CMPT 435 - Fall 2021 - Dr. Labouseur

Assignment Three

Nicholas Petrilli

November 5, 2021

1 Node

Listing 1: NodeObj Class

```
public class NodeObj {
    String item;
    NodeObj next;

    //Empty constructor
    public NodeObj() {
        item = "";
        next = null;
    }
    //Constructor to create a NodeObj with string item and a pointer to the n
    public NodeObj(String item, NodeObj next) {
        this.item = item;
        this.next = next;
    }
    //Setter function to set the item of node
    public void setItem(String item) {
        this.item = item;
    }
    //Setter function to set the pointer of next node
    public void setNext(NodeObj next) {
        this.next = next;
    }
    //Getter function to return the name of the node
    public String getItem() {
        return this.item;
    }
    //Getter function to return the next node
    public NodeObj getNext() {
        return this.next;
    }
```

```
}
```

The node class allows for chaining within the hash table. The hash table consists of an array of nodes, which include a string item and a pointer to the next node. When multiple strings compute to the same hash value, nodes are used to point to another node, creating a linked list at (possibly) each array element.

2 Hash Table

Listing 2: HashTable Class

```

public class HashTable {

    final static int HASH_TABLE_SIZE = 250;
    NodeObj[] nodes = new NodeObj[HASH_TABLE_SIZE];

    //Creates a hash table and initializes all items in the hash table to nul
    public HashTable() {
        for (int i = 0; i < HASH_TABLE_SIZE; i++) {
            nodes[i] = new NodeObj();
        }
    }
    public void put(String input, int key) {
        //need to check if there are already nodes there
        if (nodes[key].next == null){
            nodes[key].item = input;
        }
        else {
            nodes[key].next = new NodeObj();
            nodes[key].next.item = input;
        }

    }
    public void printHash() {
        for (int i = 0; i < HASH_TABLE_SIZE; i++) {
            System.out.println(nodes[i].item);
            System.out.println();
        }
    }

    public int get(String input) {
        int index = makeHashCode(input);
        int counter = 0;
        NodeObj n = nodes[index];
        while (n != null && n.getItem() != input) {
```

```
                n = n.getNext();
                counter++;
            }
            if (n == null) {
                counter++;
                return counter;
            }

            return counter;

        }
        public  int makeHashCode(String str) {
            str = str.toUpperCase();
            int length = str.length();
            int letterTotal = 0;
            // Iterate over all letters in the string, totalling their ASCII valu
            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;
            }

    }
```

As previously mentioned, the hash table class has a member which is an array of nodes. By using this array, we can put items into the hash table and also get items after they are inserted. The put function takes in the string input that is to be inserted into the table, along with at what key value it will be inserted at. The function checks if there is already a node at that key value, and if there is then another node will be created for chaining. The get function computes the key value of the string that is being looked up by using the makeHashCode function, and looks through the hash table until it is found.

3 Main

Listing 3: Hash Class

```java
import java.io.*;
import java.util.*;

public class Hash {
    final static int numItems = 666;

    public static void main(String[] args) {
        String[] magicItems = new String[numItems];
        int i = 0;

        File myFile = new File("magicitems.txt");

        try {
            Scanner fileScan = new Scanner(myFile);
        //Populates the array with the lines from text file
        while (fileScan.hasNext()) {
            magicItems[i] = fileScan.nextLine().toLowerCase();
            i++;
        }
        fileScan.close();
        }
        catch (FileNotFoundException e) {
            e.printStackTrace();
        }

        final int numElements = 42;
        String[] newArray = new String[numElements];

        //Shuffle magic items array and take the first 42 elements for the ne
        shuffle(magicItems);
        for (int j = 0; j < numElements; j++) {
            newArray[j] = magicItems[j];
        }
        //Read in magic items array again because the first one was shuffled
        String[] magicItems2 = new String[numItems];
        int j = 0;

        File myFile2 = new File("magicitems.txt");
```

```

        try {
            Scanner fileScan = new Scanner(myFile2);
        //Populates the array with the lines from text file
        while (fileScan.hasNext()) {
            magicItems2[j] = fileScan.nextLine().toLowerCase();
            j++;
        }
        fileScan.close();
        }
        catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        insertionSort(newArray);

        int linearSearchNum = 0;
        int linearSearchSum = 0;
        for (int k = 0; k < numElements; k++) {
            //Send the unshuffled magic items array
            linearSearchNum = linearSearch(magicItems2, newArray[k]);
            linearSearchSum += linearSearchNum;
        }
        int linearSearchAvg = linearSearchSum / numElements;
        System.out.println("The average number of comparisons for linear searc

        int binarySearchNum = 0;
        int binarySearchSum = 0;
        for (int x = 0; x < numElements; x++) {
            binarySearchNum = binarySearch(magicItems2, newArray[x]);
            binarySearchSum += binarySearchNum;
        }
        int binarySearchAvg = binarySearchSum / numElements;
        System.out.println("The average number of comparisons for binary searc


        HashTable hash = new HashTable();
        for (int m = 0; m < numItems; m++) {
            int hashCode = hash.makeHashCode(magicItems[m]);
            hash.put(magicItems[m], hashCode);
        }
        //hash.printHash();
        int hashComparisons = 0;
        int hashSum = 0;
        for (int k = 0; k < numElements; k++) {
            hashComparisons = hash.get(newArray[k]);
            hashSum += hashComparisons;
```

```java
        }
        int hashAvg = hashSum / numElements;
        System.out.println("The average number of comparisons for retrieving 

    }
    public static void shuffle(String[] arr) {
        Random rand = new Random();
        int index;
        String str;
        for (int i = arr.length - 1; i > 0; i--) {
            index = rand.nextInt(i + 1);
            str = arr[index];
            arr[index] = arr[i];
            arr[i] = str;
        }
    }
    public static void insertionSort(String[] arr) {
        int n = arr.length;
        String key = "";
        int i;
        for(int j = 0; j < n - 2; j++) {
            key = arr[j];
            i = j - 1;
            while (i >= 0 && arr[i].compareTo(key) > 0) {
                arr[i + 1] = arr[i];
                i = i - 1;
            }
            arr[i + 1] = key;
        }
    }
    public static int linearSearch(String[] arr, String key) {
        int comparisons = 0;
        for (int i = 0; i < arr.length; i++) {
            comparisons++;
            if (arr[i].compareToIgnoreCase(key) == 0) {
                System.out.println("The number of comparisons for linear searc
                return i;
            }
        }
        System.out.println("The number of comparisons for linear search is " +
        return -1;
    }
    //return counter
    public static int binarySearch(String[] arr, String key) {
        int low = 0;
        int high = arr.length - 1;
```

```
        int mid;
        int counter = 0;

        while (low <= high) {
            mid = (low + high) / 2;
            if (arr[mid].compareToIgnoreCase(key) == 0) {
                counter++;
                System.out.println("The number of comparisons for binary searc
                return counter;
            }
            else if(arr[mid].compareToIgnoreCase(key) < 0) {
                low = mid + 1;
                counter++;
            }
            else if (arr[mid].compareToIgnoreCase(key) > 0) {
                high = mid - 1;
                counter++;
            }

        }
        System.out.println("The number of comparisons for binary search is "
        return counter;
    }

}
```

The Hash class consists of different methods such as linear and binary search, along with code in the main method to utilize these functions and other classes. The asymptotic running times for linear search, binary search, and hashing with chaining are listed in the table below. Linear search's running time is O(n) because the entire array is being traversed in order to find the key. While the expected time is O(1/2*n), 1/2 is a constant factor that can be thrown away. For binary search, the asymptotic running time is O(logn). This is so because the array is sorted before binary search can be executed, which means that half of the elements can be eliminated by comparing the midpoint to the specified key value. Finally, for hashing and chaining the asymptotic running time for looking up an item in the hash table is a constant time operation plus the average chain length. The average chain length needs to be added because each element is a linked list, which is extra lookup time.

4 Results

	Number of Comparisons	Asymptotic Running Time
Linear Search	324	O(n)
Binary Search	9	O(log(n))
Hashing with chaining	1	O(1) + load