Assignment Three

Shannon Brady

shannon.brady2@Marist.edu

November 1, 2022

1 Linear Search

1. The Linear Search class has the following attributes:
   a) numComparisons: The number of comparisons required to complete a single search
   b) comparisonTotal: A provisional attribute used to track the total number of comparison over a certain number of searches. This is used to calculate the average number of comparisons.
2. Line 16-27: Traverse a given array. If the target element is found, assign the associated index to foundIndex and break. Increment the number of comparisons for each loop iteration. Return the index of the target element if found; otherwise, return -1.
3. Line 31-33: Print the number of comparisons for each completed search.
4. Line 36-38: Print the average number of comparisons after a given number of searches.
5. Running Time Analysis: The asymptotic running time of linear search is O(n), given that the worst-case scenario is the target element being the last element in the array. This scenario would required a complete traversal of an array of n elements. The expected running time is roughly O(n/2), since it is most likely that the target element does not exist at either extreme but rather somewhere closer to the midpoint. However, since constant factors are considered to be negligible, this would still be considered O(n) running time.

```
1  class LinearSearch {
2
3      int numComparisons = 0;
4      double comparisonTotal = 0;
5
6      LinearSearch(){
7          this.numComparisons = 0;
8          this.comparisonTotal = 0;
9      }
10
11     public int search(String[] array, String target) {
```

```
12          this.numComparisons = 0;
13          int i = 0;
14          int foundIndex = -1;
15
16          while (i < array.length) {
17              // Loop through entire array until target element is found
18              if (array[i].compareTo(target.toLowerCase()) == 0) {
19                  foundIndex = i;
20                  break;
21              }
22              this.numComparisons++;
23              i++;
24          }
25          // Track the total number of comparisons to find an average later
26          this.comparisonTotal += this.numComparisons;
27          return foundIndex;
28      }
29
30      // Print the number of comparisons for each search
31      public void printNumComparisons() {
32          System.out.println("\nNumber of Comparisons: " + this.numComparisons);
33      }
34
35      // Print the average number of comparisons after a given number of searches
36      public void printAvgComparison(int total) {
37          System.out.printf("\nAverage Number of Comparisons:  %.2f %n", this.comparisonTotal/total);
38      }
39  }
```

2 Binary Search

1. The Binary Search class has the following attributes:
   a) numComparisons: The number of comparisons required to complete a single search
   b) comparisonTotal: A provisional attribute used to track the total number of comparison over a certain number of searches. This is used to calculate the average number of comparisons.
2. Line 13: Find the midpoint index based on the given start and end values.
3. Line 14: Increment the number of comparisons for each initiated search.
4. Line 16-17: If the target element is equivalent to the element at the midpoint index, return the midpoint index.
5. Line 18-19: If the target element is lexicographically less than the element at the midpoint index, initiate another search from the start index to the index preceding the midpoint.
6. Line 20-21: Otherwise, initiate another search from the index following the midpoint index to the end index.
7. Line 23: Return the index of the target element if found; otherwise, return -1.
8. Line 27-29: Print the number of comparisons for each completed search.
9. Line 32-34: Print the average number of comparisons after a given number of searches.

10. Running Time Analysis: The asymptotic running time of binary search is $O(log_2 n)$, given that with each recursive search call, half of the remaining elements are excluded as possible matches. Unlike linear search, binary search applies the methodology of divide and conquer, whereby the initial array is recursively split in half until a match is found or no additional splits can be made (n=1).

```
1  class BinarySearch {
2
3      int numComparisons = 0;
4      double comparisonTotal = 0;
5
6      BinarySearch(){
7          this.numComparisons = 0;
8          this.comparisonTotal = 0;
9      }
10
11     public int search(String[] array, String target, int start, int end) {
12         int foundIndex = -1;
13         int midPoint = (start+end)/2;
14         this.numComparisons++;
15
16         if (target.compareTo(array[midPoint]) == 0) {
17             foundIndex = midPoint;
18         } else if (target.compareTo(array[midPoint]) < 0) {
19             foundIndex = search(array, target, start, midPoint-1);
20         } else {
21             foundIndex = search(array, target, midPoint+1, end);
22         }
23         return foundIndex;
24     }
25
26     // Print the number of comparisons for each search
27     public void printNumComparisons() {
28         System.out.println("\nNumber of Comparisons: " + this.numComparisons);
29     }
30
31     // Print the average number of comparisons after a certain number of searches
32     public void printAvgComparison(int total) {
33         System.out.printf("\nAverage Number of Comparisons:  %.2f %n", this.comparisonTotal/total);
34     }
35 }
```

3 Hash Table

1. The Hash Table class has the following attributes:
   a) hashTable: The hash table itself; an array of linked lists
   b) numComparisons: The number of comparisons required to complete a single search
   c) comparisonTotal: A provisional attribute used to track the total number of comparison over a certain number of searches. This is used to calculate the average number of comparisons.
2. Line 13-20: Method for putting an element in the Hash Table

a) Line 14-16: Create a linked list for every key within the hash table to accommodate chaining.

b) Line 19: Add the given element to the front of the linked list for the specified key (see lines 12-25 in LinkedList.java).

3. Line 22-41: Method for retrieving an element from the Hash Table.

   a) Line 24-25: Number of comparisons is at least 1 each time 'get' is called.

   b) Line 29-36: Traverse the Hash Table until the target element is found and then break. Increment number of comparisons each time a linked list element is traversed.

   c) Line 39-40: Add the number of comparisons to the total number of comparisons. Return whether or not the element was found.

4. Line 44-55: Method used to print the entire hash table.

5. Line 58-90: Method used to print the individual number of comparisons for each 'get'.

6. Line 63-65: Method used to print the average number of comparisons over a given number of 'get' calls.

7. Running Time Analysis:

   a) The asymptotic running time of each 'put' call is O(1). Given that in this implementation of a linked list new elements are added to the front of the list as opposed to the end of the list, any load factor considerations are negligible.

   b) The asymptotic running time of each 'get' call is O(1) + α, where α is the load factor associated with chaining. The load factor refers to the ratio of the number of items in the hash table to the table's size, which in this case is 250. Since each element in contained within a linked list, the linked list must first be retrieved (in O(1) time), and then traversed (α) to find the target element.

```
1  class HashTable {
2
3      LinkedList[] hashTable = null;
4      int numComparisons = 0;
5      double comparisonTotal = 0;
6
7      HashTable(int len) {
8          this.hashTable = new LinkedList[len];
9          this.numComparisons = 0;
10         this.comparisonTotal = 0;
11     }
12
13     public void put(int code, String element) {
14         if (this.hashTable[code] == null) {
15             // Create a new linked list if one doesn't exist for a given hashcode
16             this.hashTable[code] = new LinkedList();
17         }
18         // Add element to the top of the linked list
19         this.hashTable[code].add(element);
20     }
21
22     public boolean get(int code, String element) {
23         // Initial 'get' is 1 comparison
24         this.numComparisons = 0;
25         this.numComparisons++;
26
```

```
27          boolean res = false;
28          int i = 0;
29          while (i<this.hashTable[code].getSize()) {
30              // Increment num comparisons every time another item in linked list is traversed
31              this.numComparisons++;
32              if (this.hashTable[code].getNode(i).getName().compareTo(element) == 0) {
33                  res = true;
34                  break;
35              }
36              i++;
37          }
38          // Keep track of total number of comparisons
39          this.comparisonTotal += this.numComparisons;
40          return res;
41      }
42
43      // Print the entire hash table
44      public void print() {
45          for (int i=0; i<hashTable.length; i++) {
46              if (hashTable[i] != null) {
47                  System.out.println("Hash Code: " + i);
48                  System.out.println("Contains elements: ");
49                  for (int j=0; j<hashTable[i].getSize(); j++) {
50                      System.out.println(hashTable[i].getNode(j).getName());
51                  }
52                  System.out.println();
53              }
54          }
55      }
56
57      // Print the number of comparisons for each search
58      public void printNumComparisons() {
59          System.out.println("\nNumber of Comparisons: " + this.numComparisons);
60      }
61
62      // Print the average number of comparisons after a certain number of searches
63      public void printAvgComparison(int total) {
64          System.out.printf("\nAverage Number of Comparisons:  %.2f %n", this.comparisonTotal/total);
65      }
66  }
```

4 Main Class and Average Comparison Results

4.1 Main Class

1. Line 8-42: Populate the magic items in an array.
2. Line 46-47: Sort the magic items using Insertion Sort.
3. Line 50, Line 163-172: Place 42 random magic items in a new array.
4. Line 53-80: Search for the 42 set-aside items using Linear Search and count the number of comparisons for each completes search. Check if all items were found and calculate the average number of comparisons.

5. : Line 83-116: Search for the 42 set-aside items using Binary Search and count the number of comparisons for each completes search. Check if all items were found and calculate the average number of comparisons.
    a) Line 89: Reset the number of comparisons for each completed search.
    b) Line 100: Add the number of comparisons for a given search to the comparison total.
6. Line 119-160: Search for the 42 set-aside items using a Hash Table. Print the number of comparisons for each search and calculate the average number of comparisons.
    a) Line 122-125: Loop though all the magic items and assign a hash code value to each element in the array (see the assignment provided code in Hashing.java). Place the element at the designated hash code index.
    b) Line 133-149: Loop through each of the 42 set-aside items and (re)create the hash code value. Use that value to get the element from the hash table. Break out of the loop if at any point an element cannot be found.

```
1  import java.io.File;
2  import java.io.FileNotFoundException;
3  import java.util.Random;
4  import java.util.Scanner;
5
6  class Assignment_3 {
7
8      public static Random randomGen = new Random();
9
10     public static void main(String[] args) {
11
12         // Get magic items from the input file
13         File file = new File("magicitems.txt");
14         Scanner myReader = null;
15         String[] magicItems = null;
16         int i = 0;
17         int numLines = 0;
18
19         try {
20             myReader = new Scanner(file);
21             while (myReader.hasNextLine()) {
22                 // count the number of lines in the file
23                 numLines++;
24                 myReader.nextLine();
25             }
26             myReader.close();
27
28             // create magic items array
29             magicItems = new String[numLines];
30
31             myReader = new Scanner(file);
32             while (myReader.hasNextLine()) {
33                 // initialize magic items array with magic items
34                 String data = myReader.nextLine();
35                 magicItems[i] = data.toLowerCase();
36                 i++;
```

```
            }
            myReader.close();
        } catch (FileNotFoundException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
        // --------------------------------------------------------------------------

        // Sort the magic items
        InsertionSort sortObj = new InsertionSort();
        sortObj.sort(magicItems);

        // Get 42 random magic items
        String[] randItems = getRandomElements(magicItems);


        /* Linear Search ------------------------------------------------------------ */
        LinearSearch linearSearchObj = new LinearSearch();
        boolean areAllFound = false;
        i = 0;
        while (i<randItems.length) {
            int index = linearSearchObj.search(magicItems, randItems[i]);
            if (index == -1) {
                areAllFound = false;
                break;
            } else {
                areAllFound = true;
            }

            // Print individual number of comparisons
            // System.out.print("\n" + i + ": ");
            // linearSearchObj.printNumComparisons();
            i++;
        }
        // Print average number of comparisons
        linearSearchObj.printAvgComparison(randItems.length);

        System.out.println("-----------------------");
        if (areAllFound) {
            System.out.println("All items found!!");
        } else {
            System.out.println("Couldn't find " + randItems[i]);
        }
        /* END Linear Search -------------------------------------------------------- */


         /* Binary Search ------------------------------------------------------------ */
        BinarySearch binarySearchObj = new BinarySearch();
        areAllFound = false;
        i = 0;
        while (i<randItems.length) {
            // Reset the number of comparisons each time new seach begins
            binarySearchObj.numComparisons = 0;

```

```
            int index = binarySearchObj.search(magicItems, randItems[i], 0, magicItems.length);
            if (index == -1) {
                areAllFound = false;
                break;
            } else {
                areAllFound = true;
            }

            // Keep track of total number of comparisons for all searches in order to find average later
            binarySearchObj.comparisonTotal += binarySearchObj.numComparisons;

            // Print individual number of comparisons
            // System.out.print("\n" + i + ": ");
            // binarySearchObj.printNumComparisons();
            i++;
        }
        // Print average number of comparisons
        binarySearchObj.printAvgComparison(randItems.length);

        System.out.println("-----------------------");
        if (areAllFound) {
            System.out.println("All items found!!");
        } else {
            System.out.println("Couldn't find " + randItems[i]);
        }
         /* END Binary Search ------------------------------------------------------------- */


         /* Hashing ----------------------------------------------------------- */
        Hashing hashingObj = new Hashing();
        HashTable hashTableObj = new HashTable(hashingObj.HASH_TABLE_SIZE);
        for (i=0; i<magicItems.length; i++) {
            int hashCode = hashingObj.makeHashCode(magicItems[i]);
            hashTableObj.put(hashCode, magicItems[i]);
        }

        // System.out.println("-----------------------");
        // hashTableObj.print();
        // System.out.println("-----------------------");

        areAllFound = false;
        i = 0;
        while (i<randItems.length) {
            // Recalculate hashcode value for each random item
            int hashCode = hashingObj.makeHashCode(randItems[i]);
            // Check if the item exists within hashcode list
            boolean found = hashTableObj.get(hashCode, randItems[i]);
            if (!found) {
                areAllFound = false;
                break;
            } else {
                areAllFound = true;
            }
            i++;
```

```
145
146             // Print individual number of comparisons
147             // System.out.print("\n" + j + ": ");
148             // hashTableObj.printNumComparisons();
149         }
150
151         // Print average number of comparisons
152         hashTableObj.printAvgComparison(randItems.length);
153
154         System.out.println("-----------------------");
155         if (areAllFound) {
156             System.out.println("All items found!!");
157         } else {
158             System.out.println("Couldn't find " + randItems[i]);
159         }
160         /* END Hashing ------------------------------------------------------------ */
161     }
162
163     public static String[] getRandomElements(String[] array) {
164         String[] randElements = new String[42];
165         Random rand= new Random();
166
167         for (int i=0; i<randElements.length; i++) {
168             int randIndex = rand.nextInt(666);
169             randElements[i] = array[randIndex];
170         }
171         return randElements;
172     }
173 }
```

4.2 Average Comparison Results

Search Method	Average Number of Comparisons
Linear Search	342.21
Binary Search	8.55
Hash Table	3.33