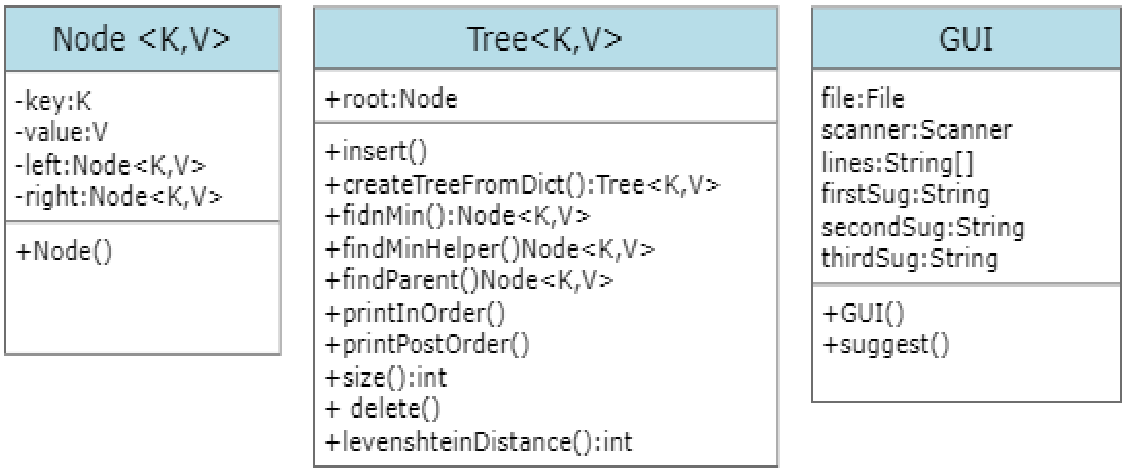
01/05/2023

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

HAMZA ALAHMEDİ– 2021221358 / RAMİ EYÜPOĞLU–1921221355

1) Design



Pseudo-code and algorthim steps:

1. Get the word in the jTextfield.
2. Store the word from the file.
3. Compare every word from the file with the word in the jTextfield using levenshtein distance algorithm.
4. Create a binary search tree with a multi values contain the word and the the result of the levenshtien distance algorthim.
5. Using the findMin method find the tree nodes with the minimum key.
6. Add the min three nodes to a new tree.
7. Get the root of the new tree and suggest it for the auto completion
8. Get all the noes in the min tree and suggest them in a jlabel.
9. Do that ever time the word changes in the jTextfield.

2) Implementation Details:

1. Correct implementation of a binary search tree to store a dictionary of words.

Methods used in this Step:

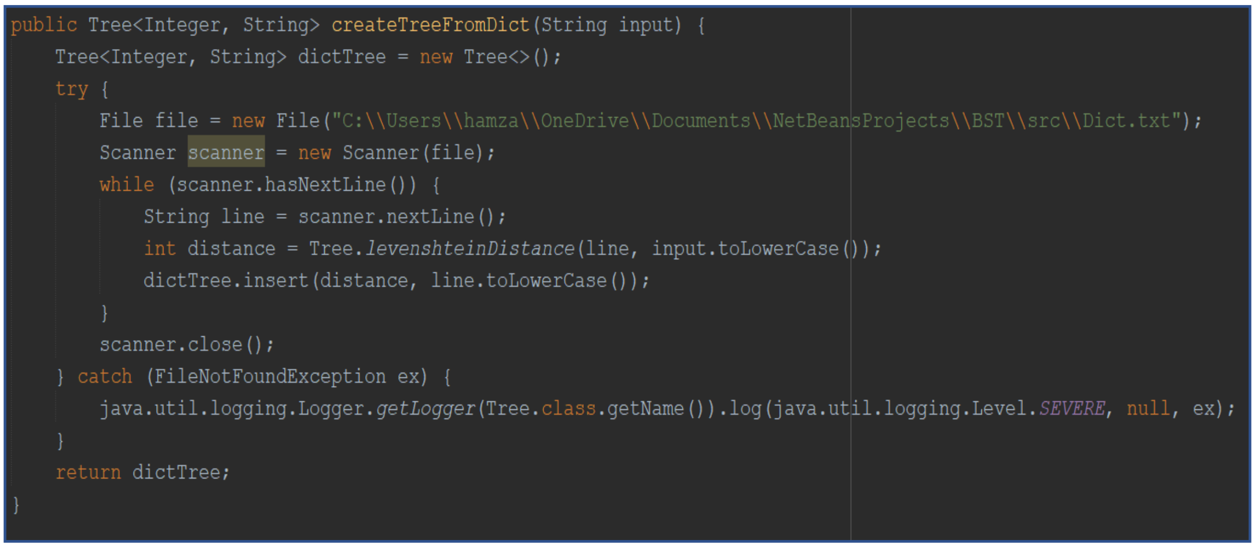
1. Insert method.
2. LevenshteinDistance.
3. createTreeFromDict.

Implementation:

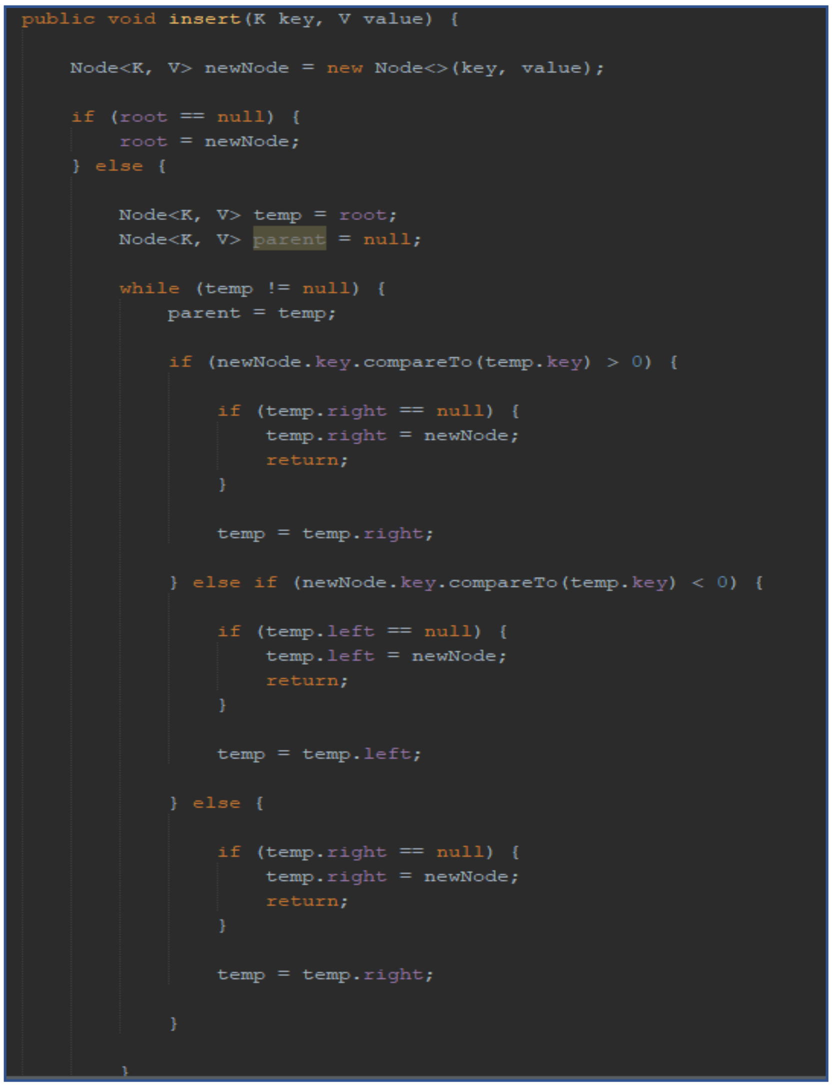
Binary search tree (BST) is implemented to store a dictionary of words using the `insert` method. The `insert` method compares the key of the new node to the key of the root of the tree, then completes the comparison in the right subtree if the key is bigger and in the left subtree if the key is smaller.

To suggest corrections for misspelled words, the Levenshtein algorithm is used. This algorithm takes two strings and returns an integer value that counts the minimum number of operations required to transform one string into the other.

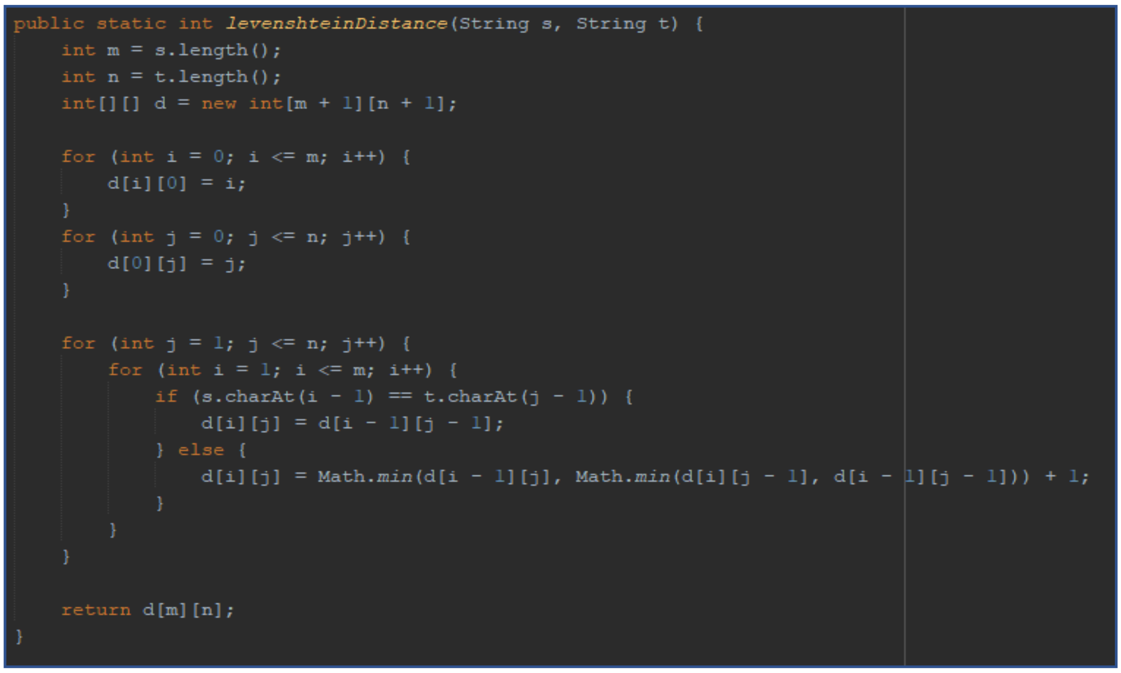
The `createTreeFromDict` function creates a new BST by storing words from a file and calling the Levenshtein algorithm to compare each word in the file with the word in the text field. The `insert` function is called to insert a new node into the tree, which contains the result of the Levenshtein distance as the key and the word itself as the value.



```java
public Tree<Integer, String> createTreeFromDict(String input) {
    Tree<Integer, String> dictTree = new Tree<>();
    try {
        File file = new File("C:\\Users\\hamza\\OneDrive\\Documents\\NetBeansProjects\\BST\\src\\Dict.txt");
        Scanner scanner = new Scanner(file);
        while (scanner.hasNextLine()) {
            String line = scanner.nextLine();
            int distance = Tree.levenshteinDistance(line, input.toLowerCase());
            dictTree.insert(distance, line.toLowerCase());
        }
        scanner.close();
    } catch (FileNotFoundException ex) {
        java.util.logging.Logger.getLogger(Tree.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
    }
    return dictTree;
}
```

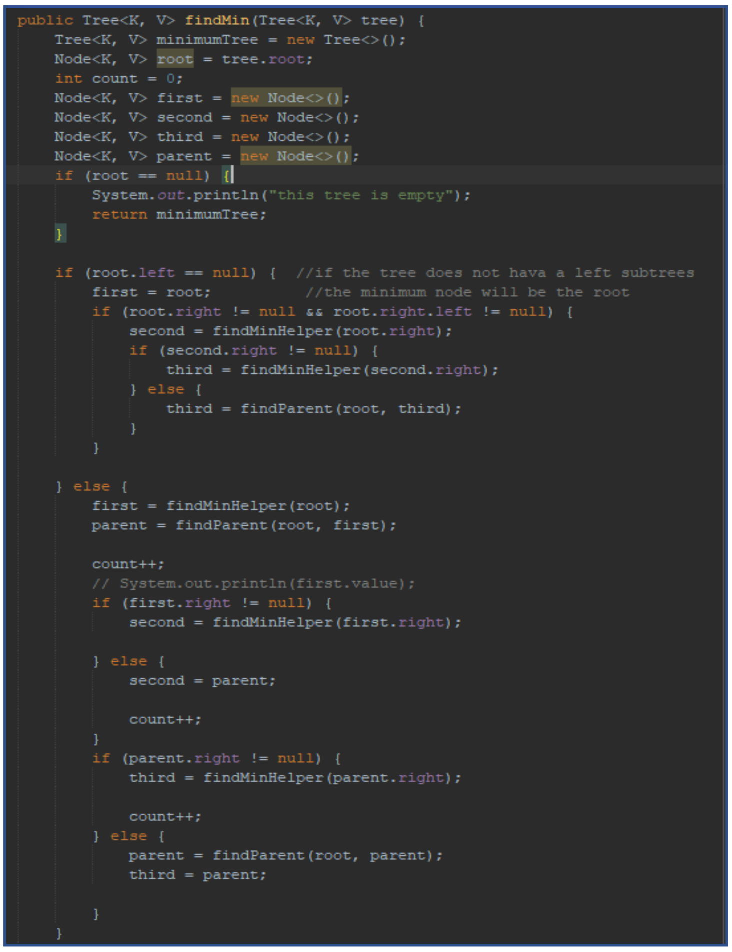


```java
public void insert(K key, V value) {

    Node<K, V> newNode = new Node<>(key, value);

    if (root == null) {
        root = newNode;
    } else {

        Node<K, V> temp = root;
        Node<K, V> parent = null;

        while (temp != null) {
            parent = temp;

            if (newNode.key.compareTo(temp.key) > 0) {

                if (temp.right == null) {
                    temp.right = newNode;
                    return;
                }

                temp = temp.right;

            } else if (newNode.key.compareTo(temp.key) < 0) {

                if (temp.left == null) {
                    temp.left = newNode;
                    return;
                }

                temp = temp.left;

            } else {

                if (temp.right == null) {
                    temp.right = newNode;
                    return;
                }

                temp = temp.right;

            }

        }
```

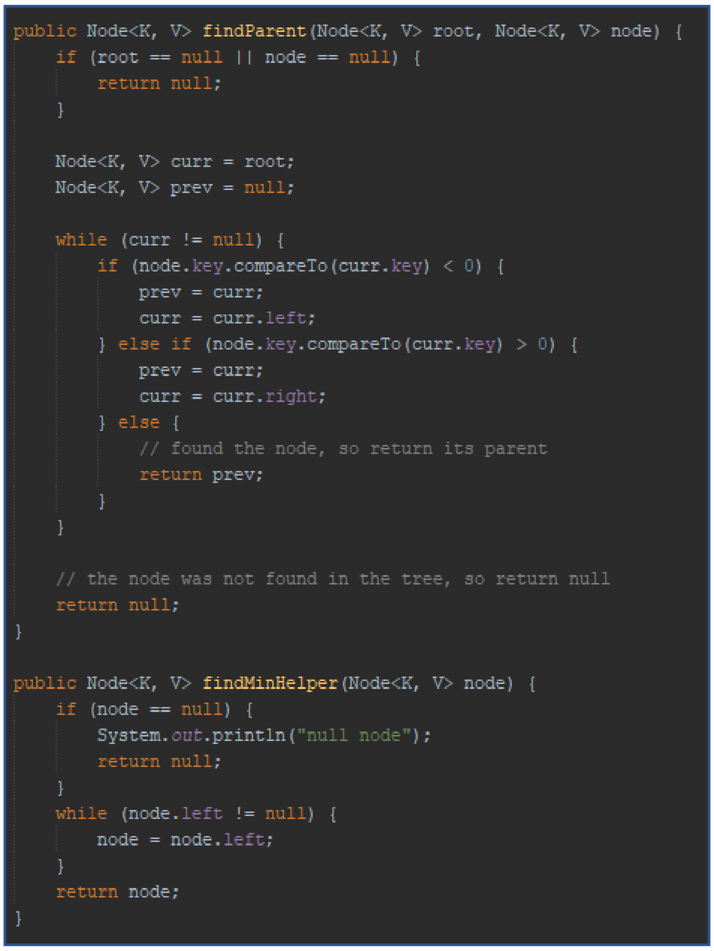


```java
public static int levenshteinDistance(String s, String t) {
    int m = s.length();
    int n = t.length();
    int[][] d = new int[m + 1][n + 1];

    for (int i = 0; i <= m; i++) {
        d[i][0] = i;
    }
    for (int j = 0; j <= n; j++) {
        d[0][j] = j;
    }

    for (int j = 1; j <= n; j++) {
        for (int i = 1; i <= m; i++) {
            if (s.charAt(i - 1) == t.charAt(j - 1)) {
                d[i][j] = d[i - 1][j - 1];
            } else {
                d[i][j] = Math.min(d[i - 1][j], Math.min(d[i][j - 1], d[i - 1][j - 1])) + 1;
            }
        }
    }

    return d[m][n];
}
```

2. **Correct implementation of a Levenshtein distance algorithm to suggest corrections for misspelled words.**
3. **Correct implementation of auto-completion**
   Method used in this step: findMin, findMinHelper , findparent and suggest.



```java
public Tree<K, V> findMin(Tree<K, V> tree) {
    Tree<K, V> minimumTree = new Tree<>();
    Node<K, V> root = tree.root;
    int count = 0;
    Node<K, V> first = new Node<>();
    Node<K, V> second = new Node<>();
    Node<K, V> third = new Node<>();
    Node<K, V> parent = new Node<>();
    if (root == null) {
        System.out.println("this tree is empty");
        return minimumTree;
    }

    if (root.left == null) {  //if the tree does not hava a left subtrees
        first = root;           //the minimum node will be the root
        if (root.right != null && root.right.left != null) {
            second = findMinHelper(root.right);
            if (second.right != null) {
                third = findMinHelper(second.right);
            } else {
                third = findParent(root, third);
            }
        }

    } else {
        first = findMinHelper(root);
        parent = findParent(root, first);

        count++;
        // System.out.println(first.value);
        if (first.right != null) {
            second = findMinHelper(first.right);

        } else {
            second = parent;

            count++;
        }
        if (parent.right != null) {
            third = findMinHelper(parent.right);

            count++;
        } else {
            parent = findParent(root, parent);
            third = parent;

        }
    }
```



```java
public Node<K, V> findParent(Node<K, V> root, Node<K, V> node) {
    if (root == null || node == null) {
        return null;
    }

    Node<K, V> curr = root;
    Node<K, V> prev = null;

    while (curr != null) {
        if (node.key.compareTo(curr.key) < 0) {
            prev = curr;
            curr = curr.left;
        } else if (node.key.compareTo(curr.key) > 0) {
            prev = curr;
            curr = curr.right;
        } else {
            // found the node, so return its parent
            return prev;
        }
    }

    // the node was not found in the tree, so return null
    return null;
}

public Node<K, V> findMinHelper(Node<K, V> node) {
    if (node == null) {
        System.out.println("null node");
        return null;
    }
    while (node.left != null) {
        node = node.left;
    }
    return node;
}
```

Implementation:

1. After creating the tree with levenshtein distances, we need to find the closest word to the input word for auto-completion and suggest alternative words.

2. The node with the minimum key has the nearest word to the input word. We can find this node by calling the `findMin` method which internally calls the `findParent` and `findMinHelper` methods.

3. The node with the minimum key is the leftmost node in the tree, so we insert it into the new tree. Then, we check if it has a right child.

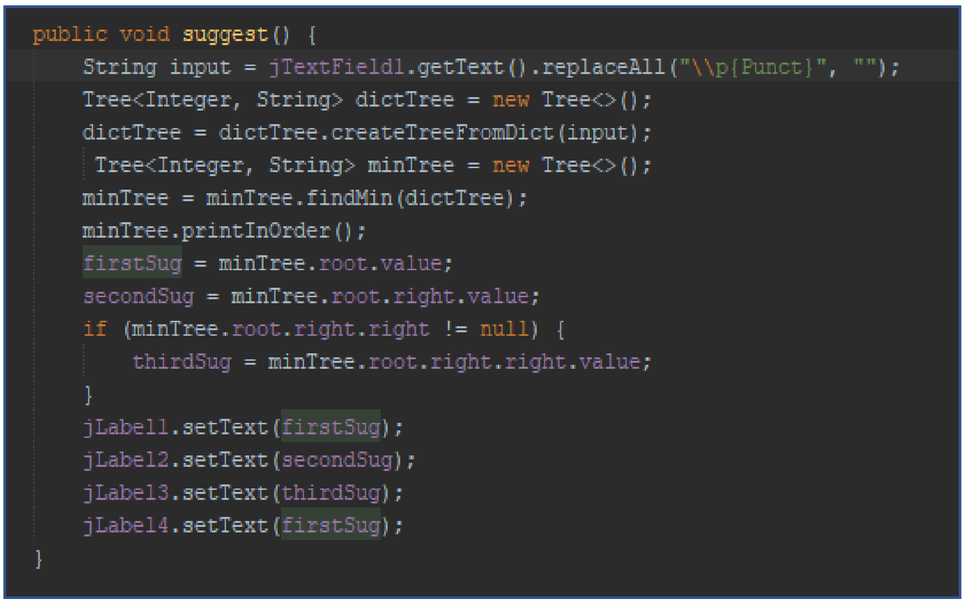
4. If it has a right child, we call the helper method to find the most left node in its right subtree and insert it into the new tree.

5. If it does not have a right child, we add the parent to the new tree by using the `findParent` method.

6. Then, we check if the parent has a right subtree. If it does, we use the helper method to add the most left node in the parent's right subtree to the new tree.

7. If the parent does not have a right subtree, we use the `findParent` method again and add the parent of the parent to the new tree.

8. This way, we have the three closest words to the input word in the new tree, which we can use for auto-completion and suggesting alternative words.

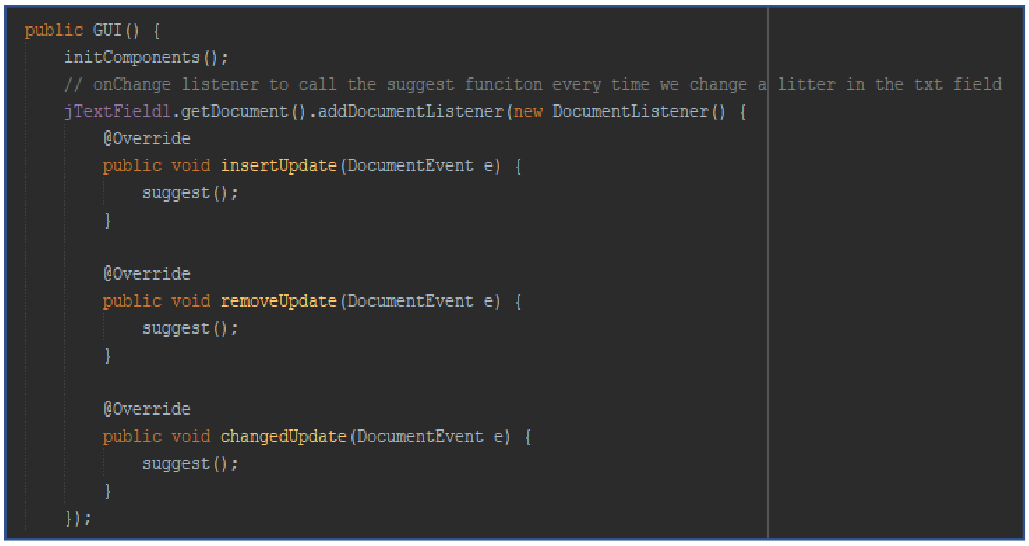


```java
public void suggest() {
    String input = jTextField1.getText().replaceAll("\\p{Punct}", "");
    Tree<Integer, String> dictTree = new Tree<>();
    dictTree = dictTree.createTreeFromDict(input);
     Tree<Integer, String> minTree = new Tree<>();
    minTree = minTree.findMin(dictTree);
    minTree.printInOrder();
    firstSug = minTree.root.value;
    secondSug = minTree.root.right.value;
    if (minTree.root.right.right != null) {
        thirdSug = minTree.root.right.right.value;
    }
    jLabel1.setText(firstSug);
    jLabel2.setText(secondSug);
    jLabel3.setText(thirdSug);
    jLabel4.setText(firstSug);
}
```

In the GUI class, the suggest method is implemented to provide suggestions for misspelled words. It first creates a new binary search tree using the createTreeFromDict method, which stores words from the file and their corresponding Levenshtein distance values as key-value pairs in nodes of the tree. Then, it calls the findMin method on this tree to find the three nodes with the smallest keys, which correspond to the three nearest words to the misspelled word. These three nodes are inserted into a minTree, which has the smallest key node as the root node. The value of the root node in the minTree is used for auto completion, while the values of all three nodes are used for suggestion and displayed in a label below the JTextField. The value of the root node is also displayed in a separate label below the auto completion button.

4. On change event listener:

The `GUI()` constructor initializes the components of the GUI and sets up a document listener for the `jTextField1` component. The document listener listens for changes made to the text in the `jTextField1` component, and calls the `suggest()` function whenever a change is detected. This ensures that the suggestions are updated dynamically as the user types in the text field. The `insertUpdate()`, `removeUpdate()`, and `changedUpdate()` methods are called by the document listener depending on the type of change made to the text field.



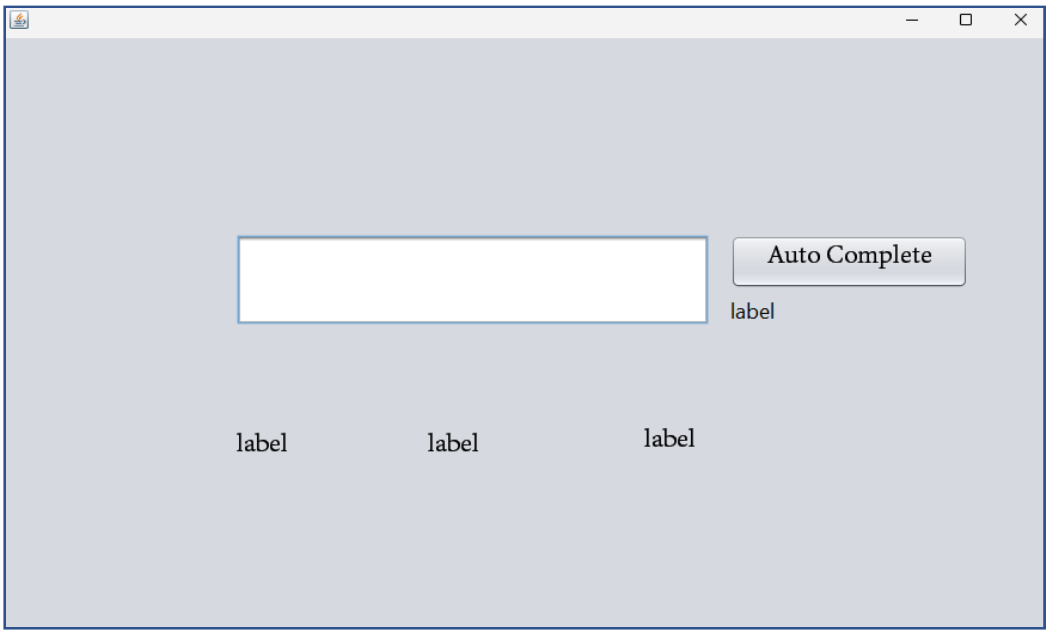
```
public GUI() {
    initComponents();
    // onChange listener to call the suggest funciton every time we change a litter in the txt field
    jTextField1.getDocument().addDocumentListener(new DocumentListener() {
        @Override
        public void insertUpdate(DocumentEvent e) {
            suggest();
        }

        @Override
        public void removeUpdate(DocumentEvent e) {
            suggest();
        }

        @Override
        public void changedUpdate(DocumentEvent e) {
            suggest();
        }
    });
```

5. Proper handling of capitalization and punctuation in input text.

To handle capitalization in the input text, the String.toLowerCase() method was used to convert both the input text and the words from the dictionary to lowercase. This ensures that capitalization differences are ignored during searching and prevents any issues.

To handle punctuation in the input text, by using input.replaceAll("\\p{Punct}", "") method all punctuation marks are removed using regular expressions before calling the Levenshtein distance algorithm. This allows the algorithm to accurately compare words without being affected by punctuation marks.

6. GUI implementation

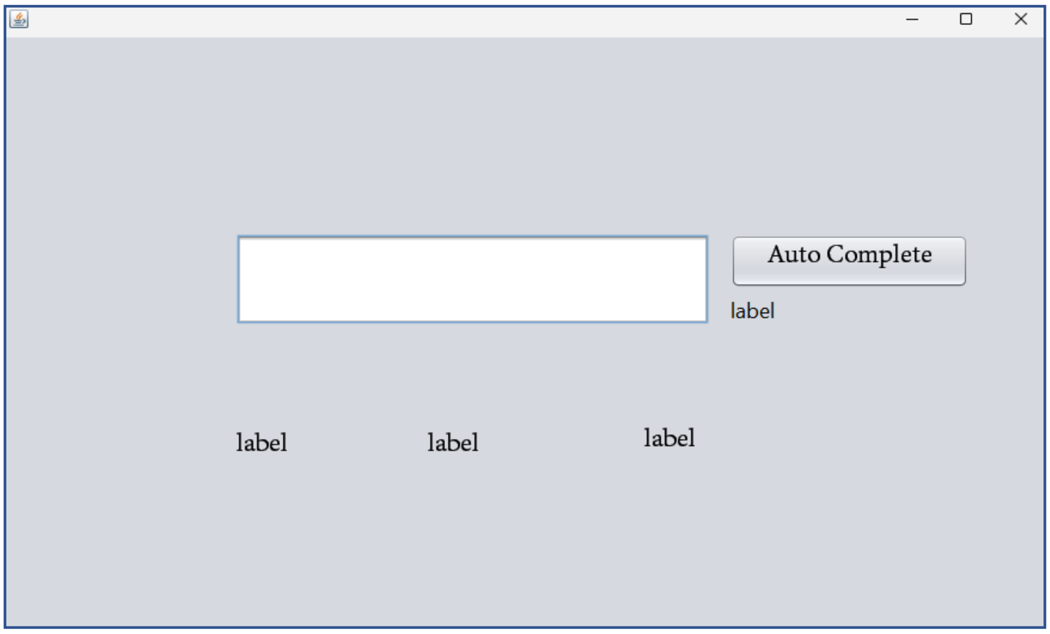


The GUI consists of a jTextField for user input and an auto complete button that sets the nearest word to the input word (displayed in a label under the auto complete button) and displays three suggestions in three labels under the textfield.

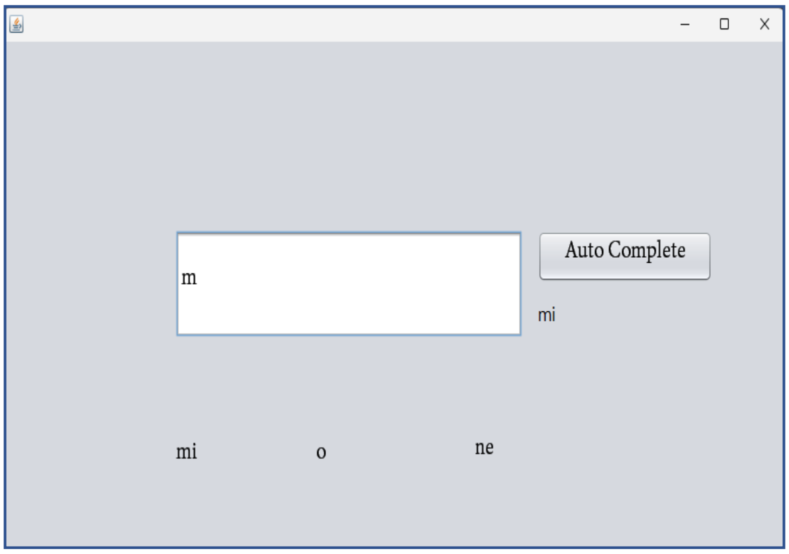
The suggest method is automatically called whenever any change is detected in the textfield, eliminating the need for a suggest button.

Output:

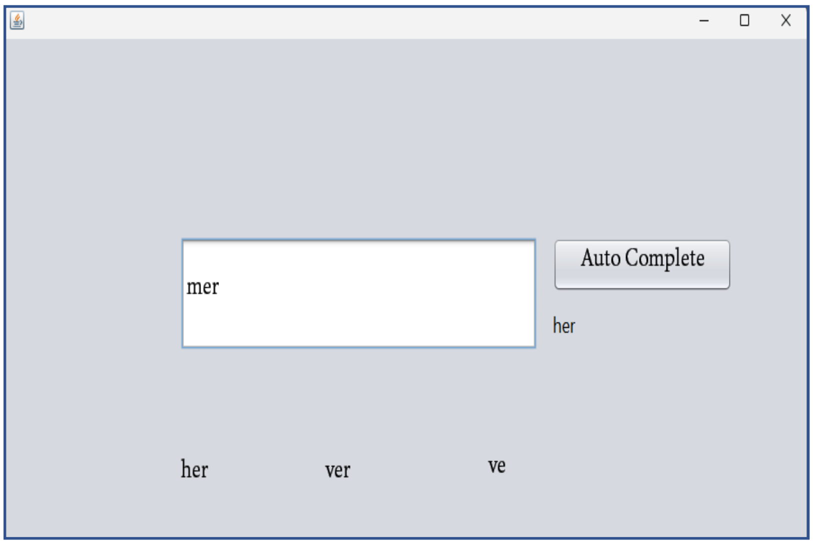
When the program start:



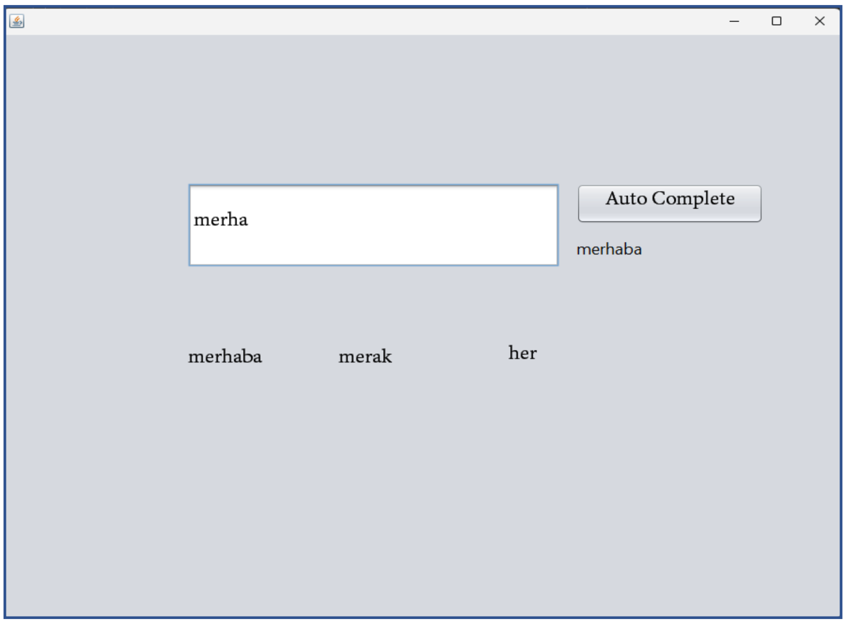
After writing one litter:



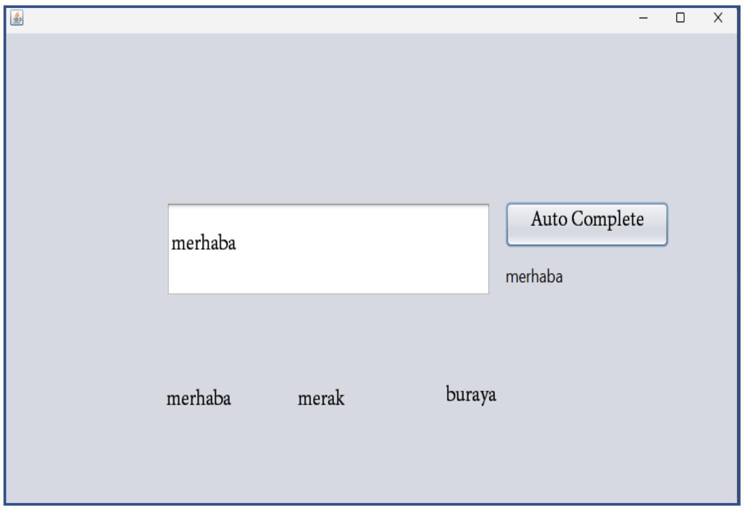
After writing three litter:

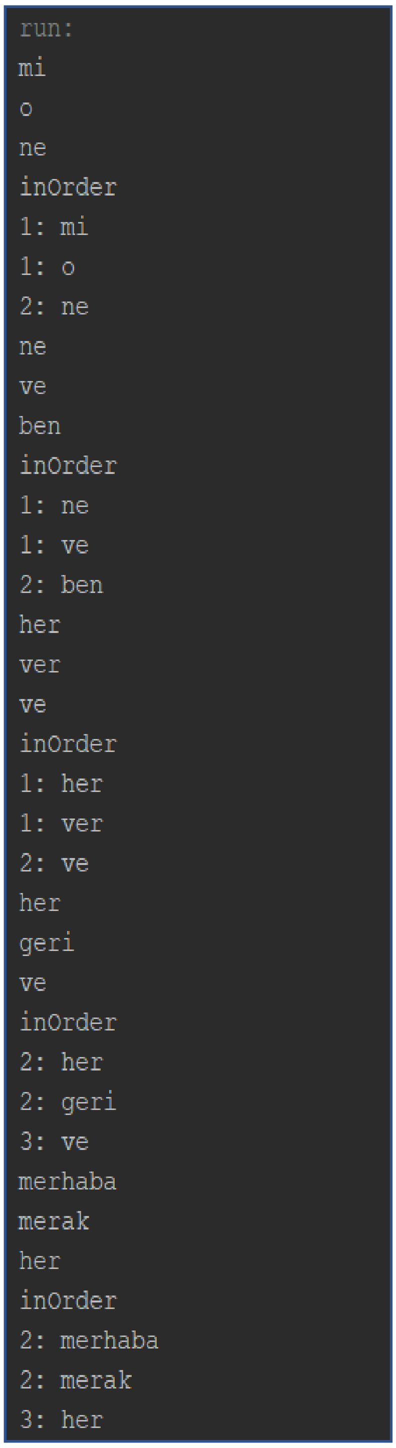


After writing more litters:



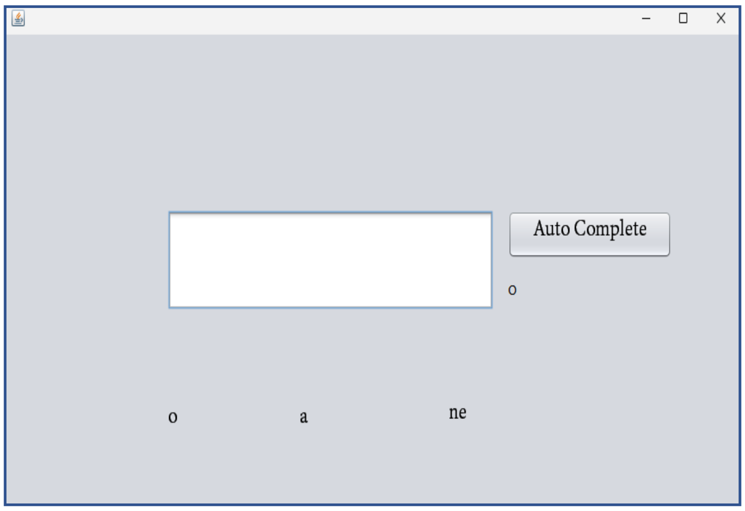
after clicking the Auto Complete button:



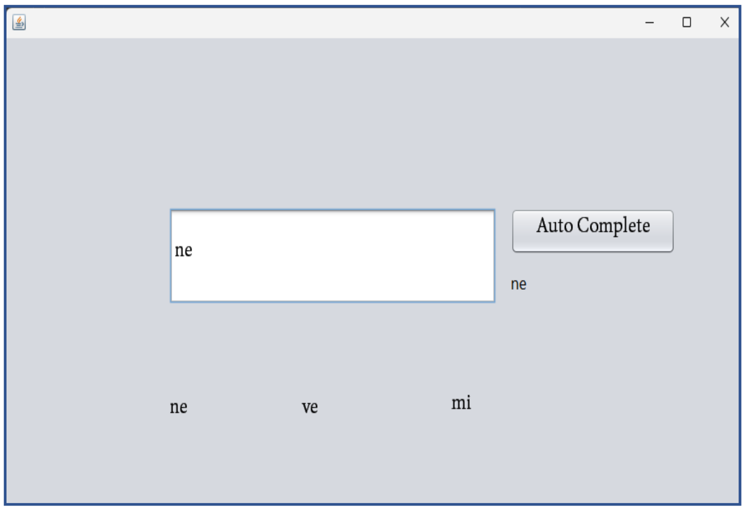


```
run:
mi
o
ne
inOrder
1: mi
1: o
2: ne
ne
ve
ben
inOrder
1: ne
1: ve
2: ben
her
ver
ve
inOrder
1: her
1: ver
2: ve
her
geri
ve
inOrder
2: her
2: geri
3: ve
merhaba
merak
her
inOrder
2: merhaba
2: merak
3: her
```

After deleting all the litters:



After clicking one of the suggested words:



The suggested words will be also printed at the output part.