Lab 3: Searching Algorithms

Nicholas Suchy
Nicholas.Suchy1@Marist.edu

April 5, 2023

1 Introduction

For this lab we were to select 42 random but unique items from magicitems.txt. Then, using these 42 items as targets, use various searching methods and find the average number of comparisons each method took to search for all items to two decimal places. The searching methods implemented in the lab were Linear Search, Binary Search, and Searching with Hash Table.

2 Code Reuse and Changes From Previous Labs

This lab reuses code from previous labs. Namely, reading the file into a LinkedList is recycled for this lab. Additionally, the LinkedList class and the Node class are also reused with some minor changes occurring in the LinkedList class.

The LinkedList now has a sort() method which sorts the contents of the list using the Quick Sort algorithm created in the last lab. This helped clean up clutter from Main. Additionally, standard use Linked Lists have a sort function associated with them. This change allows the code to follow conventions already present in world of programming. Finally, this changes allows for my LinkedList class to be more reusable in the future. Instead of having to rewrite a sorting algorithm in Main in the future, the LinkedList class now has the ability to sort itself where ever it is implemented.

3 Structure

When it came to organizing and structuring the lab, I decided to implement the Linear and Binary Search algorithms as methods of the LinkedList class. As I had moved the sorting algorithm out of Main, I also decided it would be best if the LinkedList class had a method within that allowed it to search itself, allowing it to be easily reused while also keeping clutter out of Main.

As for the hash table, I decided to implement it as it's own class. As we had discussed in class, one way to make a hash table is to implement it as an array of Linked Lists, which is how I structured it for the lab. This allows for items that have equal hash codes to be handled through the use of chaining.

4 Generating the Target Values

```
// Creates a LinkedList to hold the target values in
LinkedList targets = new LinkedList();

// Randomly select 42 uniqe items from magicItems
while (targets.length() < 42) {
    int randomIndex = (int) (Math.random() * ((magicItems.length())));
    if (targets.linearSearch(magicItems.get(randomIndex).value)[0] == 0) {
        targets.addBack(magicItems.get(randomIndex).value);
    }
}
```

The above code picks 42 items randomly from magicItems and adds them to a new LinkedList called targets. It also ensures that no item is chosen as a target twice. These items are to be used as target values when using the various searches later in the lab.

5 The Return Array

```
// First index: Determines if the target is found (0-no, 1-yes)
// Second index: Keeps track of total comparisons
int[] returnArray = { 0, 0 };
```

Every time a search is initiated, the return value will always be an integer array, named returnArray. The array holds two values:

- A 0 or a 1
    1. A 0 means that the item was NOT found after the search
    2. A 1 means the the item was successfully found during the search
- The number of comparisons made during the search of a single item

While the lab only requires the number of comparisons to be returned, the first index of the return array was used for debugging the searching algorithms to make sure that target were actually and accurately being located.

5.1 Keeping Track of Comparisons

```
// Keep track of the total comparisons each search does
     double linearComparisons = 0;
     double binaryComparisons = 0;
     double hashTableComparisons = 0;

 // Perform a Linear and Binary Search on the 42 items
 for (int i = 0; i < targets.length(); i++) {
     linearComparisons += (double) magicItems.linearSearch(targets.get(i).value)[1];
     ...
     binaryComparisons += (double) magicItems.binarySearch(targets.get(i).value)[1];
     ...
     hashTableComparisons += (double) hashTable.get(targets.get(i).value)[1];
     ...
     }
```

The above lines keep track of the number of comparisons it took to search for a given item. Once the search was completed, the array was returned and the comparisons made were added to the total number of comparisons for the given search.

The skipped lines are debugging lines that printed the first index of the return array which allowed me to see if an target value was successfully located within the list. They are omitted to save space.

```
// Performs a linear search for a target of the LinkedList
public int[] linearSearch(String target) {

    Node tempNode = this.root;

    // First index: Determines if the target is found (0-no, 1-yes)
    // Second index: Keeps track of total comparisons
    int[] returnArray = { 0, 0 };

    // Loop until target is found or tempNode is empty
    while (tempNode != null && tempNode.value != target) {
        tempNode = tempNode.nextNode;

        // Keeps track of comparisons
        returnArray[1]++;
    }

    // Return that the target was found
    if (tempNode != null && tempNode.value == target) {
        returnArray[0] = 1;
    }

    return returnArray;
}
```

The above code shows the Linear Search algorithm used within the LinkedList class. The algorithm was based off the pseudo code we had discussed in class. As the name suggests, it runs in linear time as it contains a single while loop.

The method takes a String as a parameter. This string is the target value being searched for. It then iterates through the entire list, comparing every item of the list to the target and incrementing the second index of returnArray with each comparison. If the target is not found, the next item in the list is then compared to the target. If the target is found, it stops iterating, changes the first index of returnArray to 1, and then returns return array. If the end of the list is reached, the first index of returnArray is never changed to 1 signifying that the target passed to the function is not in the list.

6 Binary Search

```
// Performs a binary search for a target of the Linked List
public int[] binarySearch(String target) {

    // Used keep track of portion of LinkedList target could be in
    int low = 0; // Lowest bounds target could be
    int high = this.length; // Highest bounds target could be
    int middle = high / 2; // The middle of the subarray
```

```java

        // First index: Determines if the target is found (0-no, 1-yes)
        // Second index: Keeps track of total comparisons
        int[] returnArray = { 0, 0 };

        // Loop until target is found or until the entire list has been searched
        while (!(target == this.get(middle).value) && low < high) {
            middle = (int) Math.floor((low + high) / 2);

            // If target comes before the middle, only look at values before the middle
            // Otherwise, only look at values after the middle
            if (target.compareTo(this.get(middle).value) < 0) {
                high = middle;
            } else {
                low = middle + 1;
            }

            // Keeps track of total comparisons
            returnArray[1]++;
        }

        // Return that the target was found
        if (target == this.get(middle).value) {
            returnArray[0] = 1;
        }

        return (returnArray);
    }
```

The above code shows the Binary Search algorithm used within the LinkedList class. The algorithm was based off the pseudo code we had discussed in class. The algorithm runs in logarithmic time has it uses a while loop that halves the amount of time required to finish the search with each iteration.

The method takes a String as a parameter. This string is the target value being searched for. The highest, lowest, and middle indices of the array are also initialized to their respective values. Then the algorithm begins the while loop, iterating until the target is equal to the value at the middle index or until the low index is greater than or equal to the upper index. If the target is not found, a new middle index is calculated. Then new upper and lower bounds are found. If the target is found before the current value at the middle index, the upper index is lowered to the middle index. Otherwise, the lower index is raised to the index after the middle index. The number of comparisons increments and then the loop restarts. Once the target is found or the list has been completely iterated through, the first index of returnArray is adjusted to see if the value was actually found, and then returnArray is returned.

7 Hash Table

```java
LinkedList[] hashTable; // hashTable is an array of LinkedLists
int size; // holds the size of the hash table NOT number of items within

// Constructor
public HashTable(int size) {
    this.hashTable = new LinkedList[size];
```

```
        this.size = size;
    }
```

The HashTable class was implemented using the LinkedLinked class. More specifically, it is an array of LinkedList objects. It has access to all of LinkedLists methods and uses them to enact methods of a typical Hash Table Data Structure. The following methods were used to help complete the lab.

7.1 hash()

```
    // Generates a hash code for a given String
    public int hash(String magicItem) {
        int letterTotal = 0;

        // Adds up ASCII values of each letter
        for (int i = 0; i < magicItem.length(); i++) {
            letterTotal += (int) magicItem.charAt(i);
        }

        // String's hash code is it's total letter value modded by size of hash table
        int hashCode = letterTotal % this.size;

        return hashCode;
    }
```

The hashing function determines the hash code for a given string. It takes a string, sums the ASCII values of each character, and then takes the total and divides it by the size of the hash table. The string's hash code is the remainder after the division.

Although the algorithm iterates through a string, we can still say that it runs in constant time when looked at in the scope of the entire list of magicItems. This is beacuse, in the context of the whole list iterating through a single item can be considered constant time.

7.2 add()

```
    // Adds a value to the hash table
    public void add(String value) {

        // Find hash code of String
        int key = hash(value);

        // If an index doesn't already have a LinkedList, create a LinkedList for the index
        if (hashTable[key] == null) {
            hashTable[key] = new LinkedList();
        }

        // Using the String hash code, store it at the equivalent index
        hashTable[key].addBack(value);
    }
```

The above method allows a string to be inserted into the hash table. First, it takes the string passed as a parameter and finds it's hash code. Then the method adds the string to the back of the linked list at the index that is equal to the calculated hash code. If the string was to be the first value added at said index, a new Linked List is created and the string gets stored in the Linked List.

The creation and use of Linked Lists allows for chaining in the scenario that two strings have similar hash codes. Adding an item to a hash table is done in constant time as no loops are present.

7.3 get()

```
public int[] get(String value) {

    // Find hash code of given value
    int key = hash(value);
    int[] returnArray = hashTable[key].linearSearch(value);

    // Increment by 1 to account for the initial get of the array
    returnArray[1]++;

    // Return the array returned by a linearSearch of the LinkedList at the values hash index
    return returnArray;
}
```

When retrieving an item from a hash table with chaining is done in constant time. This is because even though iterations may occur, the number of iterations is much smaller than when compared to if the entire data set was to be iterated through.

In order to hash an item, the item's hash code is needed. In the above method, the hash code for the item to be searched is first calculated. Then the method travels to the array index that is equal to the hash code and a linear search, using the linear search algorithm discussed before, is performed on the linked list located at this index. Before the returnArray is returned, the number of comparisons is incremented to account for the initial get of the array index. Finally, returnArray is returned.

7.4 print()

I also had created a print function which prints out all indices of the hash table and their contents. This was used for debugging purposes and was not essential to the lab.

8 Example Output

Search Algorithm	Average Comparisons
Linear Search	314.67
Binary Search	8.45
Hashing	2.02

The above table represents output from the program using the algorithms and methods explained above. Because the targets are randomly selected each time the program is run, repeating these exact values would be difficult. It is evident that the methods with lower time complexities make fewer comparisons. However, they may compensate by taking up more space in memory.