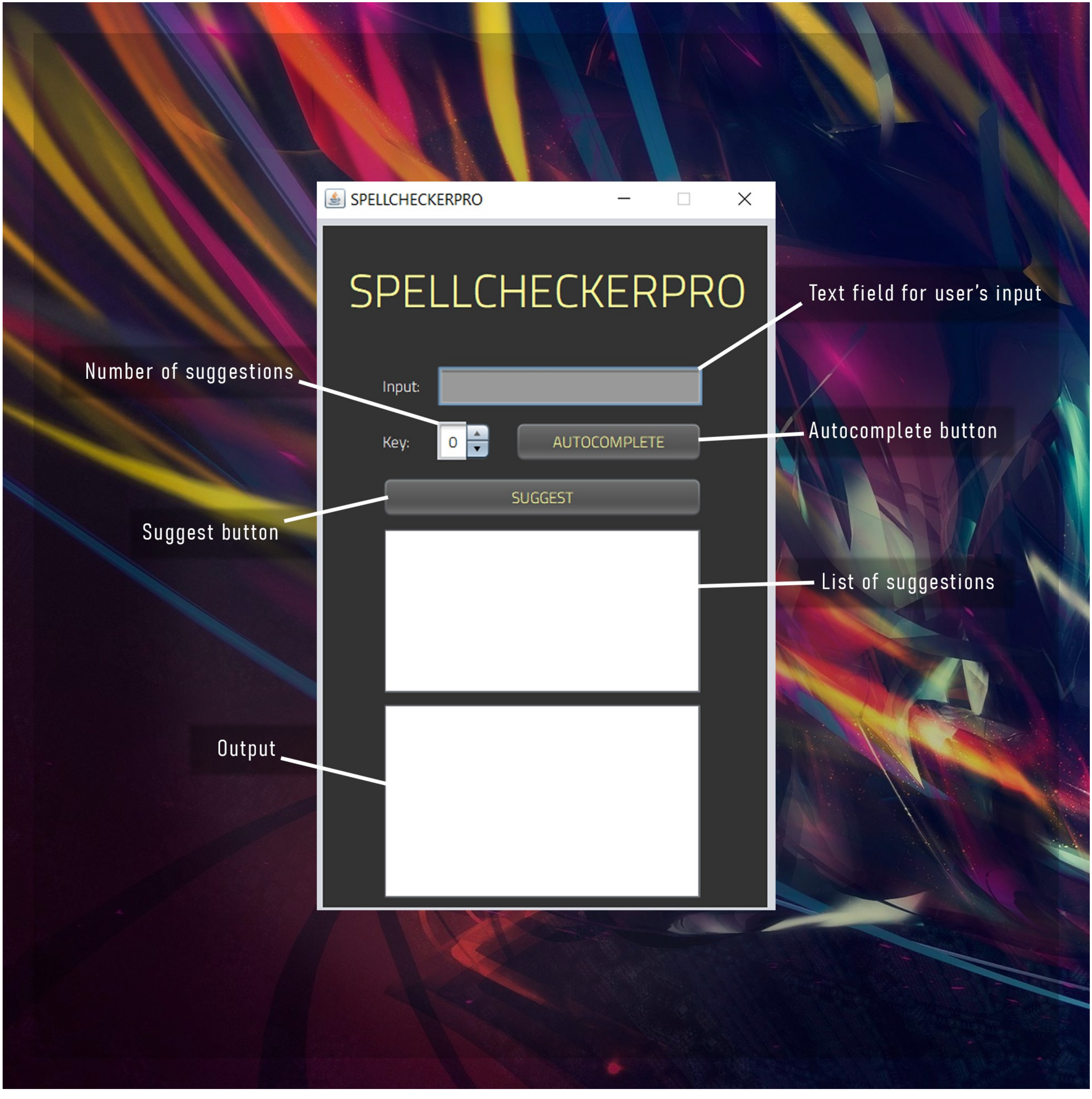
BLM19202E Data Structures Programming Assignment #2
Spell Checking and Autocomplete with Binary Search Tree

AHMED MUAZ ATİK – 2121221002

OBADA MASRİ – 2121221405


DESIGN


SPELLCHECKERPRO
SPELLCHECKERPRO
Input:
Key:
0
AUTOCOMPLETE
SUGGEST
Text field for user's input
Number of suggestions
Autocomplete button
Suggest button
List of suggestions
Output

PSEUDOCODES

LEVENSHTEINDISTANCE()

Initialize two integers, first_len and second_len, to the length of the two input strings respectively.

Create a two-dimensional array of size (first_len + 1) x (second_len + 1) and name it matrix.

Initialize the first column of the matrix to the integers from 0 to first_len.

Initialize the first row of the matrix to the integers from 0 to second_len.

Iterate from i = 1 to first_len and j = 1 to second_len:

a. If the i-th character of first is equal to the j-th character of second, set matrix[i][j] to matrix[i-1][j-1].

b. Otherwise, set matrix[i][j] to 1 plus the minimum of matrix[i-1][j-1], matrix[i][j-1], and matrix[i-1][j].

Return the value at matrix[first_len][second_len] as the Levenshtein distance between the two input strings.

CLOSESTMATCH()

```
function getClosestMatch(word):
    vary = root // start at the root of the binary search tree
    distance = MAX_VALUE // initialize distance to maximum value
    closest = "" // initialize closest to an empty string

    while vary is not null:
        // calculate the Levenshtein distance between the current node's word and the given word
        d = LevenshteinDistance(vary.word, word)

        if d < distance:
            distance = d
            closest = vary.word

        if d is 0:
            break
        else if word is less than vary.word in lexicographical order:
            vary = vary.leftChild // move to the left subtree
        else:
            vary = vary.rightChild // move to the right subtree

    return closest // return the closest matching word
```

IMPLEMENTATION DETAILS

1

```
int first_len = first.length();
int second_len = second.length();

int[][] matrix = new int[first_len + 1][second_len + 1];

for (int i = 0; i <= first_len; i++) {
    matrix[i][0] = i;
}
for (int j = 0; j <= second_len; j++) {
    matrix[0][j] = j;
}

for (int i = 1; i <= first_len; i++) {
    for (int j = 1; j <= second_len; j++) {
        if (first.charAt(i - 1) == second.charAt(j - 1)) {
            matrix[i][j] = matrix[i - 1][j - 1];
        } else {
            matrix[i][j] = 1 +
                    Math.min(matrix[i - 1][j - 1], Math.min(matrix[i][j - 1], matrix[i - 1][j]));
        }
    }
}

return matrix[first_len][second_len];
```

This is an implementation of the Levenshtein distance algorithm, which is used to calculate the minimum number of operations (insertion, deletion, substitution) needed to transform one string into another.

The algorithm works by creating a matrix where each cell (i,j) represents the Levenshtein distance between the substrings of the first string up to the i-th character and the second string up to the j-th character.
The algorithm fills in the matrix by iterating through the strings and calculating the Levenshtein distance for each pair of characters.

The first and second for loops initialize the first row and column of the matrix to the values of the index, because the distance between a string of length n and an empty string is n.

The inner for loops iterate over the remaining cells of the matrix and compute the Levenshtein distance for each cell based on the values of the neighboring cells in the matrix. If the characters at the current position are the same, the distance remains the same as the diagonal value. If they are different, the minimum value between the three neighboring cells plus one is selected as the new distance.

Finally, the Levenshtein distance between the entire first and second strings is returned, which is the value in the last cell of the matrix.

2

```
try {
    File file = new File("C:\Users\ahmed\Desktop\SpellCheckPro\words.txt");
    Scanner myReader = new Scanner(file);
    while (myReader.hasNextLine()) {
       String data = myReader.nextLine();

       String temp = "";

       for (int i = 0; i < data.length(); i++) {

          if (data.charAt(i) != ',') {
             temp += data.charAt(i);
          } else {
             bt.insert(temp);
             temp = "";
          }
       }

    }
    myReader.close();
} catch (FileNotFoundException e) {
    JOptionPane.showMessageDialog(rootPane, "Error");
    e.printStackTrace();
}
```

This code block reads from a file named "words.txt" and inserts each word into a binary tree data structure.

The first line creates a new File object with the path "C:\Users\ahmed\Desktop\SpellCheckPro\words.txt".

The try block uses a Scanner object to read from the file line by line. For each line, it removes commas and inserts the remaining substring into the binary tree using the insert method of the binary tree object named bt.

The catch block catches a FileNotFoundException and displays an error message using JOptionPane.showMessageDialog() method. It also prints the stack trace for the caught exception.

This code can be used for a spell checker program that stores a list of valid words in a binary tree data structure and checks the spelling of words against this list.

3

```
if (tf_input.getText().isEmpty()) {
        JOptionPane.showMessageDialog(rootPane, "Don't leave input area empty!");
    }

    Color red_color = Color.RED;

    String first_sub = "";
    String second_sub = "";
    String closest = "";
    String output = "";

    for (int i = 0; i < tf_input.getSelectionStart(); i++) {
        first_sub += tf_input.getText().charAt(i);
    }

    for (int i = tf_input.getSelectionEnd(); i < tf_input.getText().length(); i++) {
        second_sub += tf_input.getText().charAt(i);
    }

    closest = bt.getClosestMatch(tf_input.getSelectedText());

    output = first_sub + closest + " " + second_sub;

    tf_input.setText(output);
    ta_output.setText(output);

    highlight(ta_output, closest);
```

This code block handles the spell checking functionality of a program.

The first if statement checks if the input area (tf_input) is empty and displays an error message using the JOptionPane.showMessageDialog() method if it is.

The next few lines define some variables to store substrings and the closest match found in the binary tree.

The two for loops iterate over the characters before and after the selected text in tf_input and store them in first_sub and second_sub, respectively.

The closest variable is assigned the result of calling the getClosestMatch() method of the binary tree object bt, passing in the selected text from tf_input as a parameter. This method returns the closest matching word to the input string found in the binary tree.

The output string is constructed by concatenating first_sub, closest, and second_sub with spaces between them.

The text in input area and output area is set to the output string using the setText() method.

Finally, the highlight() method is called with ta_output and closest as parameters.
This method highlights the closest match in the output text area by changing its color to red.

4

```
public class GUI<T> extends javax.swing.JFrame {

    String new_string = "";

    boolean if_pressed = false;
```

GLOBAL VARIABLES

```
char last_letter = tf_input.getText().charAt(tf_input.getText().length() - 1);

if (tf_input.getText().length() != 0) {
    new_string = tf_input.getText().substring(0, tf_input.getText().length() - 1);
}

if (evt.getKeyChar() == 46) {
    if_pressed = true;
}

if (evt.getKeyChar() >= 97 && evt.getKeyChar() <= 122 && if_pressed == true) {
    new_string += (char) (last_letter - 32);
    tf_input.setText(new_string);
    if_pressed = false;
}
```

This code block handles capitalization of the last letter in the input string if the period (.) key is pressed followed by a letter key.

The first line initializes the variable last_letter with the last character in the input string.

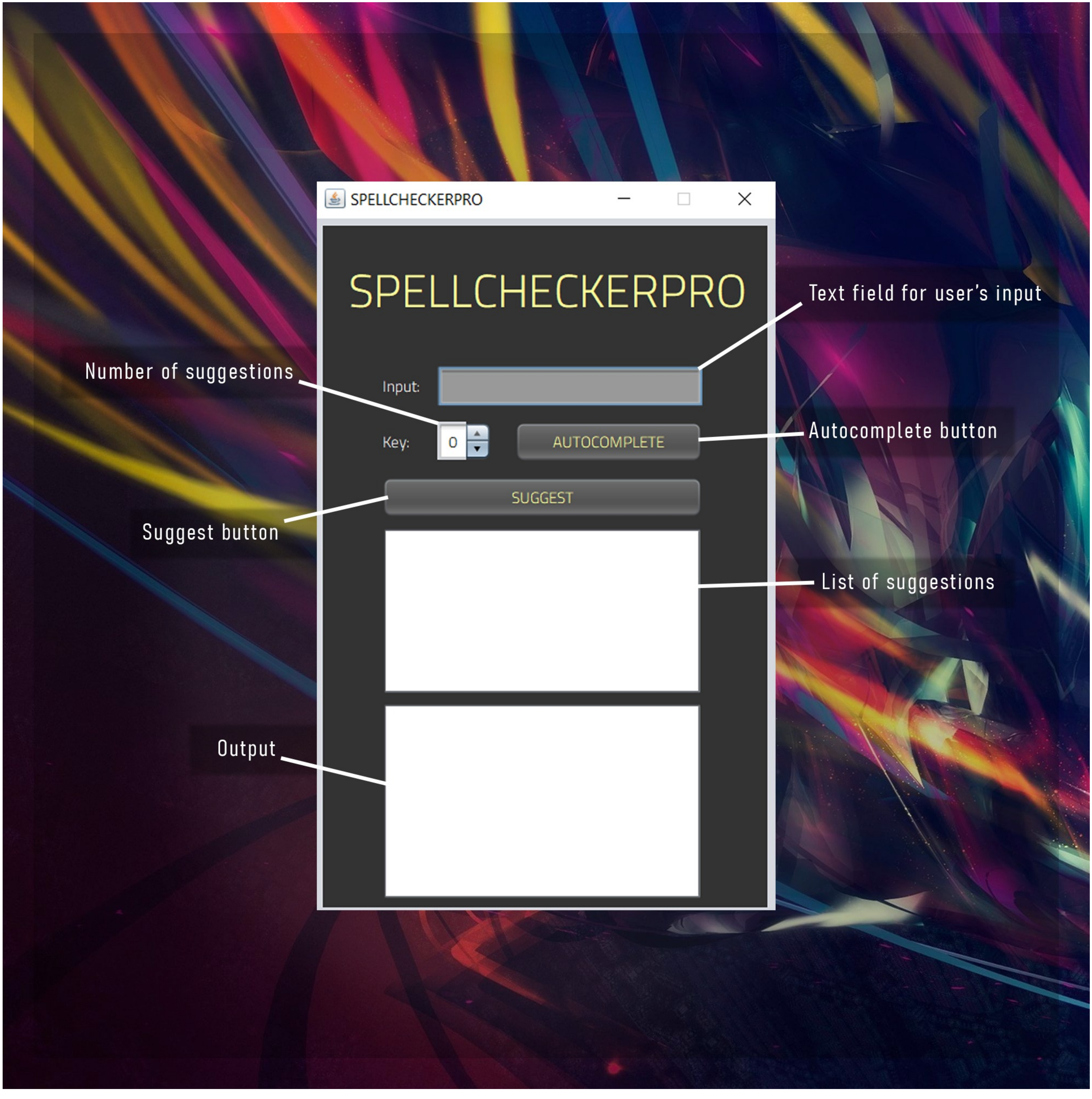
The next if statement checks if the length of the input string is not zero. If it is not zero, it creates a new string (new_string) by removing the last character of the input string using the substring() method.

The next if statement checks if the key pressed is the period (.) key (ASCII code 46) and sets the universal boolean variable if_pressed to true.

The final if statement checks if the key pressed is a letter key (ASCII codes 97-122) and if the boolean variable if_pressed is true. If both conditions are true, it adds the capitalized last letter to the new_string by subtracting 32 from its ASCII code (to convert it from lowercase to uppercase). It then sets the text of tf_input to the new_string and sets if_pressed to false.

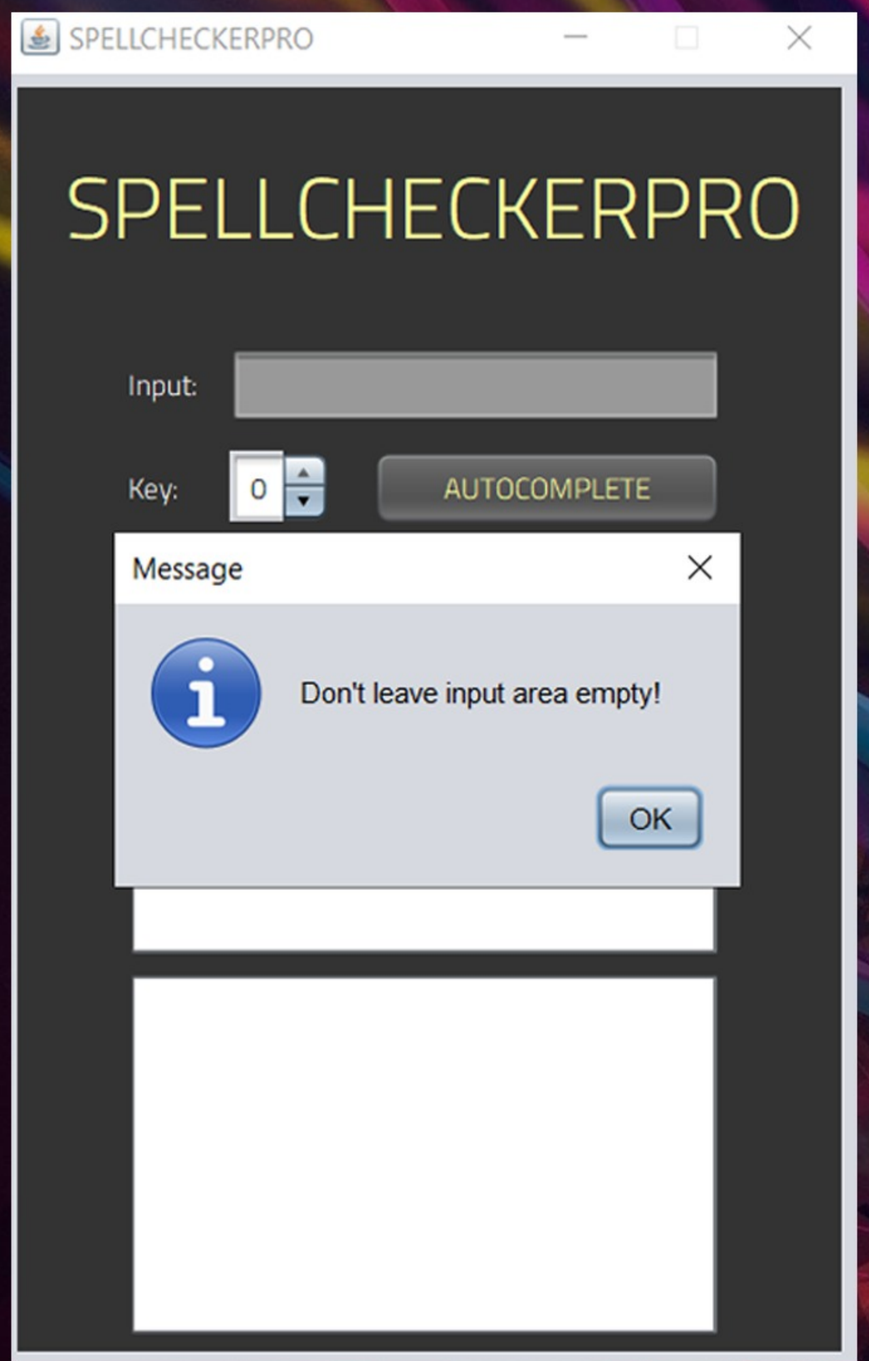
This code can be used to allow users to capitalize the last letter in the input string by pressing the period key followed by a letter key.

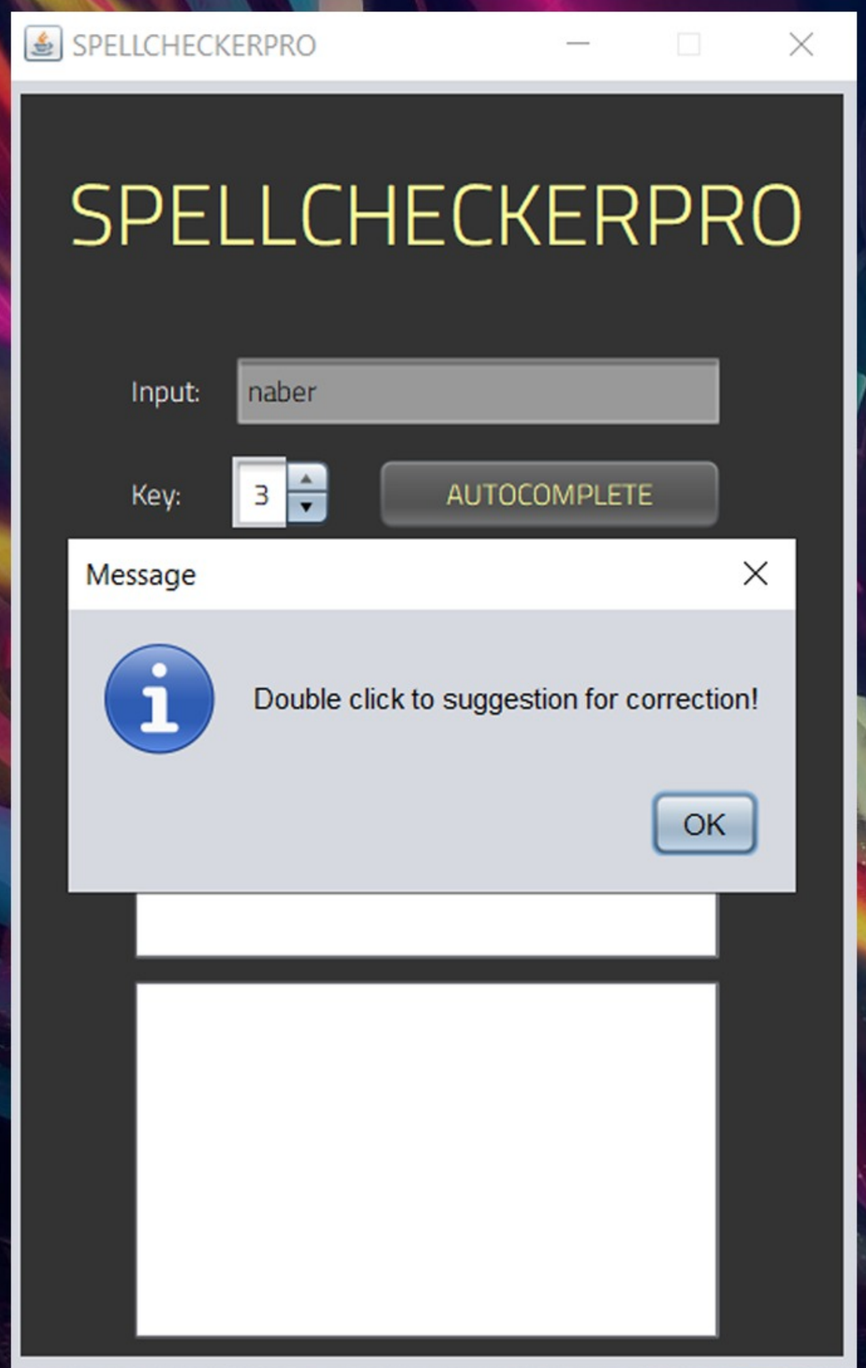
5


SPELLCHECKERPRO
SPELLCHECKERPRO
Input:
Key:
0
AUTOCOMPLETE
SUGGEST
Text field for user's input
Number of suggestions
Autocomplete button
Suggest button
List of suggestions
Output

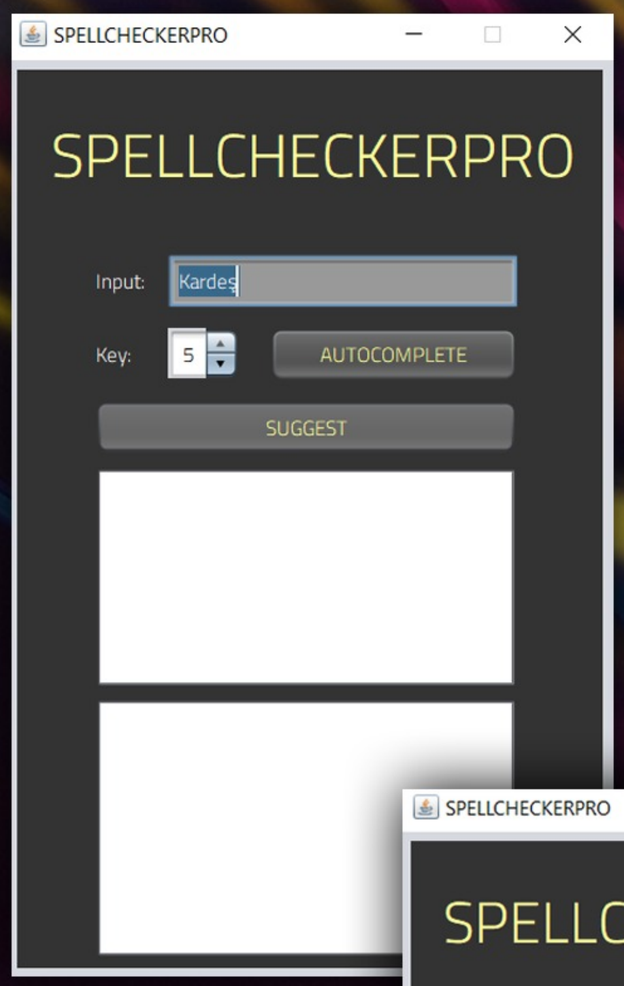
OUTPUT

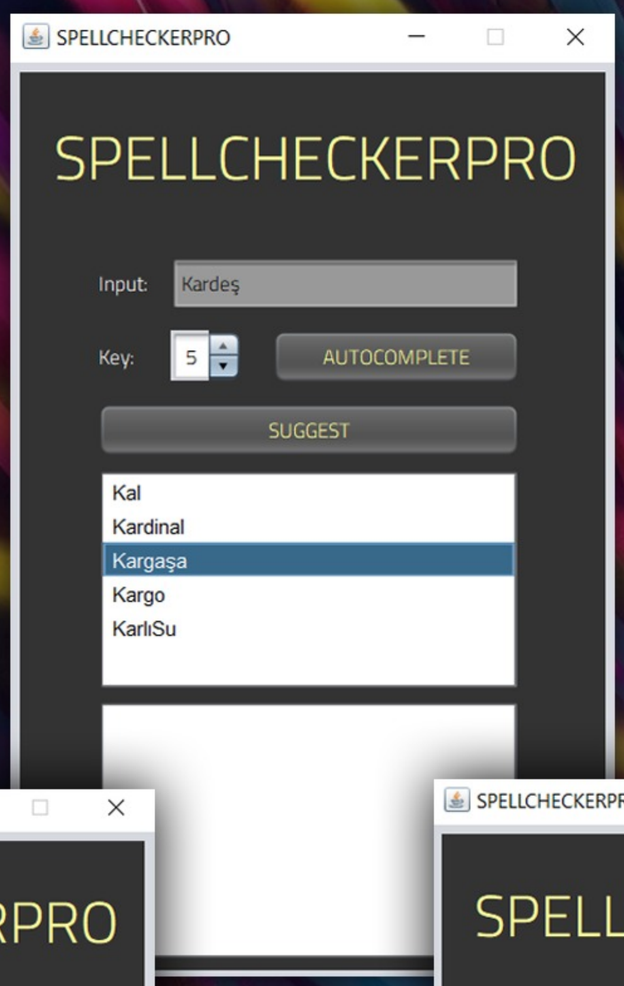
WARNINGS

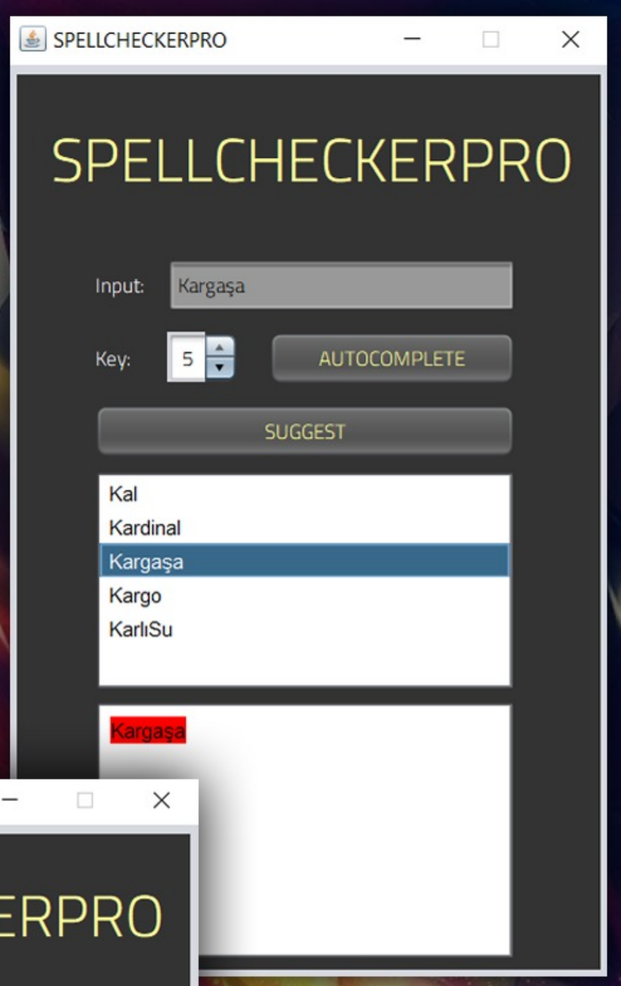

SPELLCHECKERPRO
SPELLCHECKERPRO
Input:
Key:
0
AUTOCOMPLETE
Message
Don't leave input area empty!
OK

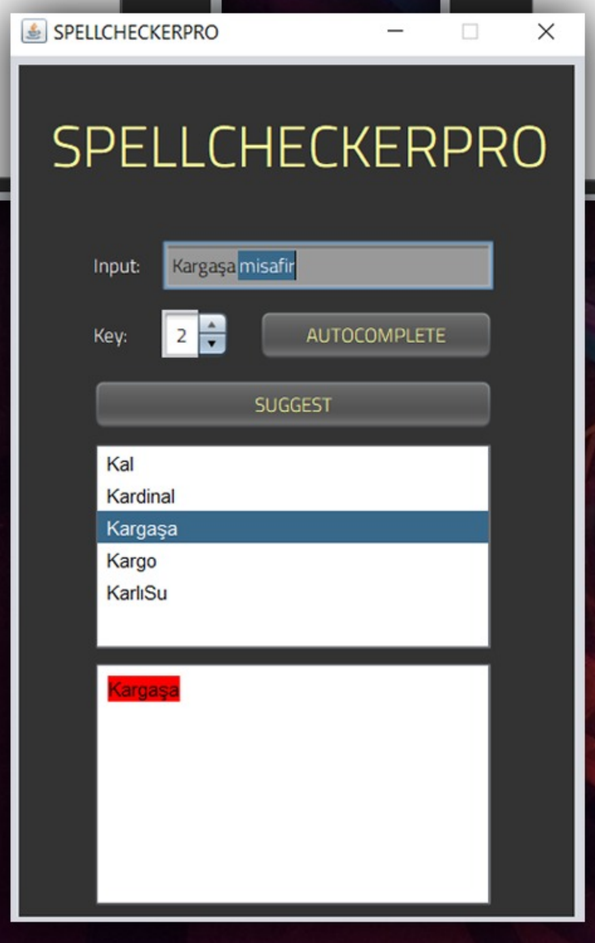

SPELLCHECKERPRO
SPELLCHECKERPRO
Input:
naber
Key:
3
AUTOCOMPLETE
Message
Double click to suggestion for correction!
OK

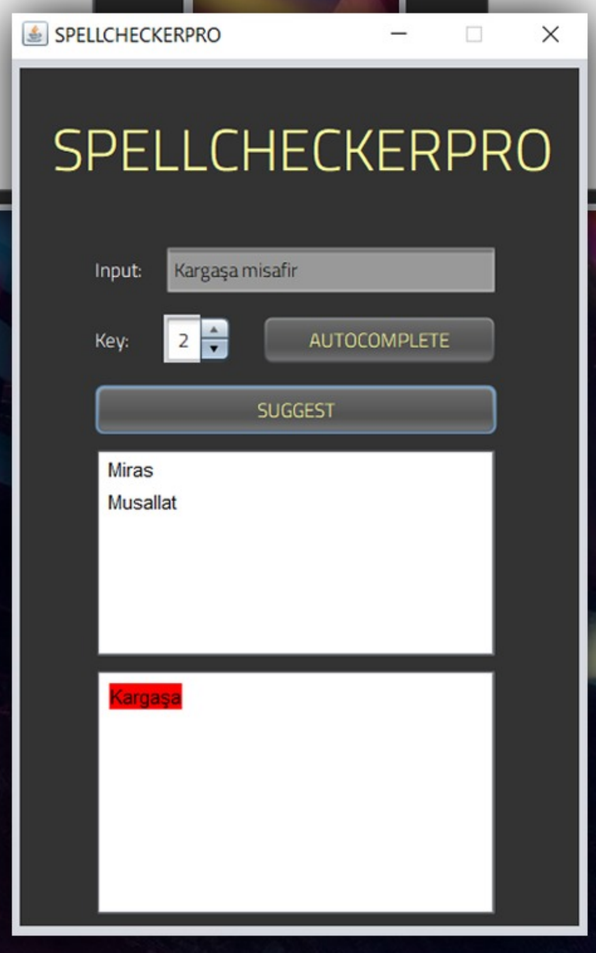
SUGGESTIONS











AUTOCOMPLETE