Assignment Three

Tyler Hernandez

Tyler.Hernandez1@marist.edu

November 4, 2022

This page left blank intentionally.

1 Main Method

```
import java.util.Random;
import java.text.DecimalFormat;

class Main {

    public static int linearCounter;
    public static int binaryCounter;
    public static int hashingTotalComparisons;

    // Driver for Assignment 3.
    public static void main(String[] args) throws Exception {
        Reader reader = new Reader("./magicitems.txt");

        // Characters we will ignore when reading from the file.
        char[] ignoreList = {};

        // Holds each char of a line
        char[] line = reader.getNextLineOfChars(ignoreList, true);

        // Holds each line as a string
        String[] fullText = new String[0]; // Start at 0 length in case given an empty list.

        // Until we've reached the end of the file, keep adding each line to our array.
        // Even empty lines in the file will have at least a '\n' character.
        while (line.length > 0) {
            // Though, we must ignore '\n' characters manually in order to keep looping
            // based on if there is a character in the next line.
            if (line[line.length - 1] == '\n') {
                // it seems that '\n' characters only show up at the end of the array (and
                // not in the last line).
                line = Utils.removeLastElementOfArray(line);
            }

            // Expand array by one every time we add a new value to it.
            fullText = Utils.expandArrayByOne(fullText);
            // Takes line of characters and puts them into fullText as a concatted string.
            fullText[fullText.length - 1] = String.valueOf(line);

            // Grab the next line
            line = reader.getNextLineOfChars(ignoreList, true);
        } // ends while

        // Unsorted Original Text.
        String[] ORIGINAL_TEXT = fullText;

        // Sort fullText.
        fullText = insertionSort(fullText);

        // Pick 42 random addresses in magicitems.txt
        String[] randomAddresses = new String[42];
        Random rand = new Random();
        for (int i = 0; i < 42; i++) {
            randomAddresses[i] = ORIGINAL_TEXT[rand.nextInt(0, ORIGINAL_TEXT.length)]
            .toUpperCase();
        }

        // Linear search with our random addresses!
        for (int i = 0; i < randomAddresses.length; i++) {
            // Linear search for each address.
            linearSearch(randomAddresses[i], fullText);
        }

        // Binary search with our random addresses!
```

```
64          for (int i = 0; i < randomAddresses.length; i++) {
65              // Binary search for each address.
66              binarySearch(randomAddresses[i], fullText, 0, fullText.length);
67          }
68
69          System.out.println("Linear Search Average: " + new DecimalFormat("#.##")
70          .format((double) linearCounter / 42));
71          System.out.println("Binary Search Average: " + new DecimalFormat("#.##")
72          .format((double) binaryCounter / 42));
73
74          HashTable htable = new HashTable();
75
76          // Append each magic item to our hashtable.
77          for (String item : ORIGINAL_TEXT) {
78              htable.storeHashFromString(item);
79          }
80
81          // Retrieve our 42 items from hashtable.
82          for (String item : randomAddresses) {
83              int currentHashCode = HashTable.makeHashCode(item);
84
85              // Plug currentHashCode into htable to find its row.
86              int currentComparisons = htable.arr[currentHashCode].findItem(item) + 1;
87              // 1 for the hash.
88              hashingTotalComparisons += currentComparisons;
89
90          }
91
92          System.out.println("Hashing Retrieval Average: "
93                  + new DecimalFormat("#.##").format((double) hashingTotalComparisons / 42));
94
95          // // Debug: print out our hashtable.
96          // for (LinkedList list : htable.arr) {
97          // System.out.println(list);
98          // }
99
100     }
```

1.1 Main Method

The goal of this assignment is to count how many comparisons each form of searching takes. Ultimately, we can see that hashing is the superior method of retrieval, followed by binary search and linear search. Below are the outputs for one run of this assignment.

Linear Search	347.31 comparisons
Binary Search	9.26 comparisons
Hashing Retrieval	3.19 comparisons

Inside of this main method, we call linear search for 42 random items, binary search on these same 42 items, and implement hashing on these 42 items and count their comparisons taken to estimate the time complexity of these storing and retrieving conventions.

1.2 Linear Search

```
1     // Performs a linearSearch for a target string over a given array. Returns index
2     // in array of where target is found.
3     public static int linearSearch(String targetString, String[] givenArray) {
4         int foundAddress = -1; // Initializes at -1 in case not found.
5
6         for (int i = 0; i < givenArray.length; i++) {
7             linearCounter++;
8
9             if (givenArray[i].equals(targetString)) {
10                foundAddress = i;
11                break;
12            }
13        }
14        return foundAddress;
15    }
```

Linear search gave me 347.31 comparisons on average for 42 numbers. This makes sense because the algorithm is O(n), however the expected time is N/2. With 666 items in our list of magic items, 333 is our expected time. This is because we can expect to find half of our numbers before we reach the halfway mark going from start to end.

1.3 Binary Search

```
    // Performs a binary search for a target string over a given Array, considering
    // min and max bounds for our recursive calls to focus on. Returns index of
    // where targetString exists in array or -1 if not found.
    public static int binarySearch(String targetString, String[] givenArray,
     int minAddress, int maxAddress) {
        int foundAddress = -1; // Initializes at -1 in case not found.
        int midPoint = (int) ((minAddress + maxAddress) / 2);

        // left and right pointer have not crossed.
        if (minAddress <= maxAddress) {
            binaryCounter++;

            // Item found, end recursive calls.
            if (givenArray[midPoint].equals(targetString)) {
                binaryCounter++;
                foundAddress = midPoint;
            }

            // Not in this part of the array, narrow down array bounds by 1/2 (lowering the
            // cieling).
            else if (givenArray[midPoint].compareTo(targetString) > 0) {
                return binarySearch(targetString, givenArray, minAddress, midPoint - 1);
            }

            // Not in this part of the array, narrow down array bounds by 1/2 (increasing
            // the floor).
            else {
                return binarySearch(targetString, givenArray, midPoint + 1, maxAddress);
            }
        }
        return foundAddress;

    }
```

Binary search gave me 9.26 comparisons for the same content as tested on linear search (Average for 42 random address searches). Because the algorithm is O(log2 (n)), this expected comparison count for this would be 9.38 for 666 items. The binary search algorithm is convenient as it allows us to throw out half of our list of numbers during every iteration, narrowing down our searching very quickly. The downside to this algorithm, however, is it must be sorted in order to be able to narrow down our search so efficiently.

1.4 Insertion Sort

```
  // Insertion Sort from previous assignment.
    public static String[] insertionSort(String[] arr) {
        for (int ptr1 = 1; ptr1 < arr.length; ptr1++) {
            String key = arr[ptr1]; // record our character to copy over.
            int ptr2 = ptr1 - 1;

            // Moves over every character greater than our key by one.
            while (ptr2 >= 0 && arr[ptr2].compareTo(key) >= 0) {
                arr[ptr2 + 1] = arr[ptr2];
                ptr2--;
            }
            arr[ptr2 + 1] = key; // paste over (insert) our recorded character.
        }
        return arr;
    }
```

This was my implementation of Insertion Sort in the previous assignment. I used this to perform a binary search, because binary search will only work if the data is sorted. See assignment 2 for more information.

1.5 The Linked List

```
class LinkedList {

    private Node head;
    private Node tail;
    public int length;

    public LinkedList() {
        head = null;
        tail = head;
        length = 0;
    }

    // Appends given node to tail.
    public void append(Node node) {
        // First append will have a null head and tail.
        if ((tail == null) && (head == null)) {
            head = node;
            tail = head;
        } else {
            tail.setNext(node);
            tail = node;
        }
        length++;
    }

    // Removes head from linked list.
    public Node removeHead() {
        // Holds head to be dequeued.
        Node temp = head;
        // Sets the new head to be the upcoming first element.
        head = head.getNext();
        length--;
        return temp;
    }

    public boolean isEmpty() {
        if (length == 0)
            return true;
        else
            return false;
    }

    public Node getHead() {
        return this.head;
    }

    // Finds a given node in our linked list and returns how many comparisons taken.
    // Returns -1 if not found.
    public int findItem(String str){
        int comparisons = 0;
        boolean isFound = false;

        Node n = head; // grab our pointer to the head of the list.
        if (n == null){
            return -1;
        }

        // From head to tail, search the list for our desired item.
        while (n!= null){
            comparisons++;
            // If we find our desired node, flag isFound and break
            if (n.getData().equals(str)){
                isFound = true;
```

```
64                  break;
65              }
66
67              n = n.getNext();
68          }
69
70          if (isFound) {
71              return comparisons;
72          }
73          return -1;
74
75
76
77      }
78
79      public String toString() {
80          String str = "Head- ";
81
82          // Retrieves head to loop through LinkedList without modifying it.
83          Node tempNode = head;
84          while (tempNode != null) {
85              str += (tempNode + ", ");
86              tempNode = tempNode.getNext();
87          }
88          return str + "- Tail";
89      }
90  }
```

In order to have pointers to the head and tail of my linked list, I've decided to create a linked list class. This was derived off of my queue from a prior assignment, however a critical function I made is the findItem(str). This function finds a specific node in our linked list through grabbing the head and traversing it. For this assignment, we wanted to see how many comparisons were taken, therefore the function either returns -1 if the item is not found, or the amount of comparisons taken when it is found.

1.6 The Hash Table

```
public class HashTable {

  // Hash Table uses chaining.

  LinkedList arr[];
  private static final int ROWS = 250;

  HashTable(){
      arr = new LinkedList[ROWS]; // Creates a new array with rows length.
  }

  // Takes in a string, creates a hash for it, then stores it in our array.
  void storeHashFromString(String str) {
      int hashCode = makeHashCode(str);

      // Use hashCode address in arr to store our given str.
      // arr[hashCode] = str;
      storeStringWithChaining(hashCode, str);;
  }

  // Properly inserts string into hashCode address by implementing chaining.
  void storeStringWithChaining(int hashCode, String str){

      Node n = new Node(str); // Wrap string in a node to insert into LinkedList.

      // Store our new node in the linked list at hashCode address.
      if (arr[hashCode] == null){
          arr[hashCode] = new LinkedList();
          arr[hashCode].append(n);
      } else {
          arr[hashCode].append(n);
      }

  }

  // Creates a hashCode given a string.
  public static int makeHashCode(String str) {
      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("] ");
           * //
           */
      }

      // Scale letterTotal to fit in ROWS.
      int hashCode = (letterTotal * 3) % ROWS;

      return hashCode;
  }

```

```
64
65 }
```

The HashTable class uses an array of linked lists for hashing. Instead of just having a two dimensional array, this linked list allows us to append to our array with the append function. It does this in constant time, thanks to our pointer to both the head and the tail (we only really need one or the other, but this makes re-usability much more convenient). Inside of our hashtable class, we have a makeHashCode() which creates a hash code for a given string, then scales it down to fit into our array. Next, we have a storeStringWithChaining() which takes in a hashcode and a given string to append our new node (derived from our string) at a given hashcode address. Finally, we have a storeHashFromString() that calls the prior two functions to give us the desired use of our hashtable; creating a hash for our string, and storing the string at that hash location. Ultimately, hashing is a convenient way of storing and retrieving information in constant time.