fragments requiring it
320 // [ ] Kept code DRY (Didn't Repeat Yourself)
321 // [ ] Appropriate definition, and usage of, helper methods
322 // [~] Spacing and indents consistent and appropriate
```

Homework 2 - Description

Title	Date Posted	Date Due
Homework 2: Yarn 2: The Weavening	9 / 27 / 17	10 / 20 / 17

Remember Yarns? They're back...

If you've somehow forgotten the focus of the first and only homework you've just completed, take a moment to remind yourself here:

Homework 1

However, unlike in Homework 1, we'll be implementing our Homework 2 Yarns as... you guessed it... Linked Lists.

In Homework 2, you will be designing the LinkedYarn class, which has an almost identical interface compared to HW1, but the details "under the hood" will be significantly different.

In addition to modifying the data structure of your LinkedYarns, you will also be adding a LinkedYarn.Iterator class to allow for iteration through the Strings in your Linked Yarn.

The details for accomplishing this task are... well... detailed next.

Specifications

Problem 1[90 points]

The following section instructs you on how to implement the LinkedYarn and LinkedYarn.Iterator classes, as well as expectations for each method. Read through this section and the "Restrictions" section that follow before you begin any coding!

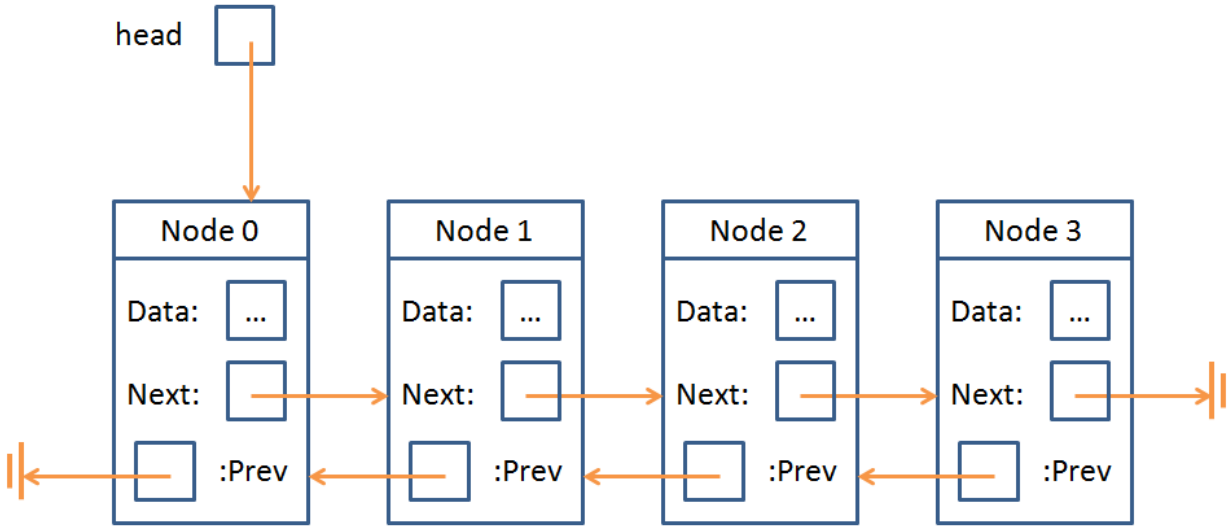
Note: if you wish to use any code in your submission from my notes or my solutions to classwork / homework, you may do so without need for attribution.

LinkedYarn

We'll begin by examining the LinkedYarn changes from HW1, and continue by detailing the Iterator's requirements.

For HW2, we will be implementing LinkedYarns using a **doubly linked list**.

As a refresher, here's what a doubly linked list looks like, pictorially:



LinkedYarn Fields

Your LinkedYarn objects must be capable of accommodating an **arbitrary** number of unique Strings (i.e., no limit any more like there was in HW1)!

Since our list is doubly linked, each Node (see class definition below) in the list holds a unique String in the LinkedYarn, the number of occurrences of that String, and references to the next and previous Nodes in the list (Nodes replace our notion of Strands from HW1).

```
class Node {
    Node next, prev;
    String text;
    int count;

    Node (String t, int c) {
        text = t;
        count = c;
    }
}
```

Note: the user of your Yarn class will never know about Node objects -- these are merely convenient record-keeping entities to be used in the LinkedYarn class' private fields.

For basic record keeping, our Yarn class should maintain the following fields (plus another detailed later):

- `private Node head;` a reference to the first Node in the list; `null` when the list is empty.
- `private int size;` the number of Strings currently in your LinkedYarn (counting duplicates separately).
- `private int uniqueSize;` the number of unique Strings currently in your LinkedYarn (counting duplicates as 1).
- `private int modCount;` (described later in LinkedYarn.Iterator spec) tracks the number of modifications made to this LinkedYarn.

You may add any additional fields that help you implement the rest of the project.

LinkedYarn Constructors

You will define:

- **A default constructor** for the LinkedYarn class that correctly instantiates its fields to represent an empty LinkedYarn (up to you to decide how to implement)
- **A copy constructor** for the LinkedYarn class that creates a deep copy of another given LinkedYarn such that each hold the same strings, but share no Nodes.

If you have any additional fields that you have defined, you should initialize them here.

LinkedYarn Methods

Your LinkedYarn class will implement the following interface, with individual method descriptions to follow.

LinkedYarnInterface.java

```
package linked_yarn;

public interface LinkedYarnInterface {

    boolean isEmpty ();
    int getSize ();
    int getUniqueSize ();
    void insert (String toAdd);
    int remove (String toRemove);
    int count (String toCount);
    void removeAll (String toNuke);
    boolean contains (String toCheck);
    String getMostCommon ();
    void swap (LinkedYarn other);
    LinkedYarn.Iterator getIterator ();

}
```

Methods
`boolean isEmpty();` Returns true if the LinkedYarn has no Strings inside, false otherwise.
`int getSize();` Returns the current size of the LinkedYarn (i.e., the number of Strings inside, counting duplicates separately).
`int getUniqueSize();` Returns the number of unique Strings in the LinkedYarn (counting duplicates as 1).
`void insert (String toAdd);` Adds the String toAdd to the LinkedYarn.Increments `modCount`.
`int remove (String toRemove);` Removes 1 occurrence of the String toRemove from the LinkedYarn, and returns the number of occurrences remaining after removal.If toRemove does not exist in the LinkedYarn, simply return 0 and do nothing.Increments `modCount`.
`void removeAll (String toNuke);` Removes ALL occurrences of the String toNuke from the LinkedYarn.If toNuke does not exist in the LinkedYarn, do nothing.Increments `modCount`.
`int count (String toCount);` Return the number of occurrences of String toCount found in the LinkedYarn.
`boolean contains (String toCheck);` Returns true if the String toCheck appears at least once inside of the LinkedYarn.
`String getMostCommon ();` Returns the String that occurs most frequently in the LinkedYarn. In the event of a tie, you may return *either* of the most frequent. If the LinkedYarn is empty, return null.
`void swap (LinkedYarn other);` Swaps the contents of the calling LinkedYarn and the other specified.Increments `modCount` for both `this` and `other`. Restriction: you may NOT use iteration/recursion to implement this method!
`LinkedYarn.Iterator getIterator ();` Returns a reference to a new LinkedYarn.Iterator object initialized at the head.If the LinkedYarn is empty, instead `throw new IllegalStateException();`

LinkedYarn Static Methods

In addition to the above methods, you must implement the following 3 static methods:

Methods
`static LinkedYarn knit (LinkedYarn y1, LinkedYarn y2);` Returns a *new* LinkedYarn object consisting of a combination of all String occurrences from y1 and y2.
`static LinkedYarn tear (LinkedYarn y1, LinkedYarn y2);` Returns a *new* LinkedYarn object consisting of all String occurrences from y1 that do NOT appear in y2.
`static boolean sameYarn (LinkedYarn y1, LinkedYarn y2);` Returns true if y1 and y2 contain the exact same unique Strings and String occurrences (i.e., the same Strings and the same counts of each String).Note: because order does not matter for Yarns, the Strings in y1 and y2 can be found in different orders but still be considered equivalent.Refer to the notes in the above Description section for examples of equivalent Yarns.

LinkedYarn.Iterator

Now that we've reviewed changes to the LinkedYarn class from HW1, let's take a look at the Iterator that we will be implementing in HW2.

Because our list is doubly linked, our Iterators will be capable of traveling forward and backward in the list.

Iterators will be used to iterate through the String **occurrences** of the LinkedYarn (not just the unique Strings).

For example, if our LinkedYarn contained: {"dup": 2, "unique": 1}, then an Iterator defined on that LinkedYarn would step through both occurrences of "dup" individually before progressing to the "unique" String (see unit tests for more on this).

We will be designing a **fast fail iterator** for our LinkedYarns, which means that our Iterators will be considered invalid (and therefore unusable) if any modifications (i.e., insertion, deletion, etc.) are done to the host LinkedYarn that were not done by the Iterator itself.

We'll track this rather cleverly:

1. Our LinkedYarns will increment a counter (`modCount`) every time they are modified.
2. New Iterators created on each LinkedYarn will have *their* modification counter (`itModCount`) set to the same value.
3. If the Iterator itself ever modifies the LinkedYarn, then both the LinkedYarn's `modCount` *and* the Iterator's `itModCount` are incremented, indicating that the two are still in sync.
4. If those values ever disagree, then it means that the LinkedYarn was modified outside of the Iterator's influence, and so that Iterator will be considered invalid.

As such, we'll design our iterators so that we will never unsafely use them to access or modify any element of the LinkedYarn.

Iterator Fields

- `private Node current;` a reference to the Node that the Iterator is currently "pointing at" in the LinkedYarn.
- `private LinkedYarn owner;` a reference to the LinkedYarn on which the Iterator was created.
- `private int itModCount;` the modification count for this Iterator. Valid iterators have the same modCount as their owner.

You may add any additional fields that help you implement your Iterator and its methods, defined below.

Iterator Constructor

Define one parameterized constructor `Iterator (LinkedYarn y);` that instantiates the Iterator at the head of the given LinkedYarn, making sure to set its `itModCounter` to that of its owner's `modCount`.

Iterator Methods

Your Iterator will implement the following interface, with method descriptions to follow:

```
1  package linked_yarn;
2
3  public interface LinkedYarnIteratorInterface {
4
5      boolean isValid ();
6      boolean hasNext ();
7      boolean hasPrev ();
8      String getString ();
9      void next ();
10     void prev ();
11     void replaceAll (String toReplaceWith);
12
13 }
```

Methods
`public boolean isValid ();` Returns true if this Iterator's `itModCount` agrees with that of its `owner`'s `modCount`, false otherwise.
`public boolean hasNext ();` Returns true if there is another String in the LinkedYarn after the `current`.This could be another occurrence of the String in the Node currently pointed at, or a separate String and Node entirely.Return false if either this Iterator is invalid or the `current` String is the last in the LinkedYarn.
`public boolean hasPrev ();` Returns true if there is another String in the LinkedYarn before the `current`.This could be another occurrence of the String in the Node currently pointed at, or a separate String and Node entirely.Return false if either this Iterator is invalid or the `current` String is the first in the LinkedYarn.
`public String getString ();` Returns the String that the Iterator is currently pointing at (i.e., the text of the Node it is referring to).If the Iterator is invalid, return `null` instead.
`public void next ();` Advances this Iterator to the next String occurrence in the LinkedYarn.This could be another occurrence of the String in the Node currently pointed at, or a separate String and Node entirely.If there is *no* next String occurrence in the LinkedYarn, `throw new NoSuchElementException();` (Note, the above requires that you `import java.util.NoSuchElementException;`, which is done for you in the solution skeleton).If the Iterator is invalid, `throw new IllegalStateException();` instead.
`public void prev ();` Regresses this Iterator to the previous String occurrence in the LinkedYarn.This could be another occurrence of the String in the Node currently pointed at, or a separate String and Node entirely.If there is *no* previous String occurrence in the LinkedYarn, `throw new NoSuchElementException();` (Note, the above requires that you `import java.util.NoSuchElementException;`, which is done for you in the solution skeleton).If the Iterator is invalid, `throw new IllegalStateException();` instead.
`public void replaceAll (String toReplaceWith);` Replaces *all* occurrences of the String that this Iterator currently points to with that given in the parameter `toReplaceWith`.Increments this Iterator's `itModCount` *and* its `owner`'s `modCount` (since this Iterator performed the modification, and so is still in sync with the host LinkedYarn).If the Iterator is invalid, `throw new IllegalStateException();` instead.

Let's make sure we understand how our Iterators are meant to behave (the following tests are also in the sample unit tests given; see next section):

```
...

@Test
public void testIteratorBasics() {
    ball.insert("a");
    ball.insert("a");
    ball.insert("a");
    ball.insert("b");
    LinkedYarn.Iterator it = ball.getIterator();

    // Test next()
    LinkedYarn dolly = ball.clone();
    while (true) {
        String gotten = it.getString();
        assertTrue(dolly.contains(gotten));
        dolly.remove(gotten);
        if (it.hasNext()) {it.next();} else {break;}
    }
    assertTrue(dolly.isEmpty());
    assertFalse(it.hasNext());

    // Test prev()
    dolly = ball.clone();
    while (true) {
        String gotten = it.getString();
        assertTrue(dolly.contains(gotten));
        dolly.remove(gotten);
        if (it.hasPrev()) {it.prev();} else {break;}
    }
    assertTrue(dolly.isEmpty());
    assertFalse(it.hasPrev());

    int countOfReplaced = ball.count(it.getString());
    it.replaceAll("replaced!");
    assertEquals(countOfReplaced, ball.count("replaced!"));
    assertTrue(it.isValid());

    ball.insert("c");
    assertFalse(it.isValid());
}

...
```

You are expected to thoroughly test your Iterator methods! Do not rely on the above as sufficient testing to verify their functionality.

Problem 2[10 Points]

Write a short report (1 paragraph min., 1 page max.) comparing your HW1 Yarn implementation to your HW2 LinkedYarn. Your report should highlight:

- Which operations felt easier / harder to program or execute between the sequential vs. linked list implementation (no formal analysis required).
- A scenario where you would want to use the sequential list implementation vs. the linked list one, and another scenario vice versa.

Save your report in a comment block at the end of your LinkedYarn.java class (i.e., the last thing in that file).

Unit Tests

You may use the following sample unit tests to verify your understanding of the specifications above. Note: these are not an exclusive list of tests that I will use to grade your assignment, so to ensure as many points as possible, you should add many tests to this list (including those required above).

Unlike in HW1, you are not required to *submit* any test case modifications, but if you choose not to make any more entirely, then ask yourself: do you feel lucky, punk? Do ya?

LinkedYarnTests.java

Solution Skeleton

The following .zip file contains a solution skeleton that you may use for your submission's starting point. It is highly recommended that you download this as a scaffold and work from there.

LinkedYarn Solution Skeleton

Solution Restrictions

Read the following list of submission restrictions carefully! **Violating any restriction will net you a 0 on this homework!**

- You may NOT use ANY data structure from the Java collections framework in your solution (that includes LinkedList). For that matter, you may not use *any* data structure that you did not create yourself! When in doubt, ask.
- You may NOT add any methods or fields to the Yarn class' public interface. You may, however, add any private fields or methods that you like.

- Your classes and therefore source files must be named exactly as intimated above (as is in the Solution Skeleton).

Hints

The implementation of this assignment requires you to make some design decisions. However, here are some hints for how you might structure your own.

- Consider making helper methods that are private to the class -- these can reduce complex code to more readable segments that better organize your thoughts, and can be used to keep your code DRY (in the case of behavior that is repeated).
- When in doubt: Draw. Everything. Out. Programming that requires tons of reference manipulation is best visualized with pictures! Got a bug? Make sure your code matches your pictorial representation of the operation, and when all else fails, try to use the debugger.

Submission

We will be using Brightspace to submit this assignment. See the submission instructions below.

To submit this assignment:

1. Find the assignment's listing on Brightspace.
2. Click the "Attach file" dialogue and add LinkedYarn.java.
3. Click "Submit" at the bottom right hand corner of the screen.

Answer and feedback (at the bottom and where you see [AF])

```
1  import java.util.NoSuchElementException;
2  /**
3   *  A Yarn is an unordered collection of Strings in which duplicates are
   allowed.
4   *  A Yarn maps Strings to the number of occurrences of each String in the
   Yarn.
5   */
6
7  public class LinkedYarn implements LinkedYarnInterface {
8
9      // ------------------------------------------------------------
10     // Fields
11     // ------------------------------------------------------------
12     private Node head;
```

```java
    private int size, uniqueSize, modCount;


    // ------------------------------------------------------------
    // Constructors
    // ------------------------------------------------------------
    LinkedYarn () {
        this.head = null;
        this.size = 0;
        this.uniqueSize = 0;
        this.modCount = 0;
    }

    LinkedYarn (LinkedYarn other) {
        this.head = null;
        this.size = 0;
        this.uniqueSize = 0;
        this.modCount = 0;
        Node current = other.head;
        for (int i = 0; i < other.uniqueSize; i++) {
            this.insertNode(current.text, current.count);
            current = current.next;
        }
    }


    // ------------------------------------------------------------
    // Methods
    // ------------------------------------------------------------
    // >> [AF] Great java-doc-umentation! :)
    /**
     * @return true if Yarn is empty
     */
    public boolean isEmpty () {
        return size == 0;
    }

    /**
     * @return size
     */
    public int getSize () {
        return size;
    }

    /**
     * @return uniqueSize
```

```
59      */
60     public int getUniqueSize () {
61         return uniqueSize;
62     }
63 
64     /**
65      * @param toAdd String to be added to the LinkedYarn.
66      */
67     public void insert (String toAdd) {
68         // >> [AF] Don't need to both check contains and call find -- find
   will always tell
69         // you whether or not the given string is in the LinkedYarn
   already
70         if (this.contains(toAdd)) {
71             find(toAdd).count ++;
72             size ++;
73         } else {
74             Node toInsert = new Node(toAdd, 1);
75             toInsert.next = head;
76             toInsert.prev = null;
77             if (head != null) { head.prev = toInsert; }
78             this.head = toInsert;
79             size ++;
80             uniqueSize ++;
81         }
82         modCount ++;
83     }
84 
85     /**
86      * @param toRemove String to be removed from the LinkedYarn (only one
   occurrance)
87      * @return the number of occurrences remaining after removal
88      */
89     public int remove (String toRemove) {
90         Node nodeToRemoveFrom = find(toRemove);
91         if ( nodeToRemoveFrom == null) {return 0;}
92         if (nodeToRemoveFrom.count == 1) {
93             removeAll(toRemove);
94         } else {
95             size --;
96             nodeToRemoveFrom.count --;
97             return nodeToRemoveFrom.count;
98             // >> [AF] Weird indent on this bracket
99             }
100         modCount ++;
101         return 0;
```

```
102     }
103
104     /**
105      * @param toNuke String to be removed from the LinkedYarn (all
    occurrances)
106      */
107     public void removeAll (String toNuke) {
108         if (!this.contains(toNuke)) {return;}
109         Node nodeToNuke = find(toNuke);
110         size -= nodeToNuke.count;
111         uniqueSize --;
112         modCount ++;
113         if (nodeToNuke == null) {return;}
114         if (nodeToNuke == head) { head = nodeToNuke.next; }
115         if (nodeToNuke.prev != null) {
116             nodeToNuke.prev.next = nodeToNuke.next;
117         }
118     }
119
120     /**
121      * @param toCount String to which the number of occurrances we want to
    know
122      * @return the number of occurrences of toCount
123      */
124     public int count (String toCount) {
125         if (!this.contains(toCount)) {return 0;}
126         return find(toCount).count;
127     }
128
129     /**
130      * @param toCheck String to which we want to check its occurrance
131      * @return true if the String toCheck appears at least once
132      */
133     public boolean contains (String toCheck) {
134         if (this.isEmpty()) { return false; }
135         // >> [AF] Oh nooo! Shouldn't ever use Iterators in your solution;
    these are
136         // meant for the user to interact with a LinkedYarn's stored
    strings, but are
137         // wildly inefficient for us to implement these methods (I
    mentioned this in class)
138         // >> [AF] Why not just use your find method like above?
139         Iterator iterator = getIterator();
140         while (!iterator.getString().equals(toCheck) &&
    iterator.hasNext()) {
141             iterator.next();
```

```
        }
        return iterator.getString().equals(toCheck);
    }

    /**
     * @return the String that occurs most frequently in the LinkedYarn (if it is a tie
     * @return *either* of the most frequent. If the LinkedYarn is empty return null.
     */
    public String getMostCommon () {
        if (size == 0) {return null;}
        Node mostCommon = head;
        Iterator iterator = getIterator();
        while (iterator.hasNext()) {
            iterator.next();
            mostCommon = iterator.current.count > mostCommon.count ? iterator.current : mostCommon;
        }
        return mostCommon.text;
    }

    /**
     * @param other LinkedYarn to swap
    */
    public void swap (LinkedYarn other) {
        Node tempHead = head;
        int tempSize = size,
            tempUniqueSize = uniqueSize,
            tempModCount = modCount;

        head = other.head;
        size = other.size;
        uniqueSize = other.uniqueSize;
        modCount = other.modCount;

        other.head = tempHead;
        other.size = tempSize;
        other.uniqueSize = tempUniqueSize;
        other.modCount = tempModCount;
    }

    /**
     * @return iterator for the LinkedYarn
    */
    public LinkedYarn.Iterator getIterator () {
```

```
185         return new Iterator(this);
186     }
187 
188 
189     // ------------------------------------------------------------
190     // Static methods
191     // ------------------------------------------------------------
192 
193     /**
194      * @param y1,y2 LinkedYarns that you want to knit (put together into
    one Linkedyarn)
195      * @return knitted LinkedYarn (y1 together with y2)
196      */
197     public static LinkedYarn knit (LinkedYarn y1, LinkedYarn y2) {
198         LinkedYarn result = new LinkedYarn(y1);
199         Node current = y2.head;
200         for (int i = 0; i < y2.uniqueSize; i++) {
201             result.insertOccurrences(current.text, current.count);
202             current = current.next;
203         }
204         return result;
205     }
206 
207     /**
208      * @param y1,y2 LinkedYarns that you want to tear (put y1 together
    with y2 except for elements of y2 that are in y1 alredy)
209      * @return teared LinkedYarn
210      */
211     public static LinkedYarn tear (LinkedYarn y1, LinkedYarn y2) {
212         LinkedYarn result = new LinkedYarn(y1);
213         Node current = y2.head;
214         for (int i = 0; i < y2.uniqueSize; i++) {
215             result.removeOccurrences(current.text, current.count);
216             current = current.next;
217         }
218         return result;
219     }
220 
221     /**
222      * @param y1,y2 two LinkedYarns
223      * @return true if y1 and y2 contain the exact same unique Strings and
    String occurrences
224      */
225     public static boolean sameYarn (LinkedYarn y1, LinkedYarn y2) {
226         return tear(y1, y2).isEmpty() && tear(y2, y1).isEmpty();
227     }
```

```
228
229
230     // ------------------------------------------------------------
231     // Private helper methods
232     // ------------------------------------------------------------
233     /**
234      * @param word to find in the LinkedYarn
235      * @return the Node that contains the word and null if LinkedYarn does
    not contain it
236      */
237     private Node find(String word) {
238         if (!this.contains(word)) {return null;}
239         Iterator iterator = this.getIterator();
240         while (!iterator.getString().equals(word) && iterator.hasNext()) {
241             iterator.next();
242         }
243         return iterator.getString().equals(word) ? iterator.current :
    null;
244     }
245
246     /**
247      * @param text to be inserted in the LinkedYarn and count how many
    times
248      */
249     private void insertOccurrences (String text, int countNumber) {
250         // >> [AF] Oh no! What if countNumber is 2,000,000?! Just find the
    node you want
251         // to insert into, and then add to its count directly rather than
    looping!
252         for (int i = 0; i < countNumber; i++) {
253             this.insert(text);
254         }
255     }
256
257     /**
258      * @param text to be removed in the LinkedYarn and count how many
    times
259      */
260     private void removeOccurrences (String text, int countNumber) {
261         for (int i = 0; i < countNumber; i++) {
262             this.remove(text);
263         }
264     }
265
266     /**
267      * @param text,count insert a node with
```

```
268      */
269     private void insertNode (String textToAdd, int countToAdd) {
270         Node currentHead = head;
271         head = new Node(textToAdd, countToAdd);
272         head.next = currentHead;
273         size += countToAdd;
274         uniqueSize++;
275         modCount += countToAdd;
276     }
277 
278     @Override
279     public String toString(){
280         if (this.isEmpty()) {
281             return "{ }";
282         } else {
283             Iterator iterator = this.getIterator();
284             String toPrint = "{ ";
285             toPrint += iterator.getString();
286             toPrint += ": ";
287             toPrint += this.count(iterator.getString());
288             while (iterator.hasNext()) {
289                 iterator.next();
290                 toPrint += ", ";
291                 toPrint += iterator.getString();
292                 toPrint += ": ";
293                 toPrint += this.count(iterator.getString());
294             }
295             toPrint += " }";
296             return toPrint;
297         }
298     }
299 
300     // ------------------------------------------------------------
301     // Inner Classes
302     // ------------------------------------------------------------
303 
304     public class Iterator implements LinkedYarnIteratorInterface {
305         LinkedYarn owner;
306         Node current;
307         int itModCount;
308         private int index; // designates the position inside each node (1
    being 1st occurrance)
309 
310         Iterator (LinkedYarn y) {
311             owner = y;
312             current = y.head;
```

```
            itModCount = y.modCount;
            index = 1;
        }

        public boolean hasNext () {
            if (owner.isEmpty()) {return false;}
            return index < current.count || current.next != null;
        }

        public boolean hasPrev () {
            if (owner.isEmpty()) {return false;}
            return index > 1 || current.prev != null;
        }

        public boolean isValid () {
            return itModCount == owner.modCount;
        }

        public String getString () {
            return this.isValid() && !owner.isEmpty() ? current.text :
null;
        }

        public void next () {
            if (!isValid()) {throw new IllegalStateException();}
            if (!hasNext()) {throw new NoSuchElementException();}
                if (index == current.count) {
                    current = current.next;
                    index = 1;
                } else {
                    index++;
                }
        }

        public void prev () {
            if (!isValid()) {throw new IllegalStateException();}
            if (!hasPrev()) {throw new NoSuchElementException();}
            if (index == 1) {
                current = current.prev;
                index = current.count;
            } else {
                index --;
            }
        }

        public void replaceAll (String toReplaceWith) {
```

```
358             if (isValid()) {
359                 current.text = toReplaceWith;
360                 itModCount ++;
361                 owner.modCount ++;
362             } else {
363                 throw new IllegalStateException();
364             }
365         }
366 
367     }
368 
369     class Node {
370         Node next, prev;
371         String text;
372         int count;
373 
374         Node (String t, int c) {
375             text = t;
376             count = c;
377         }
378     }
379 
380 }
381 
382 /**
383  * In my opinion programming both the sequential list and the linked list
    felt
384  * really similar. Sequential list was more intuitive since I am more used
    to
385  * working with arrays than I do with references. Furthermore, eventhough
    we went
386  * through the iterator class in class, it was not as intuitive how to do
    all the
387  * methods such that they would work with the LinkedYarn.
388  *
389  * On the other hand, I feel like some operatations' logic (such as add,
390  * remove all, remove) where harder to do on the Yarn class given that we
    had no
391  * prior experience of how a Yarn worked. When I got to implement the
    LinkedYarn,
392  * I already had an intuition of the steps that such methods needed to
    follow in
393  * order to work correctly, the only thing that changed was doing it in a
    Linked
394  * List rather than an array, but the algorithm was the same. Because of
    this, I
```

```
 * believe the level of "difficulty" between both balanced out such that both were
 * at the same level.
 *
 * I wold rather use a linked list implementation on a scenario where I do not
 * know a priori how many items will be on the list (and know that it will be large).
 * An array list would have to extend its size many times if the size is large but
 * for a linked list there is no need. Furthermore I would use linked lists in
 * scenarios where I want to do a stack or queue. In a sequential list this requires
 * constant shifting of the values (either to the right or left). However, on a
 * LinkedList there is no need to do this (can be done easily).
 *
 * On the other hand I would use an array list in a scenario where I would like
 * to have random access in constant time (need an iterator for LL). For example,
 * I had to code a program last semester for CMSI 282 where ecah time I had to access
 * the specific items in a list of items. I used an arraylist for this given that I
 * had to do this so many times and it is easier to do if you have a array (also
 * my list of items was fixed so there was no need for expansion).
 */

 // >> [AF] >>> Style Quality: Excellent <<<
// Legend: [X] = Needs major improvement
//         [~] = Needs some improvement
//         [ ] = You done good!
// Your Checklist:
// [ ] Variables named to clearly indicate purpose
// [ ] Adequate documentation of code fragments requiring it
// [ ] Kept code DRY (Didn't Repeat Yourself)
// [ ] Appropriate definition, and usage of, helper methods
// [ ] Spacing and indents consistent and appropriate
// [~] Appropriate consideration for runtime efficiency
// >> [AF] Stylistically excellent apart from needing to keep a closer eye on the efficiency
// of some methods
```

Homework 3 - Description

Title	Date Posted	Date Due
Homework 3: Yarnalysis	10 / 24 / 17	11 / 7 / 17

For your last assignment, you took a cursory look at the performances of Yarns and LinkedYarns.

In this assignment, you will now do so formally on a select set of methods from Homeworks 1 and 2.

The purpose: to become comfortable analyzing the asymptotic runtime complexity of code other than your own in real-application settings.

Specifications

While performing runtime analysis, sometimes our notion of input size is not clear cut. In particular:

- We might have multiple metrics of size (e.g., in a Yarn, we have both the notion of size and uniqueSize).
- We might have multiple data structures whose sizes may be different, but on which an algorithm's runtime depends (e.g., a Yarn's tear method).

For example, consider the following method that takes 2 input arrays of ints:

```
/*
 * Returns true iff all of a1's elements are found
 * within a2
 */
public static boolean isSubset (int[] a1, int[] a2) {
    for (int i = 0; i < a1.length; i++) {
        boolean contained = false;
        for (int j = 0; j < a2.length; j++) {
            if (a1[i] == a2[j]) {
                contained = true;
                break;
            }
        }
        if (!contained) {return false;}
    }
    return true;
}
```

Now, let `n = a1.length` and `m = a2.length`, then the complexity of the above is O(n*m).

Notice that it would be **incorrect** to say that `isSubset` has a complexity of O(n^2) since a1 and a2 may not have the same size.

With this in mind, we'll now explore the asymptotic complexities of some of our methods from Yarn and LinkedYarn as functions of either their size or uniqueSize, depending upon how each algorithm operates.

Problem 1[25 points]

You will use the HW1 solution for this problem:

Homework 1 Solution

For each of the following methods, provide the **worst case** Big-O asymptotic runtime complexities as a function of: `s` = the size of the Yarn (i.e., the number of individual String occurrences), OR `u` = the uniqueSize of the Yarn (i.e., the number of distinct Strings). **Show your work.**

For each method in the following section, assume that our Yarns could accommodate an infinite number of unique Strings (or else every answer would be O(1))

1. `removeAll`
2. `getNth`

Problem 2[25 points]

You will use the HW2 solution for this problem:

Homework 2 Solution

For each of the following methods, provide the **worst case** Big-O asymptotic runtime complexities as a function of: `s` = the size of the Yarn (i.e., the number of individual String occurrences), OR `u` = the uniqueSize of the LinkedYarn (i.e., the number of distinct Strings). **Show your work.**

1. `swap`
2. `insert`

Problem 3[50 points]

You will use the HW2 solution for this problem:

Homework 2 Solution

For each of the following methods, provide the **worst case** Big-O asymptotic runtime complexities as a function of: `s1, s2` = the size of the LinkedYarn (i.e., the number of individual String occurrences in y1 and y2, respectively), OR `u1, u2` = the uniqueSize of the LinkedYarn (i.e., the number of distinct Strings in y1 and y2, respectively). **Show your work.**

1. `knit`
2. LinkedYarn.java

```
  /*
   * commonThreads returns a new LinkedYarn composed of all
   * occurrences that are common between y1 and y2
   * [X] Warning: this implementation is not very good
   */
  public static LinkedYarn commonThreads (LinkedYarn y1, LinkedYarn y2)
{
      LinkedYarn result = new LinkedYarn(),
                 y2Clone = new LinkedYarn(y2);

      for (LinkedYarn.Iterator i1 = y1.getIterator(); i1.hasNext();
i1.next()) {
          String current = i1.getString();
          if (y2Clone.contains(current)) {
              result.insert(current);
              y2Clone.remove(current);
          }
      }

      return result;
  }
```

3. LinkedYarn.java

```
  /*
   * Alternative implementation of the above that is slightly better
   */
  public static LinkedYarn betterCommonThreads (LinkedYarn y1,
LinkedYarn y2) {
      LinkedYarn result = new LinkedYarn();
      for (Node curr1 = y1.head; curr1 != null; curr1 = curr1.next) {
          String text = curr1.text;
          int count1 = curr1.count,
              count2 = y2.count(text);
          if (count2 > 0) {
              result.insertOccurrences(text, Math.min(count1, count2));
          }
      }

      return result;
  }
```

Note: both of the above implementations are pretty bad; we'll look at better data structures for this task later in the course.

Hints

- Not sure where to begin? Review the examples and heuristics from our week 7 and 8 lectures.
- Worried about your answers for Problem 3? It's perfectly fine to have something that looks like O(u1 + s2); note that these are two separate size variables (u1 and s2), and so our heuristics cannot reduce the expression further.
- It's perfectly fine to simplify the cost of a particular statement with its big-O bounding. In general, you can use the properties of big-O notation to simplify your analysis, e.g.:
  - **Summation:** If f1 = O(g1(n)) and f2 = O(g2(n)) then f1 + f2 = O(g1(n) + g2(n))
  - **Product:** If f1 = O(g1(n)) and f2 = O(g2(n)) then f1 * f2 = O(g1(n) * g2(n))

Submission

We will be using Brightspace to submit this assignment. See the submission instructions below.

To submit this assignment:

1. Find the assignment's listing on Brightspace.
2. Click the "Attach file" dialogue and add a single report.txt plain text file with your work and answers. You need not copy over the entire Yarn / LinkedYarn files, but only those methods

that are relevant for the requested Yarnalysis.
3. Click "Submit" at the bottom right hand corner of the screen.

Answer

CMSI 281: Homework 3

Problem 1

removeAll:

find:

```
1  private int find (String s) {
2          for (int i = 0; i < uniqueSize; i++) { //u
3              if (items[i].text.equals(s)) { // C1
4                  return i;                  // C2
5              }
6          }
7          return -1;
8      }
```

$T(u, s) = u(C1 + C2) \implies O(u)$

removeFromBack:

```
1      private void replaceFromBack (int index) {
2          items[index] = items[uniqueSize-1]; //C1
3          items[uniqueSize-1] = null; //C2
4      }
```

$T(u, s) = C1 + C2 \implies O(1)$

removeOccurrances:

```
 private int removeOccurrences (String text, int count) {
        int index = find(text); //n

        // Case: no such string toRemove
        if (index == -1) {     //C1
            return 0;
        }

        Strand found = items[index];       // C2
        int newCount = found.count - count;//C3

        // Case: last instance to remove
        if (newCount <= 0) {// everything in here is constant lets just say C4 (all together)
            replaceFromBack(index); // it is constant (analysed before)
            size -= found.count;
            uniqueSize--;
            return 0;

        // Case: more than 1 Strand left
        } else {           // everything in here is also constatnso either way is constant worst case ( we will just keep the C4 above)
            found.count = newCount;
            size -= count;
            return newCount;
        }
    }
```

$$T(u,s) = u + C1 + C2 + C3 + C4 \implies O(u)$$

Then removeAll is...

```
public void removeAll (String toNuke) {
        int index = find(toNuke); // u
        if (index != -1) { //C1
            removeOccurrences(toNuke, items[index].count); //u
        }
    }
```

ANSWER

$$T(u,s) = u + C1n \implies O(u)$$

getNth:

```
public String getNth (int n) {
        if (n >= size || n < 0) {                    //C1
            throw new IllegalArgumentException();    //C2
        }

        for (int i = 0; i < uniqueSize; i++) {    //u
            Strand currentStrand = items[i];     //C3
            if (n < currentStrand.count) {      //C4
                return currentStrand.text;     //in worst case should never get here
            } else {
                n -= currentStrand.count;      //C5
            }
        }

        // Should never get here...
        return null;                          //C6
    }
```

ANSWER

$T(u,s) = C1 + C2 + u(C3 + C4 + C5) + C6 \implies O(u)$

Problem 2

insert:

prependNode

```
    private void prependNode (Node n) {
        Node oldHead = head;      //C1
        head = n;                //C2
        if (oldHead != null) {   //C3
            head.next = oldHead;  //C4
            oldHead.prev = head;   //C5
        }
    }
```

$T(u,s) = C1 + C2 + C3 + C4 + C5 \implies O(1)$

find

```
private Node find (String toFind) {
        for (Node curr = head; curr != null; curr = curr.next) { //u
            if (curr.text.equals(toFind)) {  //C2
                return curr;                //worst case never gets gere
            }
        }
        return null;                        //C3
    }
```

$T(u,s) = uC2 + C3 \implies O(u)$

insertOccurrences:

```
    private boolean insertOccurrences (String text, int count) {
        Node found = find(text);   //u

        // Case: new string, so add new Node
        if (found == null) {//C1(both cases in if and else are constant so either case worst case constant) lets just consider it goes in the if statement
            prependNode(new Node(text, count));           //C2
            uniqueSize++;                                 //C3

        // Case: existing string, so update count
        } else {
            found.count += count;
        }
        size += count;                                    //C4
        modCount++;                                       //C5

        return true;                                      //C6
    }
```

$T(u,s) = u + C1 + C2 + C3 + C4 + C5 + C6 \implies O(u)$

```
public void insert (String toAdd) {
        insertOccurrences(toAdd, 1); //u
    }
```

ANSWER

$T(u,s) = u \implies O(u)$

swap:

```
public void swap (LinkedYarn other) {
        Node tempHead = head;             //C1
        int tempSize = size,             //C2
            tempUniqueSize = uniqueSize; //C3

        head = other.head;                //C4
        size = other.size;               //C5
        uniqueSize = other.uniqueSize;  //C6

        other.head = tempHead;           //C7
        other.size = tempSize;          //C8
        other.uniqueSize = tempUniqueSize;   //C9
        modCount++;                          //C10
        other.modCount++;                   //C11
    }
```

$$T(u,s) = C1 + C2 + C3 + C4 + C5 + C6 + C7 + C8 + C9 + C10 + C11$$

ANSWER

$$\implies O(1)$$

Problem 3

knit:

Constructor:

```
    LinkedYarn (LinkedYarn other) {
        for (Node n = other.head; n != null; n = n.next) { //u1
            prependNode(new Node(n.text, n.count)); //C1 (constant from
above's analysis)
            size += n.count; //C2
            uniqueSize++; //C3
        }
    }
```

$$T(u1, u2, s1, s2) = u1(C1 + C2 + C3) \implies O(u1)$$

```
    public static LinkedYarn knit (LinkedYarn y1, LinkedYarn y2) {
        LinkedYarn result = new LinkedYarn(y1); //u1
        for (Node n = y2.head; n != null; n = n.next) { //u2
            result.insertOccurrences(n.text, n.count);  //u1
        }
        return result; //C1
    }
```

$$T(u1, u2, s1, s2) = u1 + (u2u1) + C1 = u1 + u1u2 + C1 \implies O(u1u2)$$

2

In this problem as in the previous ones, I will first analyze the complexity of the methods that are called in the function in which we want to measure complexity. Once this is done it will be easier to determine its complexity.

empty constructor

```
    LinkedYarn () {
        head = null;
        size = 0;
        uniqueSize = 0;
        modCount = 0;
    }
```

Clearly $O(1)$

creating an iterator

```
Iterator (LinkedYarn y) {
    owner = y;
    itModCount = y.modCount;
    current = y.head;
    onCount = 0;
}
```

Clearly $O(1)$

getIterator

```
public LinkedYarn.Iterator getIterator () {
       if (isEmpty()) {
           throw new IllegalStateException();
       }
       return new LinkedYarn.Iterator(this);
   }
```

Clearly $O(1)$

hasNext (iterator)

```
public boolean hasNext () {
    if (current.count > onCount+1) {return true;}
    return isValid() && current.next != null;
}
```

Clearly $O(1)$

veryfyIntegrity()

```
private void verifyIntegrity () {
    if (!isValid()) {
        throw new IllegalStateException();
    }
}
```

Clearly $O(1)$

next (iterator)

```
public void next () {
    verifyIntegrity(); //constant from above
    onCount++;
    if (onCount >= current.count) {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        current = current.next;
        onCount = 0;
    }
}
```

Clearly $O(1)$

isValid

```
public boolean isValid () {
    return owner.modCount == itModCount;
}
```

Clearly $O(1)$

getString

```
public String getString () {
    verifyIntegrity(); //constant
    return current.text;
}
```

Clearly $O(1)$

remove occurrances

```
private int removeOccurrences (String text, int count) {
    Node found = find(text); //u2 (the unique size) from problem 2

    // Case: no such string toRemove
    if (found == null) {
        return 0;
    }

    int newCount = found.count - count;
    modCount++;

    // Case: last instance to remove
    if (newCount <= 0) {
        deleteNode(found);
        size -= found.count;
        uniqueSize--;
        return 0;

    // Case: more than 1 entry left
    } else {
        found.count = newCount;
        size -= count;
        return newCount;
    }
```

In either case of this function (either of the if or else statements) all are constant so then $O(u2)$ is the complexity because of the find method.

remove

```
    public int remove (String toRemove) {
        return removeOccurrences(toRemove, 1); //u2
    }
```

clearly $O(u2)$ (see explanation for insert, this one is very similar)

contains

```
public boolean contains (String toCheck) {
    return find(toCheck) != null; //u2
}
```

clearly $O(u2)$

Now finally we can analize the following:

```
  /*
   * commonThreads returns a new LinkedYarn composed of all
   * occurrences that are common between y1 and y2
   * [X] Warning: this implementation is not very good
   */
  public static LinkedYarn commonThreads (LinkedYarn y1, LinkedYarn y2) {
      LinkedYarn result = new LinkedYarn(), //C1
                 y2Clone = new LinkedYarn(y2);     //u2

      for (LinkedYarn.Iterator i1 = y1.getIterator(); i1.hasNext();
i1.next()) { //s1
          String current = i1.getString(); //C2
          if (y2Clone.contains(current)) { //u2
              result.insert(current);    //u1
              y2Clone.remove(current);   //u2 (worst case y2clone is same
as y1 but in mirror order)
          }
      }

      return result;  //C3
  }
```

$T(u1, u2, s1, s2) = C1 + u2 + s1(C2 + u2 + u1 + u2) + C3$

$= C1 + u2 + s1(C2 + 2u2 + u1) + C3$

$= C1 + C3 + u2 + s1C2 + s1u2 + s1(u2 + u1)$

ANSWER

$\implies O(s1(u2 + u1))$

3

count:

```
    public int count (String toCount) {
        Node toFind = find(toCount); // u2
        return (toFind == null) ? 0 : toFind.count; //C1
    }
```

Clearly $O(u2)$

```
  /*
   * Alternative implementation of the above that is slightly better
   */
  public static LinkedYarn betterCommonThreads (LinkedYarn y1, LinkedYarn
y2) {
      LinkedYarn result = new LinkedYarn(); //C1
      for (Node curr1 = y1.head; curr1 != null; curr1 = curr1.next) { //u1
          String text = curr1.text; //C3
          int count1 = curr1.count, //C4
              count2 = y2.count(text); //u2
          if (count2 > 0) {            //C5
              result.insertOccurrences(text, Math.min(count1, count2));
//Math.min constant time. //u1
          }
      }

      return result; //C6
  }
```

$T(u1, u2, s1, s2) = C1 + u1(C3 + C4 + u2 + C5 + u1) + C6$

ANSWER

$\implies O(u1(u2 + u1))$

Homework 4 - Description

Title	Date Posted	Date Due
Homework 4: Reading Minds One Letter at a Time	10 / 31 / 17	11 / 30 / 17

Have you ever wondered how Google just seems to read your mind when you search for only part of your query?

Yes, that's right, Google was able to discern that my query was leading to "supercalifragilisticexpialidocious" (for any of you Mary Poppins fans out there).

Even more interesting, is that there's something called "superchillin", which I'm too afraid to research further.

As you might imagine, Google's specific process is the combination of many complex heuristics based on your personal search history, prevailing search trends, and the linkenesses of your search compared to those that were completed successfully.

However, in this assignment, we'll be examining a powerful tree-based data structure that can be used to efficiently perform a basic version of Google's so-called **autocomplete**.

Goal: implement a simplified version of Google's autocomplete feature using an efficient storage for known search terms.

Choosing a Data Structure

When presented with a problem like the above, your mind might race through all of the data structures you've learned hoping to find one that is objectively best for the task.

Let's review some of the putative problem requirements:

- We will be adding *many* search terms to our chosen data structure, which will consist of Strings representing search queries.
- Once we've constructed our data structure full of known search terms, we must be able to quickly look up whether any given String is contained within.
- Further, given only a portion of a String, we should be able to determine the most likely completion intended by the user.

So looking at the above, we might consider the most naive approach of storing every search term in a List and then scan through the List to find a match.

However, this is very expensive to store (doesn't take common parts of words into account), expensive to search (linear), and does not have a clear approach to suggesting autocompletions.

We might next consider binary search trees, which would allow us faster storage and search for queries if we take advantage of their alphabetical order.

However, it's still not clear how we can generate autocompletion results from a BST implementation...

Luckily, we can tweak a BST slightly to serve our purposes:

Ternary Search Trees

Ternary Search Trees are trees wherein each Node has at most 3 children with application-specific semantics for left, middle, and right children.

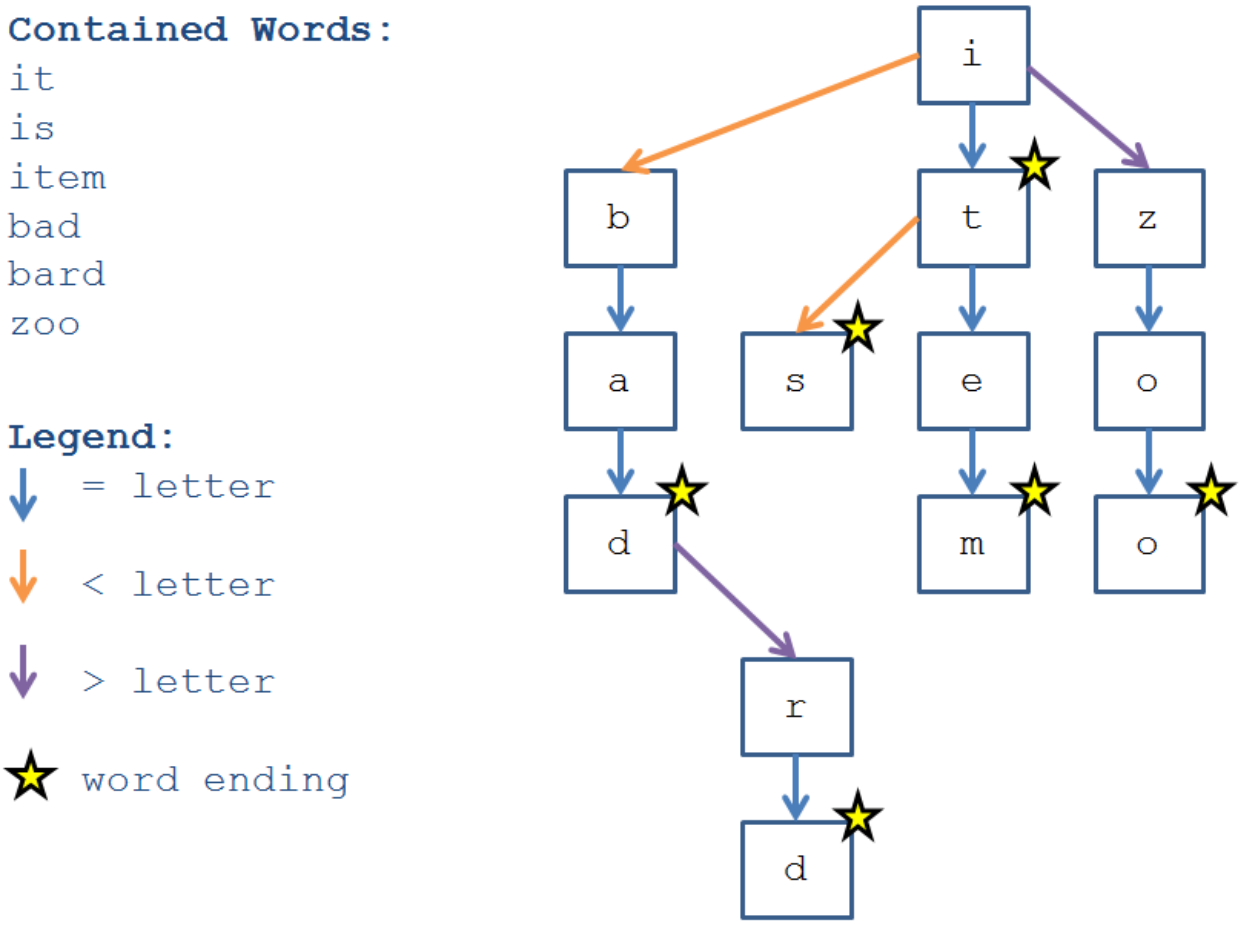
In our autocompleter application, we want to find a parsimonious way to store words that can then be queried, and find the "closest" predicted word from only the first few letters of it.

So, we'll adopt a ternary search tree for our purpose that has the following characteristics:

- Each node will store a letter of a word in the collection.
- The "middle" reference of every node will point to the next letter in the word, in sequence (just like a linked list).
- Because some words are actually prefixes of others (e.g., "it" is a word that is a prefix of "item"), we'll mark certain nodes as "word ends" to indicate that letters collected along middle paths may legally terminate at them.
- The "left" and "right" references of every node are possibly null, but when non-null, will point to nodes in which other words are to be formed using the previous middle path that led up to them as a prefix.
- In particular, a node to the left of another will possess a letter of another word that is alphabetically less than the parent, and a node to the right of another will possess a letter of another word that is alphabetically greater than the parent.
- We then use these trees to form words by starting at the root and "collecting" letters along middle paths that match our query / insertion and then act like binary search when the letters do not match at a node (i.e., look to the left if the letter is "less than" the current, or to the right when the letter is "greater").

Example

Whew! That's a lot of words. Why don't we actually look at an example?



Example Try adding another word to the above ternary tree. Where will its letters go? Which nodes will need to be marked as word ends?

Example Determine the algorithm for querying whether a word is contained within the ternary tree above. (hint: very close to binary search!)

Using the Ternary Tree

So now that you've seen the data structure, let's connect it back to our task: to create a functioning autocomplete mechanism.

The general workflow for this process will be as follows:

1. Insert search terms into the ternary tree to abide by the structure detailed above.
2. After search terms have been inserted, we may query the tree in a variety of ways:
    - Ask if the tree contains a specific search term.
    - Ask the tree to provide a suggested search term based on a given query string (which is possibly a fragment of a contained search term).
    - Ask the tree to provide a sorted list of all contained search terms.

The following specification details how we will implement the behavior above.

Specifications

Problem 1[100 points]

Implement the Autocompleter class that uses a Ternary Search Tree to provide the behavior described above / below. Each method is weighted equally for determining your final score on this assignment.

Autocompleter

Your Autocompleter class will implement the following interface.

AutocompleterInterface.java

```
package autocompleter;

import java.util.ArrayList;

public interface AutocompleterInterface {

    boolean isEmpty();
    void addTerm(String toAdd);
    boolean hasTerm(String query);
    String getSuggestedTerm(String query);
    ArrayList<String> getSortedTerms();

}
```

Methods
`boolean isEmpty();` Returns true if the Autocompleter has no search terms stored, false otherwise.
`void addTerm(String toAdd);` Adds the given search term `toAdd` to the Autocompleter by the method specified in the Ternary Tree section above.Note that for this simplified Autocompleter, the order in which terms are added to the Ternary Tree may influence the output of the `getSuggestedTerm` method detailed below. This is fine.Furthermore, the order in which terms are added can influence the efficiency of each operation if the tree becomes too linear. This, although not desirable, is fine given the time and difficulty expectations of the assignment. All inserted search terms are to be stored in their `normalized` format (see helper methods section below).
`boolean hasTerm(String query);` Returns true if the given `query` String exists within the Autocompleter, false otherwise. All query terms are to be referenced in their `normalized` format (see helper methods section below).
`String getSuggestedTerm(String query);` Returns the first* search term contained in the Autocompleter that possesses the query as a prefix (e.g., "it" is a prefix of "it" [exact match] and "item" [first two letters]).In the event that the given query is a prefix for more than one stored search term, either are acceptable return results.See the unit tests below for examples of this behavior (unit test with "goad" vs. "goat").In the event that the given query is a prefix for NO search term, return null. All query terms are to be referenced in their `normalized` format (see helper methods section below).
`ArrayList<String> getSortedTerms();` Returns an ArrayList of Strings consisting of the alphabetically sorted search terms within this Autocompleter.Alphabetic sorting is the same as how a dictionary sorts its entries, so for example, "ass" is considered a predecessor to "at" even though it has more letters.See the unit tests below for examples of this behavior.

Provided Helper Methods

I have provided two helper methods to assist you in your implementation:

Methods
`String normalizeTerm (String s);` Throws `IllegalArgumentException();` when `s` is null or empty.Used to normalize arguments to all of the assignment methods, as well as how the terms are stored.
`int compareChars (char c1, char c2);` Compares two characters and returns an integer representing their alphabetical ordering. In particular:Returns some integer less than 0 whenever `c1` alphabetically preceeds `c2`.Returns 0 whenever `c1` is the same character as `c2`.Returns some integer greater than 0 whenever `c1` alphabetically follows `c2`.This method is useful for constructing and then navigating your ternary search tree.

Assumptions

To simplify this assignment, we'll assume the following:

- You may assume there will be no punctuation, spaces, or numbers in any of the arguments to any of the above methods.
- You need make no assumptions about the order in which search terms are added to the Autocompleter so long as the above requirements are met.

Unit Tests

You may use the following sample unit tests to verify your understanding of the specifications above. Note: these are not an exclusive list of tests that I will use to grade your assignment, so to ensure as many points as possible, you should add many tests to this list (including those required above).

Unlike in HW1, you are not required to *submit* any test case modifications, but if you choose not to make any more entirely, then ask yourself: are you a gambling individual?

AutocompleterTests.java

Solution Skeleton

The following .zip file contains a solution skeleton that you may use for your submission's starting point. It is highly recommended that you download this as a scaffold and work from there.

Autocompleter Solution Skeleton

Solution Restrictions

Read the following list of submission restrictions carefully! **Violating any restriction will net you a 0 on this homework!**

- You may NOT use ANY data structure from the Java collections framework in your solution. For that matter, you may not use *any* data structure that you did not create yourself! When in doubt, ask.
- You may NOT add any methods or fields to the Autocompleter class' public interface. You may, however, add any private fields or methods that you like.
- Your classes and therefore source files must be named exactly as intimated above (as is in the Solution Skeleton), and your submission should mimic the solution skeleton's package structure.

Hints

The implementation of this assignment requires you to make some design decisions. However, here are some hints for how you might structure your own.

- The above methods can be implemented iteratively or using recursion, though some methods will be vastly simplified by a clever choice of one or the other.
- Want to use recursion to implement method but also want it to have different parameters? Make a private helper method and then just call that helper from the public one!
- Although there are only a few methods for you to implement in this assignment, beware: some of the algorithms may feel non-trivial, especially if you are unused to recursion. Leave yourself ample time to test, debug, and ask questions!

Submission

We will be using Brightspace to submit this assignment. See the submission instructions below.

To submit this assignment:

1. Find the assignment's listing on Brightspace.
2. Click the "Attach file" dialogue and add your Autocompleter.java file.
3. Click "Submit" at the bottom right hand corner of the screen.

Answer

```
1 import java.util.ArrayList;
2
```

```java
public class Autocompleter implements AutocompleterInterface {

    // ------------------------------------------------------------
    // Fields
    // ------------------------------------------------------------
    TTNode root;
    private ArrayList<String> terms;
    // ------------------------------------------------------------
    // Constructor
    // ------------------------------------------------------------
    Autocompleter () {
        root = null;
    }


    // ------------------------------------------------------------
    // Methods
    // ------------------------------------------------------------

    public boolean isEmpty () {
        return root == null;
    }

    public void addTerm (String toAdd) {
        root = addTerm(normalizeTerm(toAdd), root, 0);
    }

    public boolean hasTerm (String query) {
        return hasTerm(root, normalizeTerm(query), 0);
    }

    public String getSuggestedTerm (String query) {
        return getSuggestedTerm(root, normalizeTerm(query), 0);
    }

    public ArrayList<String> getSortedTerms () {
        terms = new ArrayList<>();
        getSortedTerms(root, "");
        return terms;
    }


    // ------------------------------------------------------------
    // Helper Methods
    // ------------------------------------------------------------

```

```
    private String normalizeTerm (String s) {
        // Edge case handling: empty Strings illegal
        if (s == null || s.equals("")) {
            throw new IllegalArgumentException();
        }
        return s.trim().toLowerCase();
    }

    /*
     * Returns:
     *   int less than 0 if c1 is alphabetically less than c2
     *   0 if c1 is equal to c2
     *   int greater than 0 if c1 is alphabetically greater than c2
     */
    private int compareChars (char c1, char c2) {
        return Character.toLowerCase(c1) - Character.toLowerCase(c2);
    }

    // [!] Add your own helper methods here!

    private TTNode addTerm (String toAdd, TTNode node, int index) {
        char[] lettersInWord = toAdd.toCharArray();
        //Base case
        if (node == null) {node = new TTNode(lettersInWord[index],
false);}

        //Recursion
        int position = compareChars(lettersInWord[index], node.letter);
        if (position < 0) {
            node.left = addTerm(toAdd, node.left, index);
        } else if (position > 0) {
            node.right = addTerm(toAdd, node.right, index);
        } else {
            if (index < lettersInWord.length - 1) {
                node.mid = addTerm(toAdd, node.mid, index + 1);
            } else {
                node.wordEnd = true;
            }
        }
        return node;
    }

    private boolean hasTerm(TTNode node, String query, int index) {
        char[] queryLetters = query.toCharArray();

        //Base case
```

```
94          if (node == null) {return false;}
95
96          //Recursion
97          int position = compareChars(queryLetters[index], node.letter);
98          if (position < 0) {
99              return hasTerm(node.left, query, index);
100         } else if (position > 0) {
101             return hasTerm(node.right, query, index);
102         } else {
103             if (index + 1 == queryLetters.length) {
104                 return node.wordEnd;
105             } else {
106                 return hasTerm(node.mid, query, index + 1);
107             }
108         }
109     }
110
111     private String getEnding(TTNode node) {
112         if (node == null) {return null;}
113         String ending = String.valueOf(node.letter);
114         return node.wordEnd ? ending : ending + getEnding(node.mid);
115     }
116
117     private String getSuggestedTerm(TTNode node, String query, int index)
{
118         char[] queryLetters = query.toCharArray();
119
120         //Base case
121         if (node == null) {return null;}
122
123         //Recursion
124         int position = compareChars(queryLetters[index], node.letter);
125         if (position < 0) {
126             return getSuggestedTerm(node.left, query, index);
127         } else if (position > 0) {
128             return getSuggestedTerm(node.right, query, index);
129         } else {
130             if (index + 1 == queryLetters.length) {
131                 return trimLast(query) + getEnding(node);
132             } else {
133                 return getSuggestedTerm(node.mid, query, index + 1);
134             }
135         }
136     }
137
138     private String trimLast(String toTrim) {
```

```
139         return toTrim.substring(0, toTrim.length() -1);
140     }
141 
142     private void getSortedTerms(TTNode node, String prefix) {
143         //Base case
144         if (node == null) {return;}
145 
146         //Recursion
147         getSortedTerms(node.left, prefix);
148         prefix += node.letter;
149         if (node.wordEnd) {terms.add(prefix);}
150         getSortedTerms(node.mid, prefix);
151         prefix = trimLast(prefix);
152         getSortedTerms(node.right, prefix);
153 
154     }
155 
156 
157     // -----------------------------------------------------------
158     // TTNode Internal Storage
159     // -----------------------------------------------------------
160 
161     /*
162      * Internal storage of autocompleter search terms
163      * as represented using a Ternary Tree with TTNodes
164      */
165     private class TTNode {
166 
167         boolean wordEnd;
168         char letter;
169         TTNode left, mid, right;
170 
171         TTNode (char c, boolean w) {
172             letter  = c;
173             wordEnd = w;
174             left    = null;
175             mid     = null;
176             right   = null;
177         }
178 
179     }
180 
181 }
```

Feedback:

--

Homework 5 - Description

Title	Date Posted	Date Due
Homework 5: The Sentinals	11 / 28 / 17	12 / 15 / 17

Have you ever read comments on some forum / online store and thought: "Wow, this person is really angry... why don't the moderators remove this comment?"

Conversely, you might remark, "Wow, what a kind, helpful individual; they should be rewarded for their tone!"

The problem with open forum moderation is that it is often infeasible to have a human manually examine every comment and then screen the negative ones or reward the positive.

As such, some systems employ a sentiment analyzer.

Sentiment Analyzers

Sentiment Analyzers are used to take text inputs and determine if the tone or vocabulary is positive, neutral, or negative. However, because we enjoy wordplay in this class, we will refer to these as **Sentinals**.

There are various ways to implement a Sentinal:

- **Top-Down Approach:** the Sentinal designer loads a (typically large / robust) database of sentimentally charged words, classifies them as positive or negative, and then scans input text for words that match those in the database.
- **Bottom-Up Approach:** the Sentinal designer labels a wide variety of sample texts as either positive or negative, and then lets the Sentinal learn the most commonly used positive / negative phrases to then determine if future texts are positive or negative.

In this assignment, we'll be designing a Top-Down Sentinal, wherein our Sentinal users will be able to load sentiment files containing phrases that will then be matched against analyzed text.

Choosing a Data Structure

Since the sentiment files will typically be very large, and we will need to consult them frequently to see if a given phrase exists within, we will use a simple hash table or two (which we'll call PhraseHashes) to contain the various sentiment phrases.

The overall process will look like this:

1. A Sentinal is created with a given positive phrase bank and negative phrase bank.

2. With these phrases loaded into the Sentinal's PhraseHashes, a document can then have its sentiment analyzed by comparing its phrases to those found in each PhraseHash.
3. At the conclusion of our Sentinal's analyses, we will conclude that the analyzed document had a positive, neutral, or negative tone.

So, since we will frequently consult our sentiment phrase database while analyzing a document, we need a DS with lightening-fast lookup -- luckily, we just learned about hash tables!

Note that lists and BSTs are not good fits for this task, since their insertion and lookup are slower, and we need not take advantage of their element orderings.

Assignment Goals

This assignment may look ostensibly trivial, but note that it requires you to do the following:

- Be able to decompose a large problem into many smaller problems that can be solved more easily; I don't exactly suggest a recursive solution here, but the ability to make meaningful helper methods will be tested herein.
- Lookup language mechanics that might make your task easier. This might mean Googling how to do something or consulting the manuals for various Java libraries.
- Manage test files in your workspace wherein your program will be required to interact with files outside of its source.

Note: sentiment analysis, and natural language processing in general, is an open problem in AI with many different approaches. The one we will explore herein is fairly brittle, but will give us some great practice with data structures and one of their important applications.

Specifications

In this assignment, you will implement two relatively small classes:

1. `PhraseHash`: a hash table for storing sentiment phrases
2. `Sentinal`: the sentiment analyzer that uses your PhraseHashes.

PhraseHash

Problem 1[40 points]

Implement a simple **PhraseHash** table in which we will store our sentiment phrases.

A **sentiment phrase** is a String consisting of 1 or more space-separated words, like "good" or "very good".

Your PhraseHash, for simplicty, will follow the **separate chaining** bucket schema with 1000 LinkedList buckets.

Your PhraseHash class implements the PhraseHashInterface.java interface, whose methods are detailed below:

Methods
`boolean isEmpty ();` Returns true if the PhraseHash has no phrases stored, false otherwise.
`int size ();` Returns the number of phrases currently stored in the PhraseHash.
`void put (String s);` Inserts the given phrase `s` into the PhraseHash, with duplicates ignored (i.e., do not insert a second copy of s into the PhraseHash if it is already contained within).Note: your put and get methods will require that you define a hash function that maps Strings to your bucket indexes. You should attempt to craft a function that exibits the desirable hash function traits we mentioned in class (see Lecture 14T). Lazy hash functions that will have needless collisions will be penalized.
`String get (String s);` Returns the given phrase `s` if it exists in the PhraseHash, `null` otherwise.
`int longestLength ();` Returns the length of the longest phrase in the PhraseHash, or 0 if it is empty.A phrase like "good" has a length of 1, a phrase like "very good" has a length of 2.

Sentinal

Problem 2[60 points]

Implement a simple **Sentinal (Sentiment Analyzer)** that stores sentiment phrases in PhraseHash fields and compares the text of target documents to the words in these PhraseHashes (with the goal of determining the tone of the target document).

A Sentinal loads its sentiment phrases from **sentiment files**, which contain **new-line separated** phrases that are each meant to be added to the Sentinal's PhraseHashes.

Your Sentinal class will implement the SentinalInterface.java interface, whose methods are detailed below (the order in which the methods are listed are my suggested order of completion):

Methods
`void loadSentiment (String phrase, boolean positive);` Loads a single `phrase` into the Sentinal.Stores the phrase in the positive-phrase hash if parameter positive is true, or the negative-phrase hash if the parameter positive is false.
`void loadSentimentFile (String filename, boolean positive) throws FileNotFoundException;` Loads all newline-separated phrases from the given sentiment file into the Sentinal's appropriate PhraseHash.Stores the phrases in the positive-phrase hash if parameter positive is true, or the negative-phrase hash if the parameter positive is false.
`Sentinal(String posFile, String negFile);` The only constructor for a Sentinal object that takes as input a positive-sentiment phrase file path, as well as a negative-sentiment phrase file.The constructor will then load each sentiment file into its appropriate posHash and negHash PhraseHashes.
`String sentinalyze (String filename) throws FileNotFoundException;` The main workhorse method of each Sentinal object, which takes the name of a file to analyze.The file will contain 1 or more **newline separated** sentences whose words must be scanned for sentiment.For each sentence in the file, the Sentinal will scan it to find phrase matches in both its positive-PhraseHash and negative-PhraseHash.It will then output a String based on the following criteria:"positive": if the file contained more positive phrases than negative."neutral": if the file contained an equal number of positive and negative phrases (including 0 of each)."negative": if the file contained more negative phrases than positive.

You must look up how to parse text from a .txt file in Java using the Scanner (which we covered at the beginning of the semester) and the File classes. This will be useful in your constructor, loadSentimentFile, and sentinalyze methods.

Example

posPhrases.txt

```
excellent
superb
terrific
awesome
positive
great job
well done
superior work
```

negPhrases.txt

```
bad
awful
terrible
horrific
negative
sad
poorly
negligent
not good
sloppy job
```

Example 1:

comDoc.txt

```
this assignment is awesome
if only this sentence was not bad
oh well i guess it is still positive
```

A Sentinal with the given posPhrases and negPhrases would classify the comDoc (with 3 sentences) as having a "positive" tone, decomposed as follows:

1. The first sentence contains 1 positive phrase, "awesome" [+1]
2. The second sentence contains 1 negative phrase, "bad" [-1]
3. The third sentence contains 1 positive phrase, "positive" [+1]

In total, there are 2 positive sentiments and 1 negative, therefore the document is considered "positive".

Example 2:

comDoc2.txt

```
andrew is not good
he is excellent
his sloppy job is actually a great job
```

Notice that our Sentinals are actually pretty brittle because they fail to pick up on some nuances of language.

In the above, the writer used dramatic impact by first saying "andrew is not good" as a setup to the next sentence: "he is excellent".

However, since we have a naive scoring system, the "not good" from the first sentence cancels the "excellent" from the second.

A Sentinal with the given posPhrases and negPhrases would classify the comDoc2 (with 3 sentences) as having a "neutral" tone, decomposed as follows:

1. The first sentence contains 1 negative phrase, "not good" [-1]
2. The second sentence contains 1 positive phrase, "excellent" [+1]
3. The third sentence contains 1 negative phrase, "sloppy job" [-1] and 1 positive phrase, "great job" [+1]

In total, there are 2 positive sentiments and 2 negative, therefore the document is considered "neutral".

Example 3:

comDoc3.txt

```
my code is sloppy
i hope andrew grades kindly
i will be quite sad
```

I know what you're thinking, and yes, that's a haiku.

A Sentinal with the given posPhrases and negPhrases would classify the comDoc3 (with 3 sentences) as having a "negative" tone, decomposed as follows:

1. The first sentence contains 0 sentimental phrases (even though sloppy is a word within "sloppy job"), [0]
2. The second sentence contains 0 sentimental phrases, [0]
3. The third sentence contains 1 negative phrase, "sad" [-1]

In total, there is 1 negative phrase, therefore the document is considered "negative".

Assumptions

To simplify this assignment, we'll assume the following:

- You may assume there will be no punctuation, spaces, capital letters, or numbers in any of the arguments to any of the above methods, nor in any input files.
- You need make no assumptions about the order in which phrases are added to each PhraseHash so long as the above requirements are met.
- You may assume that, for entered phrases into a PhraseHash, no phrase of size m is a subset of a phrase of size k for m <= k. In other words, you would not find both "great" and "great job" in a positive word dictionary, because "great" is a subset of "great job".
- You may assume that sentences do not continue onto new lines when reading from a file in `sentinalyze`. In other words, consider only phrases of length greater than one that are on the same line.
- You may assume that all given filenames represent **absolute paths** to the files on your file system. An absolute path is one that is specified from your root drive to the specific file such

as "S:/Users/aforn/workspace/cmsi-281/src/sentinal/". See SentinalTests.java for more info.
- All methods that parse files will `throws FileNotFoundException` when the provided file is not found on the system or at the path specified. You will not need to throw this exception manually since you will use the Scanner class to parse your files.
- Due to time constraints, you are not expected to do anything fancy for your phrase matching during sentinalysis, but at least challenge yourself to not scan for phrases of length 4 if, say, the largest phrase in the sentiment-specific PhraseHash that you are parsing has a longestLength of 3 (or 2 or 1).

Unit Tests

You may use the following sample unit tests to verify your understanding of the specifications above. Note: these are not an exclusive list of tests that I will use to grade your assignment, so to ensure as many points as possible, you should add many tests to this list (including those required above).

Unlike in HW1, you are not required to *submit* any test case modifications, but if you choose not to make any more entirely, then ask yourself: do you have any other masochistic tendencies?

Please download the PhraseHashTests.java and SentinalTests.java from the solution skeleton below.

Solution Skeleton

The following .zip file contains a solution skeleton that you may use for your submission's starting point. It is highly recommended that you download this as a scaffold and work from there.

Sentinal Solution Skeleton

Solution Restrictions

Read the following list of submission restrictions carefully! **Violating any restriction will net you a 0 on this homework!**

- You may ONLY use the LinkedList data structure from the Java collections framework in your solution. You may not use *any* other data structure that you did not create yourself! When in doubt, ask.
- You MAY use some convenient methods from other Java classes, like the String and Arrays classes.
- You may NOT add any methods or fields to your class' public interface. You may, however, add

any private fields or methods that you like.
- Your classes and therefore source files must be named exactly as intimated above (as is in the Solution Skeleton), and your submission should mimic the solution skeleton's package structure.

Hints

The implementation of this assignment requires you to make some design decisions. However, here are some hints for how you might structure your own.

- In analyzing files with your Sentinals, I suggest breaking down its "positive / negative phrase count" into several hierarchical steps: parse sentence by sentence in the file (i.e., line by line), then within each sentence, count each pos / neg phrase of length 1, then 2, etc. until you have a sum of the number of phrases found.
- If one step of the above feels difficult, break it down! Make a helper method that will allow you to decompose the total file scoring into smaller bits.

Submission

Submit your assignment using Brightspace given the requirements below:

1. Find the assignment's listing on Brightspace.
2. Click the "Attach file" dialogue and add your Sentinal.java and PhraseHash.java files.
3. Click "Submit" at the bottom right hand corner of the screen.