Assignment Three

Harrison Zheng

harrison.zheng1@marist.edu

November 5, 2021

1 Assignment Description

For this assignment, students were asked to develop a program to randomly pick 42 magic items from the full set of 666 and store them in an array. They then had to sort the full set of magic items in alphabetical order so that they could retrieve the 42 randomly chosen items using search algorithms. The three methods that students were required to use were linear search, binary search, and a hash table with chaining. In addition, they had to print the number of comparisons performed for each item searched, as well as compute the overall average for each method of searching.

2 Results of Search Methods From Sample Run

The table in section 2.1 contains my results from a sample run testing each search method on the array that contains all 666 magic items. The table shows that going from linear search to binary search to a hash table, the search methods get more and more efficient as they require fewer comparisons on average to find the magic items. However, as one will see in my code, the faster methods also take up more space as they store more variables in memory. This illustrates the time-space tradeoff in computer science. For explanations of the asymptotic running time of each search method, please visit their respective sections below.

2.1 Average Comparisons and Asymptotic Running Time of each Search Method

Method	**Comparisons**	**Running Time**
Linear Search	342.55	$O(n)$
Binary Search	8.17	$O(log_2 n)$
Hash Table	2.74	$O(1)$ + avg chain length

3 Linear Search

Before performing linear search, my program sorts the array of magic items using the merge sort algorithm I wrote in the last assignment. It then calls the linear search function to find the randomly chosen items in the bigger array (Listing 3.2). Linear search is a simple algorithm that works by taking the item that one is looking for and comparing it with each element in an array in linear fashion until it finds a match. As the algorithm searches for an item, it tracks each comparison used (Listing 3.1). The main method keeps a running total and prints the average number of comparisons used per search at the end. The asymptotic running time of linear search is $O(n)$ because as the size of the array being searched increases, so does the number of comparisons that need to be made and thus the search time as well. This is why linear search is, by far, the worst of the three methods for searching large data sets.

3.1 Linear Search class source code

```java
public class LinearSearch {

    //Determine how many comparisons are needed to find
    //the randomly selected item in the array using linear search
    public static int ls(String arr[], String item) {
        //Track number of comparisons used for each search
        int comparisons = 0;

        //Compare the randomly selected item to every item in
        //the array until a match is found
        for (int i = 0; i < arr.length; i++) {
            comparisons++;
            if (arr[i] == item) {
                break;
            }
        }
        return comparisons;
    }

}
```

3.2 Main Method Code to run Linear Search

```java
    //Sort array with merge sort before searching
    arr = mergeSort(arr);

    //Initialize total comparisons counter for linear search
    int ttlComparisonsLinear = 0;

    //Perform a linear search for each randomly selected item
    System.out.println("\nNumber of comparisons used for each linear search: ");
    for (int i = 0; i < randomItemArr.length; i++) {
        int comparisons = LinearSearch.ls(arr, randomItemArr[i]);
        ttlComparisonsLinear += comparisons;
        //Print the number of comparisons used for each search
        System.out.println(randomItemArr[i] + " - " + comparisons);
    }

    //Compute and print the average number of comparisons to 2 decimal places
    double avgComparisonsLinear = (double) ttlComparisonsLinear / randomItemArr.length;
    System.out.println("\nAverage number of comparisons used for linear search: " +
        String.format("%.2f", avgComparisonsLinear));
```

4 Binary Search

After finding each randomly chosen item using linear search, my program does it again using binary search. The binary search algorithm adopts a divide and conquer approach to finding items. It first finds the element in the middle of the array and compares the item held there to the item we are looking for. If they match, then we are done. If they do not, then we determine if the item would be located before or after the middle element. If it is located before, then the mid variable will point to the element in the middle of the left subarray and we compare that to the item we are looking for and if it is located after, then the mid variable will point to the element in the middle of the right subarray and we will do the same thing. This process is repeated until we land on the item we are looking for (Listing 4.1). It is important to note that this algorithm only works if we sort the array alphabetically beforehand, which we did before performing linear search. The asymptotic running time of binary search is $O(log_2n)$ because each time the algorithm compares the middle element with the item we are looking for and chooses a side, it reduces the search space in the array by half. As the search search space gets smaller, the number of comparisons the algorithm can perform also decreases drastically, making it a logarithmic function. Binary search is much faster than linear, but not as fast as searching a hash table with chaining.

4.1 Binary Search class source code

```java
public class BinarySearch {

    //Determine how many comparisons are needed to find
    //the randomly selected item in the array using binary search
    public static int bs(String arr[], int start, int stop, String item, int comparisons) {
        while (stop >= start) {
            //Calculate the midpoint in the array and use floor
            //rounding for non-whole numbers
            int mid = (int)((start + stop) / 2);

            comparisons++;

            if (arr[mid] == item) {
                //If element at the current mid matches the
                //item, return comparisons used in the search
                return comparisons;
            }
            if (arr[mid].compareTo(item) > 0) {
                //If the item comes before the current mid,
                //then search the subarray on the left
                stop = mid - 1;
            }
            else {
                //If the item comes after the current mid,
                //then search the subarray on the right
                start = mid + 1;
            }
        }
        return 0;
    }

}
```

4.2 Main Method Code to run Binary Search

```java
    //Initialize variables for binary search
    int ttlComparisonsBinary = 0;
    int startIndex = 0;
    //Comparisons counter is initialized outside of binarySearch method since
    //it is a recursive method and will reset the value each iteration
    int initialComparisons = 0;

    //Perform a binary search for each randomly selected item
    System.out.println("\nNumber of comparisons used for each binary search: ");
    for (int i = 0; i < randomItemArr.length; i++) {
        int comparisons = BinarySearch.bs(arr, startIndex,
            ITEM_COUNT-1, randomItemArr[i], initialComparisons);
        ttlComparisonsBinary += comparisons;
        //Print the number of comparisons used for each search
        System.out.println(randomItemArr[i] + " - " + comparisons);
    }

    //Compute and print the average number of comparisons to 2 decimal places
    double avgComparisonsBinary = (double) ttlComparisonsBinary / randomItemArr.length;
    System.out.println("\nAverage number of comparisons used for binary search: " +
        String.format("%.2f", avgComparisonsBinary));
```

5 Hash Table with Chaining

After finding the randomly chosen items using binary search, my program does it a third time using a hash table with chaining. Before my program can perform searching on a hash table, though, it must create it and fill it with all the magic items. It does this in the main method by first taking an item and running it through the hashing function in the Hashing.java file on the class website (Listing 5.4). The hashing function converts each letter in the item to its ASCII value and computes the total. It then performs a modulo 250 (hash table size) operation on the total and the remainder serves as the key for that magic item (Listing 5.2). Once the key has been calculated, the main method calls the Insert function in the HashTable class, which creates a Node object with the item and uses the key as the index at which it stores the node (Listing 5.3). If the hash table already has a node at that index, then the function creates a linked list and adds it at the end (Listing 5.1). The main method repeats this process of hashing and storing until the table has been populated with every item. Then, it calls the Retrieve function in the HashTable class, which recomputes the key of the item we are looking for and searches in linear fashion until it finds the node holding the item. The asymptotic running time of searching with a hash table is $O(1)$ + avg chain length. If an index in the table only has one node, then the algorithm would have found the item in one move by simply going to that index, which is described as $O(1)$. However, since many of the indexes point to linked lists, we have to add on the time it takes to run through the average number of nodes in a list. Although the hash table has the fastest *search* time of the three methods used here, it is important to consider that populating the table also requires time and this is something that we do not have to worry about when using the other two methods.

5.1 Hash Table class source code

```
//Class that contains methods for manipulating hash tables
public class HashTable {

    //Adds a node to a slot in the hash table
    public static Node[] insert(Node[] hashTable, int key, String magicItem) {
        //Create a new node using magic item passed in
        Node item = new Node(magicItem);
        if (hashTable[key] == null) {
            //If a slot in the table is null, then the new node
            //will be the first one in the slot
            hashTable[key] = item;
        }
        else {
            //If a slot in the table already has a node, then go to the last node
            //in the linked list and add the new node to its next pointer
            Node currentNode = hashTable[key];
            while (currentNode.next != null) {
                currentNode = currentNode.next;
            }
            currentNode.next = item;
        }
        return hashTable;
    }

    //Determine how many comparisons are needed to retrieve an item in the hash table
    public static int retrieve(Node[] hashTable, String magicItem) {
        //Since every get is one compare,
        //the comparisons counter starts at 1
        int comparisons = 1;

        //Use the hash code function in the Hashing class to get
        //the key/table index for the string we are searching for
        int key = Hashing.makeHashCode(magicItem);

        if (hashTable[key] == null) {
            //Tells user in case the calculated key indexes into an empty slot
            System.out.println("ERROR: No magic item in this slot of the hash table");
            return -1;
        }
        else {
            //If a slot in the table has at least one node,
            //then iterate through the linked list until we reach
            //the one with the item we are searching for
            Node currentNode = hashTable[key];
            while (currentNode.next != null) {
                comparisons++;
                if (currentNode.value.equalsIgnoreCase(magicItem)) {
                    break;
                }
                currentNode = currentNode.next;
            }
            return comparisons;
        }
    }

}
```

5.2 Hashing function code

```
public static int makeHashCode(String str) {
    //Magic item strings will already be sanitized
    //when passed into this function
    //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 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;
}
```

5.3 Node class source code

```
//Used to hold items in the hash table and create linked lists
public class Node {
    String value;
    Node next;

    //Initialize properties of new Node instances
    public Node(String value) {
        this.value = value;
        this.next = null;
    }

}
```

5.4 Main Method Code to run Hash Table Search

```
    //Initialize hash table
    Node[] hashTable = new Node[HASH_TABLE_SIZE];

    //Load hash table with all magic items
    for (int i = 0; i < arr.length; i++) {
        //Use the hash code function in the Hashing class to get
        //the key/table index for the item we are loading
        int key = Hashing.makeHashCode(arr[i]);

        //Load each magic item into table using insert method
        hashTable = HashTable.insert(hashTable, key, arr[i]);
    }

    //Initialize total comparisons counter for retrieving from the hash table
    int ttlComparisonsHT = 0;

    System.out.println("\nNumber of comparisons used for
        retrieving items from hash table: ");
    for (int i = 0; i < randomItemArr.length; i++) {
        int comparisons = HashTable.retrieve(hashTable, randomItemArr[i]);
        ttlComparisonsHT += comparisons;

        //Print the number of comparisons used for each retrieval
        System.out.println(randomItemArr[i] + " - " + comparisons);
    }

    //Compute and print the average number of comparisons to 2 decimal places
    double avgComparisonsHT = (double) ttlComparisonsHT / randomItemArr.length;
    System.out.println("\nAverage number of comparisons used for
        retrieving items from hash table: " +  String.format("%.2f", avgComparisonsHT));
```