SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

Practical Experiment 11

Research study and implementation/Analysis of Algorithm/Data Structure for ANY special Topic/ Application (mini project).

Topic: Optimizing English Dictionary Lookup with Red-Black Tree Data Structures

Team Members:

Rishabh Jain: 60004200141

Devang Shah: 60004200158

Ayush Parikh: 60004200162

PROBLEM STATEMENT

A dictionary is a data structure that has the capability to store data of various types and can be utilised subsequently to access data in a proficient manner. There are several data structures that can be utilised to construct a Dictionary, including but not limited to AVL trees, Tries, Red Black Trees, and B Trees. Given the vast amount of data that a dictionary may contain, it is imperative to carefully evaluate the temporal and spatial requirements associated with storing and accessing information within the dictionary. Red Black Trees possess certain advantages over Trie and AVL Trees and exhibit significant similarities with B Trees. The objective of this endeavour is to construct a lexicon utilising the Red Black Tree data structure.

INTRODUCTION

Use cases:

One can use a dictionary to look up the meaning of any words that one doesn't understand. A dictionary is one of the most important tools during studying at a university. A good dictionary can help one understand the subject better, improve one's communication by making sure you are using words correctly. A monolingual dictionary has lots of different information about every word in English. These dictionaries have lots of information about grammar and pronunciation. To make this

dictionary available online to everyone for use, we need to store it in some data structure for efficient management of data from the dictionary. As a dictionary can save n number of words it is necessary to maintain the storage aspect as well as retrieval of data. Thus, need of a data structure to store words of a dictionary is essential.

Working:

The project consists of a pre built English Dictionary words which can be loaded into the Red Black Tree. The program reads the text file line by line, inserts each word in the Red Black Tree using BST Insertion Property and fixes the insertion so that there are no violations of the Red Black Tree properties. Through the program option has been provided to insert a new word in the dictionary. When the user chooses insert a word option he is prompted to insert a word, the program reads the word and inserts that word in the dictionary i.e., in the RB Tree using the BST Insertion and Fixup Code to avoid any violations of properties. The program provides an option to find a particular word in the dictionary. When the user chooses the option to find a word he is prompted to search the word which is needed to be found. The search operation takes place through the normal BST Search Property. If the word is found the user gets a message that the word has been found. If the word is not found then user gets a message that the word entered is not present in the dictionary. The program also provides an option to see the size of the dictionary. When the user chooses t he size option, the program counts all the nodes of the RB Tree and displays the sum of all the nodes, which represents the size of the dictionary. The user can also see the height of the tree formed for the dictionary. When the user chooses this option the program counts the number of nodes present in one side of the tree and returns the count which represents the height of the tree.

Applications

The project can be used as a standard dictionary which contains words and their meanings. It provides fast storage and access of data with the help of Red Black Trees. The project can be used to load a predefined set of words in the dictionary. It can also be used to add new words in the dictionary. It can also be used to find whether a word is present in the dictionary or not. It provides all these operations in minimum time and space possible.

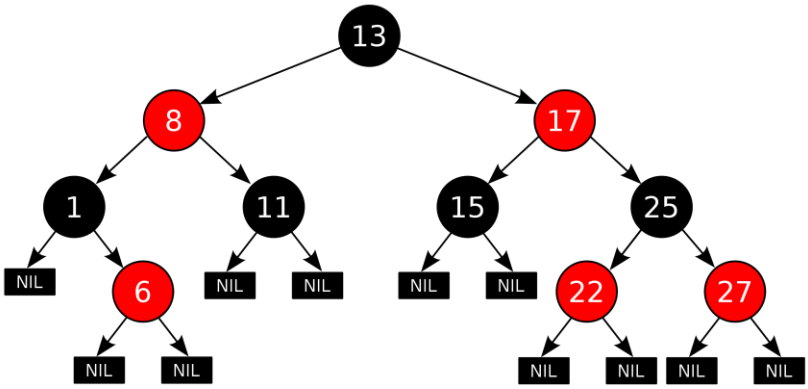
ADVANCED DATA STRUCTURE

Theory/Working

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black). These colors are used to ensure that the

tree remains balanced during insertions and deletions. Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around O(log n) time, where n is the total number of elements in the tree. Properties of RB Tree are :

- Every node has a color either red or black.
- The root of the tree is always black.
- There are no two adjacent red nodes (A red node cannot have a red parent or red child).
- Every path from a node (including root) to any of its descendants' NULL nodes has the same number of black nodes.
- All leaf nodes are black nodes



Black height is the number of black nodes on a path from the root to a leaf. Leaf nodes are also counted black nodes.

Insertion of Node in RB Tree

Algorithm for Insertion of a Node

Let x be the newly inserted node.

Perform standard BST insertion and make the colour of newly inserted node as RED.

If x is the root, change the colour of x as BLACK.

Do the following if the color of x's parent is not BLACK and x is not the root.

If x's uncle is RED

Change the colour of parent and uncle as BLACK.

Colour of a grandparent as RED.

Change x = x's grandparent, repeat steps 2 and 3 for new x.

If x's uncle is BLACK, then there can be four configurations for x, x's parent (p) and x's grandparent (g)

Left Left Case (p is left child of g and x is left child of p) : Perform Right Rotation and Swap Color

Left Right Case (p is left child of g and x is the right child of p) : Perform RL Rotation and Swap Color

Right Right Case (Mirror of case i) : Perform Left Rotation and Swap Color

Right Left Case (Mirror of case ii) : Perform LR Rotation and Swap Color

Searching a Node in RB Tree

We start at the root, and then we compare the value to be searched with the value of the root, if it's equal we are done with the search if it's smaller we know that we need to go to the left subtree because in a binary search tree all the elements in the left subtree are smaller and all the elements in the right subtree are larger. Searching an element in the binary search tree is basically this traversal, at each step we go either left or right and at each step we discard one of the sub-trees

Level Order Traversal

- Traversing every node level wise
- Nodes at level i are traversed before level i+1

Uses:

- Most of the self-balancing BST library functions like map, multiset, and multimap in C++ (or java pacakages like java.util.TreeMap and java.util.TreeSet) use Red-Black Trees.
- It is used to implement CPU Scheduling Linux. Completely Fair Scheduler uses it.
- It is also used in the K-mean clustering algorithm in machine learning for reducing time complexity.
- Moreover, MySQL also uses the Red-Black tree for indexes on tables in order to reduce the searching and insertion time.

Complexity Analysis

Insertion

There are three phases to inserting a key into a non-empty tree. The binary search tree insert is conducted in the first phase. Because a red-black tree is balanced, the BST insert operation is O(height of tree), which is O(log n). The new node is then colored red in the second stage. This step is O(1) since it only involves changing the value of one node's color field. In the third stage, we restore any red-black characteristics that have been violated. Changing the colors of nodes takes O(1) time. However, we may need to deal with a double-red issue farther along the route from the inserted node to the root. In the worst-case scenario, we wind up fixing a double-red condition all the way from the inserted node to the root. In the worst-case scenario, the recoloring performed during insertion is O(log n) i.e., time for one recoloring x maximum number of recoloring performed. As a result, restoring red-black characteristics takes O(log n), and the overall time for insert is O(log n).

Best Case: In the best case, there is no rotation. Only recoloring takes place. The time complexity is O(log n). Consider the following example.

Worst case: RB trees require a constant (at most 2 for insert) number of rotations. So in the worst case, there will be 2 rotations while insertion. The time complexity is O(log n).

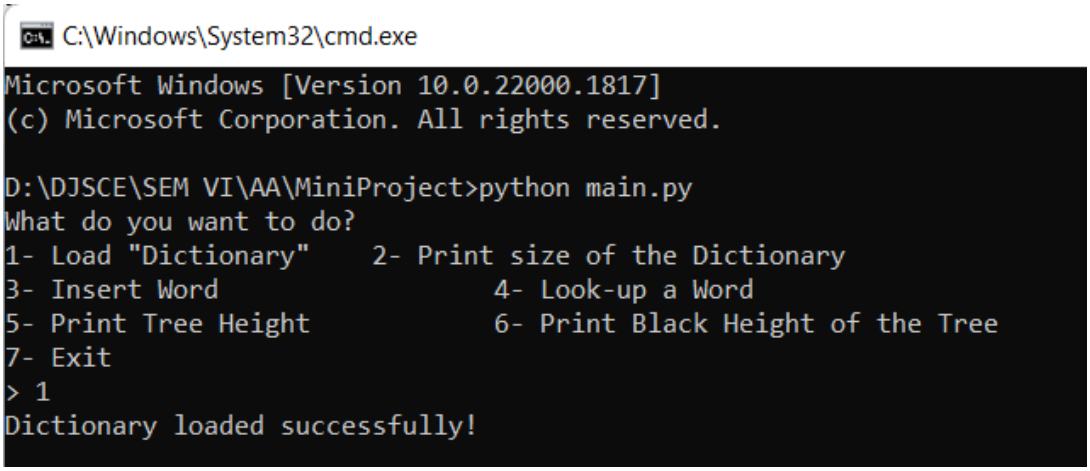
Average Case: Since the average case is the mean of all possible cases, the time complexity of insertion in this case too is O(log n).

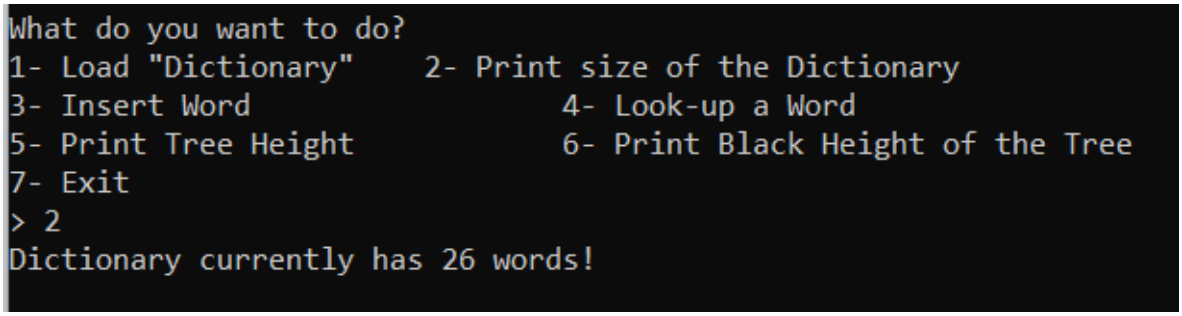
Space Complexity of RB Tree

The average and worst space complexity of a red-black tree is the same as that of a Binary Search Tree and is determined by the total number of nodes: O(n) because we don't need any extra space to hold duplicate data structures. We arrive to this conclusion because each node has three pointers: left child, right child, and parent. Each node takes up O(1) space. As a result, if the tree has n total nodes, the space complexity is n times O(1), which is O(n). Because there are just two colors, monitoring the color of each node takes only one bit of information per node. Because the tree contains no extra data that distinguishes it as a red-black tree, its memory footprint is nearly comparable to that of a conventional binary search tree. In many circumstances, the extra bit of data may be stored with no extra memory cost.

IMPLEMENTATION

Important screen shots





```
What do you want to do?
1- Load "Dictionary"    2- Print size of the Dictionary
3- Insert Word                  4- Look-up a Word
5- Print Tree Height            6- Print Black Height of the Tree
7- Exit
> 4
Enter the word you want to look-up: hello
"hello" DOES NOT EXIST IN THE DICTIONARY
```

```
What do you want to do?
1- Load "Dictionary"    2- Print size of the Dictionary
3- Insert Word                  4- Look-up a Word
5- Print Tree Height            6- Print Black Height of the Tree
7- Exit
> 3
Enter the word you want to insert: riya
"riya" inserted Successfully

What do you want to do?
1- Load "Dictionary"    2- Print size of the Dictionary
3- Insert Word                  4- Look-up a Word
5- Print Tree Height            6- Print Black Height of the Tree
7- Exit
> 3
Enter the word you want to insert: raj
"raj" inserted Successfully
```

```
What do you want to do?
1- Load "Dictionary"    2- Print size of the Dictionary
3- Insert Word                  4- Look-up a Word
5- Print Tree Height            6- Print Black Height of the Tree
7- Exit
> 2
Dictionary currently has 28 words!

What do you want to do?
1- Load "Dictionary"    2- Print size of the Dictionary
3- Insert Word                  4- Look-up a Word
5- Print Tree Height            6- Print Black Height of the Tree
7- Exit
> 4
Enter the word you want to look-up: riya
FOUND "riya"!
```

```
What do you want to do?
1- Load "Dictionary"    2- Print size of the Dictionary
3- Insert Word                  4- Look-up a Word
5- Print Tree Height            6- Print Black Height of the Tree
7- Exit
> 5
7

What do you want to do?
1- Load "Dictionary"    2- Print size of the Dictionary
3- Insert Word                  4- Look-up a Word
5- Print Tree Height            6- Print Black Height of the Tree
7- Exit
> 6
4
```

Code of important functions

DataStructure.py

```
class Node:
    # if color = 0 -> red
    # if color = 1--> black
    def __init__(self, key):  # Constructor
        self.key = key  # Node needs a key to be initialized
        self.parent = None
        self.right = None
        self.left = None
        self.color = 0


class RedBlackTree:
    def __init__(self):  # Constructor
        self.nil = Node(None)
        self.nil.color = 1  # The root and the nil are black
        self.root = self.nil
        self.number_of_nodes = 0

    def search(self, key):
        node = self.root

        while node != self.nil:  # as long as we didn't reach the end of the tree
            if node.key == key:
                return True
            elif key < node.key:
                node = node.left
            else:
                node = node.right
        return False
```

```
    def insert(self, key):
        newNode = Node(str(key).lower())
        newNode.left = self.nil
        newNode.right = self.nil
        node = self.root
        parent = None  # TBD

        while node != self.nil:  # Find the appropriate parent
            parent = node
            if newNode.key < node.key:
                node = node.left
            else:
                node = node.right
        newNode.parent = parent

        if parent is None:  # Inserted node is the first node
            newNode.color = 1
            self.root = newNode
            self.number_of_nodes += 1
            return
        elif newNode.key < parent.key:
            parent.left = newNode
        else:
            parent.right = newNode

        if newNode.parent.parent is None:  # Parent is the root
            self.number_of_nodes += 1
            return

        self.insertFix(newNode)  # Handle cases
        self.number_of_nodes += 1

    # This method handles cases of RB-tree insertions
    def insertFix(self, newNode):
        while newNode != self.root and newNode.parent.color == 0:  # Loop until we reach the root or
parent is black

            parentIsLeft = False  # Parent is considered left child by default

            # Assign uncle to appropriate node
            if newNode.parent == newNode.parent.parent.left:
                uncle = newNode.parent.parent.right
                parentIsLeft = True
            else:
                uncle = newNode.parent.parent.left

            # Case 1: Uncle is red -> Reverse colors of uncle, parent and grandparent
            if uncle.color == 0:
                newNode.parent.color = 1
                uncle.color = 1
                newNode.parent.parent.color = 0
```

```
            newNode = newNode.parent.parent

        # Case 2: Uncle is black -> check triangular or linear and rotate accordingly
        else:
            # Left-right condition (triangular)
            if parentIsLeft and newNode == newNode.parent.right:
                newNode = newNode.parent  # Take care as we made the new node the parent
                self.leftRotate(newNode)
            # Right-Left condition (triangular)
            elif not parentIsLeft and newNode == newNode.parent.left:
                newNode = newNode.parent
                self.rightRotate(newNode)
            # Left-left condition (linear)
            if parentIsLeft:
                newNode.parent.color = 1  # the new parent
                newNode.parent.parent.color = 0  # the new grandparent will be red
                self.rightRotate(newNode.parent.parent)
            # Right-right condition (linear)
            else:
                newNode.parent.color = 1
                newNode.parent.parent.color = 0
                self.leftRotate(newNode.parent.parent)

    self.root.color = 1  # Set root to black

def leftRotate(self, node):
    """
          node             y
           \     =>       / \
            y          node  d
           / \           \
          c   d           c
          """
    y = node.right
    node.right = y.left  # connect node to c
    if y.left != self.nil:  # connect c to node
        y.left.parent = node

    y.parent = node.parent  # connect y to node's parent

    if node.parent is None:  # connect node's parent to y
        self.root = y
    elif node == node.parent.left:
        node.parent.left = y
    else:
        node.parent.right = y

    y.left = node  # connect y to node
    node.parent = y  # connect node to y

def rightRotate(self, node):
    """
```

```
      node                 y
     /   \       =>       /  \
    y                   c    node
   /  \                      /
  c    d                    d
      """
      y = node.left
      node.left = y.right  # connect node to d
      if y.right != self.nil:  # connect d to node
         y.right.parent = node
      y.parent = node.parent  # connect y to node's parent

      if node.parent is None:  # connect b parent to a's parent
         self.root = y
      elif node == node.parent.left:
         node.parent.left = y
      else:
         node.parent.right = y

      y.right = node  # connect y to node
      node.parent = y  # connect node to y

   # This method returns the height of the tree
   def heightOfTree(self, node, sumval):
      if node is self.nil:
         return sumval
      return max(self.heightOfTree(node.left, sumval + 1), self.heightOfTree(node.right, sumval + 1))

   # This method returns the black-height of the tree
   def getBlackHeight(self):
      node = self.root
      bh = 0
      while node is not self.nil:
         node = node.left
         if node.color == 1:
            bh += 1
      return bh

   # Function to print used in debugging
   def __printCall(self, node, indent, last):
      if node != self.nil:
         print(indent, end=' ')  # the default end character is new line
         if last:
            print("R----", end=' ')
            indent += "     "
         else:
            print("L----", end=' ')
            indent += "|    "

         s_color = "RED" if node.color == 0 else "BLACK"
         print(str(node.key) + "(" + s_color + ")")
         self.__printCall(node.left, indent, False)
```

```
            self.__printCall(node.right, indent, True)

    # Function to call print
    def print_tree(self):
        self.__printCall(self.root, "", True)


"""
tree = RedBlackTree()
tree.insert('a')
tree.insert('b')
tree.insert('e')
tree.insert('d')
tree.insert('c')
tree.insert('f')
tree.insert('g')
tree.insert('h')
tree.insert('i')
tree.insert('j')
tree.print_tree()
print(tree.heightOfTree(tree.root, 0))
print(tree.number_of_nodes)
print(tree.search('q'))
print(tree.getBlackHeight())
"""
```

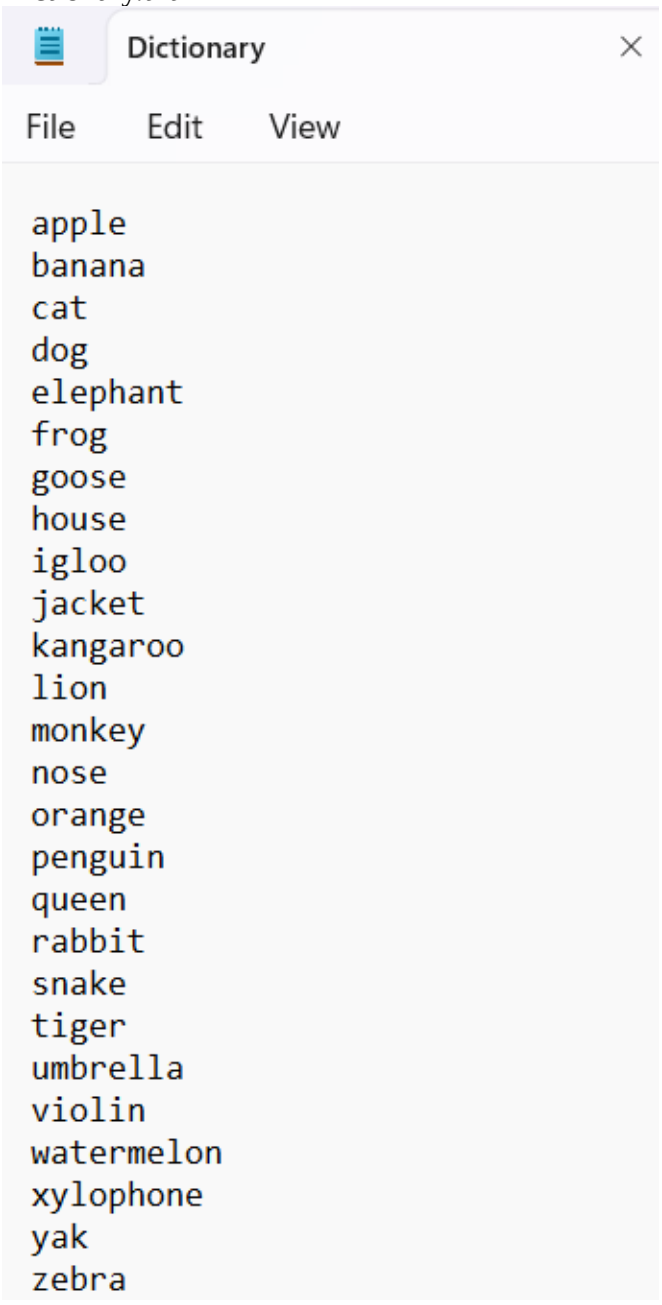
Main.py

```
import DataStructure as ds

tree = ds.RedBlackTree()  # initialize RB-tree

DICTIONARY_NAME = "Dictionary"


def readFile(fileName):
    file = open(fileName, "r")
    for i in file:
        if not tree.search(i.rstrip('\n')):
            tree.insert(i.rstrip('\n'))
    file.close()


while True:
    print("What do you want to do?")
    option = input(
        "1- Load \"" + DICTIONARY_NAME + "\"\t2- Print size of the Dictionary\n"
        "3- Insert Word            \t4- Look-up a Word\n"
```

```
    "5- Print Tree Height       \t6- Print Black Height of the Tree\n"
    "7- Exit\n"
    "> ")
if option == '1':
    readFile(DICTIONARY_NAME + ".txt")
    print(DICTIONARY_NAME + " loaded successfully!")

elif option == '2':
    print(DICTIONARY_NAME + ' currently has ' + str(tree.number_of_nodes) + ' words!')

elif option == '3':
    s = str(input("Enter the word you want to insert: ")).strip()
    if tree.search(s.lower()):
        print("\"" + s + "\" is already in the dictionary!")
    elif len(s) > 0 and not s.isspace():
        tree.insert(s.lower())
        print('\"' + s + '\" inserted Successfully')
        with open(DICTIONARY_NAME + '.txt', 'a') as file:
            file.write(f'\n{s}')
    else:
        print('Invalid entry')

elif option == '4':
    s = str(input("Enter the word you want to look-up: ")).strip()
    if tree.search(s.lower()):
        print("FOUND \"" + s + '\"!')
    else:
        print("\"" + s + '\" DOES NOT EXIST IN THE DICTIONARY')

elif option == '5':
    print(tree.heightOfTree(tree.root, 0))

elif option == '6':
    print(tree.getBlackHeight())
elif option == '7':
```

```
    print("Thank you for using our application! :)")
    break
print()
```

Dictionary.txt



COMPLEXITY ANALYSIS

Insertion of a word in the Dictionary

Insertion of a word in the dictionary takes place through the RB Tree Insertion Process. Insertion takes place in three phases where the BST insertion takes O (log n). Coloring of node is done in O (1). The Fixup code of Insertion takes best case of O(1) when only color swapping takes place and takes O (log n) if any type of rotation is to be performed. Thus the overall

average case time complexity of insertion of a word comes out to be O (log n) which is faster as compared to other data structures.

Searching a word in the Dictionary

To search a word in the Dictionary the program uses Search Operation of Binary Search Tree. The search takes place as follows : if value to be searched is less than the value of the root then the left sub tree of the tree is traversed to find the word, if value to be searched is greater than the value of the root the right subtree is searched. The process is repeated until the node with value is found. If node evaluates to NULL then the word is not found. This process is similar to Binary Search thus the complexity for searching a word is O (log n).

Height of the Tree doesn't exceeds O (log n).

The Space Complexity for the Dictionary is equivalent to the space complexity of Red Black Tree which is O (n).

Comparison of Operations on Dictionary

OPERATION	AVERAGE CASE	WORST CASE
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)

Comparison of the same operations in different data structures

METHOD	SEARCH COMPLEXITY	INSERTION COMPLEXITY
Binary Search Tree	O(n)	O(n)
Hash Table	O(n)	O(ln)
RB Tree	O(log n)	O(log n)

CONCLUSION

The present project employs a Red Black Tree data structure to instantiate a lexicon, thereby facilitating efficient word insertion and retrieval operations. The Red-Black Tree is a type of self-balancing tree that bears resemblance to the Binary Search Tree. However, it possesses an additional attribute of colour, which serves to preserve its balance. The time complexity for inserting a word in the dictionary is logarithmic with respect to the number of words in the dictionary (O(log n)). Similarly, the time complexity for searching a word in the dictionary is also logarithmic (O(log n)). The space complexity of the dictionary is linear with respect to the number of words in the dictionary (O(n)). The Dictionary has the capacity to efficiently store and retrieve a vast quantity of words.