

CMPE 343 HW 5 Report

Information

4865115490, 27673119242

Fatih Mehmet Yiğitel, Serra Sabah

Section 02, Section01

HW5

Problem Statement and Code Design

In the task we are asked to implement some methods that using the Trie algorithms in the book. We have different methods for different aims.

In the **Boolean Search(String args)** method: If the given input in the exist in our trie, this function returns true otherwise false.

In the **Void autoComplete(String prefix)** method: If the string has the given prefix, The functions prints that word. It also rearranged the order lexicographically.

In the **Void reverseAutoComplete(String suffix)** method: This function should print all strings end with given suffix in your trie, lexicographically.

In the **Void FullAutoComplete(String prefix, String suffix)** method: This function should print all strings start with given prefix and end with given suffix in your trie, lexicographically.

In the **Void findTopK(int k)** method: This function should print top k words that have most occurrences, lexicographically.

In the **(String filepath)** method: This function should read input from the given filepath and print all possible words in your trie Example file,puzzle1.txt, is as follow:

Implementation and Functionality (%40):

Tester class:

In the tester class, we use a file reader, the buffered reader using the java library and we take the txt file as input and we add the strings in the text file to “words” ArrayList. After that, we add the this word ArrayList elements into the “st” trie data structure that we create. So in the end we use this trie data structure for the methods. We create a switch case statement for the different methods that we want to call. Using scanner objects using from the java library we take the input from the user to call the methods that we want. We used System.exit(0) so that the program can terminate at the end of each case.

case 1:

It uses the constructor that uses tries as parameter in the driver class. `Public Driver(TrieST<?>t).`

The search method calls the “contains” method in that existed in the TrieST.java class.

`Return t.contains(name);`

The “contains” method has parameters as string. It takes the same parameter in the search method. The contains method checks the given keys as the parameter. If key equals to null it

returns null otherwise it returns get method in the TrieST.java class. The “get” method returns value of the given key it uses node data structure if the key exists in the trie it returns

```
(Value) x.val;
```

Driver class:

public Boolean Search(String name) { : This method returns true if we have the input is given to us in our trie, otherwise false. By using the contains a method of the trie class, we find whether the value in the parameter is in trie. **return** t.contains(name);

public void autoComplete(String prefix) { : Thanks to the keysWithPrefix(prefix) method in Trie, we can find whether the prefix entered in the parameter is in Trie.

keysWithPrefix method is the iterator method and it gives us to some strings according to perexix given as a parameter. However we need to order them lexicographically, so we decided to add them into an ArrayList to rearrange the order of strings lexicographically. To do that we use iterator.hasNext() method and we add them to ArrayList.

```
Iterator<String> iterator =t.keysWithPrefix(suffix).iterator();  
ArrayList<String> Listed = new ArrayList<>();  
while (iterator.hasNext()) {Listed.add( iterator.next());}
```

Using *Collections.sort(Listed)* we were able to sort the resulting ArrayList lexicographically. Then we print the result we found by adding commas.

Void reverseAutoComplete(String suffix) We wrote 3 methods for this function. The first is to reverse the words ArrayList in the file. We assign the inverted ArrayList to a new Trie. Here we reverse the word and look at it as a prefix to find the desired suffix. If the entered suffix corresponds to the prefix in the reversed word, we keep them in an ArrayList with the iterator method. We return the reversed words we find at the end to their original form.

1) **public** String reverseWord(String[] word) {

This method reverses the words of the .txt file received from the tester. Here it is reversed using the reverse method of StringBuilder.

```
for(String w:word){  
StringBuilder sb=new StringBuilder(w);  
sb.reverse();  
reverseWord+=sb.toString()+" "; }
```

In the tester class, in the case:3 part, we put the reverse of the words in an arraylist.

```
ArrayList<String> l2 = new ArrayList<>(Arrays.asList(d.reverseWord(word).split(" ")));
```

We add the Arraylist elements to the new “t” Trie data structure we created. Later we return the result by calling *d3.reverseAutoComplete(commandword)*;

2) **public void** reverseAutoComplete(String suffix) { As if the suffix entered in this method is looking for the same prefix, we find the words containing the suffix by using the *t.keysWithPrefix(suffix)* method.

We add an ArrayList using iterator method. We call the *reverseWord1* method to find the original versions of the words in the array. Using *Collections.sort(Listed)* we were able to sort the resulting ArrayList lexicographically. Then we print the result we found by adding commas.

- 3) **private** String reverseWord1(ArrayList<String> keysWithPrefix) { This method reverses the ArrayList values in the entered parameter.

public void FullAutoComplete(String prefix, String suffix) { Here in the function, we did the same things we did in the Void autoComplete(String prefix) and Void reverseAutoComplete(String suffix) methods. We suppressed the common ones in the ArrayList of the words containing the entered prefixes and suffixes using *Listedprefix.retainAll(Listedsuffix2);*.

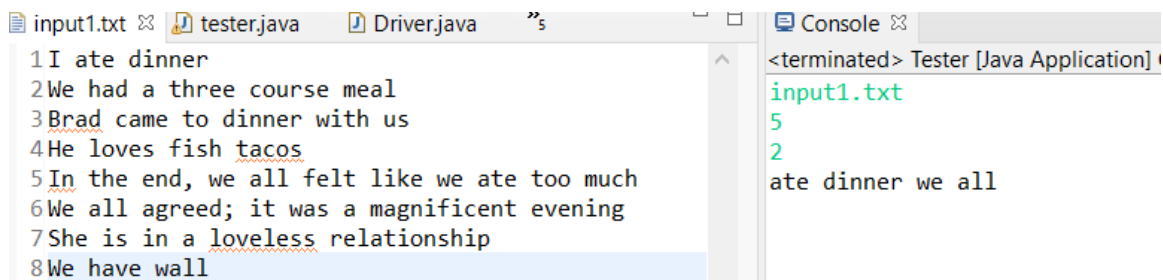
public void findTopK (int k){ In the findTopK method firstly We find occurring numbers of every string. We are holding these values in the “occurnum” array. The occurring number of any string is holding in the “occurnum” array. Strings have same index number with related occurring numbers. We are doing it in the for loop with if statement.

```
if (words.get(j).equals(element)) {
```

Secondly to print every index just once. So we use one for and in the for loop we use a if statment to do it.

```
if(occurnum[p] == k && !finedwords.contains(words.get(p))) {
```

At the end this method returns the words in the file equal to the number of the entered parameter. Like,



The screenshot shows a Java IDE with two panes. The left pane displays a text file named 'input1.txt' with the following content:

```
1 I ate dinner
2 We had a three course meal
3 Brad came to dinner with us
4 He loves fish tacos
5 In the end, we all felt like we ate too much
6 We all agreed; it was a magnificent evening
7 She is in a loveless relationship
8 We have wall
```

The right pane shows the console output for a Java application named 'Tester'. The output is:

```
<terminated> Tester [Java Application]
input1.txt
5
2
ate dinner we all
```

Puzzlefind method:

To find the words in the text file we implement Puzzle.java class. In this class we have double dimensional char array and to find the strings in the puzzle we use vector data structure. We have two methods one of them is String find method. String find method has two for loop the check row and columns. Additionally in these loops there is an if loop and it checks the puzzle has the searched word. If it finds the word it returns “confirmMatch” method.

```
if (wordFound == false && puzzle[row][column] == word.charAt(0))
```

In the confirm match method we have three parameters that taken by the “find” method. The parameters consist of the string word, row number and column number. Theconfirm match checks the every direction in the puzzle. If it finds the word string. It returns the word.

In the tester class we use file reader to read the txt file. We take the input in an array and after that we add it into our two dimensional char array. After that we create puzzle object and we set our needed parameters. For every match it returns the word. To find it I keep the returned words in another arraylist and if it returned before I did not return it again.

```
if(!printed.contains(findedwords.get(i)))
```

Testing (%15): Our Boolean Search(String args) method, Void autoComplete(String prefix) method, Void reverseAutoComplete(String suffix) method and Void FullAutoComplete(String prefix, String suffix) functions are working correctly. The test cases in the VPL work correctly.

We couldn't find the cross word in the puzzle part.

```
PS C:\Users\fatih\Desktop> java.exe -cp 'C:\Users\fatih\Desktop\src' src/input1.txt
6
src/puzzle1.txt
[a, all, wall]
[]
```

Final Assessments (%10):

The most challenging part of this assignment was finding the findTopK and Void SolvePuzzle methods. We suppressed the words that contain as many files as the number in the input entered in the findTopK method. and in the puzzle part, we could not find the crosswords.

This assignment helps us understand Trie. We learned how and where to use it.