TCSS 342 - Data Structures
Assignment 2 - Unique Words

Time Estimate: 8-12 hours

Learning Outcomes

The purpose of this assignment is to:

- Build a specialized ordered list data structure out of an array list.
- Perform a binary search.
- Sort a linked list.
- Sort an array list.
- Evaluate the performance of our linked list, array list and ordered list at a specific task.
- Read a text file and extract the unique words.

Overview

In the last assignment we built two data structures: an array list and a linked list. In this assignment we will put our data structures to the test by comparing their performance at a specific real world task. In addition, we will create a third data structure that might be well suited for this task: an ordered list.

The Task: I have a text file version of Leo Tolstoy's *War and Peace*. I would like to extract all the unique words in the file (and sort them) as the first step to counting the number of occurrences of each word in the book (we'll do this in a future assignment). We will do this by following this simple pseudocode:

```
For each word in the text:
   If the word is not in the list, add it to the list.
```

This code uses a search to find if the word is in the list, and the add method to add new words to the list.

In order to perform this task we need to know what counts as a *word* for the purposes of the task. So, for the purposes of this assignment we will define a *word* as **any string of consecutive alphanumeric characters or apostrophes**. Specifically, this means a *word* is any string of characters from this set:

Numerals	0,1,2,3,4,5,6,7,8,9
Lowercase Alphabet	a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z
Uppercase Alphabet	A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z

Apostrophe	‘

We will treat all other characters as *separators*. A *word* always has a *separator* before and after it in the text (except possibly the first and last word). Here are some examples using a line of text from *War and Peace* to make sure we understand this idea.

> It was in July, 1805, and the speaker was the well-known Anna Pavlovna Scherer, maid of honor and favorite of the Empress Marya Fedorovna.

Most *words* in this text are preceded and followed by a space. In this text we can see the *words* “July” and “1805”. Both of these *words* are preceded by a space and followed by a comma. The *words* “well” and “known” are separated by a hyphen. In these examples the spaces, commas, and hyphens are the *separators*.

To complete this you will upgrade two classes, implement three public classes:

- MyLinkedList<Type> (with a private Node)
- MyArrayList<Type>
- MyOrderedList<Type>
- BookReader
- UniqueWords

Formal Specifications

MyArrayList<Type extends Comparable<Type>>
… +comparisons : long
… +contains(item : Type) : boolean +indexOf(item : Type) : int +sort()

Field summary

- `comparisons` - Stores the total number of comparisons made by the `contains` method.

Method summary

- `contains` - This function should be upgraded to use Comparable.compareTo(Type) to compare elements.

- `indexOf` - This function should be upgraded to use Comparable.compareTo(Type) to compare elements.
- `sort` - Sorts the list in ascending order. You may select any sorting procedure you like from this list:
    a. Bubble Sort - This method should run in $O(n^2)$ time.
    b. Insertion Sort - This method should run in $O(n^2)$ time.
    c. Selection Sort - This method should run in $O(n^2)$ time.
    d. Merge Sort - This method should run in O(nlogn) time.
    e. Quick Sort - This method should run in O(nlogn) time.

MyLinkedList<Type extends Comparable<Type>>
… +comparisons : long
… +contains(item : Type) : boolean +sort()

Field summary

- `comparisons` - Stores the total number of comparisons made by the `contains` method.

Method summary

- `contains` - This function should be upgraded to use Comparable.compareTo(Type) to compare elements.
- `sort` - Sorts the list in ascending order. You may select any sorting procedure you like from this list:
    a. Bubble Sort - This method should run in $O(n^2)$ time.
    b. Insertion Sort - This method should run in $O(n^2)$ time.
    c. Selection Sort - This method should run in $O(n^2)$ time.
    d. Merge Sort - This method should run in O(nlogn) time.
    e. Quick Sort - This method should run in O(nlogn) time.

MyOrderedList<Type extends Comparable<Type>>
-list : MyArrayList<Type> +comparisons : long
+add(item : Type) +remove(item : Type) : Type +binarySearch(item : Type) : boolean

```
-binarySearch(item : Type, int start, int finish) : boolean
+size() : int
+isEmpty() : boolean
+toString() : String
```

Field summary

- `list` - The list of items.
- `comparisons` - Stores the total number of comparisons made by the `binarySearch` method and the `add` method.

Method summary

- `add` - Adds the `item` to the position of the list where it belongs. This method should run in O(n) time in the worst case.
- `remove` - Removes the `item` from the list if found. This method should run in O(n) time.
- `+binarySearch` - Uses a binary search to search the list for `item` and returns true if found, and false, otherwise. You can implement this method with a loop or recursion. This method should run in O(logn) time.
- `-binarySearch` - A private recursive method to be called by the public binarySearch. This method is optional and should only be used if you implement binarySearch in a recursive way.
- `size` - Returns the size of the list. This method should run in O(1) time.
- `isEmpty` - Returns true if the size is 0 and false otherwise. This method should run in O(1) time.
- `toString` - Returns a string that has the contents of the `list` separated by commas and spaces and enclosed in square brackets. This method should run in O(n) time.
    a. Example: [1, 2, 3, 4]

BookReader
+book : String +words : MyLinkedList<String>
+BookReader(filename : String) +readBook(filename : String) +parseWords()

Field summary

- `book` - The book as a String.

- `words` - A list of all the words in the book.

Method summary

- `BookReader` - Calls readBook on filename and then calls parseWords.
- `readBook` - Reads the contents of the file into the book.
    a. Opens the file `filename`.
    b. Reads the file character by character.
    c. Closes the file.
    d. Stores the contents of the file in `book`.

  This method should time itself and output its runtime to the console. This method should run in time O(n) where n is the number of characters in the file.
- `parseWords` - Scans the `book` for words using the specifications above for what counts as a word. When a word is found it is stored in `words`. This method should run in O(n) time where n is the length of `book`.

UniqueWords
-book : BookReader
+UniqueWords() +addUniqueWordsToLinkedList() +addUniqueWordsToArrayList() +addUniqueWordsToOrderedList()

Field summary

- `book` - The BookReader used to read the words out of the book. You should initialize this with “WarAndPeace.txt”. I also have a smaller text file I used for testing.

Method summary

- `UniqueWords` - Calls each of the `addUniqueWords` methods one at a time.
- `addUniqueWordsToLinkedList` - Adds the unique words to a MyLinkedList.
    a. For each word in the list stored in book:
        - Check to see if the word is stored in the linked list of unique words.
            - If it is not, add it to the list.
            - Otherwise, skip this word.
    b. Sort the list of unique words.

  This method should time both steps and output each runtime to the console. This method should run in time $O(n^2)$ where n is the number of words in the book.
- `addUniqueWordsToArrayList` - Adds the unique words to a MyArrayList.
    a. For each word in the list stored in book:

        - Check to see if the word is stored in the array list of unique words.
            - If it is not, add it to the list.
            - Otherwise, skip this word.
    b. Sort the list of unique words.

    This method should time both steps and output each runtime to the console. This method should run in time $O(n^2)$ where n is the number of words in the book.

- `addUniqueWordsToOrderedList` - Adds the unique words to a MyOrderedList.
    a. For each word in the list stored in book:
        - Check to see if the word is stored in the ordered list of unique words using binary search.
            - If it is not, add it to the list.
            - Otherwise, skip this word.

    This method should time itself and output its runtime to the console. This method should run in time $O(n^2)$ where n is the number of words in the book.

Testing

It is important that you test your code to ensure it works on all potential inputs. Please do this in a separate Main class, and without using additional tools (like JUnit). You will not submit your test code. Instead, your data structures will be tested by code developed by the instructor and/or grader. This code will not be provided to you, as you should test your code before submitting it. If your code fails our tests we will let you know which tests it fails so you can find and correct your errors.

Here is the output from my solution:

```
Reading input file "./WarAndPeace.txt"... 3291642 characters in 54 milliseconds.

Finding words and adding them to a linked list...  in 337 milliseconds.
The linked list has a length of 570240.

Adding unique words to an array list...  in 5 seconds.
20228 unique words
1065636950 comparisons
Insertion sorting array list...  in 1 seconds.

Adding unique words to a linked list...  in 5 seconds.
20228 unique words
1065636950 comparisons
Bubble sorting linked list...  in 3 seconds.

Adding unique words to an ordered list...  in 1 seconds.
20228 unique words
114938017  comparisons
```

Submission

You will submit a .zip file containing:

- MyLinkedList.java
- MyArrayList.java
- MyOrderedList.java
- BookReader.java
- UniqueWords.java

Grading Rubric

In order to count as complete your submission must:

1. Implement both the changes to linked list and array list efficiently and without errors.
2. Implement the ordered list efficiently and without errors.
3. Follow the API exactly so that your classes will work with any testing code.
4. Get the numbers right:
    a. Read all 3291642 characters in the file.
    b. Find all 570240 words in the book.
    c. Find the 20228 unique words in the book.
    d. Make 1065636950 comparisons to find the unique words in the linked list and array list.
    e. Make 114938017 comparisons to find the unique words in the ordered list.

Reminder: Incomplete assignments can always be corrected and resubmitted. If they are completed within 7 days of the due date they will count as late and after that period they will count as missed. Please review the grading matrix for the number of permitted late and missed assignments.