TCSS 342 - Data Structures
Extra Assignment 3 - Trie

Version 1.1.3 (April 14, 2022)

Time Estimate: 6-8 hours

Learning Outcomes

The purpose of this assignment is to:

- Build a trie data structure.
- Apply the trie data structure to a problem well suited to it.
- Gain experience testing and debugging data structures.
- Build data structures exactly to an API's specifications.
- Read a text file and extract the unique words.

Overview

A trie data structure is a special purpose data structure for storing data, in particular, strings of characters. A trie is both a compact and efficient way to store strings for re**trie**val (this is where the trie gets its name). The path to a string stored in a trie is a path through nodes representing the prefixes of the string. For this reason the trie is also called a **prefix tree**.

For example, if the word "there" is stored in a trie, the path of nodes to arrive at "there" is:

"" > "t" > "th" > "the" > "ther" > "there"

The node with the empty string, "", is the root of the trie.

This data structure would be well suited to the task on Assignment 2 - Unique Words. So for this extra assignment we will upgrade the UniqueWords class to test the trie.

To complete this you will upgrade two classes and implement one public class and one private class:

- MyTrie<Type> (with a private Node)
- MyOrderedList<Type>
- UniqueWords

Formal Specifications

MyOrderedList<Type extends Comparable<Type>>
…

```
…
+binarySearch(item : Type) : Type
-binarySearch(item : Type, int start, int finish) : Type
+get(index : int) : Type
```

Method summary

- `binarySearch` - Upgrade this method to return the item searched for. This is common when that item contains other data that we want. Our trie will need this functionality to find nodes based on the character stored in them.
- `get` - Returns the element stored at `index` and `null` if the `index` is out of bounds. This method should run in O(1) time.

MyTrie

```
-root : Node
-size : int
+comparisons : long
```

```
+insert(item : String)
+remove(item  : String)
+find(item : String) : boolean
+size() : int
+isEmpty() : boolean
+toString() : String
-addWords(node : Node, str : String, output :
StringBuilder)
```

Field summary

- `root` - Stores the root node of the trie.
- `size` - Stores the number of strings added to the trie.
- `comparisons` - Stores the number of comparisons made in the insert and find methods.

Method summary

- `insert` - Inserts the `item` into the trie. This method should run in O(m) time where m is the length of `item`.
- `remove` - Remvoes the `item` from the trie. This method should run in O(m) time where m is the length of `item`.

Note: You do not need to remove any nodes for this method. Knowing which nodes to remove is non-trivial. However, as an additional challenge you can remove nodes that only lead to non-terminal nodes.

- `find` - Returns true if the `item` is in the trie and false otherwise. This method should run in O(m) time where m is the length of `item`.
- `size` - Returns the number of elements in the trie. This method should run in O(1) time.
- `isEmpty` - Returns true if the trie is empty and false otherwise. This method should run in O(1) time.
- `toString` - Returns the contents of the trie, in alphabetical order, as a string. Uses the helper function `addWords`. This method should run in O(n) time where n is the number of words stored in the trie.
- `addWords` - This recursive function traverses the trie as a pre-order depth-first traversal. The string `str` stores the path to the current node. Words in the trie, when found, will be added to `output`.

Node implements Comparable<Node>
+terminal : boolean +character : char +children : MyOrderedList<Node>
+compareTo(Node other) : int +addChildren()

Note: Node should be a private class within MyTrie and should not be accessible outside of this class.

Field summary

- `terminal` - True if this node represents the end of a word stored in the trie, and false if the node is only part of a prefix of a word stored in the trie.
- `character` - The character you must follow to arrive at this node from its parent.
- `children` - The list of children nodes, one for each

Method summary

- `compareTo` - Compares the nodes based on `character`.
- `addChildren` - Adds one new node to `children` for each character in this set (this is the same set used by BookReader):

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	‘

`UniqueWords`
…
… `+addUniqueWordsToTrie()`

Method summary

- `addUniqueWordsToTrie` - Adds the unique words to a MyTrie.
    a. For each word in the list stored in book:
        - Check to see if the word is stored in the trie of unique words.
            - If it is not, add it to the trie.
            - Otherwise, skip this word.
    b. Calls toString from the trie to extract the words in order.

    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.

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 57 milliseconds.

Finding words and adding them to a linked list...  in 323 milliseconds.
The linked list has a length of 570240.

Adding unique words to a trie...  in 720 milliseconds.
20228 unique words
2662533 comparisons
```

Traversing the trie... in 290 milliseconds.

Submission

You will submit a .zip file containing:

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

Grading Rubric

In order to count as complete your submission must:

1. Implement the trie efficiently and without errors.
2. Follow the API exactly so that your classes will work with any testing code.
3. Pass all tests in the testing code. If you fail any tests, you will be told which test failed.
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 2662533 comparisons to find the unique words in the trie.

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.