Assignment Three

Nicholas Fiore

Nicholas.Fiore@Marist.edu

November 2, 2022

1 Searching

The purpose of this lab was to explore different algorithms for searching a (sorted) list. The algorithms explored included linear search, binary search, and searching using hashing and chaining.

1.1 Linear Search

Linear search is the most basic kind of search. It takes a list and a key, and will iterate through the entire list until either the key is found, or the end of the list is reached. The implementation of the algorithm is designed to return the index of where the key is found in the list, or -1 if the key is not found at all.

As its name implies, linear search has a worst case run-time of linear time, or $O(n)$. This is because it loops through the list iteratively once, and the worst case would be having the key be found at the last index in the list. Linear search does not need to have a sorted list to work, so it can be good for smaller, unsorted lists where the overhead of a sorting algorithm on would end up taking more time than even the $O(n)$ time that linear search would take.

1.2 Binary Search

Binary search is a more complex type of search that recursively halves the list until the key is (or isn't) found. Like linear search, the implementation takes in a key and a list, and also takes in an int value to represent the starting index and the ending index. Unlike linear search, however, binary search requires the list to be sorted in order to work, as the algorithm finds a midpoint based on the starting and ending indices, and compares that value to the key to determine how to proceed.

The workings of binary search are relatively simple. The algorithm takes in the necessary values and begins by finding a midpoint index based on the start and end indices. This midpoint is then used in an if-else block to determine the route that the algorithm takes. First, the algorithm checks to see if the start index is greater than the end index, as that means that the key was not found in the list. Next, the value at the midpoint is compared to the key, and if they are equal, the index of the midpoint is set to a return value variable and returned. If the key is less than the value at the midpoint, the list is halved, with the method being called again using only the bottom half of the list. If the key is greater, it calls recursively using the

top half instead. Technically, the midpoint is also excluded from these recursive calls, so it is not a perfect half, but it is close enough to be effectively half.

Due to the way that binary search halves the size of the list each call, its asymptomatic run-time is $O(log_2 n)$. This is an incredible improvement from linear search, but has a caveat. Since binary search ***must*** use a sorted list, any unsorted lists must first be sorted in order to work. For small lists, this could be detrimental as the additional run-time created from using a sorting algorithm may completely negate the benefits of using binary search over linear search.

2 Hashing

Hashing is a special kind of searching that has a bit more involvement than the other two search algorithms, which only need the original list in an array to function. Hashing, however, creates a table of hash values to use for searching. These hash values are determined based on a hash-making algorithm. This algorithm can function any way to create the hash, the only requirement is that an output must *always* be the same for any input that is the same. For example, a String containing the text "alpaca" will *always* result in the same hash code. This hash code might be determined by adding the values of all the characters in the String and then taking the modulus of the size of the hash table. This is the implementation that was used in the assignment. The hash table size for this implementation is 250. The hash code number refers to the index in the hash table that the item will be placed at.

Because the size of the hash table should not be huge (as it would defeat the purpose of the hash table, as well as take up a large chunk of memory), there needs to be ways to deal with collisions. Collisions are when two inputs result in the same hash code. The implementation used in this assignment is chaining. The idea of this is that instead of placing a single item at its hash table location, a pointer to a linked list is placed there, with the first item placed being the head of the list. Anytime there is a collision, the item is added to the linked list.

By itself, searching using hashing is a constant time or $O(1)$. This is due to the fact that hash codes relate to an instantaneous index lookup. However, since collisions cannot just be ignored when searching, the time is a bit more varied. Officially, the time for it would be referred to as $O(1) + \alpha$. In this instance, α refers to the amount of iterations it takes to search through the linked list at the hash location and find the item. So long as the hash table is relatively balanced, this isn't too much of an issue, but particularly large lists and small hash tables or unbalanced hash tables can cause the lookup time of a search to eclipse even binary search.

3 Appendix

3.1 Search.java

```java
/*
 * A class used to maintain static methods for searching algorithms (namely linear search and binary search). Linear search is iterative,
 * while binary search is recursive.
 */
public class Search {

    //sequentially searches the list for the key, returning the index of where it was found. If it was not found, returns -1.
    public static int linearSearch(String[] arr, String key, int[] counter) {
        int i = 0;
        while (i < arr.length && arr[i] != key) {
            i++;
            counter[0]++;
        }
        if (i >= arr.length)
            i = -1;
        return i;
    }

    //recursively searches the list by comparing the key to the item at the middle of the list, then choosing half of the array to
    //then search depending on whether the key is lesser or greater than the element at the middle. If the element is found, returns
    //the index, otherwise returns -1.
    public static int binarySearch(String[] arr, int startIndex, int endIndex, String key, int[] counter) {
        int retVal;
        int midIndex = (endIndex + startIndex) / 2;
        if (startIndex > endIndex) {
            retVal = -1;
        } else if (key.toUpperCase().compareTo(arr[midIndex].toUpperCase()) == 0) {
            counter[0]++;
            retVal = midIndex;
        } else if (key.toUpperCase().compareTo(arr[midIndex].toUpperCase()) < 0) {
            counter[0]++;
            retVal = binarySearch(arr, startIndex, midIndex - 1, key, counter);
        } else {
            counter[0]++;
            retVal = binarySearch(arr, midIndex + 1, endIndex, key, counter);
        }
        return retVal;
    }

    public static int binarySearchIt(String[] arr, int startIndex, int endIndex, String key, int[] counter) {
        int low = startIndex;
        int high = (endIndex) - startIndex;
        while (low < high) {
            int mid = (low + high) / 2;
            if (key.compareTo(arr[mid]) <= 0) {
                high = mid;
            } else {
                low = mid + 1;
            }
            counter[0]++;
        }
        return high;
    }
}
```

3.2 HASHING.JAVA

```java
/*
 * A class used for hashing and searching. Includes a function to create a hash code, and then a put() function to put the String in the
 * hash table, and a get() function to search the hash table for a String
 */

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

public class Hashing {

    private static final int LINES_IN_FILE = 666;
    private static final int HASH_TABLE_SIZE = 250;
    private static LinkedList[] hashTable = new LinkedList[HASH_TABLE_SIZE];

    //Creates a hash code based on the ASCII values of all the characters in the 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;
        }

        // Scale letterTotal to fit in HASH_TABLE_SIZE.
        int hashCode = (letterTotal * 1) % HASH_TABLE_SIZE;  // % is the "mod" operator
        // TODO: Experiment with letterTotal * 2, 3, 5, 50, etc.

        return hashCode;
    }

    //puts String into the hash table that is stored in the class as a static variable. Uses makeHashCode() to create the hash.
    public static void put(String key) {
        int hash = makeHashCode(key);
        Node newNode = new Node(key);
        if (hashTable[hash] != null) {
            hashTable[hash].push(newNode);
        } else {
            hashTable[hash] = new LinkedList(newNode);
        }
    }

    //finds the String within the hash table based on the hash table and the String itself as a key.
    public static boolean get(int hash, String key, int[] counter) {
        boolean retFlag = false;
        Node listHead = hashTable[hash].getMyHead();
        while (listHead != null && !(retFlag)) {
            if (listHead.getMyString().compareTo(key) == 0) {
                retFlag = true;
            }
            counter[0]++;
            listHead = listHead.getMyNext();
        }
        return retFlag;
    }
}
```

3.3 MainThree.java

```java
//Utility imports
import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Scanner;

/*
 * The purpose of this program is to test the asymptomatic run times of different searching algorithms. The algorithms include linear search,
 * binary search, and searching using hashing.
 */

public class MainThree {
    public static void main(String[] args) {
        //creates a new file object for the magicitems.txt file
        String fileName = "magicitems.txt";
        File file = new File(fileName);
        //long variable to store the total lines of a .txt
        long totLines;
        //array to hold all the items in the list
        String[] itemList;


        /* This try-catch block is for reading in a .txt file, putting each line onto an array, and throwing exceptions if there are any. */
        try {
            //Scanner object for scanning the file
            Scanner fileScanner = new Scanner(file);
            Path path = Paths.get(fileName);

            //grabs the amount of lines within the .txt file and prints it
            totLines = Files.lines(path).count();
            System.out.println("Total lines: " + totLines + "\n");

            //initializes the size of the array with the total lines in the .txt
            itemList = new String[(int)totLines];


            for (int i = 0; i < totLines; i ++) {
                itemList[i] = fileScanner.nextLine();
            }

            //closes the scanner after use to save resources
            fileScanner.close();


            /*
             *
             */


            //counters for the comparisons and the current search. Search count starts at 1, and will go to 42
            int[] compCounter = new int[1];
            compCounter[0] = 0;
            int currentSearch = 1;

            //array for storing the comparison counts of each search, to be used to find the average
            int[] averages = new int[42];

```

```
59              //small subprogram to get 42 random items from the list. By shuffling the original
                    list randomly, the first 42 elements
60              //will be randomly "selected" by nature of a random shuffle and then just using those
                    values. Technically, any range of
61              //42 elements within the original list would work, as they would all be randomized.
62              String[] randList = new String[42];
63              Sort.shuffle(itemList);
64              for (int i = 0; i < 42; i++) {
65                  randList[i] = itemList[i];
66              }
67
68              //temporary int array to use mergeSort without having to change it completely
69              int[] temp = new int[1];
70              temp[0] = 0;
71
72              //sorts the array using a mergeSort algorithm
73              Sort.mergeSort(itemList, 0, itemList.length - 1, temp);
74
75              /* Linear Search */
76              System.out.println("\033[1mLinear Search\033[0m");
77              //displays the comparisons for every Linear search made
78              for (int i = 0; i < randList.length; i++) {
79                  int index = Search.linearSearch(itemList, randList[i], compCounter);
80                  System.out.println("Search " + currentSearch + " number of comparisons: \033[1m" +
                         compCounter[0] + "\033[0m. Key (" + randList[i] + ") was found at index " +
                        index);
81                  averages[i] = compCounter[0];
82                  compCounter[0] = 0;
83                  currentSearch++;
84              }
85
86              //calculates and displays the average of all the searches
87              int total = 0;
88              for (int j = 0; j < averages.length; j++) {
89                  total += averages[j];
90              }
91              int average = total / averages.length;
92              System.out.println("Average number of comparisons: " + average);
93              System.out.println();
94
95              //resets counters and other values to default
96              compCounter[0] = 0;
97              currentSearch = 1;
98              average = 0;
99              total = 0;
100
101             /* Binary Search */
102             System.out.println("\033[1mBinary Search\033[0m");
103             //
104             for (int i = 0; i < randList.length; i++) {
105                 int index = Search.binarySearch(itemList, 0, itemList.length, randList[i],
                        compCounter);
106                 System.out.println("Search " + currentSearch + " number of comparisons: \033[1m" +
                         compCounter[0] + "\033[0m. Key (" + randList[i] + ") was found at index " +
                        index);
107                 averages[i] = compCounter[0];
108                 compCounter[0] = 0;
109                 currentSearch++;
110             }
111
112             //calculates and displays the average of all the searches
113             total = 0;
114             for (int j = 0; j < averages.length; j++) {
115                 total += averages[j];
116             }
```

```
            average = total / averages.length;
            System.out.println("Average number of comparisons: " + average);
            System.out.println();

            //resets counters and other values to default
            compCounter[0] = 0;
            currentSearch = 1;
            average = 0;
            total = 0;

            /* Hashing */
            //adds all the items in the list to the hash table in Hashing.java
            for (int i = 0; i < itemList.length; i++) {
                Hashing.put(itemList[i]);
            }

            //the list of items to be searched with the hash table. Numbers were all chosen randomly once with an external number generator.
            int[] choiceList = new int[]{24, 45, 64, 80, 111, 114, 123, 152, 192, 205, 225, 232, 236, 249, 262, 269, 301, 302, 320, 324, 339,
            355, 366, 387, 411, 434, 444, 445, 460, 464, 477, 491, 507, 545, 556, 577, 581, 604, 615, 639, 657, 665};

            System.out.println("\033[1mHash and Search\033[1m");
            //the actual process of searching using the hash codes. Instead of storing all the hash codes in an array, the makeHashCode
            //method is just called again. This also allows inputs of strings that are not in the itemList array.
            for (int i = 0; i < choiceList.length; i++) {
                //key and hash are used for the call to the get() function in hashing
                String key = itemList[choiceList[i]];
                int hash = Hashing.makeHashCode(key);
                //finds the value of get() from Hashing to determine outut
                boolean wasFound = Hashing.get(hash, key, compCounter);
                if (wasFound) {
                    System.out.println(key + " was found after \033[1m" + compCounter[0] + "\033[0m comparisons.");
                } else {
                    System.out.println(key + " was not found.");
                }
                averages[i] = compCounter[0];
                compCounter[0] = 0;
            }

            //calculates and displays the average of all the searches
            total = 0;
            for (int j = 0; j < averages.length; j++) {
                total += averages[j];
            }
            average = total / averages.length;
            System.out.println("Average number of comparisons: " + average);
            System.out.println();

        } catch(FileNotFoundException ex) {
                        System.out.println("Failed to find file: " + file.getAbsolutePath());
        } catch(Exception ex) {
            System.out.println("Something went wrong.");
            System.out.println(ex.getMessage());
            ex.printStackTrace();
        }
    }
}
```