King Fahd University of Petroleum and Minerals

221-ICS 202 PROJECT

[Dictionary – AVL Tree Data Structure Implementation]

MOHAMMED ALMUBARAK – 202024880 - SEC#51

HUSSAIN ALKHALAIF – 202035120 - SEC#51

```
public class Dictionary extends AVLTree<String> {
```

The dictionary holds a list of words (strings) to be used in a spell checker. The class is a subclass of the AVLTree class that also substitutes the generic T to a type String, since the dictionary only holds words (strings) and also to get rid of unnecessary casting and issues had we used a generic type T.

1- Initialization

First of all, we created a dictionary class that extends AVLTree<String>, that can be initialized using three different constructors with different parameters:

1) a single string, [public Dictionary(String s)]
2) an empty dictionary, [public Dictionary( )]
3) a text file having strings, each on a new line, [public Dictionary(File f)]

In the first constructor, we create a dictionary with only one string which would be the root in the actual AVL tree using the super keyword. This uses the original AVLTree constructor.

```
// creates a dictionary with only 1 string s.
public Dictionary(String s) {
    super(new BSTNode<String>(s));
}
```

In the second constructor, we create an empty dictionary using the super keyword. This uses the original AVLTree constructor.

```
// creates an empty dictionary.
public Dictionary() {
    super();
}
```

In the third constructor, we are reading a preformatted text file and inserting each word in it to the AVL tree. The method will throw an exception if the file does not exist or is inaccessible, which should be handled later on when using this constructor.

```java
// create a text file having strings, each on a new line.
1 usage
public Dictionary(File f) throws FileNotFoundException {
    Scanner fileScanner = new Scanner(f);
    while (fileScanner.hasNextLine())
        this.insertAVL(fileScanner.nextLine());
    fileScanner.close();
}
```

2- Adding a new word

This method adds a new word to the dictionary. It searches for a given word, if it does not exist, then it will add it to the tree using the super class's insertAVL. Otherwise, if it already exists, it will throw an exception.

```java
// Add a new word. This method adds a new word (string) to the existing dictionary
// If the word is already in the dictionary, it should throw an exception
1 usage
public void addWord(String s) throws WordAlreadyExistsException {
    if (this.search(s) == null)
        this.insertAVL(s);
    else
        throw new WordAlreadyExistsException("Word already exists.");
}
```

3- Searching for a word

This method of return type Boolean takes one string and searches for the given word. Then, if it's found, the method will return true, and if it not found, it would return false.

For searching, we are using the search method from the BST class.

```
// Search for word, This method searches for a word in the existing dictionary
// if the word is found, it should return true, otherwise it should return false.
1 usage
public boolean searchWord(String s) {
    return this.search(s) != null;
}
```

4- Removing a word

This method takes one string to delete it from the dictionary. It first searches for the given word. If it's found, it would delete the word from the AVL tree, and if it isn't found, then it would throw an exception to be handled later.

For searching, we are using the search method from BST class, and to delete, we use deleteAVL from AVLTree class, which also handles re-balancing the tree if needed.

```
// Delete word. This method deletes a word (string) from the existing dictionary.
// If the word is not in the dictionary, it should throw an exception.
1 usage
public void deleteWord(String s) throws WordNotFoundException {
    if (this.search(s) == null)
        throw new WordNotFoundException("Word does not exist.");
    else
        this.deleteAVL(s);
}
```

5- Finding similar words:

This method takes one string to find similar words to it in the dictionary. It searches for words that have the same length or differ by one letter in the length. If such words are to be found, the method stores then in a stack, then pushes them in another stack to restore the correct order, and finally formats the output as a string and returns it. It uses a helper recursive method to start searching.

```
// method for finding the similar words.
1 usage
public String findSimilar (String s) {
    Stack<String> similarWordsStack = new Stack<String>();
    Stack<String> flippedStack = new Stack<String>(); // Flipping the stack to preserve the original word order.
    this.findSimilar(s, this.root, similarWordsStack);

    while (!similarWordsStack.isEmpty())
        flippedStack.push(similarWordsStack.pop());

    String formattedString = flippedStack.pop();
    while (!flippedStack.isEmpty())
        formattedString += ", " + flippedStack.pop();
    formattedString += ".";


    return formattedString;
}
```

Helper findSimilar method: Part 1

```
// helper method that finds similar words in the dictionary, and stores them in the given stack.
3 usages
private void findSimilar (String s, BSTNode<String> node, Stack stack) {
    if (node == null)
        return;

    // search in the left subtree.
    findSimilar (s, node.left, stack);

    if (node.el.length() == s.length()) {
        int differentLetter = 0;

        for (int i = 0; i < s.length(); i++)
            if (s.charAt(i) != node.el.charAt(i))
                differentLetter++;

        if (differentLetter == 1)
            stack.push(node.el);
    }
```

In this part, we first recursively call the method to search in the left subtree, and then we check if the input word length is equal to the word stored in the current node. If lengths are equal, then we initialize a variable called differentLetter to count how many letter the two words are different from each other. We check each character and increment differentLetter as necessary. Upon finishing, if we had gotten differentLetter = 1, then we push the similar word to the stack.

Helper findSimilar method: Part 2

```
/* To search for similar words which differ by 1 letter, in case the length of the current node's word
   is greater than the given word by 1 */
else if ((node.el.length() + 1) == s.length()) {
    int differentLetter = 0;
    for (int i = 0; i < (s.length() - 1); i++) {
        if (s.charAt(i) != node.el.charAt(i)) {
            differentLetter++;

            for (int j = i; j < (s.length() - 1); j++) {
                if (s.charAt(j + 1) != node.el.charAt(j)) {
                    differentLetter++;
                    break;
                }
            }
            break;
        }
    }
    if (differentLetter == 1 || differentLetter == 0)
        stack.push(node.el);
}
```

If the previous condition (same length) is not true, then check if the input word length greater than the current node's word by 1, then we initialize a variable called differentLetter to count how many letters the two words are different from each other. We check each character, and if we get a different character, we increment our differentLetter counter. When this process has finished, and get differentLetter = 1 or = 0 (which means there is no difference or the different character is the last character in the input word), then push the similar word to the stack.

Helper findSimilar method: Part 3

```
/* To search for similar words which differ by 1 letter, in case the length of the current node's word
   is less than the given word by 1 */
else if (node.el.length() == (s.length() + 1)) {
    int differentLetter = 0;
    for (int i = 0; i < (s.length()); i++) {
        if (s.charAt(i) != node.el.charAt(i)) {
            differentLetter++;

            for (int j = i; j < (s.length()); j++) {
                if (s.charAt(j) != node.el.charAt(j+1)){
                    differentLetter++;
                    break;
                }
            }
            break;
        }
    }

    if (differentLetter == 1 || differentLetter == 0)
        stack.push(node.el);
}

// search in the right subtree.
findSimilar (s, node.right, stack);
```

If the previous condition is not true, then check if the current node's length is greater than the input word by 1, then we initialize a variable called differentLetter to count how many letters the two words are different from each other. Again, we apply the same process: we check each character, and if we get a different character, then we increment our differentLetter counter. When this process has finished and we get differentLetter = 1 or = 0 (which means there is no difference or the different character is the last character in the input word), then push the similar word to the stack.

After all that, we then recursively the call the method to find similar words in the right subtree, too.

6- Save the updated dictionary:

After adding or removing words from the dictionary, we can save it as a new text file using this method. This method uses a modified version of in-order depth-first traversing.

```
    // save the dictionary in a new text file, using in-order traversing of the dictionary
    1 usage
    public void exportDictionary(String fileName) throws FileNotFoundException {
        PrintWriter file = new PrintWriter(fileName);
        exportDictionaryAuxiliary(this.root, file);
        file.close();
        }

    // recursive helper method for the above method (traversing the dictionary in-order)
    3 usages
    private void exportDictionaryAuxiliary(BSTNode<String> p, PrintWriter file) {
        if (p != null) {
            exportDictionaryAuxiliary(p.left, file);
            file.print(p.el + "\n");
            exportDictionaryAuxiliary(p.right, file);
        }
    }
}
```

Custom exceptions:

- ### Custom WordAlreadyExistsException:

```
class WordAlreadyExistsException extends InvalidDnDOperationException {
    1 usage
    public WordAlreadyExistsException(String message) {
        super(message);
    }
}
```

- ### Custom WordNotFoundException:

```
class WordNotFoundException extends InvalidDnDOperationException {
    1 usage
    public WordNotFoundException(String message) {
        super(message);
    }
}
```

Test class:

```
1. Create a dictionary and initialize it with a text file
2. Create a dictionary and initialize it with a given single word
3. Create an empty dictionary
To quit, enter any other key.

Enter your choice>
```

```
1. Insert a word
2. Delete a word
3. Search for a word
4. Search for similar words
5. Print the dictionary
6. Save the dictionary to a new text file
7. Exit
```

- # Catches in the test class:

```
catch (WordAlreadyExistsException e) {
    System.out.println("******Exception: word already exists.******");
    dictionaryManipulationTest(dictionary);
}

catch (WordNotFoundException e) {
    System.out.println("******Exception: word not found in dictionary.******");
    dictionaryManipulationTest(dictionary);
}

catch (InputMismatchException e) { // For when the user choice is not an integer value
    System.out.println("******Exception: your choice is not an integer value.******");
    dictionaryManipulationTest(dictionary);
}

catch (FileNotFoundException e) {
    System.out.println("******Exception: file not found and/or inaccessible.******");
    dictionaryManipulationTest(dictionary);
}

catch (Exception e) {
    e.printStackTrace();
    System.out.println("******Exception: Unknown error.******");
}
```

1) Initialization

- **A text file having strings.**

```
Enter your choice> 1

Enter the file name> mydictionary.txt
Dictionary loaded successfully.
```

- **A single string.**

```
Enter your choice> 2

Enter a word for the dictionary to be initialized with> trying
Dictionary created and initialized successfully.
```

- **Empty.**

```
Enter your choice> 3

Empty dictionary created successfully.
```

2) Add new word

```
add new word>
punter
Word added successfully.
```

```
add new word> abo
******Exception: word already exists.******
```

3) Search for word

```
check word> puinter
Word not found.
```

```
check word>
abo
Word found.
```

4) Remove word

```
remove word>
abc
Word removed successfully.
```

```
remove word> ICS
******Exception: word not found in dictionary.******
```

5) Find similar words

```
Enter the number which corresponds to the operation you want> 4
search for similar words> puinter
painter, pointer, printer, punter.
```

6) Save file

```
Enter the number which corresponds to the operation you want> 6
Save Updated Dictionary (Y/N)> No
Invalid saving choice - Please either enter Y or N!
Save Updated Dictionary (Y/N)>
```

```
Save Updated Dictionary (Y/N)> Y
Enter filename> mydictionary2.txt
Dictionary saved successfully.
```

7) Printing the dictionary

```
1. Create a dictionary and initialize it with a text file
2. Create a dictionary and initialize it with a given single word
3. Create an empty dictionary
To quit, enter any other key.

Enter your choice> 1

Enter the file name> C:\Users\hossa\IdeaProjects\ICS 202\src\LabProject\try.txt
Dictionary loaded successfully.

1. Insert a word
2. Delete a word
3. Search for a word
4. Search for similar words
5. Print the dictionary
6. Save the dictionary to a new text file
7. Exit

Enter the number which corresponds to the operation you want> 5

a aaa aaas aarhus aaron aau aba ababa aback abacus abalone abandon abase abash abate abater abbas maha miha moh mohar morha muha
```