Assignment 1: Applications of the Trie Data Structure (15%)

The aim of this assignment is to use the Trie data structure in two practical contexts.

In the first context, **Task 1 (40%)**, you will make use of a standard Trie that allows us to look up words in a dictionary and use them for a predictive text function. In the second context, **Task 2 (60%)**, you will extend the Trie to be a Suffix Trie, and apply the data structure to search for particular strings in a text (i.e. exact string matching).

The skeleton code provided for this assignment includes the three Java files provided for Lab 2 (and a number of other files). Please note that you will need to have completed Checkpoints 2.4 and 2.5 of Lab 2 in order to have a foundation to work from in completing this Assignment. For those checkpoints, you are asked to implement the basic functions of:

- inserting data into a Trie,
- searching for a string, and
- searching for a prefix of a string.

This Assignment assumes that these functions have already been implemented.

The table below gives an indication as to the breakdown of the assessable components.

Assignment 1		**%**	**15**
Task 1	100	40	6
Create Dictionary	20	8	1.2
Most Frequent Word	60	24	3.6
TextAreaDemo	10	4	0.6
Code quality	10	4	0.6
Task 2	100	60	9
read/insert	40	24	3.6
get/getNode	10	6	0.9
Code quality	10	6	0.9
Testing	20	12	1.8
Extension	20	12	1.8

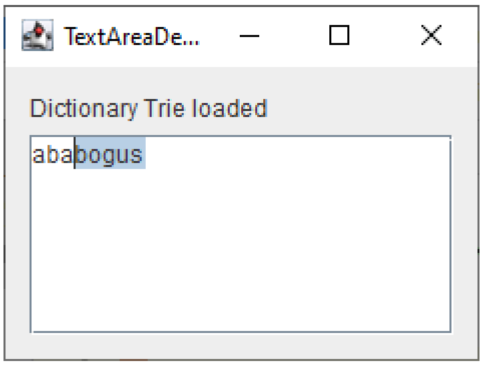
Note that 10% of each task is dedicated to code quality/style and documentation of the coding choice you have made (e.g. why a TreeMap over a HashMap?). The documentation should be in a separate document (PDF) and is submitted via FLO (Canvas).

Task 1: Predictive Text (40%)

For this task, you will make use of the classes `Trie`, `TrieNode`, `TrieDriver`, `TrieData` and `TextAreaDemo` (the last file is taken directly from the Oracle Java Trail for Java and modified for our purposes). Skeleton code has been provided for the first two classes and you will already have modified them in order to create a working Trie data structure in Lab 2.

You may optionally modify `TrieData` if you wish, but you should **not** modify `TextAreaDemo`.

`TextAreaDemo` is an extremely minimal "text editor" (it does not even allow you to save your document, or load a document from file), but it does at least provide assistance with typing, in the form of **text prediction**! As you type into the provided text area, the program will offer suggestions for words that begin with the characters that you are currently typing. Suggestions are shown by displaying the rest of the word (highlighted in blue) directly after the word prefix that you have already typed (this works with prefixes of 2 or more characters). To accept an automatic suggestion, you simply need to press Enter and the complete word will be entered and the cursor will move ahead so you can type the next word.



Dictionary Trie loaded

ababogus

You can test it out – note that at the moment, the only words that the application suggests are made up of whatever prefix the user has typed in with the word "bogus" appended to it. Examine the default implementation of the `getMostFrequentWordWithPrefix`() method in the `Trie.java` file and make sure you understand why this is happening. Your task is to replace these "bogus" suggestions with real suggestions from the dictionary.

To complete this part of the Assignment, you will need to

1. read in words from a dictionary
2. insert them into the Trie, and
3. correctly implement `getMostFrequentWordWithPrefix`

Creating the Dictionary

Complete the implementation of the static method `readInDictionary()` in Trie. The `test4` method in `TrieDriver` produces (`<-- user input` indicates input from the user):

```
4 data/word-freq.expanded.trim.txt anomaly aba hoodie <--
user input
Reading in trie...done

testing getNode
anomaly: TrieNode; isTerminal=true, data=33, #children=0
aba: TrieNode; isTerminal=false, data=null, #children=3
hoodie: null

testing get
anomaly: TrieNode; isTerminal=true, data=33, #children=0
aba: null
hoodie: null
```

You have been provided with three files for testing.

- `data/word-freq.expanded.txt` is the full expanded version of word frequencies (over 40,000 entries).
- `data/word-freq.expanded.trim.txt` is similar to `word-freq.expanded.txt` except that all the apostrophes have been removed (should be only standard lowercase a-z characters).
- `data/word-freq.grow.txt` is the "grow" example from the lecture slides for testing.

Searching for the most frequent word

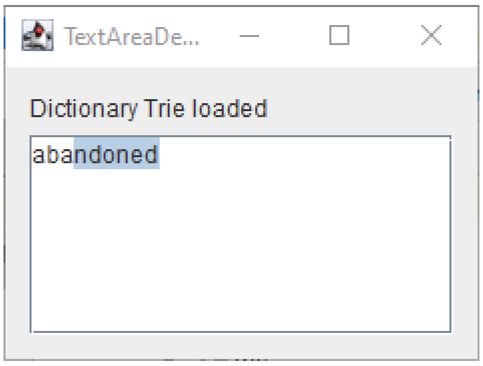
Complete the implementation of `getMostFrequentWordWithPrefix`(). This method should find the node in the trie that corresponds to the prefix supplied as its argument, and examine all candidate words starting with that prefix in order to find the one with the highest frequency of usage. `test5` produces (`<-- user input` indicates input from the user):

```
5 data/word-freq.expanded.trim.txt aba the bbb <-- user
input
Reading in trie...done
PREFIX = aba
Most Frequent Word is abandoned
PREFIX = the
Most Frequent Word is the
PREFIX = bbb
Most Frequent Word is bbb
```

NOTE: the most efficient solution is NOT finding all the prefix words and THEN finding the frequencies of those words. This will work, but will require much many more traversals of the trie than is required. It is more efficient to determine the most frequent word as you traverse the trie finding all the prefix words.

Testing `TextAreaDemo`

If you have completed the above two steps successfully, then text prediction in `TextAreaDemo` should simply work! Examine the `insertUpdate()` method of `TextAreaDemo`. You will see that the code calls the `getMostFrequentWordWithPrefix()` method for the prefix that the user has just typed in. In other words, the predictive text function uses the trie to look up the prefix that the user has typed, and the most frequent word starting with that prefix is returned as the automatic suggestion for word completion.



Dictionary Trie loaded

abandoned

Task 2: Exact String Matching (60%)

For this task, you will create a `SuffixTrie`. You are provided with the classes `SuffixTrie`, `SuffixTrieNode`, `SuffixTrieData`, `SuffixIndex` .
The `SuffixTrie` has the following methods:

- `insert()` – insert a string into the suffix trie, and associate the string with a data object. This is where the suffix'ing happens.
- `get()` – search for a particular (sub)string, and return the associated `SuffixTrieNode`

Note that the method signatures (return type and parameters) have changed – the `SuffixTrie` stores data (detailing where the substring is located) currently in the form of class `SuffixIndex`, rather than the more general `TrieData` class used in Task 1.In addition, the `SuffixTrie` features the method

- `readInFromFile()` – this loads the suffix trie with data from a supplied text file, in order to allow searching for a substring in the text (using get()).

Coding Steps

For this task, you need to:

1. implement the above three methods, and
2. test them by writing a `main(String[] args)` method that loads a text file into a suffix trie, and then uses the suffix trie to search for arbitrary substrings that you supply yourself.

Be sure to test your code not only with whole words such as `hideous`, but also with part words such as `onster`, and substrings that span multiple words and may feature punctuation, such as `, and the`. You have been provided with several example text files,

- `data/Frank01.txt`
- `data/Frank02.txt`
- `data/FrankChap02.txt`
- `data/FrankChap04.txt`
- `data/FrankMed.txt`
- `data/Frankenstein.txt`

These are modified versions of the text of the novel *Frankenstein* by Mary Wollstonecraft Shelley, taken from the Project Gutenberg site: link.

The first file is much shorter (the first paragraph) than the full `Frankenstein.txt` file, and the second file, `Frank02.txt` consists of the first two paragraphs. Both should be used during development. Use the `Frankenstein.txt` and the other files only once you are confident your code works – and be prepared to wait several minutes for the suffix trie to be loaded. In any event, your code should be able to construct a suffix trie from any text file.

You are also provided with another file for testing that contains only a single word:

- `data/mississippi.txt`

General Guidelines

For the most part, you can copy large sections of code from `Trie` and `TrieNode` to `SuffixTrie` and `SuffixTrieNode`. These implementation details are also left up to you. But below are some ideas.

You may assume that substring queries will never span across more than one sentence. In other words, you do NOT need to insert the suffixes of the entire text, but can first break the text up into sentences and then insert each individual sentence into the suffix trie.

Breaking the text up into sentences requires some thought. It is acceptable to simply split the text whenever you encounter a full-stop '.' , exclamation point '!' or question mark '?' . This leads to a number of cases that are incorrectly handled:

- there are errors of omission, for instance the first 57 or so lines of the target text would be incorrectly treated as a sentence, and
- errors of commission, for instance in the sentence *"This is a problem, i.e. it doesn't work very well."*, this approach would report that there are three sentences instead of one.

You are not required for the purpose of this Assignment to handle search patterns containing any of these three characters, so this approach (using '.!?' as sentence breaks) is acceptable for our purposes. Double-quote symbols " and " have already been removed from the supplied text files, to simplify the task.

As mentioned above, you need to store information about where each substring is located in the text. One suggested way that you might do this is by using the `SuffixIndex` class that stores a variable `sentence` that represents the index of the sentence and `character` the character index in that sentence. Also remember that a substring may appear in more than one position in the text, and so your solution should be able to store multiple positions. Your implementation should return location data that could, in principle, be used to locate the substring in the original text by traversing the text to the appropriate sentence and then to the appropriate character index.

It is *not* necessary to implement your solution as a compact suffix trie (i.e. it does not need to be a suffix tree) – **an uncompressed suffix trie is sufficient**.

Be aware that, when dealing with general text as opposed to dictionary words, some assumptions that made the implementation of the dictionary trie easier or more efficient may no longer be valid. For instance, in a dictionary application, you may be able to achieve fast access by assuming that all words are made up of alphabetic characters. When indexing into text, however, you have to be able to deal with queries involving spaces, punctuation, etc. In other words, the user might want to search for the string `, but I` as a legitimate query. Also keep in mind that all queries should be case-insensitive – this is purely to conserve space.

Please also keep in mind that a Suffix Trie can become very large, very quickly and you may run out of RAM! I know, cool right!

If this happens, try using a smaller file (e.g. the first two sentences of `Frank01.txt`). If it still happens try to modify your code so it uses a smaller amount of space. You can also add a command line argument –XmxSIZE to increase the memory allocated to the JVM, for example, adding –Xmx8G to increase to 8GB (but you might need more like 16GB for the full Frankenstein file!).

For changes using the intelliJ IDE see here:

- https://www.jetbrains.com/help/idea/increasing-memory-heap.html
- https://www.jetbrains.com/help/idea/tuning-the-ide.html#configure-jvm-options

Testing

You should note that 20% of the mark for Task 2 is for testing your implementation. You should document your **testing regime** and provide examples of input and output and any challenges that arose.

You should provide evidence that your program works. Use a simple input file (such as `mississippi.txt`) and show that it can find correct substrings and locations. The below is an example of running `SuffixTrieDriver` using the file `mississippi.txt`:

```
data/mississippi.txt          <-- user input
i                             <-- user input
is                            <-- user input
si                            <-- user input
                              <-- user input
[i]: [0.1, 0.4, 0.7, 0.10]
[is]: [0.1, 0.4]
[si]: [0.3, 0.6]
```

This indicates, for example, that `i` occurs in sentence 0 (0 based indexing) at character locations of 1, 4, 7, and 10. The format of `0.1` is the result of `toString()` calls on `SuffixTrieData` and `SuffixIndex`.

You should give an indication of the performance of your implementation. For example you could time how long it takes to build a suffix trie for `Frank01.txt` versus `Frank02.txt` and other variations in length of the text files. How much RAM does it consume? Below is an example of running `SuffixTrieDriver`:

```
data/Frank02.txt          <-- user input
and                       <-- user input
, the                     <-- user input
onster                    <-- user input
monst                     <-- user input
                          <-- user input
[and]: [1.27, 1.87, 3.34, 3.153, 4.15, 6.70, 7.66, 7.148,
8.88, 9.96, 9.120, 9.173, 9.199, 10.17, 12.67, 12.91, 13.97,
13.114, 14.27, 14.90]
[, the]: [8.16]
[onster]: null
[monst]: null
```

Here we see that `and` occurs in multiple locations, while `the` only occurs in one position, that is, sentence 8 and character position 16. The suffixes `monst` and `onster` do not exist in the file `Frank02.txt`, but in `FrankChap04.txt` the prefix `monst` does exist at `[482.60]`, but not `onster`.

Going for an HD...possible Extension Ideas

Part of being a High Distinction student is going beyond the material and demonstrating advanced knowledge. 20% of the mark for Task 2 (12% overall for Assignment 1) is given for extending Task 2 in an interesting way. Below are a list of ideas to extend this task further:

- Detailed comparison of different internal data structure for representing child nodes
- Detailed comparison with online text search (e.g. Boyer-Moore)
- Fancier output display of the result of a search (e.g. GUI)
- Concordance (give context of found words)
- Wildcard search using suffix tries
- Suffix **Tree** construction (so you can read in `Frankenstein.txt`)
- Or another extension discussed with topic coordinator

Submission of Assignment

Submission will be via FLO. You should submit the final version of all **source code files** for the assignment to FLO and you should also submit a **PDF describing your implementation and testing**. If there are any special requirements for running the program these should be detailed somewhere, preferably in a separate document about running your program. Additional documents should be in PDF format.

If you have utilised additional data files for your assignment (simple testing files, really large files) and they are too large to upload on the repository then provide a link in your documentation. Files up to a few MB in size should be okay.