CS 2610 Darshan Shetty
Homework 1 Ankita Mohapatra

# Art of Virtual Keyboarding

## Introduction

The goal in this project is to create a Virtual Keyboard that allows the users to transcend the limitations imposed by a physical device and gives them features that would not be feasible to implement physically. The virtual keyboard has the ability to type by clicking on the keyboard and by tracing words on the keyboard. The keyboard will also provide suggestions based on the user input and give feedback to the user based on that input.

## Layout

The basic layout of the virtual keyboard includes an input display text box for the expected sentence at the top where the user can enter the Expected statement (using a physical keyboard) through the **Expected** button. It also has an output display text box that shows the default word chosen from the keys typed/traced by the user right below the expected sentence, so that the difference between the two is easily visible. In Tap mode, this is the exact sequence of characters the user entered, while in Trace mode, it is the best suggestion found for the user's swipe. (The exact string of characters they swiped over is unlikely to be a sensible word). In addition, there are four suggestion buttons that display the words the program thinks the user might be trying to enter. The suggestions are displayed in an order such that the most frequent word from the dictionary file will be saved to the leftmost suggestion button. There is also a **Clear** button which is essentially used to clear out the output display text box. Finally, there is the layout of the keyboard of the traditional QWERTY-style 27 key (26 alphabet keys and a space key) keyboard

## Design Considerations

The entire framework of the virtual keyboard was built in since it can run on most devices irrespective of the Operating System, and can develop easy to use and efficient applications in a much lesser amount of time through the extensive collection of Java library files that are present on the web.

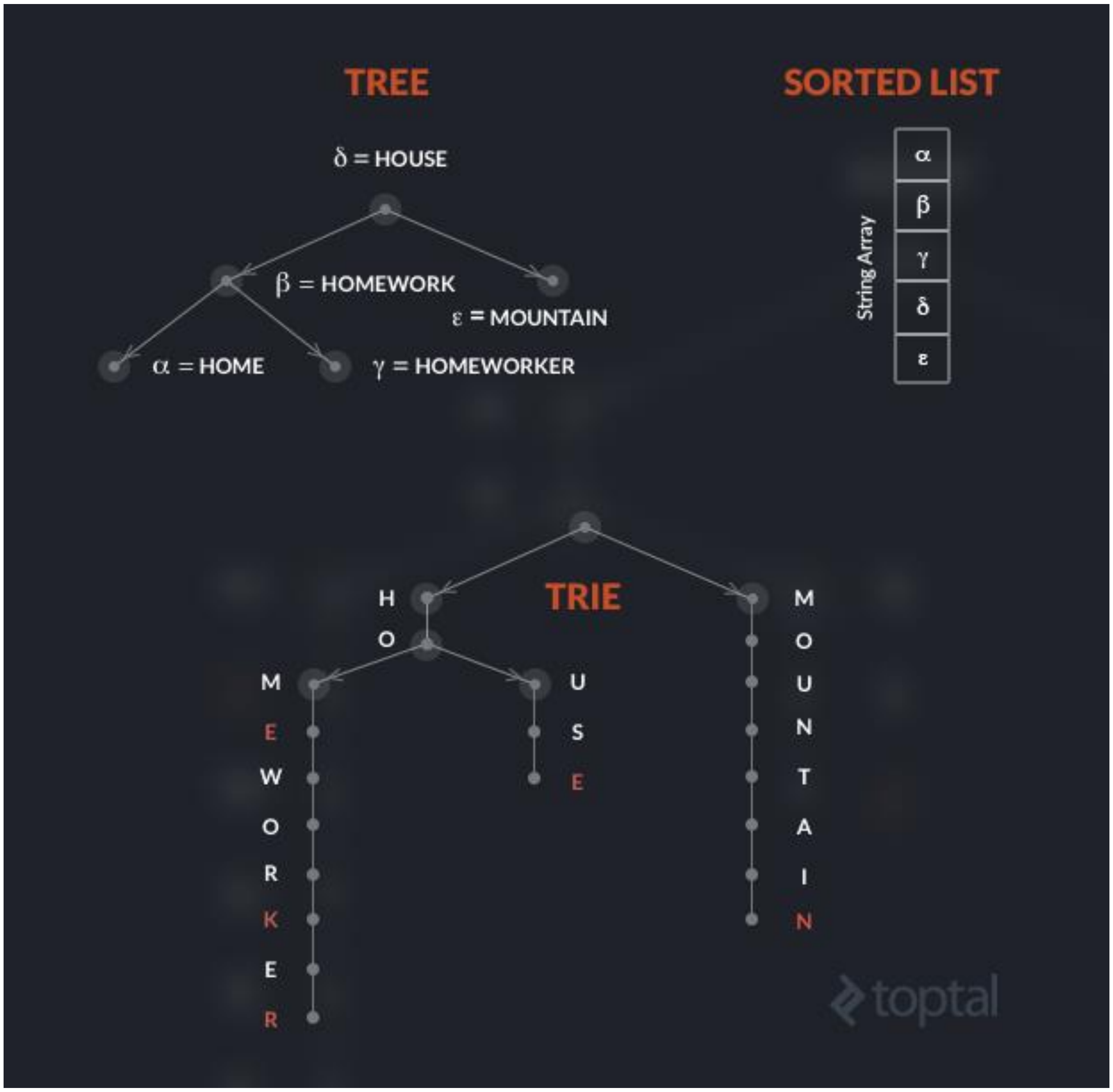
The virtual keyboard was designed keeping in mind the following design considerations:

***1.Improved affordance and feedback:***

To make the framework intuitive, we used substantial visual feedback, like lines that trace the path, keyboard buttons that light up when you go over them, keyboard buttons that change color when pressed or released. This feedback enables the user to create a firm conceptual model in their mind which enables the user to further use the application efficiently. Another consideration was to use custom made rectangles as keys instead of buttons since rectangles offered more affordance than a keyboard button since they could change color and texture as visual feedback. Also the while one clicked the button and dragged it, it would only look like one button was pressed down whereas for a rectangle, all the rectangles on the traced path would light up sequentially. Therefore, since rectangles provided us much better feedback we decided to use custom drawn rectangles.

***2.Shorthand Writing/Word Suggestions:***

For the two modes of input, we use slightly different suggestion-finding algorithms. However, both make use of the same data structure. When the user starts the application, it reads through the dictionary file to find each word and its frequency of use. It constructs a 2-dimensional 26-by-26 array of tries, where the trie in array index [i][j] contains all of the words that start with character i and end with character j (where character 0 is A, character 1 is B, etc.). We chose this structure because in both modes, we know the starting letter of the user's word, and in trace mode, we know the likely ending letter that the user wants. Thus, this structure narrows down the searching the program must do.



**Trace Mode:** The keyboard portion of the program feeds the search algorithm a sequence of (letter, necessity) pairs in the order that the user swipes over the letters. Necessity indicates how likely the program believes the letter is in the word the user wants. The first and last characters swiped have a high necessity, others have a low necessity, and duplicates of letters that are added to account for double-letter words have no necessity (so a word with double letters will not necessarily appear over a word with one of those double letters reduced to a single letter). The algorithm searches in two tries, both beginning with the letter the user pressed the mouse down on, but one ending with the letter on which the user released the mouse and one ending with the letter the user traveled over before that one. The algorithm finds all words that are subsequences of the sequence of letters, ordering them first by the total necessity of the matched letters and then by highest frequency. The five best words appear as suggestions.

**Tap Mode:** The algorithm moves down each trie node which starts with the first letter tapped, entering the nodes for each successive letter in order. It then recursively visits the rest of each of those tries' nodes below the node for the last letter the user tapped. It keeps track of the four words with the highest frequency values that it finds this way, and displays them as suggestions.

## *3.Handling of Duplicate Characters:*

Handling of words like "bee" and "be", i.e., words with duplicate letters, is done by requiring the user to leave the letter button and select any other in its vicinity and returning back. Only if the neighboring letter is traversed, the words with duplicate letters (bee) are displayed. In all other cases, "be" is displayed.

**Pros:** There is no ambiguity as to what the application will return. On selecting an arbitrary letter in the vicinity of the letter we want twice and returning back does not affect the output at all since the last character is constant. The application will make a button visible with the probable otherwise which can be selected on clicking.

**Cons:** This method involves extra tracing and the user has to be told about this (discoverability). In case the output is not obtained directly we still have the button appearing, but is it slower because it involves an extra click.

## *4. Ambiguity:*

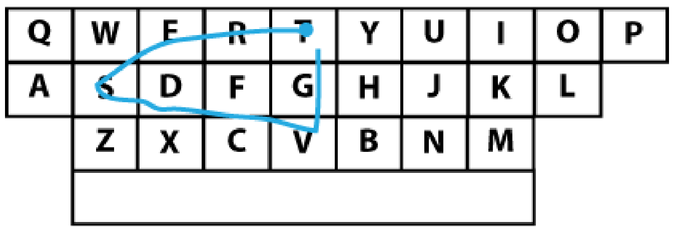
In the case that one input trace maps to multiple valid words, the word with the highest frequency is displayed since they correspond to the same starting and ending character. Based on the frequencies stored for every word in the Dictionary, the four most frequent words are displayed of which the highest frequent word is displayed.

**Pros:** Returns the word closest to what was intended on tracing, plus the next four most likely words.

**Cons:** May miss out on a low frequency word (values obtained from dictionary provided) since we have limit of 4. If the limit was not the problem then only selecting the word would require extra time, because it returns all other possible combinations.

## *5. Displaying a word other than the most frequent word:*

The trace below will look like the sequence "TRESDVGT". The application uses corners of a polygon method to decide some letters that the user must want since we went to an extreme to select it. The application returns the words that matches with all the corners. At the same time, it also returns the other words (limit 4) arranged by highest frequency first.



**Pros:** The user is shown the possible words with highest frequency first. He can select it if that is what he wants, else he will have to retrace. Does not display the traced characters, so no need of deleting to try again, saving time.

**Cons:** The user is not perfect, for speed he may lose accuracy and so prone to make errors which will not let word be displayed in the display text box.

*6. **Levenshtein Distance Measurement and Visualization:***

As the user enters a word, before the user has pressed the Space key or chosen a suggestion, the text for that word in the user text box is blue. After the user confirms a word (by pressing Space or clicking the "Current Word" or any of the Suggestion buttons), that word in the User and Goal text boxes changes colors to show the user's accuracy based on the Levenshtein Distance. Letters in the corresponding user word that have been omitted by the user when compared with the expected word turn yellow. Letters that would have to be inserted in the user word to produce the expected word appear purple in the user text box. If a letter in the expected word does not match the letter in the corresponding space in the user word, then that letter in the user word would turn red. Green letters indicate matched letters. In each case, the program decides which letters are considered omitted, extra, or mismatched based on which choice results in the lowest edit distance between the words, where each letter designated as "omitted", "extra", or "mismatched" counts as 1 unit of distance. The colors make it easy for users to see if they have any mistakes, and with a bit more effort, they can figure out what their mistakes were.

## Inspiration

The primary inspiration of our system came from the android third party application called "Swype" which works on the same principles and uses slightly different algorithms to produce similar results

## Citations

http://www.toptal.com/java/the-trie-a-neglected-data-structure

http://www.krishnabharadwaj.info/how-swype-works/

http://www.swype.com/