Homework 6

Due: 4:00pm on December 10, 2024 (Tuesday)

Objectives:

- Implement a binary search tree from scratch and use it in an application
- Try to implement Comparable and Comparator interfaces and use them in your application.

In homework 4, you implemented `MyHashTable` class that could help you search for a word very quickly. The `MyHashTable` class also has a few other capabilities and one of them is keeping track of the frequency of each word.

That was pretty useful. But, suppose that you were not totally satisfied with it. Now, you want to have a lot more useful information and features such as printing values in any order that you can specify.

While reading a book, one of the things you would do is using the index at the end of the book in which you can find pages where a word or a phrase is used in the book.

Your major goal, this time, is to make your own program that builds an index of words from a file that you are to parse.

Wouldn't you want to see not only how many words are in a file but also how often each word appears and where they are in terms of line numbers of the file?

Also, it would be great to sort them by alphabetically as well as by their frequencies.

You are to write a program that parses words from a text file and produces a very useful index stored in a binary search tree that you ought to implement for this assignment.

There are a few classes that you need to write for the program.

1. Word class

```
public class Word implements Comparable<Word>
```

This class has three private fields:

- String word
- Set<Integer> index
- int frequency

For word and frequency variables, there should be getters and setters as follows.

```
public void setWord(String newWord)
public String getWord()
public void setFrequency(int freq)
public int getFrequency()
```

Make sure to implement a constructor with a String parameter.

For index variable, there should be a method called, addToIndex, that has an integer parameter (use Integer type parameter, not int type, to make it consistent with the Generics data type of Set) that indicates a new line number for the word that should be added to the index. The getter (getIndex) returns the Set<Integer> data structure that contains all of the line numbers for the word.

```
public void addToIndex(Integer line)
public Set<Integer> getIndex()
```

Also, you are to override toString and compareTo methods in this class. For the format of the toString method, please refer to the expected output of HW6Driver.java file. The natural order of this class is defined by alphabetical order of the `word` field (String).

The index variable is there to store line numbers of each word.

A word is a sequence of letters [a..zA..Z] that does not include digits [0..9] and does not include other characters than the sequence of letters [a..zA..Z]. Here are examples of non-words: abc123, a12bc, 1234, ab_c, abc_

The process to validate words should happen in the Index class, not in the Word class.

2. BST class

```
public class BST<T extends Comparable<T>> implements Iterable<T>,
BSTInterface<T>
```

This class has only two variables. No other variables are allowed:

- Comparator<T> comparator
- Node<T> root

There is also the private static nested class Node:

```
private static class Node<T>
```

Note that you may store Word objects in your program. However, you need to make your BST class very general so that it can handle any type of objects. You should NOT have any reference to Word class in your BST class.

You need to implement a few methods. But, the most important two methods are insert and search.

```
public void insert(T toInsert)
```

This method accepts any comparable object toInsert and ***recursively*** inserts the object into the BST based on either natural ordering (provided by the Comparable interface) or the specified comparator, passed into the tree through a constructor.

Note: In this method, if there is already the existing object, such as a word (String value), in the BST, keep the old one. Do not update or combine.

```
public T search(T toSearch)
```

This method ***recursively*** searches the BST. Just like insertion, this search method should be able to compare objects in the BST either using compareTo() or compare().

Hint: For this reason, the BST class must have comparator as its variable.

```
Iterator()
```

There should be Iterator implementation. You need to implement in-order iterator that traverses the elements of the BST in an ordered sequence. But, it has to be done iteratively, NOT recursively. Keep in mind that the first element that your iterator should output is the leftmost child of the root. How can we do this? Make sure to implement hasNext() and next() methods in your own iterator class.

In this class, you are not allowed to use the Java Collections Framework except for your Iterator implementation where you may need. In other words, you can use classes in the Java Collections Framework only to implement the iterator. Iterator and Comparator in the util package have been already imported in the file given to you as they are needed and used for sure.

3. Index class

```
public class Index
```

This class is responsible for building an index tree in three different ways using the following three different methods

```
public BST<Word> buildIndex(String fileName)
```

that pares an input text file (using the same order as they appear in a file) and build an index tree using a natural alphabetical order.

```
public BST<Word> buildIndex(String fileName, Comparator<Word> comparator)
```

that parses an input text file (using the same order as they appear in a file) and build an index tree using a specific ordering among words provided by a specific comparator.

For the two `buildIndex` methods above, there is already the existing object, such as a word (String value), in the BST, keep the old one and increase its frequency by one and update its line number information.

Split text using any non-word character in the same way you used in HW #5.

```
public BST<Word> buildIndex(ArrayList<Word> list, Comparator<Word>
comparator)
```

that takes Word objects in a list from the first to the end and allows to rebuild an index tree using a different ordering specified by a comparator.

For example, you call the second method using one comparator and realized that you want to organize words with the same frequency to be sorted by alphabetically.

For this `buildIndex` method that takes ArrayList of Word objects and a Comparator, please keep in mind that, if there is already the existing object in the BST, simply keep the existing one. Do not update or combine.

Example usages of these methods are provided in the HW6Driver.

The process to validate words should happen in the Index class, not in the Word class.

Invoking the following methods in the Index class should not modify the tree structure of the argument, BST.

```
public ArrayList<Word> sortByAlpha(BST<Word> tree)

public ArrayList<Word> sortByFrequency(BST<Word> tree)

public ArrayList<Word> getHighestFrequency(BST<Word> tree)
```

In this class, you may use the Java Collections Framework.

4. Comparator classes

`public class IgnoreCase implements Comparator<Word>`

that sorts words by case insensitive alphabetical order. Also, if this comparator is passed into buildIndex method, then all the words need to be converted into lowercase and then added into the BST. The conversion step should be done in the Index class.

`public class Frequency implements Comparator<Word>`

that sorts words according to their frequencies (a word with highest frequency comes first). Do not get confused! This is one of the general comparator classes that maybe used in some other applications too. This comparator will NEVER be used to build the BST.

`public class AlphaFreq implements Comparator<Word>`

that sorts words according to alphabets first and if there is a tie, then words are sorted by their frequencies in ascending order (a word with lowest frequency comes first).

The examples used in the lecture note 9 will be helpful for your implementation of the classes above.

To save your time, starter code files (all the class files along with the `HW6Driver.java` file) have been provided. Use them for your implementation and don't forget to write your andrew id and name and proper JavaDoc comments in all the classes you are to implement!

Deliverables:

- Your source code files. Place `Word.java, BST.java, Index.java, IgnoreCase.java, Frequency.java, and AlphaFreq.java` files into a temporary folder and use the following command to zip them. Do NOT include `BSTInterface.java` and `HW6Driver.java` files in your zip file. Use WinZip, 7-Zip, or Compress option (with right-clicking on the java files). The following ZIP command can be used to do this:

```
zip homework6.zip *.java
```

- Using single-line or multi-line comments, make sure to comment about base case(s) and recursive cases in all recursive methods.
- Submit your homework6.zip file using Autolab (https://autolab.andrew.cmu.edu).
- Once again, the HW6Driver is just a starting point for your testing. Test extensively!

Grading:

We will be STRICTER in grading since this is your last homework.

Autolab will grade your assignment as follows.

- Working code: 90 points
  - File exists, File compiles, and Author Comment: 5 points
  - Word.java: 5 points
  - BST.java: 40 points
  - Index.java: 35 points
  - Comparators: 5 points
- Coding style (refer to the guideline): 10 points

In case you do not pass test case(s), please spend some time to think about edge cases you may have missed and test thoroughly before asking questions to the TAs and resubmitting. Please, make sure to read the spec more carefully. Also, read previously asked questions and answers on Piazza before posting duplicate questions.

Autolab will show you the results of its grading within approximately a few minutes of your submission. You may submit multiple times to correct any problems with your assignment. Autolab uses the last submission as your grade.

The Autolab score is not final. The TAs will look into your source code to check correctness and design and deduct points accordingly. The most important criterion is always correctness. Buggy code is useless (even if you may think a found bug is very minor). It is also critical that your code is efficient and follows the specifications properly. Additionally, it should be readable and well organized. This includes proper use of clear comments. Points will be deducted for poor design decisions and unreadable code, etc.

As mentioned in the syllabus, we will be using the Moss system to detect software plagiarism. Make sure to read the cheating policy and penalty in the syllabus. *Any cheating incident will be considered very seriously. The University also places a record of the incident in the student's permanent record.*

There will be NO EXTESIONS, and late submissions will NOT be accepted and, if you have multiple versions of your code file, make sure you do submit the correct version.