Autocomplete

Topics: Trie data structure/prefix tree, recursive data structures, comparing records, UML diagrams, testing, algorithm analysis, file I/O

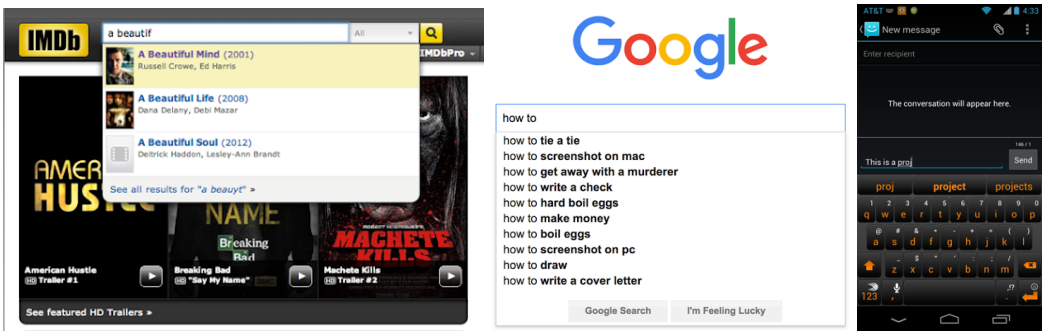
Due Date: April 2 @ 11:59PM

Starter Code

Background

In this assignment, you will write a program to implement autocomplete for a given set of N terms, where a term is a query string with an associated non-negative weight. That is, given a prefix, return all queries starting with that prefix in descending order of weight.

Autocomplete is pervasive in modern applications. As a user types, the program predicts the complete query (typically a word or phrase) they intend to type. Autocomplete is most effective when there are a limited number of likely queries. For example, IMDb uses it to display names of movies as a user types; search engines use it to display suggestions as a user enters search queries; cell phones use it to speed up text input.



In these examples, the application predicts how likely it is that the user is typing each query and presents them with a list of the top-matching queries, in descending order of weight. These weights are determined by historical data, such as box office revenue for movies, frequencies of search queries from other Google users, or the typing history of a cell phone user. For this assignment, you will have access to a set of all possible queries and associated weights (these queries and weights will not change).

The performance speed of autocomplete is critical in many systems. Consider a search engine that runs an autocomplete application on a server farm. According to one study, the application only had 50ms to return a list of suggestions for it to be useful to the user. Moreover, in principle, it must perform this computation for every keystroke typed into the search bar for every user!

You will implement autocomplete using a Trie. Your program will find all query strings (suggestions) starting with a prefix and sort the matching terms by weight or lexicographic order.

Assignment Steps

Step 0: Understanding the Problem

You are expected to submit a class diagram describing the program you write for this assignment. Specifically, you should submit a domain model diagram, as described in lecture. Consider reading through the assignment now all the way through and creating your diagram now, both to get it out of the way and to make sure you understand how all of the pieces will relate to each other.

Step 1: Term

Write a data type `Term.java` that implements the `ITerm.java` interface. This class represents an autocomplete `term`: a string and an associated weight (the data fields). Create getters/setters for these data fields in the class as well.

> **NOTE**
> In Java, static methods can be defined in interfaces. Static methods are initialized at compile time and are associated with the class or interface in which it is defined (rather than an object). Dynamic methods are initialized at runtime and are associated with an object. You should implement the static methods in `ITerm.java` directly.

Terms can be compared by three different orders (for lexiographic orderings, you can use the String `compareTo()` method):

1. **Lexicographic order by query string:** The natural order.
2. **Lexicographic order by query string using only the first r characters:** A family of alternate orderings. This order may seem a bit odd, but you will use it in Task 3 to find all query strings that start with a given prefix of length r. This corresponds to the `byPrefixOrder(int r)` method.
3. **Descending order by weight:** Corresponds to `byReverseWeightOrder()`. Your `Term.java` class should have the following constructor:

```
// Initializes a Term with the given query string and weight
public Term(String term, long weight) {
}
```

> **CORNER CASES**
> 1. The `Term` constructor should throw an `IllegalArgumentException` if `term` is null or `weight` is negative.
> 2. `byPrefixOrder()` should throw an `IllegalArgumentException` if `r` is negative.

> **TESTING TIP**

> Use the `Collections.sort()` method to test your orders. For example, if you create a List of `ITerm(l)`, to sort list `l` using reverse weight order, you will do: `Collections.sort(l, ITerm.byReverseWeightOrder());`
>
> Then check if the items are properly ordered: that the first element in `l` is the `ITterm` with the highest weight, the second is the one with the second largest weight, etc.

Step 2: Node

Write a data type `Node.java` that represents a node in your Trie. This data type will maintain a reference to a `Term` object. The Node class must also keep track of the number of words (in the subtrie rooted at node) matching the query string, the number of words with prefixes (in the subtrie rooted at node), and references to the next nodes. Create getters and setters for all data fields in this class using Eclipse's naming suggestions. The Node class has the following attributes:

1. `term`: Holds a `Term` object (a word and its weight). For nodes that do not correspond to a complete word, initialize the `Term` object so its `Rerm value is an empty string and its weight value is 0.
2. `words`: This field indicates whether this node represents a complete word (for simplicity, a complete word is also considered a prefix). For example, in the "pokemon.txt" file, if a query string is "mew", then the words value would be 1 for the node, since there is a Pokémon whose complete word is "Mew". If the query string was "char", then the words value would be 0 for that node. For this assignment all input words are unique, so this value won't be greater than 1.
3. `prefixes`: This field indicates how many words have the prefix of the node. For example, if a query string is "mew", then the prefixes value would be 2, since there are two Pokémon that have that prefix (Mew and Mewtwo). If the query string is "char", then the prefixes value would be 3 for that node (Charmander, Charmeleon, Charizard). An empty string is a prefix to all `Term`s.
4. `references`: Array of references to this node's children. Each vertex must have references to all his possible children (in our case 26). You must initialize all references to null in your constructor.

Your Node class should have two constructors:

```
// Initializes a Node with empty string and 0 weight
public Node() {
}
// Initializes a Node with the given query string and weight
public Node(String query, long weight) {
}
```

> **CORNER CASES**
> The Node constructor should throw an `IllegalArgumentException` if `query` is `null` or if `weight` is negative.

Step 3: Autocomplete

In this part, you will implement a data type that provides autocomplete functionality for a given set of strings and corresponding weights, using `Term` and `Node`. Your `Autocomplete.java` class will implement a `Trie` data structure. Your class should keep a reference to the root `Node` of the `Trie`.

Organize your program by creating a data type `Autocomplete.java` that implements the `IAutocomplete.java` interface. Most of the implementation details for `Autocomplete.java` are listed in the JavaDocs found in `IAutocomplete.java`.

Some important things to keep in mind when implementing `Autocomplete.java`:

- When building your trie from a file, you should convert all strings to lowercase.
- Preserve encapsulation by returning copies of the `Terms` that you suggest instead of returning references to the `Term` objects that constitute your `Trie`.
- For inputs that contain special characters or prefixes that are not contained in the Trie`, do nothing and return null/0.

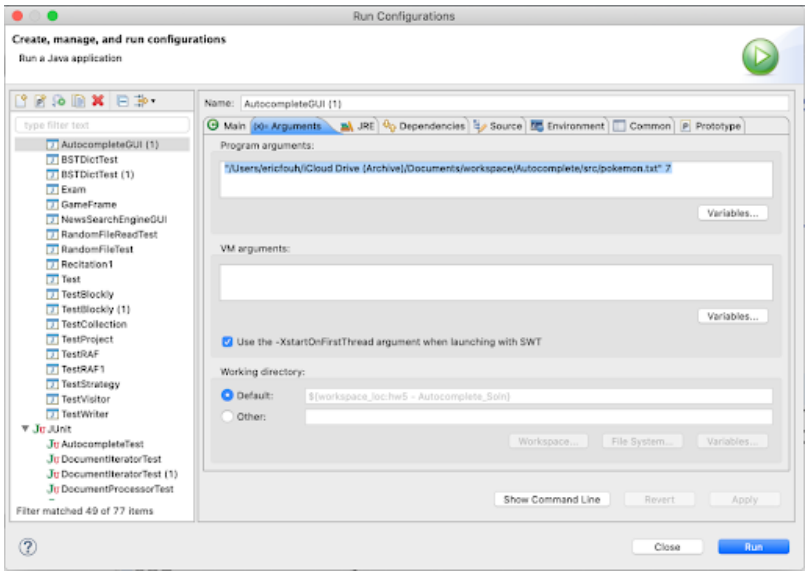
> **NOTE**
> Feel free to create additional private helper methods if they will help you implement your solution.

Step 4: Testing & Running the GUI

Create test classes `TermTest.java`, `NodeTest.java`, and `AutocompleteTest.java`. Submit any text files your tests use.

Running the GUI

Right click on `AutocompleteGUI.java` **-> Run As -> Run Configurations -> select "(x) = Arguments" tab**. In the "Program argument:" box enter the full path of the file of terms (in double quotes), a space, then an integer (see image below).



Do not forget to test and polish your code!

Step 5: Submit to Gradescope

Submit all of your classes (`Autocomplete.java`, `AutocompleteTest.java`, `Node.java`, `NodeTest.java`, `ITerm.java`, `Term.java`, `TermTest.java`), the `readme.txt` file, the class diagram file, and all relevant testing files to Gradescope. Make sure that you follow all of the CIT 5940 testing guidelines so that the

autograder runs properly on your submission. Submissions without a test class will result in a grade of zero on the assignment.

Grading

Your grade will consist of points you earn for submitting the homework on time and for the final product you submit.

The assignment is worth 205 points.

Description	Points
Autograder	120
Documentation/Style	15
Testing (correctness & code coverage)	50
Class Diagram	15
Readme	5

This assignment was adapted by Eric Fouh, based on Matthew Drabick and Kevin Wayne assignment.