Predictive Text Dictionary

readInDictionary()

The following that I will be referring to will be the readInDictionary method which is shown in figure 1, where I will refer to pieces of code to explain how this method works. The data structure used to help do this was the trie data structure. To store all the dictionary entries we had to initialise a new dictionary object in the form of a trie data structure and open a text file using a scanner. If the file path is wrong, or doesn't exist then the method catches a FileNotFoundException, which can be seen in the terminal.

The output of the file should include the index, the word and then the frequency which can be shown in test 4s figures. Split(" ") is used to parse through the words and the frequency is taken through parts[1] and parts[2]. A try catch block is in place to determine whether the frequency is an integer, if not the word is skipped and the method continues. Once complete the word is inserted into the dictionary object created earlier using the insert() method. The insert() method stores the frequency using a TrieData object as a sequence of child nodes for each character of the word, as shown in figure 2. Then the readInDictionary() method is complete and the created dictionary object will contain the words and their frequencies and now a functioning predictive dictionary.

```
public static Trie readInDictionary(String fileName) {
    Trie newDictionary = new Trie();

    try {
        Scanner fileScanner = new Scanner(new File(fileName));
        while (fileScanner.hasNextLine()) {
            String[] parts = fileScanner.nextLine().split( regex: " ");
            if (parts.length >= 3) {
                try {
                    newDictionary.insert(parts[1], new TrieData(Integer.parseInt(parts[2])));
                } catch (NumberFormatException ignored) {
                }
            }
        }

        fileScanner.close();
    } catch (FileNotFoundException exception) {
        System.out.println("Trying to open: " + new File(fileName).getAbsolutePath());
        System.err.println("File not found: " + fileName);
    }

    return newDictionary;
}
```

Figure 1: readInDictionary() method code to refer to throughout my explanation

```java
public TrieNode insert(String str, TrieData data) {
    // hint you can use str.toCharArray() to get the char[] of characters
    char[] Characters = str.toCharArray();
    TrieNode nodeToInsert = root;

    for (char index : Characters){
        if (nodeToInsert.getChild(index) == null) {
            TrieNode newNode = new TrieNode();
            nodeToInsert.addChild(index, newNode);
        }
        nodeToInsert = nodeToInsert.getChild(index);
    }
    nodeToInsert.addData(data);
    nodeToInsert.setTerminal(true);
    return nodeToInsert;
}
```

Figure 2: The code for the insert() method

getMostFrequentWordWithPrefix()

The following code that I will be referring to will be the getMostFrequentWordWithPrefix() method which is shown in figure 3. The getMostFrequentWordWithPrefix() method finds the most frequently used word in the trie that starts with a given prefix. A text file is read through first with numerous words. It first uses getNode() to locate the node and matches the last character of the prefix. If the prefix doesn't exist in the trie, it returns null.

If the prefix is found, the method performs a depth-first search from that node, exploring all possible word continuations and checks the frequency stored in the node's TrieData. It keeps track of the word with the highest frequency and returns it as the most relevant suggestion. The depth-first search uses a stack for all the word combinations from the letters imputed from the user. The string builder is used to build each of the words given the characters inputted by the user. This search method is chosen over breadth-first search because it's more memory efficient. The depth-first search uses a stack which is most optimal for this problem as it explores one path at a time, compared to breadth-first search which explores multiple paths at a time making it more memory costly. This method is what creates the predictive text feature in the TextAreaDemo GUI which provides words to finish when typing in, as shown in figure 4.

```java
public String getMostFrequentWordWithPrefix(String prefix) {
    TrieNode startNode = getNode(prefix);
    if (startNode == null) return null;

    String greatestFreq = null;
    int biggestFreq = 0;

    Stack<Pair> stack = new Stack<>();
    stack.push(new Pair(startNode, new StringBuilder(prefix)));

    while (!stack.isEmpty()) {
        Pair pair = stack.pop();
        TrieNode node = pair.node;
        StringBuilder word = pair.word;
        if (node.isTerminal()) {
            TrieData data = node.getData();
            if (data != null && data.getFrequency() > biggestFreq) {
                biggestFreq = data.getFrequency();
                greatestFreq = word.toString();
            }
        }
        node.getChildren().forEach((ch, childNode) -> {
            stack.push(new Pair(childNode, new StringBuilder(word).append(ch)));
        });
    }
    return greatestFreq;
}
```

Figure 3: getMostFrequentWordWithPrefix() method code to refer to throughout my explanation

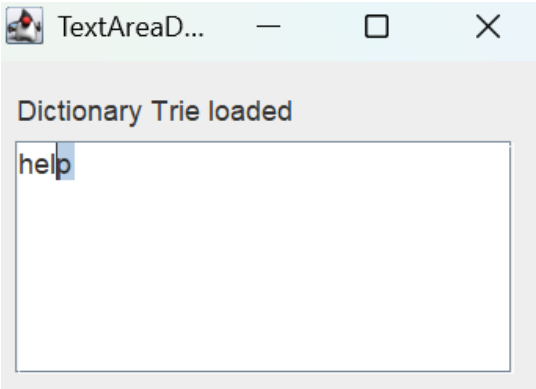

Figure 4: Example of the predictive text feature in the TextAreaDemo GUI helping finish off the word help ‘from’ the input ‘fr’

Extension idea 1

My extension idea was to make a toggle button that changes the lights and the colour of the background and the text, similar to how it can be done on an application like instagram. I usually like interacting with darker colours and modes on my phone like this, similar to instagram how I have it on the darker mode. Now I thought that this mode could potentially be more visually appealing to users as it is to me. Luckily enough I have had previous experience with swing so creating a feature like this was something that I was capable of doing on my own. This button was created using a JButton where I chose to place this button on the bottom of the screen. The method changeColour has an if else statement that contains two options. The if statement explains that when 'Dark Mode' is pressed, it changes the background of the background to black, and the text colour to white. The else statement explains that when 'Light Mode' is pressed, it changes the background of the textbox back to its original colour of white, and the text colour back to black. This is shown in figure 18.

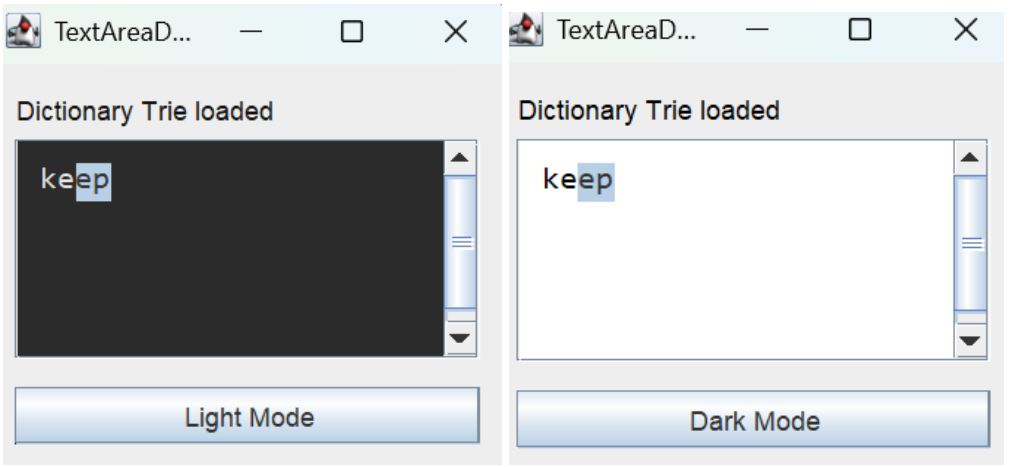

Figure 18: Pictures of the light and dark mode in the GUI for textAreaDemo

Extension idea 2

My extension idea was to make a clear text button that clears the text on the screen. This makes it easier for users that are typing long words or have decided to type something very long to click the clear button to clear the text instead of having to delete all the text manually. This was done by creating another Jbutton that was coloured red to stand out in the GUI and show to the users that this will clear/delete the text on the screen. This is styled very similar to my dark mode button, and is created very similarly. The line of code clearButton.addActionListener(e -> textArea.setText("")); is the line that gives the function to when the clear text button is clicked, the text is removed from the GUI. The implementation of this button is shown in figure 19.

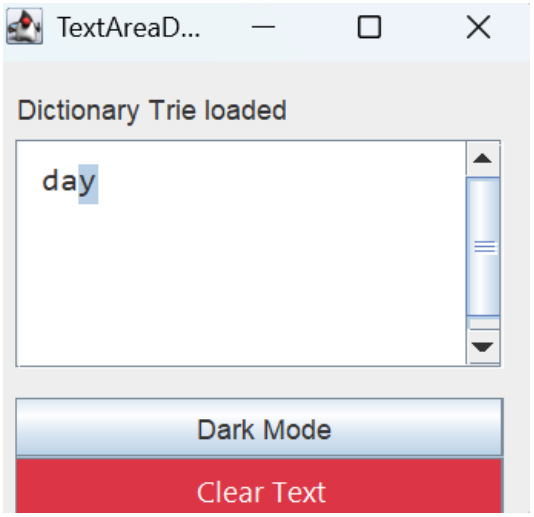

Figure 19: Shows the clear text button on the bottom of the figure