String Index Assignment 3

Manoj Kumar (manoj18.iitd@gmail.com)

December 12, 2020

Problem Statement

We need to implement a *practical, thread-safe* and *scalable* data structure for stroing 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*.

Interfaces:

- StringIndexInterface.java

```
public interface StringIndexInterface {
    public int insert(String s);
    public Result stringsWithPrefix(String s);
    public Result stringsWithSuffix(String s);
}
```

- ResultInterface.java

```
public interface ResultInterface {
    public int size();
    public int remove();
}
```

Implementation Details

I implemented an ***optimized Trie*** Data Structure to store the strings. It is different from an ordinary Trie in the following ways:

- It is space optimized. I used an ordered *TreeMap (Inbuilt Red-Black Tree)* to store the *TrieNodes*. An Red-Black tree is memory efficient as compared to a Hash Table.
- It is Thread-safe.

Interface of the implemented trie data structure is given below:

```
public interface TrieInterface {
    public int insert(String word, TrieValue v);
    public TrieNode startsWith(String s);
    public boolean remove(String word, TrieValue v);
}
```

All the data is maintained by two different Trie objects *Trie1* and *Trie2*, where strings are stored in *Trie1* and their reverse ordering strings are being stored in *Trie2*. *Trie2* is maintained to get the list of strings with a perticular *suffix*.
A *synchronized Queue* is also maintained to store and execute the instructions. Whenever a new instruction arrives, the program adds it in the queue and starts a new thread.
Each thread pulls an instruction from the queue in a *synchronized* way and begins to execute it. It calls respective functions to execute the instruction.

Thread Safety & Synchronization:

- The input queue is *synchronized* while *en-queuing* and *de-queuing* by threads.
- **Locks while inserting:** A *lock* blocks the more than one threads to access the same resource. Each *trieNode* represents a character. Also, each object of the *trieNode* has its own **ReentrantLock**. While inserting a string, we lock the current *trieNode*. Later, when we move to the next *trieNode* to insert the next character of the string, we unlock the previous *trieNode* and lock the the next *trieNode*. The same process runs til we insert the whole string into the trie.
- **Thread Safety in startsWith function**: function *startsWith* in a *trie* returns respective *trieNode* of the last letter of the given *prefix*. Since, it is very fast compared to inserting a string, we check locks whether a strings with the given prefix is currently inserting or not. While reading the trie, we do not need to lock a *trieNode* so we just check here if it is unlocked move forward.
- **Thread Lock while removing an element**: While deleting an element, we locks its last *TrieNode* to prevent collisions.

Working of the program:
Input format of the instructions is of 5 types:

(i) **insert str** : This instruction invokes the *insert* function of *StringIndex* to store the String *str* in our data structure trie. String *str* supports all type of strings including empty strings.
it creates a new *TrieValue* object and inserts it in both Tries : *Trie1* and *Trie2*.

(ii) **stringsWithPrefix object prefix**: This invokes the *startsWith* function of StringIndex with the prefix string. And returns and object of class *Result* which consists of all the strings of the StringIndex object containing the prefix *prefix* and their corresponding *TrieValues*.

(iii) **stringsWithSuffix object suffix**: Same as *stringWithPrefix*, but it works on *Trie2* and the string it uses is a reversed string but while storing in Result object, it again changes the strings to their original format.

(iv) **remove obj**: It removes all the strings that were returned in the object of class *Result* named *obj*. It returns total number of removed strings from the trie by this operation. Result object stores the corresponding TrieValues of the strings, so this operation changes them to null as well. It helps to prevent deleting an element that is inserted after creating the Result object.

(v) **exit**: It exits the program.

Requirements Satisfied

1. **All operations should be case-sensitive.**
   We are using the ascii values of the characters to store them. Hence, every character which have a positive ascii value is supported in our data structure.
   Also, I am storing empty strings in the *root* of the Trie. So it supports *empty strings* as well.

2. **Time Complexity:**

   (i) **insertion**: on micro level, average time complexity of inserting an element in the trie is $O(h * log(k))$ where h is length of the string and k is total number of ASCII characters. Since, I am storing the child *TrieNodes* in a **TreeMap (RB Tree)** to optimize the memory, hence, to lookup a TrieNode, it can take upto $log(k)$ time where k is total number of ascii characters (128). Furthermore, we are again using an **RB Tree** to store values on a TrieNode, it takes $O(log(n))$ time in insertion. If we take the length of the string constant.
   Average case, Time Complexity : $O(1)$
   Worst Case, Time complexity : $O(log(n))$: if all strings are same.

   (ii) **stringsWithPrefix/stringsWithSuffix**: trie takes $O(1)$ time to search the *trieNode* with the given prefix (if we do not consider prefix length too much). Now we use dfs in the trie to collect all the strings and *TrieValues*. Time complexity of this operation is $O(m)$ where m is total number of strings.

   (iii) **remove**: Time complexity of removing a string from the trie is $O(1)$ (considering string length not too large).
   Therefore, For all strings of Result object, Final Time Complexity: $O(m)$

   Here we can see that time complexity of all operations in worst case is $O(m + log(n))$.

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.
   We are using separate locks and synchronized blocks for each trieNode and shared items to make is thread safe and efficient.

4. You should attempt to minimize unnecessary blocking of threads so that the data structure can support a greater number of concurrent operations.
   To minimize the unnecessary blocking of threads, I use separate locks for each *TrieNode*. So it unlocks the previous *TrieNode* and locks the next *TrieNode* before moving to the next *TrieNode*.

5. You must assume that the object will be active for an indefinite period of time.
   Whenever a new input arrives, the main thread creates and starts a new thread to execute the operation. So the object is active for an indefinite period of time.

Testing and Correctness

Let's first check the program when a no two consecutive threads running. We manually give inputs to the console by maintaining a short duration of time between two inputs so that each thread completes before the next instruction and no two threads run concurrently.

<terminated> StringIndex [Java Application] /home/manoj/.p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.j

```
insert Manoj
Manoj inserted 0
insert Ayush
Ayush inserted 0
stringsWithPrefix o1 Ay
Ay stringsWithPrefix returned : o1 Size : 1 [Ayush, ]
stringsWithSuffix o2 oj
oj stringsWithSuffix returned : o2 Size : 1 [Manoj, ]
insert Ayush
Ayush inserted 1
insert Manoj
Manoj inserted 1
remove o1
o1 Removed. total items removed: 1
remove o2
o2 Removed. total items removed: 1
remove o1
o1 Removed. total items removed: 0
insert Ayush
Ayush inserted 1
insert
 inserted 0
stringsWithPrefix o3
 stringsWithPrefix returned : o3 Size : 4 [, Ayush, Ayush, Manoj, ]
exit
```

Operations:

(i) **insert Manoj** : it inserts the string *Manoj* and returns 0 (the number of strings equivalent to *Manoj* already in the collection prior to the new insertion.)

(ii) **insert Ayush** : it inserts the string *Ayush* and returns 0 (the number of strings equivalent to *Ayush* already in the collection prior to the new insertion.)

(iii) **stringsWithPrefix o1 Ay** : it returns the Result object *o1* where total strings in *o1* is 1 (*Ayush*).

(iv) **stringsWithSuffix o2 oj** : it returns the Result object *o2* where total strings in *o2* is 1 (*Manoj*).

(v) **insert Ayush** : it inserts the string *Ayush* and returns 1 (the number of strings equivalent to *Ayush* already in the collection prior to the new insertion.)

(vi) **insert Manoj** : it inserts the string *Manoj* and returns 1 (the number of strings equivalent to *Manoj* already in the collection prior to the new insertion.)

(vii) **remove o1** : it removes all the strings that was present in *Result* object *o1*. total strings removed : 1

(viii) **remove o2** : it removes all the strings that was present in *Result* object *o2*. total strings removed : 1

(ix) **remove o1** : When I tried the same operation again, it returns 0, as all the strings of object o1 were deleted previously and it does not delete the string *Ayush* that was inserted after creating the object.

(x) **insert Ayush** : it inserts the string *Ayush* and returns 1 (the number of strings equivalent to *Ayush* already in the collection prior to the new insertion.)

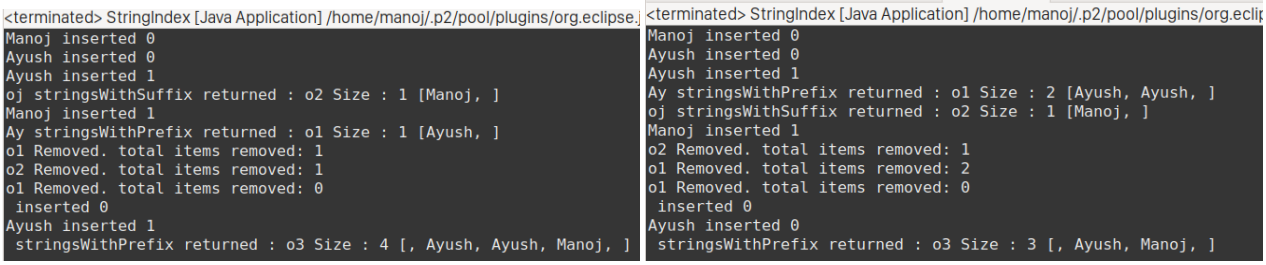
(xi) **insert** : It inserts an empty string.

(xii) **stringsWithPrefix o3** : it returns the Result object *o3* with all strings.

No we tried the same inputs by taking inputs from a file to show the effect of multithreading.
input.txt :

```
insert Manoj
insert Ayush
stringsWithPrefix o1 Ay
stringsWithSuffix o2 oj
insert Ayush
insert Manoj
remove o1
remove o2
remove o1
insert Ayush
insert
stringsWithPrefix o3
```

The output changes every time we run the program.



These two different outputs for the same input file shows the working of multithreadings.

Commands to execute

The program in implemented in java. JDK 14 (openjdk 14.0.2)
IDE used to compile the files : Eclipse IDE for Java

- To run the program without IDE, run the command:

```
chmod +x run.sh
./run.sh path_of_input_file
```

- To delete all the .class files, run:

```
./run.sh clean
```

Implemented by:
Manoj Kumar
manoj18.iitd@gmail.com