CMPT 435 - Fall 2022 - Dr. Labouseur

Assignment Three – Searching

Genevieve Anderson
Genevieve.anderson1@marist.edu

November 4, 2022

1 Node Class

The node class is where I build the framework for my linked list. The linked list allows for each phrase of the magic items file to be put in their own node. Inside the Node class I am building the framework for each node, the linked list is made up of several nodes.

I declared that inside each node there would be a name, and a next pointer [lines 4, 5] which will be initialized in the constructor. In the constructor [line 11] I initialized the name to ”n” and the next pointer to null to begin with. The pointer is what connects the nodes in the linked list.

Within my Node class I have two ”getters”, and two ”setters”. The ”getters” [lines 17, 21] will provide information for the node, and the ”setters” [lines 27, 31] will actually set the name and the pointer in the node. I built ”getName()” and ”getNext()” which get the name variable, and the pointer for the node. Then I built ”setName()” and ”setNode()”. These are specifically meant for setting the name in the node to a string value, and setting the pointer to the next node.

Lastly, in the Node class I created a ”toString()” method [line 36] in order for the program to set the name of the node to the result. This is so that when it comes time to print the magic items phrases from the node it will return the actual phrase rather than the object identifier.

I was able to recycle this code from Assignment 1!

```
1 // Format for linked list
2 // Building the framework for each node. the linked list is going to be made up of several nodes.
3 public class Node { // Creating the node for the linked list
```

```java
    String name = ""; // Declaring and initializing the name inside the node
    Node next = null; // Declaring and initializing the node pointer


    // Node constructor
    // Uppercase N is referring to the node class
    // Lowercase n is referring to the actual node with the information in it (could be called anything)
    public Node(String n, Node node) { // The first parameter is for the information the node is holding, the second parameter is for the pointer
        this.name = n; // Initializing the name
        this.next = null; // Initializing the pointer
    }

    // I will now build two getters: information for the node, information for the pointer
    public String getName() { // Getting the name variable from the node, returns a string
        return name; // Returns the information from the variable name
    }

    public Node getNext() { // Returning a node
        return next; // Next because we are calling the pointer next, returns the node from the variable next
    }

    // I will now build two setters: setting the name, and setting the pointer
    // Void because it isnt going to return anything
    public void setName(String n) { // Parameters always go inside parenthesis, I am only updating the name value, so we only need one parameter
        name = n;
    }

    public void setNode(Node m) { // "m" is the node that we are going to set next equal to, for the pointer
        next = m; // I am using m so that the pointer is not null. the pointer will not be null until the end of the linked list.
    }

    // toString so that the program prints what is actually inside the node rather than the object identifier
    public String toString() {
        String result = name; // setting a string equal to what is inside the node
        return result;
    }
}
```

2 Linear Search

In this class I implement the algorithm for linear search. First, I declared the String array for magic items, the target array for the 42 items, and the integer for comparisons [2-4]. Then I built a constructor [7] where I set that the linear search will use the string array of the magic items, the target array will contain the 42 items we are searching for, and the comparison counter will begin at 0.

Next, I created my "linear()" function [14] which is where the searching takes place. I used a for loop [15] that ran until the entire length of the magic items was "searched". If the target value we are searching for is found [16], then the search is complete. Inside of my linear search function I also keep track of how many comparisons are made [19]. The comparison count is increased at the end of the for loop, since this is where each comparison takes place.

Lastly, I have my "toString()" [25] function. This is where the comparisons and average are calculated and printed for the main program.

```
1 public class Linear {
2     String[] targetArray;
3     String[] magicItems;
4     int comparisons;
5 
6     // Constructor
7     public Linear(String[] magicItems, String[] targetArray) {
8         this.magicItems = magicItems;
9         this.targetArray = targetArray; // 42 items
10        comparisons = 0; // Initialize comparisons
11    }
12
13    // Function for linear search
14    public int linear(String target) {
15        for (int j = 0; j < magicItems.length; j++) { // Loop through all the magic items
16            if (target.equals(magicItems[j])) {
17                j = magicItems.length; // Ends loop without break statement - the loop goes while j is less than the magic items length
18            }
19            comparisons++;
20        }
21        return comparisons;
22    }
23
24    // Function for toString
25    public String toString() {
26        String result = ""; // Initializing result variable as empty
27        int totalComparisons = 0;
28        for (int i = 0; i < targetArray.length; i++) { // Beginning at 0; go until the end of target array; increment by 1
29            totalComparisons += comparisons;
30            comparisons = 0;
31            result += "\n" + (i+1) + ": target word: " +
```

```
    targetArray[i] + " and the number of comparisons is: " + linear
    (targetArray[i]);
        }
        double average = ((double)(totalComparisons))/targetArray.
    length; // Dobule for decimal places
        result += "\n Average: =" + String.format("%.2f", average);
     // Labeling average
        return result;
    }
}
```

3 Binary Search

In this class I implement the algorithm for binary search. First, I declared the String array for magic items, the target array for the 42 items, and the integer for comparisons [2-4]. Then I built a constructor [7] where I set that the binary search will use the string array of the magic items, the target array will contain the 42 items we are searching for, and the comparison counter will begin at 0.

Next, I created my "binary()" function [14] which is where the searching takes place. First, I find the midpoint of the items, then I repeatedly divide the list of items in half until the item I am searching for is found. If the item is not found, then I return a negative number [18] because it can't be true that there is a negative index in the list of items. Then I considered whether the item we are searching for happens to be the midpoint [20], which should eventually be true after some number of times splitting the array. Before the target is found, for the first and second half of the arrays (since we split them in the beginning) the binary search is recursively called [20-24].

I also count comparisons inside my "binary()" function after each split of the list happens [16]. Lastly, I created my "toString()" function [32]. This is where the comparisons and average are calculated and printed for the main program.

```
1  public class BinarySearch {
2      String[] targetArray;
3      String[] magicItems;
4      int comparisons;
5  
6      // Constructor
7      public BinarySearch(String[] magicItems, String[] targetArray) {
8          this.magicItems = magicItems;
9          this.targetArray = targetArray;
10         comparisons = 0;
11     }
12 
13     // Function for Binary Search
14     public int binary(String [] A, int start, int stop, String target) {
15         int midPoint = (start+stop)/2; // Finding the midpoint
16         comparisons++; // Counting comparisons here
17         if (start > stop) {
18             return -1; // This is so that we know the target was no found because negative indexes dont exist in arrays
19         }
20         else if (target.equals(A[midPoint])) {
21             return midPoint; // midPoint is the index of what we are searching for
22         }
23         else if (target.compareTo(A[midPoint])<0) {
24             return binary(A, start, midPoint, target); // Recursion for the first half of the array
25         }
```

```java
        else { // It wasnt the other scenarios so the target must be larger than the midpoint or not there at all
            return binary(A, midPoint + 1, stop, target); // Recursion for the second half of the array
        }
    }

      // Function for toString
      public String toString() {
        String result = ""; // Initializing result variable as empty
        int totalComparisons = 0;
        for (int i = 0; i < targetArray.length; i++) { // Beginning at 0; go until the end of target array; increment by 1
            totalComparisons += comparisons;
            comparisons = 0;
            result += "\n" + (i+1) + ": target word: " + targetArray[i] + " which was found at index:  " + binary(magicItems, 0, magicItems.length, targetArray[i]) + " and the number of comparisons is: " + comparisons;
        }
        double average = ((double)(totalComparisons))/targetArray.length; // dobule for decimal places
        result += "\n Average: =" + String.format("%.2f", average); // Labeling average
        return result;
    }
}
```

4 Quick Sort

In this class I implement the algorithm for quick sort. In binary search, it is important that the list is sorted otherwise it would not work properly and not be as efficient! First, I declared the String array which will contain magic items and the declared the comparisons variable [4, 5]. Then, I built a constructor [8] where I set that the insertion sort will use the string array of the magic items.

Next, I created my "choosePivot()" function [18] for choosing a pivot value. The function works by choosing a random value located somewhere between the highest value and the lowest value. This is so that the algorithm will not have to pivot around the lowest or highest value, which would take much longer.

Then, I created a simple "swap()" function [27] to perform the swapping needed in the algorithm. This function uses a temporary string array to swap the "lesserIndex" and the "greaterIndex".

Next, I created my "partition()" function [35]. This function splits the values into lesser and greater arrays. Within this function, the "choosePivot" function I created is called, and chooses a pivot value. Inside this function I have a for loop with an if statement inside. This function handles the swapping within the quick sort. I also count comparisons within this function.

Inside of my "partition()" function I also keep track of how many comparisons are made [44].

Then, I have my "quick()" function [51]which is where the algorithm recursively calls upon itself until the list of items is sorted. Lastly, I have my "toString()" function [61]. This is where I set the result returned equal to the amount of comparisons made during the quick sort.

I was able to recycle this code from Assignment 2!

```
1 import java.util.*;
2
3 public class Quick {
4     String[] magicItems; // Declaring that there is a string array of magic items
5     int comparisons = 0; // Declaring comparisons
6
7     // Quick sort constructor
8     public Quick(String[] magicItems) {
9         this.magicItems = magicItems;
10    }
11
12    // Thought process
13        // First, choose random value as pivot
14        // Compare all other values to the pivot and store them in arrays for lesser values, and greater values
15        // Repeat this process until the magic items are fully sorted
16
17    // To preseve O(nlogn) I will attempt to choose a pivot which is not the highest or lowest value
18    public void choosePivot(String[] A, int low, int high) { //
19        Random random = new Random();
```

```
        int pivot = random.nextInt(high - low) + low; // Doesn't allow for pivot value of the highest or lowest value, but rather a random value inbetween
        String temp = A[pivot]; // Setting the temporary variable to pivot
        A[pivot] = A[high]; // Swapping
        A[high] = temp;
    }

    // Function for swapping
    public void swap(String[] A, int lesserIndex, int greaterIndex) { // Array that will be used for swapping, the index of the lesser value, the index of the greater value
        String temp = A[lesserIndex]; // Setting the temporary variable to lesser index
        A[lesserIndex] = A[greaterIndex]; // Swapping
        A[greaterIndex] = temp;
    }

    // Function for partition
    // Splitting the magic items around the pivot into lesser and greater arrays
    public int partition(String[] A, int low, int high) {
        choosePivot(A, low, high); // Choosing pivot value
        String pivot = A[high]; // Setting pivot value
        int i = low - 1;
        for (int j = low; j < high; j++) {
            if (A[j].compareTo(pivot) < 0) { // If the second value is less than the last item in the array (pivot)
                i = i + 1;
                swap(A, i, j); // Swapping the value located at i, with a smaller value
            }
            comparisons++; // Counting comparisons here
        }
        swap(A, i + 1, high); // Swapping so that the pivot value is where it belongs
        return i + 1;
    }

    // Function for quick sort
    public void quick(String[] originalA, int startIndex, int endIndex) { // Parameters start and end index so that when the array is getting split we can reference where to split
        if (startIndex < endIndex) {
            int split = partition(originalA, startIndex, endIndex);
            // Recursion
            quick(originalA, startIndex, split - 1); // Quick method from 1 to end of the values before the pivot value
            quick(originalA, split+1, endIndex); // Quick method from first value after the pivot to the end of the array
        }
    }

    //Function for toString
    public String toString() {
```

```
        String result = ""; // Initializing result variable as empty
        /*for (int i = 0; i < magicItems.length; i++) { // Beginning at 0; go until the end of magic items; increment by 1
            // Setting result to a string of all of the magic items
            result += magicItems[i] + "\n"; //+= is adding each result on to the previous result, \n is so that it gets split up by line
        }*/
        result += "Quick sort comparisons: " + comparisons; // Labeling the comparisons
        return result;
    }
}
```

5 Hash table

In this class I implement the algorithm for hash table. First, I declared that the size of the hash table would be 250 [5]. Then, I declared the String array for magic items, the target array for the 42 items, the Node array for the hash table, and the integer for comparisons [7-9]. Then I built a constructor [13] where I set that the hash table will use the string array of the magic items, the target array will contain the 42 items we are searching for, and the comparison counter will begin at 1 since we will always do at least one comparison. I also populate the hash table in my constructor.

Next, I created my "populate()" function [21] which is how the hash table is populated in the constructor. Each magic item will go in one of the 250 spots. Since there are only 250 spots though, there will be collisions so the magic items will be in "chains" at some spots in the hash table. I used an if else statement [26, 29] to deal with actually putting the magic items in nodes. Inside my else statement, I have a while loop [30] for how to add the magic items to the "chain" at some location is there is already a magic item there.

Next, I created my "find()" function [39] which is how we find the target value in the hash table. I use a while loop [43] to search through the hash table by using the index of where the target should be inside the hash table. If the item is in a chain at said location, then my code also accounts for that. If the value is not found, then my code returns a negative number [48] since a negative index doesn't exist in the list.

Next, I used the "makeHashCode()" function [53] given to us in class. This function takes the ascii values from the string and turns them into integers. The function adds the ascii value of each character together which makes up the final value. This is how we figure out where things go in the hash table.

I count comparisons [42, 44] in the "find()" function. I count them in the prior to my while loop since since there will always be one comparison made at each index, then I also count comparisons inside my while loop to account for the chaining of some items.

Lastly, I created my "toString()" function [69]. This is where the comparisons and average are calculated and printed for the main program.

```
1  import java.util.Arrays;
2  
3  public class Hash {
4  
5      private final int HASH_TABLE_SIZE = 250;
6  
7      String[] magicItems;
8      String[] targetArray;
9      Node[] hashTable = new Node[HASH_TABLE_SIZE];
10     int comparisons;
11 
12     // Constructor
13     public Hash(String[] magicItems, String[] targetArray) {
14         this.magicItems = magicItems;
```

```java
        this.targetArray = targetArray; // 42 items
        comparisons = 1; // Because we always do at least one comparison
        populate(); // Actually populating the hash table
    }

    // Function for populating the hash table
    public void populate() {
        for (int i = 0; i < magicItems.length; i++) {
            int hash = makeHashCode(magicItems[i]); // Finding a number from 0 to 250 for where to put the magic item in the hash table
            Node hashNode = new Node(magicItems[i], null); // Putting the magic items in node with null pointer
            Node currentNode = hashTable[hash]; // hash is the current spot that we are looking at
            if (currentNode == null) {
                hashTable[hash] = hashNode; // The node with the magic item - actually putting the magic items in the hash table
            }
            else {
                while (currentNode.getNext() != null) {
                    currentNode = currentNode.getNext(); // Getting the next item from the chain
                }
                currentNode.setNode(hashNode); // Adding magic item to the chain if something is already there
            }
        }
    }

    // Function for finding the target value
    public int find(String target) {
        int hash = makeHashCode(target); // Getting the index of where the target should be inside the hash table
        Node currentNode = hashTable[hash];
        comparisons++;
        while (currentNode != null && !currentNode.getName().equals(target)) { // While the current node is not null and does not equal the target value
            comparisons++;
            currentNode = currentNode.getNext(); // Getting the next item from the chain
        }
        if (currentNode == null) {
            hash = -1; // Negative number becuase we know there isnt a negative index
        }
        return hash; // Reutrning the index where the target is
    }

    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 values.
```

```java
        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
        return hashCode;
    }

    // Function for toString
    public String toString() {
        String result = ""; // Initializing result variable as empty
        int totalComparisons = 0;
        for (int i = 0; i < targetArray.length; i++) { // Beginning at 0; go until the end of target array; increment by 1
            totalComparisons += comparisons;
            comparisons = 1; // Because every get is one compare
            result += "\n" + (i+1) + ": target word: " + targetArray[i] + " which was found at index:  " + find(targetArray[i]) + " and the number of comparisons is: " + comparisons;
        }
        double average = ((double)(totalComparisons))/targetArray.length; // dobule for decimal places
        result += "\n Average: =" + String.format("%.2f", average); // Labeling average
        return result;
    }
}
```

6 Main Program

My main program is where I run my searches and analyze the outputs! To actually upload the file of magic items, I created a try and catch statement [7-22]. I put all of the phrases into their own node, because a linked list is made up of nodes [16].

Before each search is run, I sort my magic items [25, 26] using my quick sort from assignment 2. It doesn't matter if the items are sorted for linear search, but it is very important that the items are sorted for binary search. Then, I created a string array "targetArray" [29] for 42 items that will be searched. The same items will be searched for each search algorithm. I used a for loop [32] which uses the build in random function to randomly choose the 42 items to search for.

Then, I run linear search [39]. The running time for linear search is $O(n)$ since we are simply just going through the items one by one in a single loop. You can notice that the words that are being searched which have a first character near the beginning of the alphabet have a small number of comparisons vs. the words that begin with a character towards the end of the alphabet. This is because the list was sorted prior to the algorithm being run!

Then, I run binary search [45]. The running time for binary search alone is $O(log_2 n)$, however for binary search to work the items must be sorted prior to running the search. So we also have to take into account the $O(nlog_2 n$ running time of the sort algorithm.

Lastly, I run my hash table [49]. The running time for the "get" is $O(1) + \frac{number\ of\ items}{size\ of\ list}$. The "get" running time for this specific code would be $O(1) + 2.66$. The running time for "put" is simply $O(1)$.

I included a table below which provides the results of each search!

Linear search	Binary search	Hash table
278.12	8.33	3.5

```
1 import java.util.*;
2 import java.io.*;
3
4 public class Main {
5     public static void main(String[] args) {
6         String[] magicItems = new String[666];
7         try { //Trying to find the file
8             File file = new File("magicitems.txt");
9             Scanner sc = new Scanner(file);
10            int index = 0;
11
12            while (sc.hasNextLine()) {
13                String item = sc.nextLine();
14                item = item.toUpperCase();
15                item = item.replaceAll("\\s+","");
16                Node magicItemsNode = new Node(item, null);
17                magicItems[index ++] = magicItemsNode.getName();
```

```java
            }
        }
        catch (FileNotFoundException e) { // If we cant find the file
            e.printStackTrace();
        }

        // Sorting the magic items
        Quick quick = new Quick(magicItems);
        quick.quick(magicItems, 0, magicItems.length - 1);

        // Search the same 42 items for each search and setting them to target array
        String[] targetArray = new String[42]; // String array with 42 items

        // Actually getting 42 random values
        for (int i = 0; i < targetArray.length; i++) {
            Random random = new Random();
            int randInt = random.nextInt(magicItems.length);
            targetArray[i] = magicItems[randInt]; // Randomly choosing the items we are going to search for in the binary search
        }

        // Linear search
        Linear linear = new Linear(magicItems, targetArray);
        System.out.println(linear);
            // Notice that some comparisons are much higher than others
            // The phrases that start with "A" have much smaller number of comparisons since we alphabetically sorted the list prior to searching

        // Binary search
        BinarySearch binary = new BinarySearch(magicItems, targetArray);
        System.out.println(binary);

        // Hashing
        Hash hash = new Hash(magicItems, targetArray);
        System.out.println(hash);
    }
}
```