The Solution to Rhyme Word Matching

Chengkang Xu
cx33@duke.edu
July 2018

INTRODUCTION

This solution was written in Java because I am most familiar with Java when building data structures. My solution used principles in OOP, such as inheritance, abstraction. Codes are refactored that they are scalable and very easy to add additional functionality or adding a different implementation of the fundamental data structure. It took me 4 hours to develop the solution and 2.5 hours to test and write the documentation.

Build & Run Instruction

This application is written in Java. Any machine with Java 1.8 or older should be able to compile and run it in any IDE. Eclipse is recommended to run the application since JUnit is used. You will need JUnit 5 on the build path to get the JUnit testing running in eclipse. Or you can move the TestReverseTrieApplication.java from the src to other place, and re-run the program using TestRunner.java, which outputs the test result to console.

ASSUMPTIONS

I assume each word does not contain special characters, such as: "-" and the application only needs to handle English words. Besides, I am assuming the fair random needs to be achieved when randomly select a word from the rhyme words pool.

SOLUTION

Algorithm

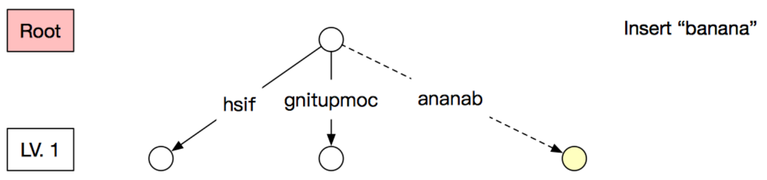
To solve the problem, I defined a data structure called compressed reverseTrie which is based on the idea of compressed Trie. The compressed reverseTrie is a tree-like data structure that stores substrings and characters within TrieNode. It is compressed because it not only stores the character but also stores substrings that begin with that character. Each TrieNode contains two hashmaps: one of type <Character, TrieNode>, which maps each ending character of a substring in the rhyme wordlist to a reference of TrieNode that contains the information for that substring. Another hashmap is map <Character, StringBuilder>, which maps the same character to the substring that this character corresponds to. Each TrieNode also has a Boolean field that indicates whether the node is the end of a complete word. Since we are tracking rhyme words, the string is stored reversely in the StringBuilder.

There are two main operations defined in ReverseTrie class, insert and search. Both have a similar procedure for examining data.

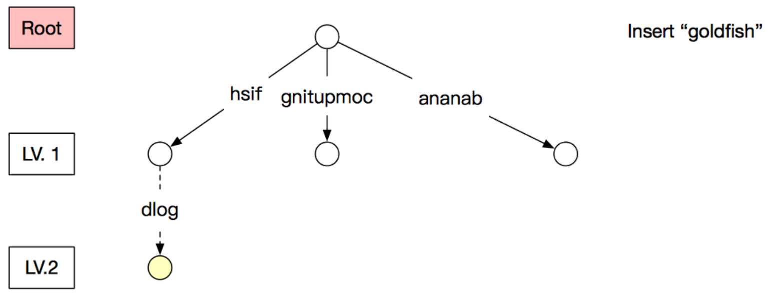
Insertion

There are 5 cases in insertion; yellow circles are new nodes created in the insertion step:

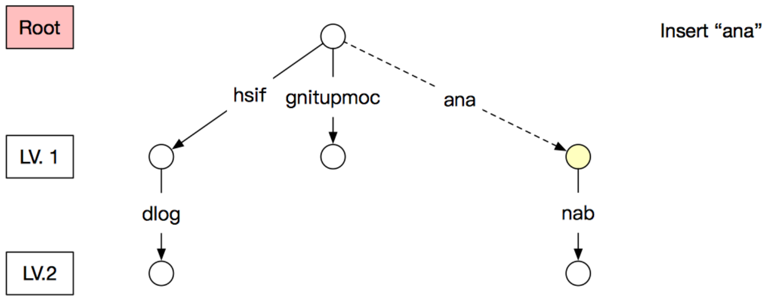
1. Insert a word that does not overlap with the previous word



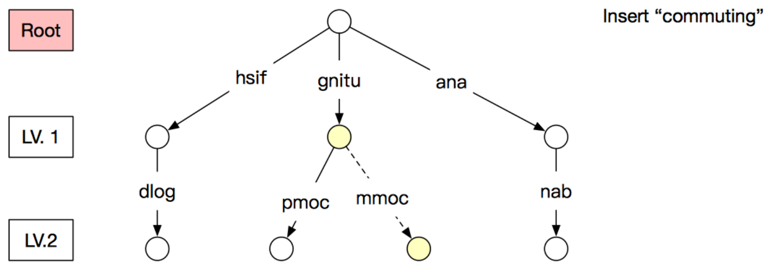
2. Insert a node that already has its suffix stored



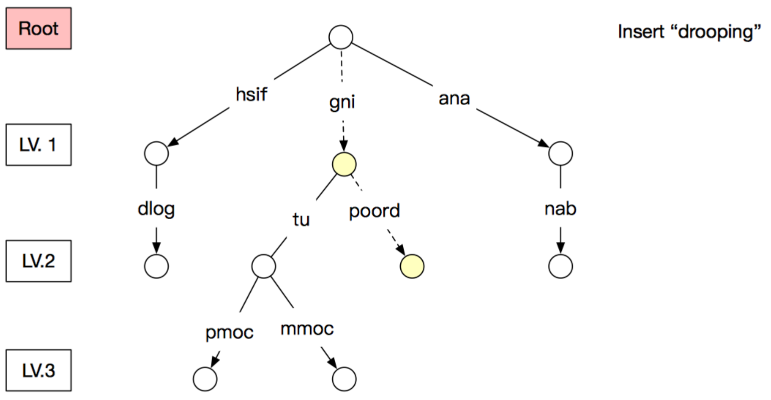
3. Insert a node that is a prefix of the stored node.



4. Insert a node that has the common suffix with another word in the Trie



5. Inserting a node that has a common suffix with another word in the Trie, which causes reposition of existing node.



To insert a new node, the algorithm starts with the first character in the word. It checks whether the root of Trie's children map contains this character as key. If there is, comparing the string with the string associates the character key stored in the map. Otherwise, store the new node (case 1). If the input string contains the current substring stored in the map and it is longer than the substring, get the first character of the remaining characters in the input string, then check the next child substring that corresponds to this character. Whenever the character does not map to a string, the algorithm breaks loop and check whether all characters in input has been checked. If it is, it means input is part of an existing node, set current node to completed (case 3). If it is not, means input is longer than matching keywords in this trie. Thus, a new node containing the remainder of the string will be added to Trie (case 2). When there is a miss matched character between current node's string and input string (case 4, 5), modify the string in the node to be the common parts of two strings. Then create two new nodes to store remainder of the input and rest of the string in the node. Copy children mappings from the original node to the newly created node that contains the remainder of the original string. Then modify the mapping in original to map to these two new nodes.

Search

Search is similar to insert in terms of finding the last matching substring node. It uses the same technique to detect whether the input string matches with any of the nodes in the Trie. If it does not find any, return null. While it is searching for more matches, it stores the matched substring in a StringBuilder. After it finds the last matching substring node in the Trie, it uses Depth First Search to search the entire spanning tree rooted at the last matching substring node. Before each time it goes a level deeper in the Trie, it creates a new StringBuilder and passes it into the recursion call, which makes sure that each branch does not interfere with each other. Whenever it finds a node with a completed indicator, it stores the String to the ArrayList. After all leaves of the sub-trie are checked, it gets the size of the ArrayList, and randomly select an integer in [0, size - 1] as the index in the ArrayList. Then it returns the randomly selected word.

Complexity

Time Complexity

Inserting a new word requires comparison of the new word with keys stored in Trie, so iterating a word with length n takes O(n) time. Modifying the Trie structure when split or swim down is needed takes O(1) time. So average case for insertion is O(n). If there are m words to be inserted, it takes O(mn) to insert all words.

Searching a word also requires iterating through the word, which takes O(k) for a word that has length k. Also, since randomness of selecting words is required, it needs to retrieve sub spanning tree using DFS if it finds a matching suffix, and then randomly return one string from the tree. It is necessary to traverse the entire sub spanning tree since I am assuming there maybe skewness in the data, in which some subtrees are longer. In this case, we cannot merely randomly pick one in each level and combine the substrings since the probability of reaching each leaf node will be unequal. The number of nodes highly depends on the commonality among words, if words are very similar to each other, time complexity will be reduced. Assuming there are m words, the number of nodes in the trie will be:

$$S_n = \frac{a_1(1-q^n)}{1-q}$$

Where a_1 is m, q is one divide by the number of children per node, and n is the number of levels, i.e., how many substrings word is divided into. In the worst-case scenario, only first ending character is matched, and entire trie is biased toward that character, the time complexity will be Sn, which is the number of nodes in the trie. From my observation, English words in Trie typically divided into a finite number of substrings, i.e., 5~6 substrings per word. Each node in the tree usually have a limited number of children, i.e., there are 5~6 words have that suffix. If that case,

$$S_n = \frac{m\left(1-\left(\frac{1}{5}\right)^5\right)}{1-\frac{1}{5}} \sim m$$

In a typical case, the time complexity of getting the sub-trie will be close to O(m). So, the total time complexity of searching is O(m) + O(k).

Space Complexity

In my solution, compression is used to reduce space complexity. Each node not only stores a key character but also stores a partial string that is unique given its ancestors. As discussed above, there are Sn nodes in the Trie where:

$$S_n = \frac{a_1(1-q^n)}{1-q}$$

In general, S_n is close to m as there is the finite number of both children per node and levels of the Trie. Thus, the space complexity in general would be close to O(m).

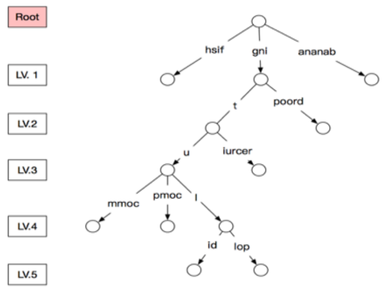
Test Cases

To examine the correctness of random-select, un-comment line 215 in ReverseTrie.java, and then re-run the application. It will print the pool of most rhyming words (subtree) it randomly selects from for each input. You may run TestRunner.java if you want to see the result printed in console; otherwise run TestReverseTrieApplication.java to use JUnit testing.

Given Sample rhyme list 1

Wordlist: Computing, Polluting, Diluting, Commuting, Fish, Recruiting, Drooping, Banana

Reverse Trie built:



```
parent 0 level 1 a ananaB true
parent 0 level 1 g gni false
parent gni level 2 p poorD true
parent gni level 2 t t false
parent t level 3 u u false
parent u level 4 p pmoC true
parent u level 4 l l false
parent l level 5 i iD true
parent l level 5 l loP true
parent u level 4 m mmoC true
parent t level 3 i iurceR true
parent 0 level 1 h hsiF true
```

The results above is the Reverse Trie built by the given word list. Each line indicates the information about each node. The string after *parent* is the string value of their parent, and the number after *level* is the level of each node. The single character after level number is the first character of the string of current node followed by the full string of current node. The *true* and *false* indicates whether this is a completed word. Therefore, the trie above has five levels and 13 nodes including the head, and their positions are correct.

Test input:

Below test output shows the pool of the most matched rhyming words and the randomly selected word from the pool. If the input is invalid, it will also display invalid target.

Empty input: ""

```
invalid target
null
```

Null input: Null

```
invalid target
null
```

Suffix input: "ting"

```
[Computing Diluting Polluting Commuting Recruiting]
Computing
```

Has common suffix with many words in the trie: "Shooting"

```
[Computing Diluting Polluting Commuting Recruiting]
Diluting
```

Has common suffix with one word: "Disputing"

```
[Computing]
Computing
```

Has common suffix with many words in the trie:"Convoluting"

```
[Diluting Polluting]
Polluting
```

Does not have rhyme word: "Orange"

```
null
```

Same word as in the tree & also word at first level: "Banana"

```
[Banana]
Banana
```

Word that has a word in the Trie as its prefix: "kbBanana"

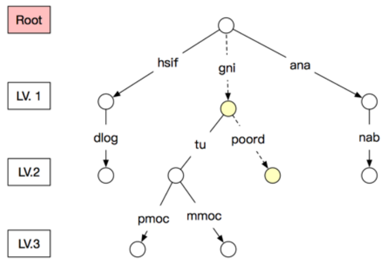
```
[Banana]
Banana
```

The outputs are correct, and the word pools for each case are as expected.

Sample rhyme list 2

Wordlist: banana, computing, fish, goldfish, ana, commuting, drooping

Reverse Trie built:



```
rhymelist2
parent 0 level 1 a ana true
parent ana level 2 n nab true
parent 0 level 1 g gni false
parent gni level 2 p poord true
parent gni level 2 t tu false
parent tu level 3 p pmoc true
parent tu level 3 m mmoc true
parent 0 level 1 h hsif true
parent hsif level 2 d dlog true
```

As shown above, the tree is built as expected.

Test input:

suffix of a word: "ana"

```
[ana banana]
banana
```

suffix of a word: "fish"

```
[fish goldfish]
goldfish
```

Suffix of word: "ping"

```
[drooping]
drooping
```

Entire matching word: "goldfish"

```
[goldfish]
goldfish
```

The outputs are correct, and the word pool for each case are as expected.

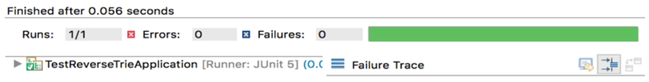
Sample rhyme list empty list 3 & null list 4:

```
rhymelist3 & 4
invalid rhymelist
invalid rhymelist
```

Return invalid as expected.

Automated Testing:

Above test cases are also used in JUnit test. All tests passed.


Finished after 0.056 seconds

Runs: 1/1 Errors: 0 Failures: 0

TestReverseTrieApplication [Runner: JUnit 5] (0.(Failure Trace

EVALUATION

Accuracy

As shown in tests, the trie is built as intended and search function gets the correct sub-trie and randomly select one from the sub-trie. Besides, users can choose whether it is case sensitive by passing a boolean value to the constructor of ReverseTrieApplication.

Speed

Assuming a client-server architecture, ReverseTrieApplication is run on the server only once to build the trie. It takes at most O(mn) to construct the Trie as discussed previously. After the Trie is built, every search only takes O(m), which is efficient. Since Siri is based on the internet at this time, the building part can be done on the server where the trie can be stored. When the client requires for execute, we only need to use the search function to return the results. Currently search in ReverseTrie is returning a Result object for testing purposes (pool of rhyming words are contained in the object with prefix added). To boost performance, we can simple return the selected word instead of the Result object which saved time adding prefixes to words in the pool.

Scalability

Since the time complexity of search is linear and build operation only run once, it can scale to a large set of rhyme words without impacting the performance badly.

Reusable

Trie abstract class can be easily reused to implement different Trie. Also, many common methods are put in Trie abstract class to increase reusability.

Modularity

The fundamental data structure used in the solution is TrieNode and ReverseTrie. ReverseTrie inherits from Trie, which is an abstract class that can be easily extended to a different implementation of Trie. Current ReverseTrie inserts rhyme list reversely and search would return a randomly select word from the sub-trie. If we want to implement a Trie that inserting words in normal order and search methods return boolean instead of a word, we can easily inherit from Trie and implement the abstract insert and search method differently. Another layer in the application is the ReverseTrieApplication. It builds a reverse trie during initialization, and it has the rhyming method. This class is used to put methods related to reverse trie, and it is straightforward to add new ways into it.

DISCUSSION

Random Issue

According to the sample cases from the assignment, when there are several words considered as the best matches, the result should be a random one from this list. Thus, when achieving the requirement, every word in the list has the same possibility to be returned. However, in the real-world application, if we don't have to keep the randomness, we may save more time and space by returning the closest rhyme nodes to the given key. By doing that, the application does not need to traverse to the leaves of the tree whenever it finds rhyme words exist in the Trie. Or, if we can assume that Trie is balanced that each branch has the same number of children nodes, we can simply randomly select a branch when traversing down the trie and return the node found at a leaf. By doing this, we do not need to traverse the entire subtree which reduces the runtime to O(k) where k is the length of the input word.