Informatics Institute of Technology
Department of Computing

Bsc (Hons) Artificial Intelligence and Data Science

Module: CM1602 Data Structures and Algorithms

Coursework Report

Runisi Nikoya Samaranayake – 2237028

EXECUTIVE SUMMARY

In task 1 The code builds a Java software that functions as a capital gains calculator, allowing a user to track their share purchases and sales while also calculating their overall capital gains. In order to record the transactions, the application asks the user for input and employs a deque data structure to keep track of them. When a selling transaction is entered, the computer will first try to fulfill the order based on the oldest buying transactions before calculating the capital gain for each order that is completed. It will be canceled if there are not enough buying transactions to complete the selling transaction. Unless the user presses "Q" to exit, the program will continuously ask for input before outputting the entire capital gain.

In task 2A The code replicates a card game that is played with a regular 52-card deck. The user is first asked to enter the number of participants, which might be 2, 4, or 13. The computer distributes out cards to each player after receiving input, shuffles the deck, and then sorts each player's hand. Once all players have played all of their cards, the program permits each player to play one card every turn. Each time a player takes a turn, the program prints the cards that are currently in their hands and asks them to enter the card they want to play. The program ends once all cards have been played.

In task 2B The code creates a straightforward console-based spell checker that asks the user to enter a word and then determines whether it is spelled correctly. To compare the user's input to a list of correctly spelled words, it consults a dictionary file called "dictionary.txt."Until the user enters "exit" as the input, the program uses a while loop. The program asks the user to type a word, reads their input, and then sends it to the SpellChecker class for a spelling check within the loop.If the dictionary contains the entered word, the program prints "Word is correct." If not, "Word is incorrect." is displayed along with a list of synonyms. It prints "No" if the dictionary does not contain a word that is similar.

CONTENTS

TASK 1

This is a Java program that simulates a capital gain calculator that processes a sequence of buy and sell transactions and calculates the total capital gain for the user. The program prompts the user to enter a command (B for buying, S for selling, or Q for quitting), and then prompts the user for additional input depending on the command.

The program uses a Deque (double-ended queue) data structure to keep track of the buy and sell transactions. The Deque is implemented using a linked list, with each node in the list representing a single transaction. It uses a dequeue to store the buying transactions and when a selling transaction is made, it removes the oldest buying transactions first to calculate the capital gain.

```
Deque<Transaction> buyQueue = new Deque<Transaction>(); //initialize Dequeue with max
size 100
//initializes a Deque (double-ended queue) named buyQueue that can store Transaction
objects for buying
Deque<Transaction> sellQueue = new Deque<Transaction>(); //initialize Dequeue with max
size 100
Scanner = new Scanner(System.in);
String input;
double totalGain = 0.0; // initializes a double variable named totalGain to store the
total gain or loss
// from all the transactions.
```

The Transaction class is a private nested class that represents a single buy or sell transaction. Each transaction has two fields: the number of shares and the unit price. The main method of the program starts by creating two empty Deques, one for the buy transactions and one for the sell transactions, and then enters a loop that prompts the user for a command and processes the input until the user chooses to quit by entering the Q command.

```
while (true) {//while loop that continues running until the program is manually stopped
    System.out.println("Enter B for buying transaction, S for selling transaction or Q to
exit:");
    input = scanner.nextLine().trim().toUpperCase();
```

When the user enters a B command, the program prompts the user for the number of shares and the unit price, creates a new Transaction object with this information, and adds it to the buy Deque.

```
if (input.equals("B")) {
    System.out.println("Enter number of shares bought and unit price, separated by a
space:");
    String[] tokens = scanner.nextLine().trim().split(" ");
    if (tokens.length != 2) {
        System.out.println("Invalid input, please try again.");
        continue;
    }
    int shares = 0;
    double price = 0.0;
    try {
        shares = Integer.parseInt(tokens[0]);//tries to parse the input tokens as
integers and doubles
        price = Double.parseDouble(tokens[1]);
    } catch (NumberFormatException e) {
        //If the parsing fails, an error message is printed, and the program continues to
the next
        // iteration of the while loop.
        System.out.println("Invalid input, please try again.");
        continue;
    }
    if (shares <= 0 || price <= 0.0) {
        System.out.println("Invalid input, please try again.");
        continue;
    }
    buyQueue.addLast(new Transaction(shares, price));
    //input is valid, this creates a new Transaction object with the shares and price
variables
    // and adds it to the end of the buyQueue.
```

When the user enters an S command, the program prompts the user for the number of shares and the unit price, calculates the capital gain by comparing the sell price to the oldest buy price and sell shares until the sell order is fulfilled. Then, the program creates a new Transaction object with this information, adds it to the sell Deque, and updates the total capital gain.

```
else if (input.equals("S")) {
    System.out.println("Enter number of shares sold and unit price, separated by a
space:");
    String[] tokens = scanner.nextLine().trim().split(" ");
    if (tokens.length != 2) {
        System.out.println("Invalid input, please try again.");
        continue;
    }
    int shares = 0;
    double price = 0.0;
    try {
        shares = Integer.parseInt(tokens[0]);
        price = Double.parseDouble(tokens[1]);
    } catch (NumberFormatException e) {
        System.out.println("Invalid input, please try again.");
        continue;
    }
    if (shares <= 0 || price <= 0.0) {
        System.out.println("Invalid input, please try again.");
        continue;
    }
```

When the user enters a Q command, the program sells all remaining shares in the buy Deque and prints the total capital gain and remaining items (shares) in the buy Deque.

The quantity of unsold shares is indicated by the remainingShares variable. The gain variable shows the amount of capital gains or losses that have already been realized. As long as there are transactions in the queue (!buyQueue.isEmpty()) and shares still need to be sold (remainingShares > 0), the loop will keep running.

Inside the loop, the oldest transaction is retrieved using buyQueue.removeFirst(). This transaction is the one that has been held the longest according to the FIFO protocol. If the remaining shares are greater than or equal to the shares in the oldest transaction, then all of the shares in that transaction are sold. The number of remaining shares is updated by subtracting the number of shares in the oldest transaction, and the capital gain or loss is updated by multiplying the difference in price (price - oldestTransaction.price) with the number of shares in the oldest transaction (oldestTransaction.shares).

Only a portion of the shares in the oldest transaction are sold if the number of remaining shares is less than the shares in the oldest transaction. The remaining shares are subtracted to update the number of shares in the earliest transaction, and the price difference is multiplied by the remaining shares to update the capital gain or loss. Since all shares have been sold, the remainingShares variable is updated to 0. Since the oldest transaction still has some shares that have not been sold, it is then brought back to the front of the line.

After the loop has executed, the gain variable contains the total capital gain or loss based on the FIFO protocol for the given sequence of transactions.If the remaining shares are greater than or equal to the shares in the oldest transaction explanation...In this part of the code, the program checks if the remaining shares to be sold are greater than or equal to the shares in the oldest transaction in the buy queue.

```java
int remainingShares = shares;
double gain = 0.0;
while (remainingShares > 0 && !buyQueue.isEmpty()) {
    Transaction oldestTransaction = buyQueue.removeFirst();
    if (remainingShares >= oldestTransaction.shares) {
        // sell all shares in the oldest transaction
        remainingShares -= oldestTransaction.shares;
        gain += (price - oldestTransaction.price) * oldestTransaction.shares;
    } else {
        // sell only some shares in the oldest transaction
        oldestTransaction.shares -= remainingShares;
        gain += (price - oldestTransaction.price) * remainingShares;
        remainingShares = 0;
        buyQueue.addFirst(oldestTransaction);
    }
}
```

Using the remainingShares variable, the code inserts a conditional statement that determines whether there are enough shares to fulfill the sell order. If not enough shares are present, the transaction is canceled and a message to that effect is printed by the program.If there are sufficient shares, the program uses the addLast() method to add a new Transaction object to the sellQueue. The shares being sold at the specified price are represented by this Transaction object.

By deducting the purchase price of the shares being sold from the current price and multiplying the result by the quantity of shares being sold, the computer program determines the capital gain from the sale. The gain variable is increased by this gain.The gain variable is added to the totalGain variable by the program to calculate the total capital gain.A new conditional statement is added by the program if the user types the command "Q." By iterating over the buyQueue and adding the product of each transaction's price and shares to the gain variable, the program here determines the total capital gain from all the completed trades.

```java
    if (remainingShares > 0) {
        // not enough shares to fulfill the sell order
        System.out.println("Not enough shares to sell, transaction cancelled.");
    } else {
        sellQueue.addLast(new Transaction(shares, price));
```

```
            System.out.printf("Transaction recorded, capital gain: $%.2f%n", gain);
            totalGain += gain;
        }
} else if (input.equals("Q")) {
    double gain = 0.0;
    while (!buyQueue.isEmpty()) {
        Transaction = buyQueue.removeLast();
        gain += transaction.price * transaction.shares;
    }
    System.out.printf("Total capital gain: $%.2f%n", totalGain);

    break;
} else {
    System.out.println("Invalid input, please try again.");
}
```

The Deque class is implemented as a nested public class with generic type T. The Deque has three fields: a Node object representing the head of the Deque, a Node object representing the tail of the Deque, and an integer representing the size of the Deque. The Node class is a private nested class that represents a single node in the linked list. Each Node object has a reference to its data, its previous Node object, and its next Node object.

```
public static class Deque<T> {
    private Node head; // pointer to the head of the deque
    private Node tail; // pointer to the tail of the deque
    private int size; // size of the deque

    // Node class to represent each item in the deque
    private class Node {
        private T data;
        private Node prev;
        private Node next;

        public Node(T data) {
            this.data = data;
            this.prev = null;
            this.next = null;
        }
    }

    // constructor to create an empty deque
    public Deque() {
        head = null;
        tail = null;
        size = 0;
    }
```

The Deque class has several methods for adding and removing items from the Deque, including addFirst, addLast, removeFirst, and removeLast. These methods all update the head, tail, and size fields of the Deque as needed. The Deque class also has several methods for checking the size and emptiness of the Deque, including isEmpty and size.

```
public void addFirst(T item) {
            Node newNode = new Node(item);
            if (head == null) {
                head = newNode;
                tail = newNode;
            } else {
                newNode.next = head;
                head.prev = newNode;
                head = newNode;
            }
            size++;
        }

        // method to add an item to the back of the deque
        public void addLast(T item) {
            Node newNode = new Node(item);
            if (tail == null) {
                head = newNode;
                tail = newNode;
            } else {
                newNode.prev = tail;
                tail.next = newNode;
                tail = newNode;
            }
            size++;
        }

        // method to remove and return the item at the front of the deque
        public T removeFirst() {
            if (head == null) {
                throw new NoSuchElementException();
            }
            T data = head.data;
            head = head.next;
            if (head == null) {
                tail = null;
            } else {
                head.prev = null;
            }
            size--;
            return data;
        }

        // method to remove and return the item at the back of the deque
        public T removeLast() {
            if (tail == null) {
                throw new NoSuchElementException();
            }
            T data = tail.data;
            tail = tail.prev;
            if (tail == null) {
                head = null;
            } else {
                tail.next = null;
            }
            size--;
            return data;
        }

        // method to check if the deque is empty
        public boolean isEmpty() {
            return size == 0;
        }

        // method to get the size of the deque
        public int size() {
            return size;
        }
    }
}
```

EXPLANATION AND EVALUATING THE APPROPRIATENESS OF THE ALGORITHMS AND THE DATA STRUCTURES USED TO IMPLEMENT

A nested class called Transaction and a general class called DequeT> are both present in the code snippet above. The Transaction class represents a transaction with share and price fields, whereas the DequeT> class is an implementation of a double-ended queue data structure with generic type T.

A doubly-linked list is the data structure employed by the DequeT> class. Two pointers, one to the node before it and one to the node after it, are also present at each node of the list along with the data. To the first and last nodes of the list, respectively, are pointed by the head and tail pointers. Using a variable size, the size of the deque is monitored.

The head pointer must be modified, which is a constant-time operation, so the time complexity of the addFirst() and removeFirst() methods is O(1). The methods addLast() and removeLast(), which involve changing the tail pointer, are equivalent. Because they only return the value of the size variable, the isEmpty() and size() methods both have a time complexity of O(1).

WHY DEQUEUE IS USED INSTEAD OF STACK OR QUEUE?

The use of a deque (Double Ended Queue) in this code makes sense as it needs to support the insertion and removal of elements at both ends of the queue.A regular queue would only allow elements to be inserted at the rear and removed from the front, which does not support the functionality required in this program.

This program uses the Deque implementation because it makes it simple to implement the FIFO (First-In-First-Out) protocol. We must identify the shares sold in the same order they were bought, as stated in the problem statement. We use the addLast() method to add the shares we buy to the end of the deque. Using the removeFirst() method, we remove shares sold from the front of the deque. By following the FIFO protocol, this makes sure that we are selling the shares that were bought first and in the order in which they were bought.

Using a standard queue would not have been appropriate for this issue because we cannot always be sure that we are removing the oldest elements when we remove items from the front of a queue. The oldest item in a queue is always at the front, but removing items from the front does not always mean that we are also removing the oldest items, especially if there have been additions to the queue's back. The FIFO protocol can therefore be correctly implemented by using a Deque that can add and remove elements from both ends.

A stack, on the other hand, allows only insertion and removal of elements from one end (top of the stack), but it does not meet the requirements of the program since the program needs to remove items from both the front and rear ends of the deque.Therefore, a deque is chosen for this program since it allows for elements to be inserted and removed from both ends.

TASK 2A

A simulation of a card game is provided by this application. In order to construct an array of Player objects, the application first asks the user how many players will be participating in the game.

In order to prompt the user to enter the number of players, the program employs a while loop until a proper input is provided (i.e., 2, 4, or 13). Also, a while loop is utilized by the software to keep dealing cards until each and every card has been dealt.

Each object in the array serves as a player in the game. Once a normal deck of cards has been created, shuffled, and dealt to each player, the program ends. The game continues until all cards have been played after the program urges each participant to play a card of a specified suit.

A Player class is defined by the software, and it has fields for the player's name and card hand. The Player class also has methods for adding cards to the player's hand,taking cards out of the player's hand, playing cards of a certain suit, and sorting the player's hand.

A number of exception handling statements are also used by the application to deal with probable issues. The application will raise a NumberFormatException and request the user to enter a valid input, for instance, if the user inputs an invalid value for the number of participants. A player will be prompted to play a legal card if the computer detects an attempt to play a card that is not in their hand and throws an IllegalArgumentException.

This stimulation uses 3 classes Player class ,Positional List class and Card class

The program defines the Player Java class, which symbolizes a participant in a card game. The class has methods for adding cards to the player's hand, playing cards from the hand, iterating over the cards in the hand, and sorting the hand by suit and rank. A constructor for the class is also available, which initializes the player's hand after receiving a player name.

The hand of the player is represented by the code as a list of Card objects in a PositionalList data structure. The methods in the Player class manipulate this data structure to add, remove, and iterate over the cards in the hand.

The Card class is defined with two properties, rank and suit, that correspond to the rank and suit of a playing card. In its constructor, the associated properties are initialized by taking a rank and a suit. Moreover, it offers two getter methods for retrieving the card's rank and suit. A string representation of the card is produced using the toString() method by translating the integer rank to a string value (for example, "Jack" for 11) and the suit character to a string value (for example, "Clubs" for 'C').

EXPLANATION AND EVALUATING THE APPROPRIATENESS OF THE ALGORITHMS AND THE DATA STRUCTURES USED TO IMPLEMENT

A doubly linked list serves as the data structure in this implementation. With references to the nodes before and after it, as well as the card's rank and suit, each node in the list represents a playing card.

The use of a doubly linked list is appropriate for this implementation because it enables efficient node addition and removal, as well as forward and backward iteration. Additionally, the doubly linked list makes it simple to navigate the list in both directions, which is required for iterating over cards of a particular suit.

The two iterator classes, CustomIterator and SuitIterator, are suitable for this implementation because they enable flexible iteration over the list's nodes. While the SuitIterator allows iteration over nodes with a specific suit, the CustomIterator allows iteration over all nodes in the list. Both classes have a time complexity of O(n), as they need to visit every element of the list to iterate over all the cards.

```
class CustomIterator {  // iterate over all cards in the hand.
    Node current; //Node object representing the current position of the iterator.

    CustomIterator(Node start) {
        current = start;
    }  //object representing the starting position of the iterator.

    boolean hasNext() {
        return current != trailer;
    }
```

```
class SuitIterator {  // used to iterate over all cards of a given suit that are
currently in the hand.
    Node current;
    char suit;


    SuitIterator(Node start, char suit) {
        this.suit = suit;
        current = start;
        while (current != trailer && current.card.suit != suit) {
            current = current.next;
        }
    }
```

To add new cards to the list, use the addACard() method. The node is added to the list between the trailer.prev node and the trailer node by creating a new card object and new node object. This guarantees that the new node is added to the end of the list and that its previous and next pointers are correctly updated. This method has a time complexity of O(1), as it performs a constant number of operations to add a new element to the list.

```
void addACard(int rank, char suit) {  //adds a new card with the given rank and suit to
the hand
    // represented by the PositionalList.
    Card = new Card(rank, suit); // creates a new Card object using the given rank and
suit.
    Node = new Node(card, trailer.prev, trailer); //it creates a new Node object with the
    // Card object, trailer.prev (the node before the trailer), and trailer as its
previous,
    // current, and next nodes, respectively.
    trailer.prev.next = node;
    trailer.prev = node;
    size++;  //The Node is inserted between the trailer.prev node and the trailer node by
updating
    // the next and prev pointers of the adjacent nodes.
}
```

A specific card can be eliminated from the list using the removeCard() method. It moves through the list, comparing each card to the target card, looking for a match. Then, by altering the previous and next pointers of the neighboring nodes, it removes the corresponding node. This method has a time complexity of O(n), as it needs to visit every element of the list to find the node that stores the given card.

```
void removeCard(Card card) {
    Node current = header.next;
    while (current != trailer) {
        if (current.card.equals(card)) {
            Node prevNode = current.prev;
            Node nextNode = current.next;
            prevNode.next = nextNode;
            nextNode.prev = prevNode;
            size--;
            break;
        }
        current = current.next;
    }
```

The list can be represented in a string by using the toString() method, which iterates through every node in the list and adds each node's string representation to a StringBuilder object. . This method has a time complexity of O(n), as it needs to visit every element of the list to construct the string representation.

```
public String toString() {
    StringBuilder sb = new StringBuilder();
    Node current = header.next;
    while (current != trailer) {
        sb.append(current.card.toString());
        if (current.next != trailer) {
            sb.append(", ");
        }
        current = current.next;
    }
    return sb.toString();
}
```

WHY LINKED LIST IS USED WITHOUT ARRAYLISTS?

The sequence of cards in a hand are represented by the code using a doubly linked list data structure, with each node carrying a Card object to represent a single card in the hand. This data structure is suitable for the presented issue as it enables card insertion and removal in constant time O(1). To traverse the linked list and retrieve all cards in the hand or all cards of a specific suit, the code employs two iterator classes, CustomIterator and SuitIterator. In order to efficiently traverse the linked list without having to reveal the underlying implementation details, iterators are a suitable solution for this issue.

ArrayList is not suitable for this Because they prohibit constant time insertion and removal at any arbitrary index, arraylists are ineffective for this issue. Every element to the right of the insertion point must be moved by one place when an element is inserted into an array list in order to make room for the new element. The elements to the right of the removal point in an ArrayList must all be moved by one position to fill the void when an element is removed from the list. The worst-case time required for this shifting operation, where n is the number of members in the ArrayList, is O(n). ArrayLists are inapplicable since the issue necessitates the continual insertion and removal of cards.

TASK 2B

In this task the code is a straightforward spell checker that reads user input, compares the spelling of the entered word to a dictionary file, and suggests alternative words if the entered word is misspelled.

When the user is asked to enter a word, the program first uses a SpellChecker class to check the spelling of the word if it is not "exit." The SpellChecker class examines a dictionary file containing a list of acceptable words to see if the entered word matches any of them. When a word is entered and checked against a dictionary, the program prints "Word is correct." Otherwise, the program displays the message "Word is incorrect." and provides a list of synonyms using the SpellChecker class. Until the user types "exit," the program will repeatedly ask them to enter a word.

EXPLANATION AND EVALUATING THE APPROPRIATENESS OF THE ALGORITHMS AND THE DATA STRUCTURES USED TO IMPLEMENT

The code creates a simple spellchecker using a binary tree data structure. During construction, it reads words from a file and inserts them into the binary tree. When a word is passed to the spellChecker method, the binary tree is searched to see if the word is present in the tree. If the word is found, it is returned correctly spelled. If the word cannot be found, a list of suggestions is returned after searching the binary tree for nodes that contain words that are similar to the input word.

The binary tree data structure is appropriate for use in the implementation of a spellchecker due to its efficient word search capabilities. It typically takes O(log n) time to search for a word in a binary tree, where n is the number of nodes in the tree. This is quicker than word searches, which typically take O(n) time. Furthermore, putting a word into a binary tree only takes O(log n) time, which is faster than doing the same for a list.

The binary tree is implemented using an associated word and a node class with left and right children. This method of implementing binary trees is widely used. When using the insert method, a word is recursively inserted into the left or right child of the current node, depending on whether it is less than or greater than the word linked to the current node. In order to insert the word into the binary tree, this compares it to the word that is linked to the current node. The search method recursively looks for the given word in the left or right child of each node in the binary tree, depending on whether the search term is less than or greater than the word of the current node.

The words read from the file are momentarily stored in the ArrayList data structure by the SpellChecker class before being added to the tree. Due to its support for dynamic scaling and easy access to elements, the ArrayList data structure is suitable. When reading a sizable text file, the ArrayList's size could drastically grow and use up plenty of memory.

```
BufferedReader reader = new BufferedReader(new FileReader(filename)); //which reads the
specified file.
String line = null;
List<String> words = new ArrayList<>(); //creates a new ArrayList called words to store
the words read from the file.

while ((line = reader.readLine()) != null) {////reads each line of the file and adds it
to the words ArrayList
    // after trimming any whitespace.
    words.add(line.trim());
}
```

The insert() method contrasts the word to be entered with the word at the current node before iteratively putting each word into the tree. The word is added to the right subtree if it is longer than the word in the current node or the left subtree if it is shorter. In the event that a node has a null value, the process should stop and a new node should be generated.

```
private Node insert(Node node, String wordtocheck) {
    if (node == null) { //f the Node is null, it creates a new Node with the input word
        return new Node(wordtocheck);//return the input word
    }
    int compare = wordtocheck.compareTo(node.word); //compares the input word with the
Node's word
    if (compare < 0) {
        //determine whether to insert the word to the left or right of the current nod
        node.left = insert(node.left, wordtocheck);
    } else if (compare > 0) {
        node.right = insert(node.right, wordtocheck);
    }
    return node; //returns the node it was passed.
}
```

The spellChecker() method invokes the search() method to check the input word's spelling before using it. In the event that the input word is identified, it is returned and added to the results list. If that doesn't happen, the searchNodes() method is then employed to hunt for words that are comparable to the input word. This approach scans the tree structure to locate nodes that include words that are similar to the input term. If a word in a node shares any letters with a word in the list of nodes, that node is added to the list. The isSwapped(), isInserted(), isDeleted(), and isReplaced() methods can all be used to detect which of the four categories of spelling errors the input word includes.

```
private boolean isSwapped(String wordtocheck1, String wordtocheck2) {
        if (wordtocheck1.length() != wordtocheck2.length()) {//checks if the lengths of
the 2 words are different.
            return false;//so they can't be swapped so returns false
        }
        char[] chars1 = wordtocheck1.toCharArray(); //converts string to character
        char[] chars2 = wordtocheck2.toCharArray();
        int count = 0;//variable will be used to count the number of characters that are
different between the two input Strings.
        for (int i = 0; i < chars1.length; i++) { //oop that goes through each character
```

```
in the chars1 array.
            if (chars1[i] != chars2[i]) {
                //checks if the character at index i in chars1 is different from the
character at the same index in chars2.
                count++;// characters are different, the count variable is incremented.
            }
            if (count > 2) { //This checks if the count variable has exceeded 2. If it
has,
                // it means that more than two characters are different between the two
Strings,
            }
        }
        return count == 2;
    }

    private boolean isInserted(String wordtocheck1, String wordtocheck2) {
        if (Math.abs(wordtocheck1.length() - wordtocheck2.length()) != 1) {//length of
the two strings differs by exactly one character
            return false;//if its not returns false
        }
        int i = 0;
        int j = 0;
        int count = 0;
        while (i < wordtocheck1.length() && j < wordtocheck2.length()) { // iterates over
the characters of both strings
            if (wordtocheck1.charAt(i) != wordtocheck2.charAt(j)) {
                //checks whether the characters at index i in wordtocheck1 and j in
wordtocheck2 differ
                count++;//that means character inserted
                if (count > 1) { //check whether more than one character has been
inserted.
                    return false;
                }
                if (wordtocheck1.length() > wordtocheck2.length()) { //ncrement either i
or j, depending on which
                    // string is longer.

                    i++;
                } else {
                    j++;
                }
            } else {
                i++;
                j++;
            }
        }
        return true; //it has not found more than one difference between the two strings
        //wordcheck2 can be obtained by wordcheck1 by inserting a character
    }

    private boolean isDeleted(String wordtocheck1, String wordtocheck2) {
        return isInserted(wordtocheck2, wordtocheck1);
        //returns the result of calling the isInserted method with the parameters
reversed.
    }

    private boolean isReplaced(String wordtocheck1, String wordtocheck2) {
        if (wordtocheck1.length() != wordtocheck2.length()) {//checks if the two words
have different lengths
            return false;
        }
        int count = 0;
        for (int i = 0; i < wordtocheck1.length(); i++) {
            //starts a loop that iterates through the characters in the two words.
            if (wordtocheck1.charAt(i) != wordtocheck2.charAt(i)) {
                // increments the counter if the characters at the current position are
different.
                count++;
            }
            if (count > 1) {
                //returns false if the counter is greater than 1, meaning more than one
character was replaced.
                return false;
            }
        }
        return count == 1;
    }
}
```

Due to its efficient searching capabilities (O(log n) time complexity) and ability to store and access words in sorted order, BST is an ideal data structure for storing and searching for words in dictionaries. The code also incorporates the essential features that check for misspelled words and offer potential corrections. Similarly, the search operation for locating a misspelled word traverses the tree and is O(log n) sized.

The algorithm for inserting a word requires traversing the tree in order to find the ideal location to insert the new word, so it has an O(log n) time complexity. Due to the similar tree traversal required, the search operation for locating a misspelled word is also O(log n).

The number of nodes in the tree and the length of the input word both affect how time-consuming the spellchecker approach is. If the input word is k characters long and the tree is a chain, the worst-case time complexity is O(k * n), where n is the number of nodes in the tree. The input word will be located at the root node, and the time complexity for the best scenario will be O. (1). The typical time complexity is O(log n) where n is the number of nodes in the tree.

CONCLUSIONS

In conclusion task 1 has used a dequeue data structure using linked list which it makes it simple to implement the FIFO (First-In-First-Out) protocol. Then task 2A uses positional list using doubly linked list.This data structure is suitable for the presented issue as it enables card insertion and removal in constant time O(1). Then task 2B uses binary search tree. The binary tree data structure is appropriate for use in the implementation of a spellchecker due to its efficient word search capabilities and it is done faster.

REFERENCES

DEQUEUE USING DOUBLY LINKED LIST

https://www.youtube.com/watch?v=0UkQnQAt4lo

DOUBLY LINKED LIST

https://www.youtube.com/watch?v=bQlw5tJM6_0&list=PLt4nG7RVVk1gIcVQAo8laecQWkzOdYe6i

BINARY SEARCH TREE

https://www.youtube.com/watch?v=zIX3zQP0khM&t=37s