Artificial Intelligence and Data Science

2022 1st year semester 2

Data structures and Algorithms CM1605

20221333-Esandu Obadaarachchi RGU-2237041

Module leader – Ms. Malsha Fernando

Table of Contents

1. Executive Summery

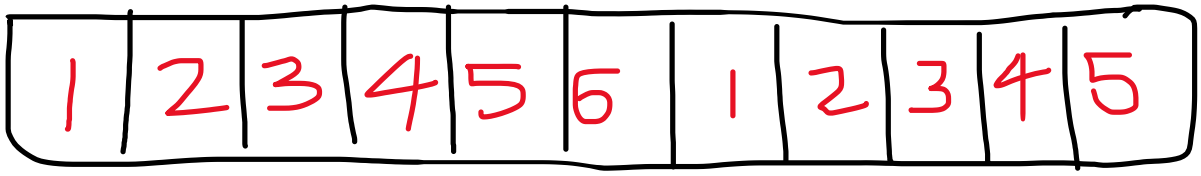
This report provides a in detail report about the 3 tasks in the coursework. This report goes in depth about the explanations for the implementations and the justifications for the use of the appropriate data structures and the algorithms. The time complexities are also compared to identify the most efficient data structure and algorithm.

This report contains TASK 1 – STOCK PROFIT CALCULATOR
TASK 2A – CARD SIMULATOR
TASK 2B – SPELL CHECKER PROGRAM

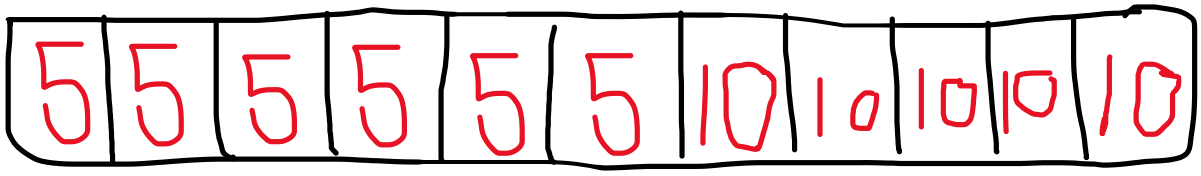
2. TASK 1 – STOCK PROFIT CALCULATOR

- In this question, since the stocks we bought FIRST are being sold FIRST, this program needs a FIFO data structure therefore a Queue data structure is used. The queue was implemented using Linked Lists as the first bought stocks are the ones that must be sold first.
- The concept used in this program is if the user buys 6 shares for 5 dollars each, and then the next day the user buys another 5 shares at 10 dollars each. The diagram below shows how the data is stored in the queue.
- There will be 2 queues used,
    1. Buying amount of shares Queue – to store no of shares
    2. Price of each stock Queue – to store price at which the user bought.

Buying amount of stock queue



Buying price of each stock queue



- The price queue stores the price of each individual share, which makes it easy when it comes to removing shares (selling). When it comes to selling shares, if the user is selling 5 shares from the above example the first 5 shares are removed and also in the price queue the first 5 elements are removed. The 5 elements removed from the price queue are added and stored in a variable called “moneyCounter” which is been used to calculate the profit after each sale.
  Profit = (selling price *selling amount of stock) – moneyCounter.
- All the user inputs are validated so that the user can enter only integers and the user has no restriction when it comes to how many days he buys or sells.

Implementation Explanation and justification

- The queue is implemented using Linked lists.
- When using a singly-linked list, adding an element to the rear takes O(1) time as we simply need to create a new node and update the "next pointer" of the last node to the new node we created. Similarly removing an element from the front of the queue takes O (1) time as we simply need to update the "next" pointer of the head node to the next behind node.
- If we compare the time complexities of linked list implementation of queues to arrays and arraylists, the time complexity of adding and deleting from the front and rear of the queue are different.
    - For arrays, adding an element to the rear of the queue takes O (1) time, but O (n) in the worst case when it comes to the point where the array needs to be resized. Removing an element from the front of the queue takes O (n) time as all the remaining elements need to be shifted one position to the left.
    - For arraylists, adding an element to the rear of the queue takes O (1) time on average, but O (n) in the worst case when it comes to the point where the array needs to be resized. Removing an element from the front of the queue takes O (n) time as all the remaining elements need to be shifted one position to the left.
    - Also implementing queues using 2 stacks take the same time as Arrays and arraylists.

- In this case, the advantage of using a linked list for the queue implementation is that adding and removing both take O (1) time, regardless of the size of the queue. This makes the linked list implementation more time efficient than the array and the arraylist, therefore using arrays and arraylists for larger queues is not efficient as removing takes 0(n) where n will be large.

The adding and removing queue implementation.

```
    4 usages
    public void enqueue(int data){
        Node newNode = new Node(data);
        if(head == null){
            head = newNode;
            return;
        }
        Node currentNode = head;
        while(currentNode.next !=null){
            currentNode = currentNode.next;
        }
        currentNode.next=newNode;
    }
    no usages
    public void printing(){
        if(head == null){
            System.out.println("list is empty");
        }
        Node currentNode = head;
        while(currentNode!=null){
            System.out.print(currentNode.data+"->");
            currentNode = currentNode.next;
        }
    }

    2 usages
    public int dequeue(){
        if(head == null){
            System.out.println("the list is empty");
            return 0;
        }
        size--;
        int removedItem = head.data;
        head = head.next;
        return removedItem;
    }

    4 usages
    public int gettingSize() { return size; }
}
```

// printing method is used to print the queue which can be used to see whether all the elements are being added properly

3. TASK 2 A – The card simulator program.

In this program, the method used to do the program is,

- there is an array list with the initial size of 52 called deck which stores all the cards in the deck.
- There is a class called Player where each player object has an arraylist called hand, with the capacity of 13 as the max a hand will get is 13 cards.
- In the player class, there are methods called playACard and addACard, which are being used to add and play cards from the player hand arraylist.
- There are two iterators used to print each card in the hand and one to check the suits of the cards in the hand to play.
- To sort the cards, each rank and suit has been given a value which is used to sort using Insertion sort.

<u>Implementation Explanation and justification</u>

- We have used the arraylist data Structure to implement the deck of cards and the player ’s hands. An array list is appropriate for the deck and hand as it provides dynamic resizing, random access, and efficient traversal. When adding an element at the end the time complexity is O (1), O (1) to access any element and inserting and deleting an element from the ends but from the middle it is O (n). in our program we are accessing the middle cards only when playing a card, so this makes arraylists an ideal choice for the data structure.

- Also, the arrraylist for the deck has been given a intial size of 52 which eliminates the need to re size when adding an element so the time complexity when adding an element will always be O (1).

- compared to other alternatives such as linked lists and arrays, arraylists are better for this program as array lists provide constant time access to any element within the list, as each element is stored in a contagious block of memory where in linked lists it can be slower as the element are not stored in the contagious memory. Therefore, the arraylist is relatively efficient for this program rather than the others.

- The sorting algorithm we have used is insertion sort to sort the cards in player's hand. Insertion sort Is appropriate for this as the worst-case time complexity Is O (n^2). Since we are sorting small array lists of size 13, insertion sort gives us efficient sorting. Therefore insertion sort is the most appropriate algorithm rather than more complex sorting algorithms like merge sort, quick sort and bubble sort.
- The iterator is used to iterate over the cards in the players hands. It is efficient and provides us with the methods like next, hasnext and remove.

The player class code

```
public class Player{
    private ArrayList<String> hand;

    public Player() { //constructor
        this.hand = new ArrayList<String>();
    }
    public void addACard(String rank,String suit){ //adding the card to the
player
        this.hand.add(rank+"-"+suit);
    }

    public String playACard(String suit){
//method to play a card where the suit has to be passed
        Iterator<String> suitIterator = new HandIterator();
        while (suitIterator.hasNext()) {
            String card = suitIterator.next();
            if (card.endsWith(suit)){
                System.out.println(" the card removed is " + card);
                suitIterator.remove();
                return card;
            }
        }
        String arbitraryCard = hand.remove(0);
        System.out.println(" the card removed is " + arbitraryCard);
        return arbitraryCard;
    }
    //setter and getter
    public void setHand(ArrayList<String> hand){
        this.hand = hand;
    }
    public ArrayList<String> getHand() {
        return hand;
    }
    //-----------------------------------------------------------------

    public String iterateToPrint() {            //this is used to iterate
through the hand and make all the cards to one string
        StringBuilder sb = new StringBuilder();
        for (String card : hand) {
            sb.append(card);
            sb.append(",");
        }
        if (sb.length() > 0) {                      //only does this if
hand is not empty
            sb.deleteCharAt(sb.length() - 1); // removes the last comma
```

```
        }
        return sb.toString();
    }

//=============================================================================


    private class HandIterator implements Iterator<String> {
        //iterator implementation
        private int currentIndex;

        public HandIterator() {
            this.currentIndex = 0;
        }

        public boolean hasNext() {
            return currentIndex < hand.size();
        }

        public String next() {
            return hand.get(currentIndex++);
        }

        public void remove() {
            hand.remove(currentIndex - 1);
            currentIndex--;
        }
    }
```

```
//insertion sort code

for (Player player : players) {
    ArrayList<String> hand = player.getHand();
    for (int i = 1; i < hand.size(); i++) {
        String key = hand.get(i);
        int j = i - 1;
        while (j >= 0 && comparingCards(hand.get(j), key) > 0) {
            hand.set(j + 1, hand.get(j));
            j--;
        }
        hand.set(j + 1, key);
    }
    player.setHand(hand);
}
```

- Key is the variable where the card currently being sorted is stored.
- The variable int j is the pointer, is the index of the element before the element being sorted. That index is used to get the element before to compare and re arrange.
- In the while loop, j should be greater than zero as when j becomes minus is when we have gone outside the loop from the front.
- The comparing cards method return the value when the two comparing cards are being subtracted. And that should be greater than zero which means that the card we are sorting currently is smaller than the before card and must go to the front. Pointer is used to check like that with all the cards before that card.

4. TASK 2 B – The spell checker program.

The method used in this is, there is a class called tree where a tree constructor is there where the text file name must be passed. Then all the words in the dictionary text file will be read and inserted to the tree.

The user is asked for the word and if the word is present straightaway a message is shown “the spelling is correct”. If the word is not present, there are 3 cases we check to see whether the word can be found. To do this string manipulation Is used.

```
//case 1
// Check if a word can be formed by adding one letter to another word
for (int i = 0; i <= word.length(); i++) {
    for (char c = 'a'; c <= 'z'; c++) {
        String newWord = word.substring(0, i) + c + word.substring(i);
        if (checkWord(newWord)) {
            boolean alreadyExists = false;
            for (String suggestion : suggestions) {
                if (suggestion.equals(newWord)) {
                    alreadyExists = true;
                    break;
                }
            }
            if (!alreadyExists) {
                suggestions.add(newWord);
            }
        }
    }
}
```

All the alphabet letters are added to each place of the word to check whether a letter is missing from the word. If not found it goes to case 2 and case 3.

```
//case 2
// Check if a word can be formed by deleting one letter from another word
for (int i = 0; i < word.length(); i++) {
    String newWord = word.substring(0, i) + word.substring(i + 1);
    if (checkWord(newWord)) {
        boolean alreadyExists = false;
// checking if the word is already added from the previous cases
        for (String suggestion : suggestions) {
            if (suggestion.equals(newWord)) {
                alreadyExists = true;
                break;
            }
        }
        if (!alreadyExists) {
            suggestions.add(newWord);
        }
```

```
    }
}
```

Each letter is deleted one by one and checked whether the word is present In the tree.
If the word is already there in the list, then it is not added again.

```
//case 3
// Check if a word can be formed by switching adjacent letters in another
word
for (int i = 0; i < word.length() - 1; i++) {
    String newWord = word.substring(0, i) + word.charAt(i + 1) +
word.charAt(i) + word.substring(i + 2);
    if (checkWord(newWord)) {
        boolean alreadyExists = false;      // checking if the word is
already added from the previous cases
        for (String suggestion : suggestions) {
            if (suggestion.equals(newWord)) {
                alreadyExists = true;
                break;
            }
        }
        if (!alreadyExists) { // if the word is not there, then we add to
our list
            suggestions.add(newWord);
        }
    }
}
```

A for loop is used to get the indexes to swap the letters 2 by 2 and each time it is checked whether the word is present in the tree.

Implementation Explanation and justification

```
public class Trees {

    static ArrayList<String> suggestions = new ArrayList<>();

    private static Node root; //the root node

    // Binary search tree node
    private static class Node {
        String data;
        Node left, right;

        public Node(String data) {
            this.data = data;
            left = right = null;
        }
    }
//=============================================================================
===
    // Insert words into binary search tree
    private Node insert(Node node, String data) {
        if (node == null) {
            node = new Node(data);
            return node;
```

```
        }
        if (data.compareTo(node.data) < 0)
            node.left = insert(node.left, data);
        else if (data.compareTo(node.data) > 0)
            node.right = insert(node.right, data);
        return node;
    }


//============================================================================
    public Trees(String filename) {
        // Load words from file into binary search tree
        try {
            Scanner scanner = new Scanner(new File(filename));
            while (scanner.hasNext()) {
                String word = scanner.next().toLowerCase();
                root = insert(root, word);
            }
            scanner.close();
        } catch (FileNotFoundException e) {
            System.out.println("Error: File not found");
        }
    }


//============================================================================
    // Check if word is spelled correctly
    public static boolean checkWord(String word) {
        return search(root, word.toLowerCase());

    }


//============================================================================
    // Search for word in binary search tree
    private static boolean search(Node node, String data) {
        if (node == null)
            return false;
        if (data.equals(node.data))
            return true;
        else if (data.compareTo(node.data) < 0)
            return search(node.left, data);
        else
            return search(node.right, data);
    }
```

this is the tree implementation where the class has the methods search (), insert(), checkWord() and getsuggestions().

In the constructor the scanner method is used to load the words to the text file. Exception handling is used to prevent any errors and program stopping.

The checkword method returns a Boolean value where its true If the word is found.

In the search method, first it checks whether the node is null, root null means the tree is empty and returns false. If not empty,

```
(data.compareTo(node.data) < 0)
```

If this is less than zero, the word is searched in the left side as the words are put according to the no.of characters in the word and vice versa.

- In this the data structure used is a Tree, the tree is more appropriate to use than arrays and array lists as they have a size and dynamic resizing worst case time complexity is O (n) and in this case since the size of the array to be made is not known, creating an arraylist with a greater size is not efficient interms of memory.
- The tree has a worst-case time complexity of O (n) when inserting if the tree is inbalanced, but if the tree is balanced the time complexity Is O (log n).
- Trees are faster than arraylists in this case as searching for a element in a tree is O (log n) whereas in arrays and arraylists it is O (n) as each element must be checked.
- Trees can handle any amount of data without the need for resizing whereas in arrays and array lists we don't know the no of elements and the array list could be really large, searching would be slower as n will be larger.
- Binary search trees are more memory efficient than arraylists for larger datasets as array lists use a contigious memory which can be insufficient for lager data sets.
- Therefore, we can say that trees are relatively more efficient than the other alternatives.

5. TASK 3 – The spell checker program with phonetic algorithm.

- The new code added to the program

```
public static ArrayList<String> gettingSuggestionsForWord(String word) {
    suggestions.clear();
    // Generating Soundex code for the misspelled word
    String soundexCodeForUserWord = Soundex.gettingCodeForTheWord(word);

    // Generate suggestions by checking with the dictionary Soundex code
for all the words
    Node node = root;
    while (node != null) {
        String dictionWord = node.data;
        // Generating Soundex code for the word in the dictionary file
        String dictionaryWordSoundex =
Soundex.gettingCodeForTheWord(dictionWord);
        // If the Soundex codes match, that means common words and added to
the list
        if (soundexCodeForUserWord.equals(dictionaryWordSoundex)) {
            suggestions.add(dictionWord);
        }
        // If the Soundex code of the misspelled word is less than the
Soundex code of the dictionary word, search the left subtree
        if (soundexCodeForUserWord.compareTo(dictionaryWordSoundex) < 0) {
```

```
            node = node.left;
        } else {
            node = node.right;
        }
    }
public class Soundex{
    public static String gettingCodeForTheWord(String word){
        char[] x = word.toUpperCase().toCharArray();


        char firstLetter = x[0];

        //Convert letters to numeric code
        for (int i = 0; i < x.length; i++) {
            switch (x[i]) {
                case 'B':
                case 'F':
                case 'P':
                case 'V': {
                    x[i] = '1';
                    break;
                }

                case 'C':
                case 'G':
                case 'J':
                case 'K':
                case 'Q':
                case 'S':
                case 'X':
                case 'Z': {
                    x[i] = '2';
                    break;
                }

                case 'D':
                case 'T': {
                    x[i] = '3';
                    break;
                }

                case 'L': {
                    x[i] = '4';
                    break;
                }

                case 'M':
                case 'N': {
                    x[i] = '5';
                    break;
                }

                case 'R': {
                    x[i] = '6';
                    break;
                }

                default: {
                    x[i] = '0';
                    break;
                }
```

```
            }
        }



        //Remove duplicates

        String output = "" + firstLetter;

        //RULE [ 3 ]
        for (int i = 1; i < x.length; i++)
            if (x[i] != x[i - 1] && x[i] != '0')
                output += x[i];


        //Pad with 0's or truncate
        output = output + "0000";
        return output.toLowerCase().substring(0, 4);
    }
```

EXPLANATION

A new class is made called soundex, It has a phonetic algorithm which is used to encode words into 4 character codes based on their pronounciation.

The class has a method in which a word can be passed and the 4 character code is returned.

Using switch case statements each letter in the alphabet is given a character and at the end using a for loop the 4-character code.

In the tree class, now when getting the suggestions, it is also checked whether the user entered word Soundex code is also equal to any Soundex code generated using the words in the dictionary.

Using a while loop (node!= null) each node left or right is checked by generating the code and if the generated code is equal to the user's word code then that word is added to the suggestions list.

The phonetic code generating method has a time complexity of O (n) where n is the no.of letters in the word.

6. Reference list

Insertion sort
college, a., n.d. [Online]
Available at: https://youtu.be/PkJIc5tBRUE
[Accessed 20 3 2023].

Stacks implementation and examples
college, a., n.d. [Online]
Available at: https://youtu.be/7m1DMYAbdiY
[Accessed 5 3 2023].

queues implementation and example
college, a., n.d. [Online]
Available at: https://youtu.be/va_6RmSrKCg
[Accessed 14 3 2023].

trees implementation and example
college, a., n.d. [Online]
Available at: https://youtu.be/-DzowlcaUmE
[Accessed 29 3 2023].

the phonetic substitutions implementation example
howtodoinjava, n.d. [Online]
Available at: https://howtodoinjava.com/algorithm/implement-phonetic-search-using-soundex-algorithm/
[Accessed 6 4 2023].