Assignment Three – Magic Items Searcher

C. Marcus DiMarco
C.DiMarco1@Marist.edu

November 5, 2021

1 Objectives

- Create a program that searches for a set of 42 random Strings within a given text file using linear search and binary search.
- In this same program, populate a hash table and retrieve the aforementioned set of Strings from the table.
- Print the number of comparisons for each search/retrieval, as well as the average comparisons for each method after all 42 searches/retrievals.

2 Conditions

- The average comparisons for each method must print only two decimal places.
- The program must only use custom algorithms for the searches and hashing, with the exception of the hash function provided at `https://www.labouseur.com/courses/algorithms/Hashing.java.html`. Any of these which already exist in the language may not be used.

3 Overview

In order to enhance readability, each search algorithm is a method contained in the class MarcusSort.

3.1 linearSearch()

The most straightforward searching algorithm in this program is linear search. Linear search starts at the first index and iterates through the array until it finds the target value.

```java
// Linear search will search the entire array in order,
// halting once the target has been found
public void linearSearch(String[] array, String target) {

  // Start with counter at 0
  this.resetCounter();

  // Search the array in indexed order for the target
  for (int i = 0; i < array.length; i++) {
    counter++;
    // If found, print completion message and break
    if (array[i].compareToIgnoreCase(target) == 0) {
      this.printCompletionMessage();
      return;
    }
  }

  // If not found, print message and counter
  this.printFailureMessage();
}
```

3.2 binarySearch()

Improving on linear search is binary search. Binary search uses a logarithmic approach to searching, which greatly reduces the amount of comparisons needed on average. Binary search is only effective, however, if the array is sorted. We compare our target to the midpoint of the array. If they are equal, we have found the target. If the target is lower, we search the lower half of the array in the same way. The same is true for the upper half if the target is higher.

```
  // Abstracting binarySearch() for ease-of-use and to ensure
  // proper counter functionality
  public void binarySearch(String[] array, String target) {

    // Start with counter at 0
    this.resetCounter();

    // Private binary search method
    int indexOfTarget = binarySearch(array, 0, array.length - 1, target);

    // Print message conditional on if target was found
    if (indexOfTarget == -1) {
      this.printFailureMessage();
    } else {
      this.printCompletionMessage();
    }
  }

  /// Binary search will take a midpoint of the sorted array and
  // compare the target, recursively calling binary search on the
  // half of the array that would contain the sorted target
  // until found or a base case is reached.
  private int binarySearch(String[] array, int leftIndex,
                int rightIndex, String target) {

    // Check for IndexOutOfBoundsException
    if (rightIndex >= 0) {
      int midpoint = (leftIndex + rightIndex) / 2;

      // If the target is at the midpoint index, return the index
      // If less than the value at midpoint, search the left half
      // If greater than the value at midpoint, search the upper half
      if (target.compareToIgnoreCase(array[midpoint]) == 0) {
        counter++;   // Increment for comparison
        return midpoint;
      } else if (target.compareToIgnoreCase(array[midpoint]) < 0) {
        counter++;   // Increment for comparison
        return binarySearch(array, leftIndex, midpoint - 1, target);
      } else {
        counter++;   // Increment for comparison
        return binarySearch(array, midpoint + 1, rightIndex, target);
      }
    } else {
      return -1;      // Return -1 if value not found
    }
  }
```

4 MarcusHash

In our custom hashing library, we have four primary functions: makeHashCode(), loadToTable(), chainToTable(), and retrieve().

4.1 makeHashCode()

makeHashCode() is a hashing function sourced from `https://www.labouseur.com/courses/algorithms/Hashing.java.html`. It uses the sum of a String's ASCII values to generate a hash value.

```java
  // Hash function
  public int makeHashCode(String string, int hashTableSize) {
    string = string.toUpperCase();
    int length = string.length();
    int letterTotal = 0;

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

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

    // Scale letterTotal to fit in hashTableSize
    int hashCode = (letterTotal * 1) % hashTableSize;

    return hashCode;
  }
```

4.2 loadToTable()

Once we've generated the hash value for a given String, we must populate the hash table with it. loadToTable() is an abstraction for user-friendliness that houses a try/catch block to prevent an IndexOutOfBoundsException from passing conflicting parameters. It checks if the index at the hash value has been populated, and if not, adds the MarcusNode passed. If the index already has a pointer to a MarcusNode, loadToTable() begins a (potentially recursive) call to chainToTable().

```java
  // Add MarcusNode to table
  public void loadToTable(MarcusNode[] hashTable, MarcusNode node,
              int hashCode) {
    // Try/catch block to ensure param @hashCode will fit into param @hashTable
    try {
      if (hashTable[hashCode] == null) {
        hashTable[hashCode] = node;
      } else {
        chainToTable(hashTable[hashCode], node);
      }
    } catch (IndexOutOfBoundsException e) {
      System.out.println("Whoops! You're passing a hash value greater " +
                "than the size of the hash table.");
      e.printStackTrace();
    }
  }
```

4.3 chainToTable()

chainToTable() is where the magic happens. If nodeInTable points to null, then we set its pointer to nodeToChain. If nodeInTable points to a MarcusNode, we call chainToTable() again on the next node in the chain until we reach the base case of pointing to null.

```java
  // Recursive chain function for hash table - only for use by loadToTable()
  private void chainToTable(MarcusNode nodeInTable, MarcusNode nodeToChain) {
    if (nodeInTable.getNext() == null) {
      nodeInTable.setNext(nodeToChain);
      return;
    } else {
      chainToTable(nodeInTable.getNext(), nodeToChain);
    }
  }
```

4.4 retrieve()

In order to retrieve values from the hash table, we first need to get the hash value of the target. Then, using currentNode, we can travel the chain at the index of the hash value and compare each String along the way.

```java
  // Retrieve value from hashTable, if exists
  public void retrieve(MarcusNode[] hashTable, String target) {
    int hashCode = makeHashCode(target, hashTable.length);
    MarcusNode currentNode = hashTable[hashCode];

    // Reset counter
    this.resetCounter();

    while (currentNode != null) {
      counter++;
      if (target.compareTo(currentNode.getItem()) == 0) {
        this.printCompletionMessage();
        return;
      } else {
        currentNode = currentNode.getNext();
      }
    }

    // If not found, print failure message
    this.printFailureMessage();
  }
```

4.5 Other helper functions

Below are the remaining functions present in both MarcusSearch and MarcusHash (the two classes print different completion/failure messages, but these methods are otherwise the same).

```java
  // Getter for counter
  public int getCounter() {
    return counter;
  }

  // Controlling setter functionality to only reset to 0
  public void resetCounter() {
    counter = 0;
  }

```

```java
  // Message to print upon completing retrieval which includes counter
  public void printCompletionMessage() {
    System.out.println("Retrieval complete! Number of comparisons: "
               + counter);
  }

  // Message to print if search completed without finding target
  public void printFailureMessage() {
    System.out.println("Retrieval unsuccessful - target not found." +
               "\nNumber of comparisons: " + counter);
  }
```

4.6 Assignment3

With the methods we have just defined, we are ready to create our searcher.
We'll need some imported libraries: namely, java.io.File to import a file, java.io.FileNotFoundException to account for errors finding the input file, and java.util.Scanner to read the file.

```java
/**
 * A program which reads a constant number of Strings
 * from a file, sorts it using a custom sort library,
 * then searches for a random 42 items using custom
 * linear and binary search implementations. Then,
 * hashes the Strings into a table and retrieves the
 * 42 items.
 */
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class Assignment3 {
  public static void main(String[] args) {
    final int NUM_OF_ITEMS = 666;           // Length of file as constant
    final int NUM_OF_ITEMS_TO_FIND = 42;       // Number of items to find
    final int HASH_TABLE_SIZE = 250;          // Size of hash table
    String[] magicItems = new String[NUM_OF_ITEMS]; // Array of file strings
    MarcusSort sorter = new MarcusSort();        // Instance of MarcusSort
    MarcusSearch searcher = new MarcusSearch();    // Instance of MarcusSearch
    MarcusHash hasher = new MarcusHash();        // Instance of MarcusHash
```

```java

    // Try/catch block for file import and reading
    try {
      File file = new File("magicitems.txt");
      Scanner read = new Scanner(file);
      for (int i = 0; i < NUM_OF_ITEMS; i++) {
        magicItems[i] = read.nextLine();
      }
      read.close();
    } catch (FileNotFoundException e) {
      System.out.println("Whoops! Couldn't find magicitems.txt");
      e.printStackTrace();
    }

    // Select 42 items at random from magicitems[] and populate a subarray
    sorter.notRosannaShuffle(magicItems);
    String[] magicItemTargets = new String[NUM_OF_ITEMS_TO_FIND];
    for (int i = 0; i < magicItemTargets.length; i++) {
      magicItemTargets[i] = magicItems[i];
    }

    // Sort magicitems[]
    sorter.quickSort(magicItems);

    // Use linear search and print comparisons
    double averageComparisons = 0;
    for (int i = 0; i < magicItemTargets.length; i++) {
      searcher.linearSearch(magicItems, magicItemTargets[i]);
      averageComparisons += searcher.getCounter();
    }
    averageComparisons /= magicItemTargets.length;
    System.out.printf("Average comparisons for linear search: %.2f",
              averageComparisons);
    System.out.println();

    // Use binary search and print comparisons
    averageComparisons = 0;
    for (int i = 0; i < magicItemTargets.length; i++) {
      searcher.binarySearch(magicItems, magicItemTargets[i]);
      averageComparisons += searcher.getCounter();
    }
    averageComparisons /= magicItemTargets.length;
    System.out.printf("Average comparisons for binary search: %.2f",
              averageComparisons);
    System.out.println();

```

```
        // Hash magicItems[]
        MarcusNode[] hashTable = new MarcusNode[HASH_TABLE_SIZE];
        for (int i = 0; i < magicItems.length; i++) {
          int hashCode = hasher.makeHashCode(magicItems[i], hashTable.length);
          MarcusNode node = new MarcusNode(magicItems[i]);
          hasher.loadToTable(hashTable, node, hashCode);
        }

        // Retrieve the 42 items from the hash table, printing comparisons
        averageComparisons = 0;
        for (int i = 0; i < magicItemTargets.length; i++) {
          hasher.retrieve(hashTable, magicItemTargets[i]);
          averageComparisons += hasher.getCounter();
        }
        averageComparisons /= magicItemTargets.length;
        System.out.printf("Average comparisons for hash table retrieval: %.2f",
                  averageComparisons);
        System.out.println();
      }
    }
```

5 Results

Due to the pseudorandom nature of notRosannaShuffle(), exact results will vary from execution to execution. The below table shows the results obtained during 5 trials, along with the average number of comparisons and the expected order of growth for each sort.

Search/Retrieval Performance (in number of average comparisons)			
	linearSearch()	binarySearch()	retrieve()
Trial 1	358.19	8.55	2.48
Trial 2	353.50	8.71	2.33
Trial 3	350.31	8.43	2.71
Trial 4	295.26	8.05	2.24
Trial 5	372.71	8.38	2.55
Average	345.994	8.424	2.462
$O(g(n))$	$O(n)$	$O(log(n))$	$O(1)$

Linear search belongs to $O(n^2)$ due to its worst-case iteration over the entire array. Binary search expects to run in $O(log(n))$ time due to its logarithmic approach, recursively removing half of the array to search per call. Finally, retrieval from a hash table is expected to be $O(1)$, but it can degrade to $O(n)$ if there is a great deal of chaining due to a poorly designed hash function.