Assignment Three

Shannon Cordoni
Shannon.Cordoni@Marist.edu

November 6, 2021

1 Problem One: Searching

1.1 The Data Structure

Given a list of strings our job was to create an algorithm to go through this list and pick 42 random items and search for them using different sorting algorithms. These sorting algorithms included: Linear Search, Binary Search, and Hashing.

1.2 Main Class

1.2.1 Description

First we took the first 42 items in the word array and placed them in their own array to search for later. Then we called selection sort from assignment 2 and sorted the word array. After sorting we then called each of the searching methods to search for the 42 random words. First came linear search, then binary search and then hashing, after each of these methods were called we averaged the number of comparisons each took to find the 42 random words.

```

/*
 *
 * Assignment 3
 * Due Date and Time: 11/5/21 before 12:00am
 * Purpose: to implement searching and hashing, and to understand their performance.
 * Input: The user will be inputting a file containing a list of words/statements
 * Output: The program will sort them and search through them to find certain elements
 * @author Shannon Cordoni
 *
 */

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Arrays;
import java.util.Scanner;

public class Cordoni {

```

```java
    //Declare keyboard
    static Scanner keyboard = new Scanner(System.in);

    public static void main(String[] args) {

        //Declare and initialize variables
        String line;
        String[] wordarray = new String[666];
        String[] randomarray = new String[42];
        int[] hashValues = new int[666];
        NodeCordoni[] hashTable = new NodeCordoni [250];
        HashCordoni HashCordoni = new HashCordoni();

        int startindex = 0;
        int endindex = 665;

        int linearsum = 0;
        double linearaverage = 0.0;

        int binarysum = 0;
        double binaryaverage = 0.0;

        int hashsum = 0;
        double hashaverage = 0.0;

        //create new file object
        File myFile = new File("magicitems.txt");

        try
        {
            //create scanner
            Scanner input = new Scanner(myFile);
            line = null;

            int i = 0;

            //while there are more lines in the file it inputs them into a word array
            while(input.hasNext())
            {
                //Input into array
                wordarray[i] = input.nextLine();
                i++;
            }//while

            input.close();

        }//try

        //error for file not found
        catch(FileNotFoundException ex)
        {
          System.out.println("Failed to find file: " + myFile.getAbsolutePath());
        }//catch

        //Error in case of a null pointer exception
        catch(NullPointerException ex)
        {
            System.out.println("Null pointer exception.");
            System.out.println(ex.getMessage());
        }//catch

        //General error message
        catch(Exception ex)
        {
            System.out.println("Something went wrong");
```

```java
            ex.printStackTrace();
        }//catch

        //take the first 42 items before sorted and place them into an array to be searched
        //for later
        for (int i = 0; i < randomarray.length; i++){
            randomarray[i] = wordarray[i];
        }//for

        //pass array to selection sort to be sorted
        selectionSort(wordarray);

        //Call linear search to search for the 42 random items
        for (int i = 0; i < randomarray.length; i++){
            linearsum = linearsum + linearSearch(wordarray, randomarray[i]);
        }//for

        //find the average number of comparisons for linear search
        linearaverage = linearsum / randomarray.length;
        System.out.println("The Linear Search average is " + linearaverage);

        //Call binary search to search for the 42 random items
        for (int i = 0; i < randomarray.length; i++){
            binarysum = binarysum + binarySearch(wordarray, randomarray[i], startindex,
                                                    endindex);
        }//for

        //find the average number of comparisons for binary search
        binaryaverage = binarysum / randomarray.length;
        System.out.println("The Binary Search average is " + binaryaverage);


        //hashing

        //make hashcode for each string and place that hashcode in a new array
        for (int i = 0; i < wordarray.length; i++){
            hashValues[i] = HashCordoni.makeHashCode(wordarray[i]);
            //System.out.println(hashValues[i]);
        }//for

        //set hashcode so that it is not null
        for (int i = 0; i < hashTable.length; i++){
            hashTable[i] = new NodeCordoni();
        }//for


        //input the node containing the string to either start or continue the chain
        for (int i = 0; i < hashValues.length -1; i++){
            //System.out.println(hashValues[i]);
            //System.out.println(hashTable[hashValues[i]].getData());
            //System.out.println((HashCordoni.makeChain(wordarray[i])).getData());

            hashTable[hashValues[i]].setData((HashCordoni.makeChain(wordarray[i])).getData());

        }//for

        //print hash table
        for (int i = 0; i < hashTable.length - 1; i++){
            //System.out.println(hashTable[i].getData());
        }//for

        //Call hashing to search for the 42 random items
        for (int i = 0; i < randomarray.length; i++){
            hashsum = hashsum + hashsearch(wordarray, randomarray[i], hashValues, hashTable);
            //System.out.println(hashsum);
```

```java
        }//for

        //find the average number of comparisons for binary search
        hashaverage = hashsum / randomarray.length;
        System.out.println("The Hashing average is " + hashaverage);

    }//main

    //This method is the selection sort method that goes through and sorts the array using a
    //Big Oh of n squared
    public static String[] selectionSort(String[] wordArray)
    {

       //to loop through the array to determine the next smallest position
       for(int i = 0; i < wordArray.length - 2; i++){

            int smallpostion = i;
            numberOfSortComparisons++;

            //to loop through the array to to compare small position with the rest of the array
            for(int j = i + 1; j < wordArray.length - 1; j++){

                numberOfSortComparisons++;

                //compares to see if the value of j comes before the value of small position
                //in the alphabet
                if (wordArray[j].compareToIgnoreCase(wordArray[smallpostion]) < 0){
                    smallpostion = j;
                }//if

            }//for j

            //swap wordarray[i] with wordarray[smallpostion]
            if (wordArray[smallpostion]!= wordArray[i]){

                String temp = wordArray[i];
                wordArray[i] = wordArray[smallpostion];
                wordArray[smallpostion] = temp;

            }//if

       }//for i

       //System.out.println("Selection Sort Comparisons: " + numberOfSortComparisons);

       return(wordArray);
    }//selection sort

    //This method uses linear search to find the 42 items
    public static int linearSearch(String[] wordArray, String target)
    {
        int numberofLinearComparisons = 0;

        int index = 0;

        for (int i = 0; i < wordArray.length; i++){

            numberofLinearComparisons++;

            if (target.compareToIgnoreCase(wordArray[i])==0){
                i = index;
                return numberofLinearComparisons;
            }//if

        }//for
```

```java

        return numberofLinearComparisons;

    }//Linear Search

    //This method uses binary search to find the 42 items
    public static int binarySearch(String[] wordArray, String target, int startindex,
                                    int endindex)
    {
        int numberofBinaryComparisons = 0;
        int low = 0;
        int high = 0;
        int mid = 0;
        int temp = 0;

        low = startindex;
        high = endindex;


        while (low < high){
            mid = (low + high)/2;
            numberofBinaryComparisons++;
            if ( target.compareToIgnoreCase(wordArray[mid]) < 0){

                high = mid;
            }//if

            else {
                low = mid + 1;
            }//else

        }//while

        return numberofBinaryComparisons;

    }//Binary Search

    //This method uses hashing to retrieve the 42 items
    public static int hashsearch(String[] wordArray, String target, int[]hashValues,
                                NodeCordoni[] hashTable)
    {

        int numberofHashComparisons = 0;

        //Go through the hash table and search for the 42 items
        for ( int i = 0; i < hashValues.length; i++){

            numberofHashComparisons++;

            if((target.compareToIgnoreCase(hashTable[hashValues[i]].getData().toString())!=0))
            {

                return numberofHashComparisons;
            }//if

            else{
                hashTable[hashValues[i]].getNext();
            }//else

        }//for

        return numberofHashComparisons;

    }//Hash

```

```
}//Assignment3Cordoni
```

1.2.2 Description of Main Code

The code above is the code inside the main class, this code reads the magic items file into a word array, then we take the first 42 items of this array and place them into their own random word array to be searched for later. Then we call the selection sort method to sort the word array, and we call each searching method to search for the 42 items. First we call linear search, we do this by going through the array of random items and calling linear search for each item and adding up the value of comparisons returned, to then take the average of, then we do the same for binary search. To complete hashing we first call the *makeHashCode* method to create a hash code for each string in the word array and place each of these hash codes in their own array. Now we go through the hash table and set each index to a new Node so that we can input new nodes into the table. Then we go through the hash table and input each new node containing a string from the word array into the hash table at the intended index from the hash code or value. Now we can call the hashing method for each of the 42 items to search for them, and add up the value of comparisons returned, then take the average of them.

The *selectionsort* method takes in the word array and sorts it one index at a time with an asymptotic running time of O(n^2). Once the array is in order, it passes it back to the main method to then go on and search through using linear and binary search and hashing.

The *linearsearch* method takes in the word array and the target string to search for. It then goes through the word array and compares each string in the array with the target, adding up the number of comparisons along the way. The method then returns the number of comparisons it took to find the string. This method gets repeated for each of the 42 items in the array and the sum is taken of all the comparisons and divided by 42 to find the average of the linear search comparisons.

The *binarysearch* method takes in the word array, the target string to search for, and the start and end index of the word array. Then we set the start index to the temp variable low, and the end index to the temp variable high. While low is less then high we loop through the array and if the target value comes before the middle value then we set the high value to the mid, or we set the low value to be the middle value plus one. While we loop through we add up the number of comparisons to be passed back to main for the average to be taken.

The *hashsearch* method takes in the word array, the target string, the hash values, and the hash table. The method then goes through the hash table and compares the target string with the hashtable value at the given hash value index. While comparing these values the the comparisons are counted and then returned to then be averaged to find the overall hash comparison average.

1.3 Hashing Class

1.3.1 Description

This class contains the methods to create the hashcode for each value in the word array and create the chain to be stored at each index.

```

/**
 *
```

```
 * Assignment 3
 * Due Date and Time: 11/5/21 before 12:00am
 * Purpose: to implement searching and hashing, and to understand their performance.
 * @author Shannon Cordoni
 *
 */

import java.io.BufferedReader;
import java.io.FileReader;
import java.util.Arrays;

public class HashCordoni
{
   /**
    * Declare Variables
    */
    private  final String FILE_NAME = "magicitems.txt";
    private  final int LINES_IN_FILE = 666;
    private  final int HASH_TABLE_SIZE = 250;
    private static  NodeCordoni myHead;
    private static  NodeCordoni myTail;
    Cordoni Assignment3Cordoni = new Cordoni();

    //This method creates the hashcode for the string, courtesy of Professor Labouseur!
    public int makeHashCode(String str) {
        int hashTableSize = 250;
        str = str.toUpperCase();
        int length = str.length();
        int letterTotal = 0;

        // Iterate over all letters in the string, totalling their ASCII values.
        for (int i = 0; i < length; i++) {
            char thisLetter = str.charAt(i);
            int thisValue = (int)thisLetter;
            letterTotal = letterTotal + thisValue;

            // Test: print the char and the hash.
            /*
            System.out.print(" [");
            System.out.print(thisLetter);
            System.out.print(thisValue);
            System.out.print("] ");
            // */
        }//for

        // Scale letterTotal to fit in HASH_TABLE_SIZE.
        int hashCode = (letterTotal * 1) % hashTableSize;  // % is the "mod" operator

        return hashCode;
    }//make hash code

    //This method adds a node to the chain
    public NodeCordoni makeChain(String newword)
    {
        //this sets a temp variable to hold the current tail node
        NodeCordoni oldHead = myHead;

        //this sets the tail to be a new node and its data to be the new string
        myHead = new NodeCordoni();
        myHead.setData(newword);

        //This checks to see if the hash index is empty
        //if it is not empty then the old tail is set to now point to the new Node
        if (!isEmpty()){
            myHead.setNext(oldHead);
```

```
        }//if


        else{
            myHead = myTail;
        }//else

        return myHead;

    }//make chain

        //This checks to see if the queue is empty
        public static  boolean isEmpty()
        {
                boolean empty = false;

                if(myHead == null)
                        {
                        empty = true;
                        }//if
                return empty;
        }//empty

}//hashCordoni
```

1.3.2 Description of Hash Code

The code above is the code inside the Hash class, this class involves the methods *makeHashCode*, *makeChain*, and *isEmpty*.

The *makeHashCode* method takes in the string and totals up the ASCII values for each letter, and makes that the hash code for the string. This method then returns the integer value of the hashcode and returns it to then place the value in the array.

The *makeChain* method takes in the string and creates a new node to be added to the chain either containing only the new string or adding the new string to the beginning of the chain and then returns the head of the list.

The *isEmpty* method takes checks to see if the hash table is empty by checking to see if *myHead* is null.

1.4 Node Class

1.4.1 Description

For each word in the chain, a node was created so that we could traverse the linked list of words at each index easier. Each node's data was set to a string and it's next was set to the next string node in the list should a collision occur in the hash table.

```
/**
 *
 * Assignment 3
 * Due Date and Time: 11/5/21 before 12:00am
 * Purpose: to implement searching and hashing, and to understand their performance.
 * @author Shannon Cordoni
 *
 */
```

```java

public class NodeCordoni
{
    /**
     * Instance Variable for word data and node
     */
    private String myData;
    private NodeCordoni myNext;

    /**
     * The default Constructor for NodeCordoni
     */
    public NodeCordoni()
        {
        myData = new String();
        myNext= null;
        }//Node Cordoni

    /**
     * The full constructor for NodeCordoni
     * @param newData the incoming data of the item
     */
    public NodeCordoni(String newData)
        {
        myData = newData;
        myNext = null;
        }//NodeCordoni

    /**
     * the setter for the item data
     * @param newData the incoming data of the item
     */
    public void setData(String newData)
        {myData = newData;} //set data

    /**
     * The getter for the item data
     * @return the incoming data of the item
     */
    public String getData()
        {return myData;}//get data

    /**
     * The setter for the node
     * @param NewNext the incoming node data
     */
    public void setNext(NodeCordoni newNext)
        {myNext = newNext;}//set Node

    /**
     * the getter for the node
     * @return the incoming node data
     */
    public NodeCordoni getNext()
        { return myNext;}//get node

}//NodeCordoni
```

1.4.2 Description of Node Code

This code for the Node Class was created by in class lessons but also previous knowledge from Software Development 1. Using the same set up each node was created so that it consisted of a string and a myNext linking each node to the next. Getters and setters were created for both the nodes themselves and the data inside of them so that we would be able to call $node.getNext()$, $node.setNext()$, $node.getData()$, and $node.setData()$ in the hash class and main class to create the chain of nodes at each index in the hash table.

1.5 Overall:

Overall, these searching methods were successful in implementation and effective in searching for the 42 items. To go through each of these searches we can create a table for better data understanding, this table will show each search and their asymptotic running time:

Binary Search	Linear Search	Hashing
O($log(n)$)	O(n)	O($1 + averageChainLength$) or O(n)

The table above shows a quick understanding of the searching methods used here. To go into more detail Binary Search has an asymptotic running time of O($log(n)$) this is because this search involves going through sorted data rather than unsorted data. Along with that once a number is selected for comparison, if the target value falls say below that selected number then everything above is eliminated from the search. Thus, allowing for the data to be searched through faster than linear search. Then moving on to Linear Search, which has a running time of O(n), this is because linear search involves looking through a list of unsorted data. Along with that each element in said list is checked sequentially, so linear search goes through each element of the list until a match is found. This can lead to an average search time of O($n/2$), then we remove the constant for O(n). Moving on to Hashing, or Hashing with chaining specifically, this has a asymptotic running time of O(n) for searching through the table, and an asymptotic running time of O(1) for insertion into the hash table. To dive in deeper to searching, this could have an asymptotic running time of O(1), however this is very hard to accomplish. It is more along the lines of O(1) $+$ the average chain length for a total asymptotic running time of O(n). Overall, this was a successful implementation of three different types of searching.