STRING INDEX

This problem evaluates your skills in *object-oriented design*, *concurrent programming* and *testing*. You are free to choose any appropriate OO programming language to implement your solution. However, do try to use only standard APIs and libraries. Feel free to explain or discuss your design choices/assumptions as comments in your code.

Problem

You are to implement a *practical*, *thread-safe* and *scalable* data structure for storing a collection of strings. Users may query for the current number of strings in the collection that have a certain prefix or suffix, and subsequently remove these strings from the collection if they choose to. The data structure should be encapsulated in a class named `StringIndex`, with a supporting class named `Result`. The interface specifications of these classes can be found on the next page.

You are free to implement the data structure internally in any way you see fit as long as the requirements are met and interfaces are implemented to specification, but do note that your design choices will be scrutinized. In a short report, describe (1) any additional assumptions you have made, (2) how your implementation accomplishes the above requirements, (3) the limitations and/or tradeoffs in your implementation, and (4) any 'smart' design choices you want to highlight.

Finally, your submitted program should instantiate the data structure and execute a series of test routines on it to verify the *correctness* of your implementation, especially when multiple threads are involved. Explain, in your code, the purpose of each test routine and how it contributes to the overall verification process. Note that a 'correct' implementation must never experience *deadlock*, *livelock* or *inconsistent* results from operations. You may also provide a convincing argument (i.e. proof) of correctness in your report if you feel that the test routines alone are not sufficient.

Requirements

1. All operations should be case-sensitive.
2. The time complexity of query operations should be $O(m + \log n)$, where m is the expected size of the result of the operation.
3. The implementation must be *thread-safe*, so concurrent reads or writes by multiple threads should never result in operation errors or show inconsistencies in the return values. For example, it should be impossible for the sum of all `remove()` return values to exceed the number of `insert()` calls made so far.
4. You should attempt to minimize unnecessary blocking of threads so that the data structure can support a greater number of concurrent operations. You can assume that there will be significantly more reads than writes and that strict operation-level atomicity is not required.
5. You must assume that the object will be active for an *indefinite* period of time (e.g. in a high-availability database system) during which insertions, queries and deletions would be carried out. Consider this in your design.

CLASS StringIndex

- `insert(s) : int`

  Inserts string `s` into the collection. Duplicate and empty strings are acceptable. The method should return the number of strings equivalent to `s` *already* in the collection prior to the new insertion. If this is the first insertion of string `s`, the method should return zero.

- `stringsWithPrefix(s) : Result`

  Returns a `Result` instance representing a *snapshot* of all strings with prefix `s` in the collection *at invocation time*. In other words, the contents of the result object are *unaffected* by subsequent insertions or deletions of strings. An empty string is an acceptable prefix.

- `stringsWithSuffix(s) : Result`

  Same requirements as the above method, but returns strings with suffix `s` instead. An empty string is an acceptable suffix.

CLASS Result

- `size() : int`

  Returns the number of strings in this `Result`, including duplicates. Will be zero if there are no matching strings for the query.

- `remove() : int`

  Removes all (and only) the strings represented in this `Result` from the parent `StringIndex` collection. This method must *not* remove any strings that were inserted *after* the result was generated.

  Returns the number of strings actually removed, x. Possible outcomes: $x = size()$, if all of the referenced strings were successfully removed; $x = 0$, if all referenced strings have been previously removed by this `Result` or other `Result` instances; $0 < x < size()$, if $size() - x$ referenced strings were previously removed by other instance(s) of `Result` and only x strings were removed in this invocation.