Assignment Three

Veronica Longley

November 5, 2021

Our Goal: In this assignment we will compare Linear Search, Binary Search and a Hash Table. We will be searching for a random 42 magic items from a list of 666 magic items. We will count the number of comparisons to find each of the 42 sub-array items for each of the searching implementations as well as the average number of comparisons for each of the three searches.

1 Linear Search

```
1   static int linearSearch(String items[], String target)
2   {
3           int comparisons = 0;
4           int l = 0;
5           //while the item we are checking does not match the target
6           // move to next item and increment comparisons
7           while(items[l] != target)
8           {
9                   comparisons++;
10                  linearComparisons ++;
11                  l++;
12          }//while
13
14          return comparisons;
15  }//linear search
```

Linear search works by going through the entire list of items until the one that matches the one were looking for is found. As seen in line 8, while the item is not equal to the target we increment the number of comparisons as well as the counter through the list of 666 magic items. In main we call this method for each of the 42 sub-array items and output the number of comparisons for each one as well as the average number of comparisons for linear search. The global variable on line 10 is incremented for all comparisons for all 42 sub-array items and then divided by 42 later in main to output the average number of comparisons. This will be around 333 because sometimes the items will be in the beginning of the list and require less comparisons while sometimes the item will be towards the end of the list and require more. On average over the 42 items, it will be very close to 333 as this is half of 666; similarly while linear search is O(n), on average it will be much closer to O(n/2).

2 Binary Search

```
1     static int binarySearch(String[] items, String target)
2       {
3           int left = 0;
4           int middle = 0;
5           int right = items.length - 1;
6           int pos = 0;
7           int index = -1;
8           int indivComp = 0;
9
10          //while left is less than or equal to right and index has not
11          // been found increment comparisons and divide again
12          while (left <= right && index ==-1) {
13
14                  binaryComparisons++;
```

```
                    indivComp++;
                middle = left + (right - left) / 2;

                pos = target.compareTo(items[middle]);

                // Check if x is present at mid
                if (pos == 0)
                    index = middle;

                // If x greater, ignore left half
                if (pos > 0)
                    left = middle + 1;

                // If x is smaller, ignore right half
                else
                    right = middle - 1;
            }//while

            return indivComp;
        }//binary search
```

Binary search is a bit more complicated. Binary search works by picking the middle element of an ordered list and checking if the target is equal to, less than, or greater than it. If it is equal to the search is over and the number of comparisons is returned. If the value we look at is less than the target then we ignore the lower half of the ordered items as if the one in the middle is less than the target all the ones before it must also be less than the target because the list is ordered. This is run recursively until the middle element picked is the target. Because we are able to cut the list in half each time we recursively call binary search, it is O(log(n)). For this case, we saw about 8 comparisons were needed to locate the items on average.

3 Hash Table

```
public class HashTable {

        private static final int HASH_TABLE_SIZE = 250;

        public static class HashObj
        {
                public String key;
                public String value;
        }//HashObj

        private LinkedList<HashObj>[] arr = new LinkedList[HASH_TABLE_SIZE];

        //initialize to null
        public HashTable()
        {
                for(int i = 0; i < HASH_TABLE_SIZE; i++)
                {
                        arr[i] = null;
                }//for
        }//HashTable

        private HashObj getObj(String key)
```

```
23          {
24                  int index = -1;
25
26                  if (key == null)
27                          return null;
28                  index = makeHashCode(key);
29                  LinkedList<HashObj> items = arr[index];
30                  if(items == null)
31                          return null;
32                  for(HashObj item : items)
33                  {
34                          if(item.key.equals(key))
35                                  return item;
36                  }//for
37
38                  return null;
39          }//HashObj
40
41          //count comparisons
42          public int objComps(String key)
43          {
44                  int hashTableComparisons = 0;
45
46                  if(key == null)
47                          return -1;
48                  int index = makeHashCode(key);
49                  LinkedList<HashObj>items = arr[index];
50
51                  if(items == null)
52                  {
53                          return -1;
54                  }//if
55
56                  for(HashObj item : items )
57                  {
58                          hashTableComparisons ++;
59                          if(item.key.equals(key))
60                                  return hashTableComparisons;
61                  }//for
62
63                  return -1;
64          }//objComps
65
66          public String get(String key)
67          {
68                  HashObj item = getObj(key);
69                  if(item == null)
70                          return null;
71                  else
72                          return item.value;
73          }//get
74
75
76          public void put(String key, String value)
77          {
78                  int index = makeHashCode(key);
79                  LinkedList<HashObj> items = arr[index];
80                  if(items == null)
```

```
81                  {
82                          items = new LinkedList<HashObj> ();
83                          HashObj item = new HashObj();
84                          item.key = key;
85                          item.value = value;
86                          items.add(item);
87                          arr[index ] = items;
88                  }//if
89
90                  else
91                  {
92                          for(HashObj item : items )
93                          {
94                                  if(item.key.equals(key))
95                                  {
96                                          item.value = value;
97                                          return;
98                                  }//if
99                          }//for
100
101                         HashObj item = new HashObj();
102                         item.key = key;
103                         item.value = value;
104                         items.add(item);
105                 }//else
106         }//put
107
108
109         private static int makeHashCode(String str) {
110               str = str.toUpperCase();
111               int length = str.length();
112               int letterTotal = 0;
113
114               // Iterate over all letters in the string, totaling their ASCII values.
115               for (int i = 0; i < length; i++) {
116                  char thisLetter = str.charAt(i);
117                  int thisValue = (int)thisLetter;
118                  letterTotal = letterTotal + thisValue;
119
120                  // Test: print the char and the hash.
121                  /*
122                  System.out.print(" [");
123                  System.out.print(thisLetter);
124                  System.out.print(thisValue);
125                  System.out.print("] ");
126                  // */
127               }
128
129               // Scale letterTotal to fit in HASH_TABLE_SIZE.
130               int hashCode = (letterTotal * 1) % HASH_TABLE_SIZE;
131               // % is the "mod" operator
132               // TODO: Experiment with letterTotal * 2, 3, 5, 50, etc.
133
134               return hashCode;
135            }//makeHashCode
136
137
138 }//HashTable
```

Hash tables work by converting each string into ASCII values then storing the string in the element of the array associated with the ASCII value mod the size of the hash table. We used a hash table of size 250 which means we will have to deal with collisions, multiple strings being stored at the same bucket of the hash table, as we have 666 items to organize and only 250 buckets. To deal with this, each bucket references a linked list where as many items as needed can be added. This means items with (ASCII values mod 250) values that are the same can reference each other, and when we go to look an item up we convert it to ASCII, calculate it's mod 250, go to that bucket, and linear search that bucket until we find it or establish it is not present.

We have several methods in the Hash Table class such as getObj, objComp, get, put, and makeHashCode. Put is responsible for adding the 666 items into the hash table. Get and getObj are used to locate the 42 items from the sub-array in the hash table of all 666 items. MakeHashCode is used to determine what bucket the item we are trying to add or look up should be in. It converts the string to ASCII values then calculates that value mod 250 to locate the bucket we will be adding to or searching through. ObjComp will be used for this assignment to count the number of comparisons to locate each item in the hash table. We will also have a global variable hashTableComparisons in main that will sum all of the comparisons for all the 42 look ups and divide by 42 to find the average number of comparisons to locate an item using a hash table.

4 ...In Main Method

```
1  public class main {
2
3          public static int linearComparisons;
4          public static int binaryComparisons;
5          public static int hashTableComparisons;
6
7  static Scanner keyboard = new Scanner(System.in);
8
9          public static void main(String[] args)
10         {
11                 // TODO Auto-generated method stub
12                 // TODO Auto-generated method stub
13
14                 //initialize variables used to read in items from list
15                 String fileName = "";
16                 final int NUMOFITEMS = 666;
17                 String [] magicItems = new String [NUMOFITEMS];
18                 final int SUBARRAYSIZE = 42;
19                 String [] subArray = new String [SUBARRAYSIZE];
20                 String theitem = null;
21                 fileName = "magicitems.txt";
22                 int linSearchComp = 0;
23                 int binSearchComp = 0;
24
25                 //input item names from file and store in array
26                 try
27                         {
28                         Scanner readFile = new Scanner (new File (fileName));
29                         int i = 0;
30                         while(readFile.hasNextLine())
```

```
                        {
                                theitem = readFile.nextLine();
                                magicItems[i] = theitem;
                                i++;
                        }//while
                        readFile.close();
                        }//try
                catch(FileNotFoundException ex)
                        {
                        System.out.println("Failed to find file: "+ fileName);
                        }//catch
                catch(InputMismatchException ex)
                        {
                        System.out.println("Type mismatch. ");
                        System.out.println(ex.getMessage());
                        }//catch
                catch(NumberFormatException ex)
                        {
                        System.out.println("Failed to convert String. ");
                        System.out.println(ex.getMessage());
                        }//catch
                catch(NullPointerException ex)
                        {
                        System.out.println("Null pointer exception. ");
                        System.out.println(ex.getMessage());
                        }//catch
                catch(Exception ex)
                        {
                        System.out.println("Oops, something went wrong. ");
                        ex.printStackTrace();
                        }//catch

                //shuffle the 666 magic items
                shuffle(magicItems, NUMOFITEMS);

                //pick the first 42 and store them in a sub-array
                for (int j = 0; j < 42; j++)
                {
                        subArray[j] = magicItems[j];
                }//for

                //use merge sort to order the original 666 magic items
                mergeSort(magicItems);

                //linear search for each of the 42 sub-array items
                //print the number of comparisons associated with each item
                System.out.println("Linear Search: ");
                for(int i = 0; i<SUBARRAYSIZE; i++)
                {
                        linSearchComp = linearSearch(magicItems, subArray[i]);
                        System.out.println(linSearchComp+ " comparisons to locate "
                        + subArray[i]);;

                }

                //Print the average number of comparisons for linear search
                System.out.println("The average number of comparisons using
                linear search: "+ linearComparisons/SUBARRAYSIZE);
```

```
                System.out.println();

                System.out.println("Binary Search: ");
                //binary search for each of the 42 sub-array items
                //print the number of comparisons associated with each item
                for(int i = 0; i<SUBARRAYSIZE; i++)
                {
                        binSearchComp = binarySearch(magicItems, subArray[i]);
                        System.out.println(binSearchComp+ " comparisons to locate "
                        + subArray[i]);
                }

                //Print the average number of comparisons for binary search
                System.out.println("The average number of comparisons using binary
        search: "+ binaryComparisons/SUBARRAYSIZE);

                System.out.println();
                System.out.println("Hash Table: ");
            HashTable ourHashTable = new HashTable();

                //fill hash table
                for(int i = 0; i<NUMOFITEMS; i++)
                {
                        ourHashTable.put(magicItems [i], magicItems[i]);
                }


                //locate the 42 sub-array items in hash table
                //print the number of comparisons associated with each item
                for(int j = 0; j < SUBARRAYSIZE; j++)
                {
                        int itemcomp = 0;
                        itemcomp = ourHashTable.objComps(subArray[j]);
                        if (itemcomp == 1)
                                System.out.println(itemcomp + "comparison to locate"
                                + ourHashTable.get(subArray[j]));
                        else
                                System.out.println(itemcomp + "comparisons to locate"
                                + ourHashTable.get(subArray[j]));
                        hashTableComparisons += itemcomp;
                }//for


                System.out.println();
                System.out.println("The number of comparisons using a hash table:"
                + hashTableComparisons / SUBARRAYSIZE);

        }//main
```

To being main we will introduce three global variables to count all the comparisons for each of the three searches then divide by 42, the number of sub-array items, to determine the average number of comparisons for each search. Next, we will read in the file containing the 666 magic items. We then shuffle the 666 magic items to ensure the 42 we choose are truly random. After they are shuffled, we choose the first 42 items to be our sub-array. These will be the items we locate in the list of 666 items. Next we use merge sort to sort the original 666 magic items (the items will need to be sorted for binary search). First we

will look at linear search. We use a for loop to call the linear search method for each of the 42 sub-array items and store and output the number of comparisons for each item. We also output the average number of comparisons by dividing the global counter, which is updated in the linear search method, by 42, the number of searches we conducted. Next is binary search. Similarly to linear search, we use a for look to call the binary search method for each of the 42 sub-array items, output the number for each item as well as the global counter divided by 42 to indicate the average for binary search. Lastly, for the hash table we must first create the hash table, line 107, then fill it using the put method for each of the 666 magic items. To locate the items we use another for loop and the objComps method to output the number of comparisons for each item. Lastly, we want to add this to the global counter for hash tables and once all 42 items are located, we can divide by 42 to output the average number of comparisons for searching using a hash table.

5 Conclusion

The table below shows the searching method, the running time, and the average number of comparisons when searching through 666 items.

Search	Running Time	Comparisons
Linear	O(n)	333
Binary	n^2	8
Hash Table	O(1) + load	2

It is clear from the table above the ideal searching method would be a hash table. Maybe if you were only going to be searching for an element a few times using binary search would be more efficient just because creating a hash table is much more complicated than creating a binary search method. As for linear search, this may be useful if you were looking for an item in a list of 15 or 20 items, but probably not much more. It is overly clear here that linear search, which uses on average 333 comparisons, does not measure up to a hash table, which only needs 2 on average. It is also important to note that depending on the number of collisions, the average chain length of each bucket will vary. This means the look up time of O(1) + load is truly dependent on the load/average chain length. This is calculated by the number of items, 666, divided by the number of buckets, 250. In our case the load of our hash table is 2.664 which means on average the lookup time for an item will be O(1) + 2.664, which makes sense with our results.