**iTASK 1**

public class CapitalGainCalculator {  
  
 public static void main(String[] args) {  
 Deque<Transaction> buyQueue = new Deque<Transaction>(); //initialize Dequeue with max size 100  
 Deque<Transaction> sellQueue = new Deque<Transaction>(); //initialize Dequeue with max size 100  
 Scanner = new Scanner(System.*in*);  
 String input;  
 double totalGain = 0.0;  
  
 while (true) {  
 System.*out*.println("Enter B for buying transaction, S for selling transaction or Q to exit:");  
 input = scanner.nextLine().trim().toUpperCase();  
  
 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]);  
 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;  
 }  
 buyQueue.addLast(new Transaction(shares, price));  
 System.*out*.println("Transaction recorded.");  
 } 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;  
 }

The program uses two Deques (double-ended queues), buyQueue and sellQueue, to store the buy and sell transactions, respectively. It prompts the user to enter a character 'B' for buying a transaction, 'S' for selling a transaction, or 'Q' to exit the program.

If the user enters 'B', the program prompts the user to enter the number of shares bought and the unit price, separated by a space. It then checks the validity of the input by ensuring that there are exactly two tokens, both of which can be parsed into an integer and a double, respectively, and are greater than zero. If the input is invalid, the program prints an error message and prompts the user to try again. If the input is valid, the program creates a new Transaction object with the specified number of shares and unit price and adds it to the end of the buyQueue.

If the user enters 'S', the program follows a similar process to the 'B' case, but instead creates a new Transaction object with the specified number of shares and unit price and adds it to the end of the sellQueue.If the user enters 'Q', the program exits the while loop.

After the while loop is exited, the program calculates the total capital gain (or loss) by iterating through the sellQueue. For each sell transaction, it iterates through the buyQueue until the specified number of shares have been sold, using the FIFO protocol. For each buy transaction used to fulfill a sell transaction, the program calculates the capital gain (or loss) by subtracting the unit price of the buy transaction from the unit price of the sell transaction and multiplying the result by the number of shares sold. If the buy transaction was used to fulfill multiple sell transactions, its capital gain (or loss) is added to a running total. If the sell transaction was not completely fulfilled by the buy transactions in the buyQueue, the program prints an error message and exits. Finally, the program prints the total capital gain (or loss).

This part of the code handles the case when the user inputs "S" to indicate a selling transaction. It first prompts the user to enter the number of shares sold and the unit price, which are then stored in the tokens array after being split by spaces and trimmed to remove any leading/trailing whitespace.

Next, the code checks if the length of tokens is equal to 2, meaning that both the number of shares sold and the unit price have been entered. If the length is not equal to 2, the code prints an error message and continues with the loop to prompt the user for input again.If the length is valid, the code attempts to parse the values of shares and price from the tokens array using the parseInt() and parseDouble() methods respectively. If either of these operations fail, meaning that the input is not a valid integer or double, the code prints an error message and continues with the loop to prompt the user for input again.

Finally, if both shares and price are successfully parsed and are greater than zero, the code creates a new Transaction object with these values and adds it to the sellQueue deque using the addLast() method. The code then prints a message to indicate that the transaction has been recorded.

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);  
 }  
}

The remainingShares variable represents the number of shares that are yet to be sold. The gain variable represents the capital gain or loss that has been realized so far. The loop continues to execute as long as there are shares yet to be sold (remainingShares > 0) and there are transactions in the queue (!buyQueue.isEmpty()).

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 tran saction, 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).

If the remaining shares are less than the shares in the oldest transaction, then only some of the shares in the oldest transaction are sold. The number of shares in the oldest transaction is updated by subtracting the remaining shares, and the capital gain or loss is updated by multiplying the difference in price with the remaining shares. The remainingShares variable is updated to 0, since all shares have been sold. The oldest transaction is then added back to the front of the queue, since it still has some shares that have not been sold.

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.If the remaining shares are greater than or equal to the shares in the oldest transaction, it means that all shares in the oldest transaction can be sold.

In this case, the program subtracts the shares in the oldest transaction from the remaining shares to be sold and calculates the gain from the sale.The gain from the sale is calculated by taking the difference between the selling price and the price of the oldest transaction and then multiplying it by the number of shares sold in that transaction. The gain is then added to the total gain.This process is repeated until either all the shares have been sold or the buy queue is empty.

int remainingShares = shares;

This line initializes a variable called remainingShares to the total number of shares to be sold. This variable is used later in the loop to keep track of how many shares are left to be sold.

double gain = 0.0;

This line initializes a variable called gain to 0.0. This variable is used to keep track of the total capital gain (or loss) so far.

while (remainingShares > 0 && !buyQueue.isEmpty()) {

This line starts a while loop that will execute as long as there are remaining shares to sell and there are transactions left in the buyQueue.

Transaction oldestTransaction = buyQueue.removeFirst();

This line removes the oldest transaction from the buyQueue and stores it in a variable called oldestTransaction. The removeFirst() method removes and returns the first element in the buyQueue.

if (remainingShares >= oldestTransaction.shares) {

This line checks if the number of remaining shares is greater than or equal to the number of shares in the oldest transaction. If this is true, all shares in the oldest transaction can be sold.

remainingShares -= oldestTransaction.shares;

gain += (price - oldestTransaction.price) \* oldestTransaction.shares;

These two lines subtract the number of shares in the oldest transaction from remainingShares, and then calculate the capital gain (or loss) for those shares. This gain is added to the total gain variable.

} else {

oldestTransaction.shares -= remainingShares;

gain += (price - oldestTransaction.price) \* remainingShares;

remainingShares = 0;

buyQueue.addFirst(oldestTransaction);

These lines sell only some shares in the oldest transaction, and then add the remaining part of the transaction back to the buyQueue.

The capital gain (or loss) for the sold shares is calculated and added to the total gain variable. The remainingShares variable is set to 0 because all shares have been sold.

The while loop continues until all the remaining shares have been sold or the buyQueue is empty. At the end of the loop, the total capital gain (or loss) for the entire sequence of transactions is stored in the gain variable.

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.");  
 }  
 System.*out*.println();  
 }  
}

if (remainingShares > 0) {

This condition checks if there are any remaining shares that need to be sold. If there are, the program prints a message saying that there aren't enough shares to fulfill the sell order, and the transaction is cancelled. If there aren't any remaining shares, the program adds the transaction to the sell queue.

sellQueue.addLast(new Transaction(shares, price));

This line adds a new transaction to the sell queue, which presumably contains all of the shares that are available for sale. The Transaction class seems to represent a single transaction of buying or selling a certain number of shares at a certain price.

System.out.printf("Transaction recorded, capital gain: $%.2f%n", gain);

This line prints a message indicating that the transaction was recorded, along with the capital gain or loss from the transaction. However, the variable gain doesn't seem to be defined anywhere in this code snippet, so it's unclear where it comes from.

totalGain += gain;

This line adds the gain or loss from the current transaction to a running total of the total capital gain or loss for all transactions processed so far.

} else if (input.equals("Q")) {

This condition checks if the user has entered the letter "Q", which presumably stands for "quit" or "exit". If so, the program calculates the total capital gain by iterating over all of the transactions in the buy queue and summing up the gains from each one. The gains are calculated by multiplying the number of shares by the price for each transaction. The program then prints the total capital gain and the remaining items in the buy queue.

while (!buyQueue.isEmpty()) {

This line starts a loop that iterates over all of the transactions in the buy queue. The loop continues until the buy queue is empty.

Transaction = buyQueue.removeLast();

This line removes the last transaction from the buy queue and assigns it to a variable called transaction.

gain += transaction.price \* transaction.shares;

This line calculates the gain or loss from the current transaction by multiplying the price by the number of shares, and adds it to the running total gain.

System.out.printf("Total capital gain: $%.2f%n", totalGain);

This line prints the total capital gain or loss for all transactions processed so far.

break;

This line exits the while loop and ends the program.

} else {

If the user enters any other input besides "S" or "Q", the program prints a message saying that the input is invalid.

System.out.println("Invalid input, please try again.");

This line prints an error message indicating that the user input is invalid.

}

This curly brace ends the if/else block that processes user input.

System.out.println();

This line prints a blank line, presumably for formatting purposes.

**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**

public String toString() {  
 String rankStr;  
 String suitStr;  
 switch (rank) {  
 case 11:  
 rankStr = "Jack";  
 break;  
 case 12:  
 rankStr = "Queen";  
 break;  
 case 13:  
 rankStr = "King";  
 break;  
 case 14:  
 rankStr = "Ace";  
 break;  
 default:  
 rankStr = String.*valueOf*(rank);  
 }  
 switch (suit) {  
 case 'C':  
 suitStr = "Clubs";  
 break;  
 case 'D':  
 suitStr = "Diamonds";  
 break;  
 case 'H':  
 suitStr = "Hearts";  
 break;  
 case 'S':  
 suitStr = "Spades";  
 break;  
 default:  
 suitStr = "";  
 }  
 return rankStr + "-" + suitStr;

* The class overrides the **toString()** method from the **Object** class. This method returns a string representation of a **Card** object.
* Inside the **toString()** method, two local variables **rankStr** and **suitStr** are declared to store the string representations of the rank and suit, respectively.
* A **switch** statement is used to assign the appropriate string representation to **rankStr** based on the value of **rank**. If **rank** is 11, the string "Jack" is assigned to **rankStr**. If **rank** is 12, the string "Queen" is assigned, and so on. If **rank** doesn't match any of the cases, the **default** case is executed, and the string representation of **rank** is assigned to **rankStr** using **String.valueOf(rank)**.
* Another **switch** statement is used to assign the appropriate string representation to **suitStr** based on the value of **suit**. If **suit** is 'C', the string "Clubs" is assigned to **suitStr**. If **suit** is 'D', "Diamonds" is assigned, and so on. If **suit** doesn't match any of the cases, the **default** case is executed, and an empty string is assigned to **suitStr**.
* Finally, the method returns the concatenation of **rankStr**, "-", and **suitStr**
* import java.util.\*;  
    
  public class Player {  
   String name; //represents the name of the player  
   PositionalList hand; //The player's hand is represented using a PositionalList  
   //The hand is sorted by suit and then by rank  
    
   Player(String name) { //constructor for the Player class  
   this.name = name; //takes a string argument name  
   hand = new PositionalList(); //initializes the instance variables name and hand.  
    
    
   }  
    
   void addCard(int rank, char suit) { //method takes an integer representing the rank of the card  
   // and a character representing the suit of the card.  
    
   hand.addACard(rank, suit); // It adds a new card with the given rank and suit to the player's  
   // hand.  
   }  
    
   Card playACard(char suit) { //method takes a character representing the suit of the card to be played.  
   if (hand.size == 0) { //checks if there are any cards left in the player's hand  
   throw new IllegalStateException("No cards left in hand");// If there are no cards left,  
   // it throws an exception.  
   }  
    
    
   PositionalList.SuitIterator iterator = hand.suitIterator(suit); //If there are cards left,  
   // it gets an iterator for all cards of the specified suit using the suitIterator() method of  
   // hand  
   if (iterator.hasNext()) {  
   Card = iterator.next(); //If there are cards of the specified suit, it removes  
   // the first card and returns it.  
   hand.removeCard(card);  
   return card;  
   } else {  
   throw new NoSuchElementException("No cards of that suit in hand"); // If there are no  
   // cards of the specified suit, it removes an arbitrary card and returns it.  
   }  
   }  
    
   java.util.Iterator<Card> iterator() {  
   return (Iterator<Card>) hand.iterator();  
   }  
   //This method returns an iterator for all cards currently in the player's hand.  
    
    
   PositionalList.SuitIterator suitIterator(char suit) {  
   return hand.suitIterator(suit);  
   }  
   //takes a character suit representing the suit of the cards to iterate over, and returns an  
   // iterator for all cards of that suit that are currently in the player's hand.  
    
    
   void finalDeal(List<Card> deals, int numCards) { //a list of Card objects named deals, and an  
   // integer numCards.  
   List<Card> allCards = new ArrayList<>(); //creates a new ArrayList object named allCards that  
   // will be used to store all the Card objects from the hand and the deals list.  
   PositionalList.CustomIterator iterator = hand.iterator(); //used to iterate over the Card  
   // objects in the hand list.  
   while (iterator.hasNext()) { //iterator object to loop through all the Card objects in the  
   // hand list  
   allCards.add(iterator.next()); // add them to the allCards list.  
   }  
   allCards.addAll(deals); //adds all the Card objects from the deals list to the allCards list.  
   Random = new Random();  
   for (int i = 0; i < numCards; i++) {  
   if (allCards.size() == 0) {  
   break;  
   }  
   int index = random.nextInt(allCards.size()); // select a random Card object from the allCards list.  
   Card = allCards.remove(index); //selected Card object is then removed from the allCards list  
   addCard(card.getRank(), card.getSuit()); // added to the hand using the addCard method  
   }  
   }  
    
   void sortHand() {  
   PositionalList sortedHand = new PositionalList(); //list used to sort the cards  
   for (char suit : new char[]{'C', 'D', 'H', 'S'}) { //uses a for each loop to assign each suit to  
   //variable suit  
   for (int rank = 2; rank <= 14; rank++) { //iterates over each rank from 2 to 14 (inclusive)  
   PositionalList.SuitIterator iterator = hand.suitIterator(suit);  
   //This line creates a new "SuitIterator" object for the current suit and assigns it to the variable "iterator"  
   while (iterator.hasNext()) { //iterates over each card in the hand with the current suit.  
   Card = iterator.next(); // gets the next card from the iterator and assigns it to the variable "card".  
   if (card.rank == rank) { // checks if the rank of the current card matches the rank being searched for in the outer loop.  
   sortedHand.addACard(rank, suit); // checks if the rank of the current card matches the rank being searched for in the outer loop.  
   hand.removeCard(card); // line removes the current card from the "hand" object.  
   }  
   }  
   }  
   }  
   hand = sortedHand; //replaces the original unsorted hand with the sorted one.  
    
    
   }  
    
   @Override  
   public String toString() {  
   return name + ": " + hand.toString();  
   }  
  }

The Player class is a Java class that represents a player in a card game. The player's hand is represented using a PositionalList, which is a custom data structure that represents a sequence of cards. The goal of the application is to support a person arranging a group of cards in hand so that cards of the same suit are kept together. The hand is sorted by suit and then by rank. The class includes methods to add a card to the hand, play a card from the hand, return an iterator for all cards currently in the hand, and return an iterator for all cards of a specific suit that are currently in the hand.

The addCard() method takes an integer representing the rank of the card and a character representing the suit of the card, and adds a new card with the given rank and suit to the hand.The playACard() method takes a character representing the suit of the card to be played. If there is a card of the specified suit in the hand, that card is removed from the hand and returned. If there is no card of the specified suit in the hand, an arbitrary card is removed from the hand and returned.

The iterator() method returns an iterator for all cards currently in the hand.

The suitIterator() method takes a character representing the suit of the cards to iterate over, and returns an iterator for all cards of that suit that are currently in the hand.

The finalDeal() method takes a list of Card objects representing the cards that have been dealt so far, and the number of cards that should be dealt to the player. It adds the dealt cards to a list of all cards and then randomly selects the specified number of cards from the combined list to add to the player's hand.

The sortHand() method sorts the player's hand by suit and then by rank. It does this by creating a new PositionalList called "sortedHand" and iterating over each suit and rank in turn, adding each card that matches the current suit and rank to the sorted hand and removing it from the original hand.

The toString() method returns a string representation of the player's name and their current hand of cards.

import java.util.NoSuchElementException;  
  
  
public class PositionalList {  
 Node header;  
 Node trailer;  
 int size;  
  
 PositionalList() {  
 header = new Node(null, null, null);  
 trailer = new Node(null, header, null);  
 header.next = trailer;  
 size = 0;  
 }  
  
 // Adds a new card with rank r and suit s to the hand  
 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.  
 }  
  
 public int size() {  
 return size;  
 } //This method returns the size of the PositionalList.  
  
 // CustomIterator class  
 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;  
 }  
  
 Card next() {  
 if (!hasNext()) {  
 throw new NoSuchElementException("No more elements in the iterator");  
 }  
 Card = current.card;  
 current = current.next;  
 return card;  
 }  
 }  
  
 // SuitIterator class  
 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;  
 }  
 }  
  
 boolean hasNext() {  
 return current != trailer;  
 }  
  
 Card next() {  
 if (!hasNext()) {  
 throw new NoSuchElementException("No more elements in the iterator");  
 }  
 Card = current.card;  
 current = current.next;  
 while (current != trailer && current.card.suit != suit) {  
 current = current.next;  
 }  
 return card;  
 }  
 }  
  
 CustomIterator iterator() {  
 return new CustomIterator(header.next);  
 }  
  
 SuitIterator suitIterator(char suit) {  
 return new SuitIterator(header.next, suit);  
 }  
 //Node object representing the starting position of the iterator, and a char representing the  
 // suit of cards to iterate over  
  
 @Override  
 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();  
 }  
  
  
 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;  
 }  
 }  
}

This code implements a positional list for a card game where cards of the same suit should be kept together. The list consists of nodes, and each node contains a Card object, a reference to the previous node, and a reference to the next node. The Card object contains a rank and a suit.

The addACard method takes a rank and a suit and creates a new Card object with those values. It then creates a new node with the Card object as its data and adds the node to the end of the list (before the trailer node).

The CustomIterator class is an iterator for all cards currently in the hand. It contains a reference to the current node and has two methods: hasNext() and next(). The hasNext() method returns true if there are more cards in the list, and the next() method returns the next Card object in the list.

The SuitIterator class is an iterator for all cards of a given suit that are currently in the hand. It contains a reference to the current node and a suit value. It also has the hasNext() and next() methods. The next() method returns the next Card object in the list that has the same suit as the one passed to the constructor. If there are no more cards of that suit in the list, the iterator stops.

The iterator() method returns a CustomIterator object that starts at the first card in the list (after the header node).

The suitIterator(char suit) method returns a SuitIterator object that starts at the first card in the list with the given suit.The removeCard(Card card) method removes the given card from the list. It iterates through the list until it finds a node with a Card object that equals the given card. It then removes the node from the list by updating the previous and next nodes to point to each other, effectively removing the current node from the list.

header: A Node object representing the head of the list.

trailer: A Node object representing the tail of the list.

size: An integer representing the size of the list.

The PositionalList class has a constructor that initializes the header and trailer nodes and sets the size to zero. The header and trailer nodes are linked to each other to form an empty list.

The class also has the following methods:

addACard(int rank, char suit): Adds a new card with rank and suit to the hand represented by the PositionalList. The method creates a new Card object using the given rank and suit, and then creates a new Node object with the Card object, the node before the trailer, and the trailer node as its previous, current, and next nodes, respectively. The Node is inserted between the node before the trailer and the trailer node by updating the next and prev pointers of the adjacent nodes.

size(): Returns the size of the PositionalList.

iterator(): Returns a CustomIterator object that can be used to iterate over all cards currently in the hand.

suitIterator(char suit): Returns a SuitIterator object that can be used to iterate over all cards of suit that are currently in the hand.

removeCard(Card card): Removes the given Card object from the PositionalList.

The PositionalList class also has two nested classes: CustomIterator and SuitIterator.

CustomIterator is an iterator class that can be used to iterate over all cards in the hand. It has the following instance variables:

current: A Node object representing the current position of the iterator.

The CustomIterator constructor takes a Node object representing the starting position of the iterator. The hasNext() method returns true if there are more elements to iterate over, and false otherwise. The next() method returns the next Card object in the iteration.

SuitIterator is an iterator class that can be used to iterate over all cards of a given suit that are currently in the hand. It has the following instance variables:

current: A Node object representing the current position of the iterator.

suit: A char representing the suit of cards to iterate over.

The SuitIterator constructor takes a Node object representing the starting position of the iterator, and a char representing the suit of cards to iterateover. The constructor initializes the current position to the first node in the list that has the given suit. The hasNext() method returns true if there are more elements to iterate over, and false otherwise. The next() method returns the next Card object in the iteration. If there are no more cards of the given suit, the method returns the next card in the list regardless of its suit.

Finally, the PositionalList class overrides the toString() method to provide a string representation of the hand. It iterates over all the cards in the hand and adds their string representation to a StringBuilder object, separating them by commas.

**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.

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

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.

**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**

import java.io.\*;  
import java.util.\*;  
  
public class SpellChecker {  
 private Node root;  
  
 private class Node {  
 String word;  
 Node left, right;  
  
 public Node(String wordtocheck) {  
 this.word = wordtocheck; //The constructor takes a string called wordtocheck as input  
 this.left = null;  
 this.right = null;  
 }  
 }  
  
 public SpellChecker(String filename) throws IOException {  
 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());  
 }  
 reader.close();  
 for (String wordtocheck : words) {//iterates through each word in the words ArrayList and inserts it into the  
 // SpellChecker object by calling the insert method.  
 insert(wordtocheck);  
 }  
 }  
 public void insert(String wordtocheck) {  
 root = insert(root, wordtocheck);  
 }  
  
 private Node insert(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.  
 }  
  
 public List<String> spellChecker(String wordtocheck) {  
 List<String> results = new ArrayList<>();//returns a List of Strings that contains suggestions for correctly spelled  
 // words  
 if (search(wordtocheck)) {  
 // input word is found in the SpellChecker object, it is added to the results list and returned  
 results.add(wordtocheck);  
 return results;  
 }  
 List<Node> nodes = new ArrayList<>();  
 searchNodes(root, wordtocheck, nodes); //if word not found search nodes is called to see for similar words to the  
 //input word  
 for (Node : nodes) {  
 results.add(node.word); // adds their words to the results list  
 }  
 return results;  
 }  
  
 private void searchNodes(Node node, String wordtocheck, List<Node> nodes) {  
 //takes a Node object and a String called wordtocheck as input.  
 if (node == null) {  
 return;  
 //searches the SpellChecker object's tree structure to find Nodes that contain words similar  
 // to the input word.  
 }  
 if (isSwapped(wordtocheck, node.word) || isInserted(wordtocheck, node.word) || isDeleted(wordtocheck, node.word) || isReplaced(wordtocheck, node.word)) {  
 ////checks if the input word is one of four types of spelling errors  
 nodes.add(node);  
 //If a node's word contains a similar word to the input word, it adds the node to the list of nodes  
 }  
 searchNodes(node.left, wordtocheck, nodes);  
 searchNodes(node.right, wordtocheck, nodes);  
 }  
  
 private boolean search(String wordtocheck) { //takes the words as input and returns a node object that has the input word  
 Node = search(root, wordtocheck);  
 //compares the input word to the current node word  
 if (node == null) {  
 return false;  
 }  
 return true;  
 }  
  
 private Node search(Node, String wordtocheck) { //if words there searches the tree structure  
 if (node == null) {  
 return null;  
 }  
 ////compares the input word to the current node word  
 int compare = wordtocheck.compareTo(node.word);  
 if (compare < 0) {  
 return search(node.left, wordtocheck); //If the input word is less than the current Node's word calls itself  
 //on the left child of the node  
 } else if (compare > 0) {  
 return search(node.right, wordtocheck); //if greater calls itself on the right child  
 } else {  
 return node;  
 }  
 }  
  
 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 it’s 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;  
 }  
}

The code represents a basic implementation of a spell checker program that uses a tree data structure to store and organize words from a text file.

The tree data structure used is a binary search tree, where each node in the tree represents a word, and the left and right child of each node represent words that are lexicographically less and greater than the node's word, respectively.

The program loads a file of words into the tree by inserting each word into the tree with the insert method, which recursively traverses the tree and places the word in the correct position based on its lexicographical ordering. The program then provides spelling suggestions for a given word by searching for words in the tree that are similar to the given word.

The searchNodes method is called to traverse the tree and find nodes that contain words that are similar to the given word. This method checks for four types of spelling errors: swapping adjacent characters, inserting an additional character, deleting a character, and replacing a character. If a node's word contains a similar word to the input word, the node is added to a list of nodes. The program then returns a list of words that are contained in thenodes in the list.

The tree structure provides an efficient way to store and search for words, making the spell checker program fast and effective.

public class SpellChecker {  
 private Node root;  
  
 private class Node {  
 String word;  
 Node left, right;  
  
 public Node(String wordtocheck) {  
 this.word = wordtocheck; //The constructor takes a string called wordtocheck as input  
 this.left = null;  
 this.right = null;  
 }  
 }  
  
 public SpellChecker(String filename) throws IOException {  
 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());  
 }  
 reader.close();  
 for (String wordtocheck : words) {//iterates through each word in the words ArrayList and inserts it into the  
 // SpellChecker object by calling the insert method.  
 insert(wordtocheck);  
 }  
 }

* The **SpellChecker** class has a private member variable **root**, which represents the root node of a binary tree.
* Inside the **SpellChecker** class, there is an inner class **Node** representing a node of the binary tree. Each **Node** has a **word** (representing a word to be checked), as well as **left** and **right** pointers to its left and right child nodes.
* The **Node** class has a constructor that takes a string parameter **wordtocheck**. It initializes the **word** with the value of **wordtocheck**, and sets both **left** and **right** pointers to **null**.
* The **SpellChecker** constructor takes a **filename** as a parameter. It reads the contents of the file specified by **filename**.
* It creates a **BufferedReader** to read the file using **FileReader** and initializes a **String** variable **line** to **null**.
* It creates an **ArrayList** called **words** to store the words read from the file.
* It enters a loop that reads each line from the file using the **readLine()** method of the **BufferedReader**. The loop continues until there are no more lines to read (**line = reader.readLine()** returns **null**).
* Inside the loop, it trims any leading and trailing whitespace from the line using the **trim()** method and adds the trimmed line (word) to the **words** list.
* After reading all the lines from the file, it closes the **BufferedReader** to release system resources.
* Finally, it iterates through each word in the **words** list using an enhanced **for** loop and calls the **insert** method, passing each word as an argument, to insert the word into the binary tree.

public void insert(String wordtocheck) {  
 root = insert(root, wordtocheck);  
}  
  
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 **insert** method takes a string parameter **wordtocheck** and is responsible for inserting a new word into the binary tree.
* The public **insert** method serves as an interface to the private **insert** method that performs the actual insertion.
* The public **insert** method assigns the root of the binary tree to the result of the private **insert** method.
* The private **insert** method takes a **node** parameter (representing the current node in the binary tree) and the **wordtocheck** to be inserted.
* If the **node** is **null**, indicating an empty subtree, a new **Node** is created with **wordtocheck** and returned.
* If the **node** is not **null**, the **wordtocheck** is compared with the **word** stored in the current **node**.
* If the **wordtocheck** is lexicographically smaller than the current **node.word**, the **insert** method is called recursively on the left child of the current node.
* If the **wordtocheck** is lexicographically greater than the current **node.word**, the **insert** method is called recursively on the right child of the current node.
* The method returns the current **node**, whether it was newly created or an existing node, to maintain the correct tree structure.

public List<String> spellChecker(String wordtocheck) {  
 List<String> results = new ArrayList<>();//returns a List of Strings that contains suggestions for correctly spelled  
 // words  
 if (search(wordtocheck)) {  
 // input word is found in the SpellChecker object, it is added to the results list and returned  
 results.add(wordtocheck);  
 return results;  
 }  
 List<Node> nodes = new ArrayList<>();  
 searchNodes(root, wordtocheck, nodes); //if word not found search nodes is called to see for similar words to the  
 //input word  
 for (Node node : nodes) {  
 results.add(node.word); // adds their words to the results list  
 }  
 return results;  
}

* The **spellChecker** method takes a **wordtocheck** parameter and returns a list of suggestions for correctly spelled words.
* It first creates an empty **ArrayList** called **results** to store the suggestions.
* It calls the **search** method to check if the **wordtocheck** is already present in the binary tree. If it is found, the **wordtocheck** is added to the **results** list and returned.
* If the **wordtocheck** is not found in the binary tree, it creates an empty **ArrayList** called **nodes** to store the nodes with similar words.
* It then calls the **searchNodes** method, passing the root node, **wordtocheck**, and the **nodes** list as parameters, to find nodes with words similar to the **wordtocheck**.
* After the **searchNodes** method returns, it iterates through the **nodes** list and adds the **word** of each node to the **results** list.
* Finally, it returns the **results** list containing the suggestions for correctly spelled words.

private boolean search(String wordtocheck) { //takes the words as input and returns a node object that has the input word  
 Node node = search(root, wordtocheck);  
 //compares the input word to the current node word  
 if (node == null) {  
 return false;  
 }  
 return true;  
}

* The **search** method takes a string parameter **wordtocheck** and checks if the word exists in the binary tree.
* It calls the private **search** method, passing the **root** node and the **wordtocheck**.
* The private **search** method is responsible for traversing the binary tree to find the node that contains the **wordtocheck**.
* If the **node** returned by the private **search** method is **null**, it means the word was not found in the binary tree, so the method returns **false**.
* If the **node** is not **null**, it means the word was found, so the method returns **true**.

private Node search(Node node, String wordtocheck) { //if words there searches the tree structure  
 if (node == null) {  
 return null;  
 }  
 ////compares the input word to the current node word  
 int compare = wordtocheck.compareTo(node.word);  
 if (compare < 0) {  
 return search(node.left, wordtocheck); //If the input word is less than the current Node's word calls itself  
 //on the left child of the node  
 } else if (compare > 0) {  
 return search(node.right, wordtocheck); //if greater calls itself on the right child  
 } else {  
 return node;  
 }  
}

* The **search** method is a private recursive method that takes a **node** parameter (representing the current node in the binary tree) and the **wordtocheck** to be searched.
* If the **node** is **null**, indicating an empty subtree or the end of the tree, the method returns **null** to indicate that the word was not found.
* If the **node** is not **null**, the **wordtocheck** is compared with the **word** stored in the current **node** using the **compareTo** method.
* If the **wordtocheck** is lexicographically smaller than the current **node.word**, the **search** method is called recursively on the left child of the current node.
* If the **wordtocheck** is lexicographically greater than the current **node.word**, the **search** method is called recursively on the right child of the current node.
* If the **wordtocheck** is equal to the current **node.word**, it means the word was found in the binary tree, and the method returns the current **node**.
* The recursive calls continue until either the word is found and returned as a **node**, or until the end of the tree is reached, returning **null**.

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;  
}

* The **isSwapped** method takes two strings, **wordtocheck1** and **wordtocheck2**, as parameters.
* The method first checks if the lengths of the two words are different. If they are different, it means the words cannot be swapped, so the method returns **false**.
* If the lengths of the words are the same, the method continues to compare the characters of the two words.
* The method converts the two input words into character arrays using the **toCharArray** method.
* It initializes a variable **count** to keep track of the number of characters that are different between the two input words.
* The method then iterates over each character in the **chars1** array using a for loop.
* For each character at index **i**, it checks if the character at the same index in **chars1** is different from the character at the same index in **chars2**.
* If the characters are different, the **count** variable is incremented.
* After checking all characters, the method checks if the **count** variable has exceeded 2. If it has, it means that more than two characters are different between the two words. In this case, the method returns **false** because the words cannot be considered as swapped.
* Finally, if exactly two characters are different between the two words, the method returns **true** indicating that the words can be considered as swapped versions of each other.

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  
}

* The **isInserted** method takes two strings, **wordtocheck1** and **wordtocheck2**, as parameters.
* The method first checks if the absolute difference in length between the two words is exactly 1. If the difference is not 1, it means that more or less than one character has been inserted, so the method returns **false**.
* If the length difference is 1, the method initializes two variables **i** and **j** to keep track of the current index in **wordtocheck1** and **wordtocheck2**, respectively.
* It also initializes a variable **count** to keep track of the number of differences encountered.
* The method then enters a loop that continues as long as both **i** and **j** are within the bounds of their respective word lengths.
* For each iteration, the method checks if the characters at index **i** in **wordtocheck1** and index **j** in **wordtocheck2** are different.
* If the characters are different, it means that a character has been inserted. The **count** variable is incremented.
* If the **count** variable exceeds 1, it means that more than one character has been inserted, and the method returns **false**.
* If the lengths of **wordtocheck1** and **wordtocheck2** are different, the method increments either **i** or **j** depending on which word is longer.
* If the characters at the current indices are the same, both **i** and **j** are incremented.
* After checking all characters, if the **count** variable is still 1, it means that only one character has been inserted, and the method returns **true**.
* If the loop completes without finding more than one difference between the two words, it means that **wordtocheck2** can be obtained from **wordtocheck1** by inserting a single character.

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

* The **isDeleted** method simply calls the **isInserted** method with the parameters **wordtocheck2** and **wordtocheck1** reversed.
* The **isInserted** method, as explained earlier, checks if one word can be obtained from another word by inserting a single character.
* By reversing the parameters, the **isDeleted** method essentially checks if one word can be obtained from another word by deleting a single character.
* Since the logic for deletion and insertion is the same (except for the order of the parameters), calling **isInserted(wordtocheck2, wordtocheck1)** effectively checks for deletion.

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;  
 }  
}

* The **isReplaced** method takes two words, **wordtocheck1** and **wordtocheck2**, as parameters.
* First, the method checks if the lengths of the two words are different. If they are different, it means that more than one character would need to be replaced to transform one word into another, so the method returns **false**.
* If the lengths are the same, the method initializes a variable **count** to keep track of the number of differences encountered.
* The method then iterates through the characters of both words using a loop.
* For each character at index **i**, the method checks if the characters at the corresponding positions in **wordtocheck1** and **wordtocheck2** are different.
* If they are different, the **count** variable is incremented to indicate a replacement.
* If the **count** variable exceeds 1, meaning more than one character has been replaced, the method returns **false**.
* After checking all characters, if the **count** variable is 1, it means that only one character has been replaced, and the method returns **true**.
* If the loop completes without finding more than one difference, it means that **wordtocheck2** can be obtained from **wordtocheck1** by replacing a single character.

**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

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

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.

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.