KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

INFORMATION AND COMPUTER SCIENCE DEPARTMENT

ICS-202 Project:

Designing and Implementing a Dictionary Data Structure for Spell Checker

Done By:
Syed Abidullah Fareed
202185410
Section: 56
Major: Computer Engineering

9th December 2023

I. Introduction

The goal of this project is to develop a dictionary which is to be used in a spell checker. The spell checker will help users identify and correct misspelled words by comparing input words against the dictionary.

a. Data Structure Selection

For an efficient dictionary in spell checker, a suitable data structure is crucial. The chosen data structure should facilitate quick lookups and insertions. One commonly used data structure for this purpose is an AVL tree. It offers faster lookup, insertion and deletion times, providing log(n) complexities for average and worst-case performance.

b. Design Decision

The required operations for a dictionary in spell checker:
1. Adding a new word.
2. Deleting an existing word.
3. Searching for a word.
4. Finding similar words.

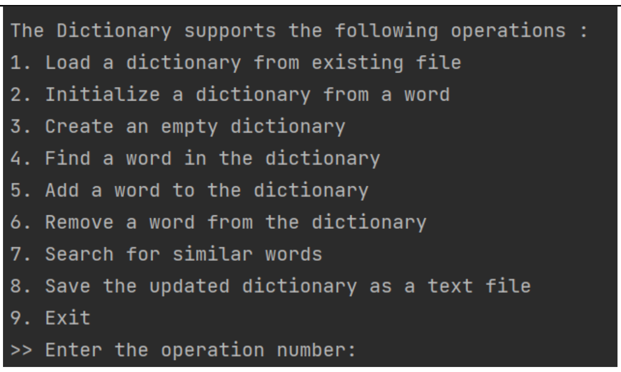
AVL are self-balancing trees, which makes them a better choice to be used in designing the dictionary with the above-mentioned operations. Here are some of the benefits:

- Efficient Lookups: Balanced nature ensures O (log n) lookup time compared to O(n) in unbalanced structures.
- Memory Management: AVL trees efficiently manage memory with dynamic resizing, unlike fixed-size arrays.

c. Motivation

Three main motives:
1. Efficient Operations
2. Efficient Memory Management
3. Scalability

II. Operations of the Dictionary



```
The Dictionary supports the following operations :
1. Load a dictionary from existing file
2. Initialize a dictionary from a word
3. Create an empty dictionary
4. Find a word in the dictionary
5. Add a word to the dictionary
6. Remove a word from the dictionary
7. Search for similar words
8. Save the updated dictionary as a text file
9. Exit
>> Enter the operation number:
```

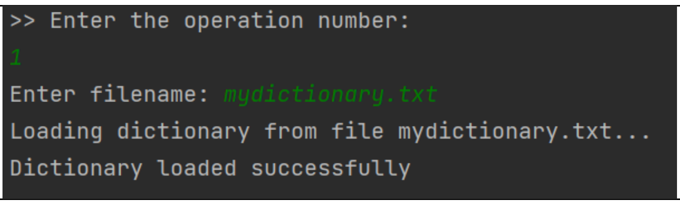
Figure 1. Main Dashboard

1. Initializing dictionary:

Pseudocode:

a. From a file

```
loadFileIntoDictionary:
    Open the file.
    if file does not exist or cannot be opened:
        Display an error message.
        Return
    while not end of file:
        Read the next word from the file.
        Insert the word into the AVL tree and balance it.
    Close the file.
```

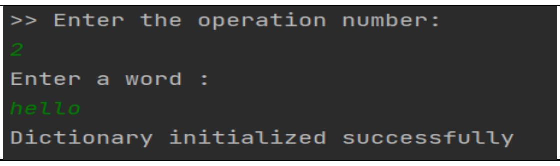


```
>> Enter the operation number:
1
Enter filename: mydictionary.txt
Loading dictionary from file mydictionary.txt...
Dictionary loaded successfully
```

Figure 2. loading from file

b. A single word

```
Read the word.
Insert the word as root into the AVL tree.
```

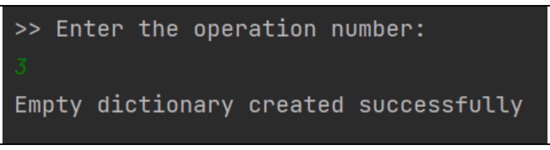


```
>> Enter the operation number:
2
Enter a word :
hello
Dictionary initialized successfully
```

Figure 3. loading from a single string

c. Empty Dictionary

Initialize AVL with root as null.



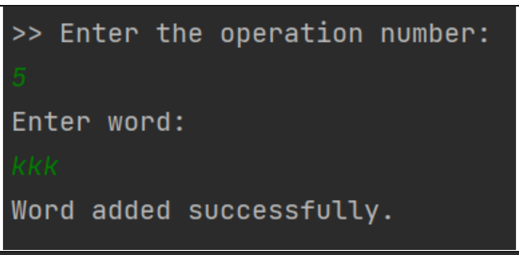
```
>> Enter the operation number:
3
Empty dictionary created successfully
```

Figure 4. empty dictionary

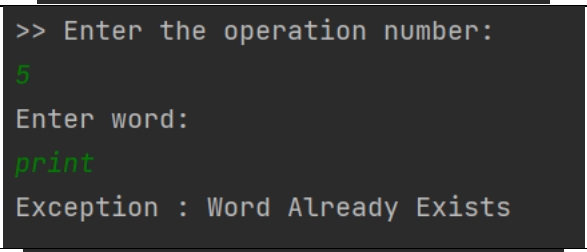
2. Adding a word:

Pseudocode

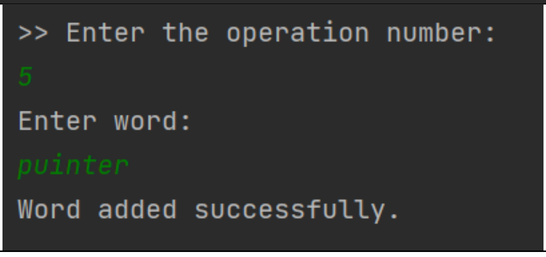
```
addWord(word):
traverses through the levels in the direction of node to be added
    If word is not found in the dictionary and appropriate position is reached:
        Insert the word into the AVL tree and balance it.
    Else:
        Throw a custom WordAlreadyExistsException.
```



```
>> Enter the operation number:
5
Enter word:
kkk
Word added successfully.
```



```
>> Enter the operation number:
5
Enter word:
print
Exception : Word Already Exists
```



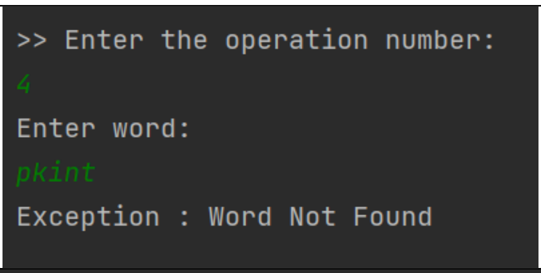
```
>> Enter the operation number:
5
Enter word:
puinter
Word added successfully.
```

Figure 5. adding a word

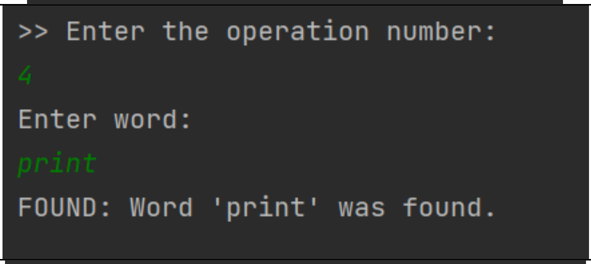
3. Finding a word:

Pseudocode

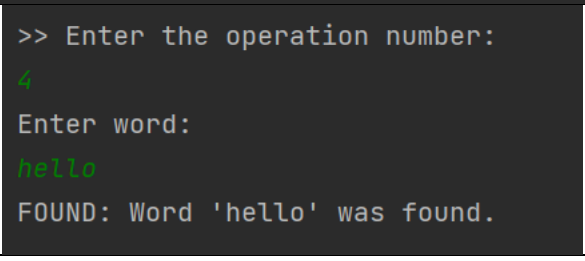
```
findWord(word):
traverses through the levels in the direction of the word to be searched.
    If word is found:
        Return true
    Else return false
```



```
>> Enter the operation number:
4
Enter word:
pkint
Exception : Word Not Found
```



```
>> Enter the operation number:
4
Enter word:
print
FOUND: Word 'print' was found.
```



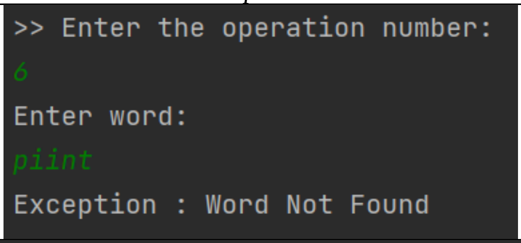
```
>> Enter the operation number:
4
Enter word:
hello
FOUND: Word 'hello' was found.
```

Figure 6. finding a word

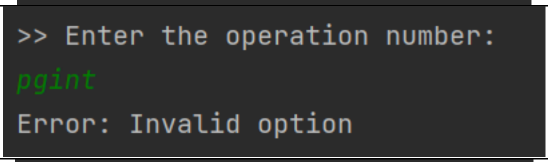
4. Remove a word:

Pseudocode

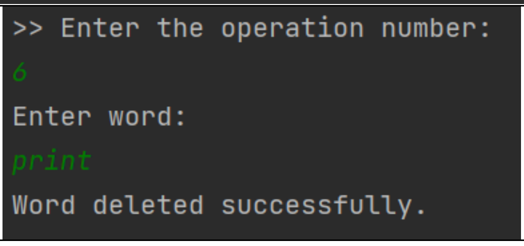
```
removeWord(word):
traverses through the levels in the direction of the word to be deleted.
   If word is found in the dictionary:
      delete the word into the AVL tree and balance it.
   Else:
      Throw a custom WordNotFoundException
```



```
>> Enter the operation number:
6
Enter word:
piint
Exception : Word Not Found
```



```
>> Enter the operation number:
pgint
Error: Invalid option
```



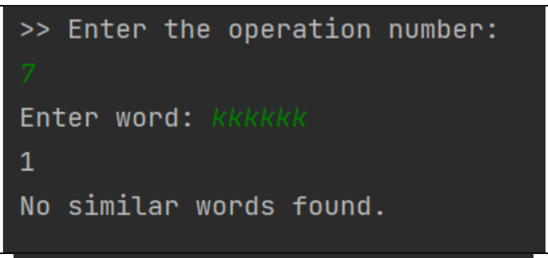
```
>> Enter the operation number:
6
Enter word:
print
Word deleted successfully.
```

Figure 7. removing a word

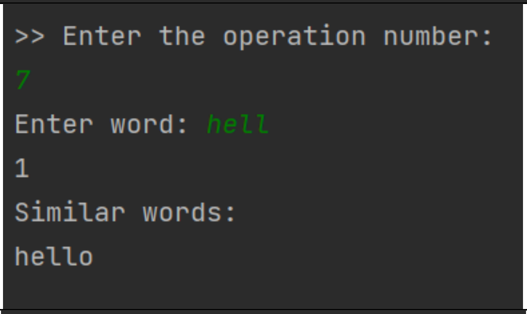
5. Search for similar words:

Pseudocode

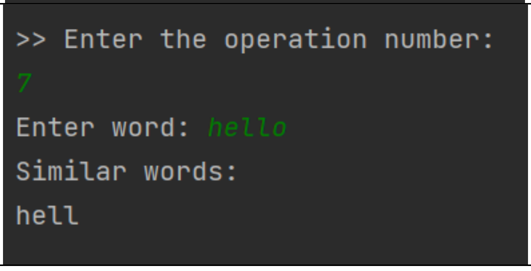
```
SimilarWords(word):
Traverse through the AVL tree, for each node
    If node is similar (i.e. differs only by one letter or one in length) to word
        Add to similar words list
Return similar words
```



```
>> Enter the operation number:
7
Enter word: kkkkkk
1
No similar words found.
```



```
>> Enter the operation number:
7
Enter word: hell
1
Similar words:
hello
```



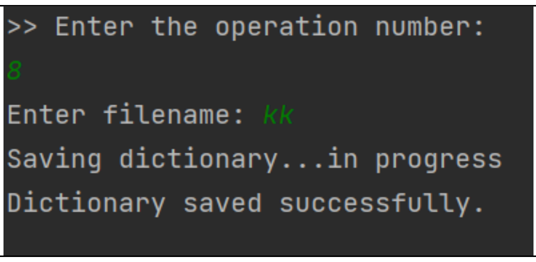
```
>> Enter the operation number:
7
Enter word: hello
Similar words:
hell
```

Figure 8. finding similar words

6. Saving the updated dictionary:

Pseudocode

```
Open the file.
Traversing the AVL tree using in order traversal
    Writing the word to the file
 Close the file.
```



```
>> Enter the operation number:
8
Enter filename: kk
Saving dictionary...in progress
Dictionary saved successfully.
```

Figure 9. saving dictionary

III. Challenges Faced

- Selecting Optimal Structure: Choosing the most efficient data structure for the dictionary amidst options like AVL trees, BSTs, and SLLs required careful consideration and analysis. Striking a balance between memory usage and optimal performance for large datasets was a key challenge. However, with the help of the course materials I was able to select an efficient data structure.

- Testing and Debugging: Extensive testing and debugging were essential to ensure the AVL tree handled various scenarios without errors.

- Time Management: Balancing project timelines and ensuring an efficient AVL tree implementation within the deadline posed time management challenges.

IV. Conclusion

This project gave me an opportunity to test my understanding of the data structures and efficiently use it in real life applications. While AVL trees demonstrated efficiency in lookups and insertions, exploring hashing techniques could further optimize operations.

V. Future Works

- Investigating hash-based structures to potentially enhance dictionary efficiency.
- Exploring hybrid structures combining AVL trees and hashing for improved performance.