CMPT 435 - Fall 2021 - Dr. Labouseur

Assignment Three

Jake Vissicchio
Jake.Vissicchio1@Marist.edu

November 5, 2021

1 Results

These are overall results through the searching of 42 random items within the array.

Sorting Magic Items		
Search Type:	Average Comparisons:	Asymtotic Running Time:
Linear Search	292.93	$O(n)$
Binary Search	9.45	$O(log(n))$
Hash Table Get	1.95	$O(1)$

2 Explanations

Linear Search is $O(n)$ because it uses a loop to go through the array until it is empty or until it reaches the index where the item is.
The average should be around n/2.

Binary Search is $O(log(n))$ because it recursively divides the array until it finds the item or goes through the array without finding it. Think of how Merge Sort is $O(nlog(n))$ because the divide step is $O(log(n))$ and the conquer step is $O(n)$, but Binary Search does not have a conquer step. This leaves us with $O(log(n))$.

Hash Table Get is $O(1)$ because it will instantly go to the index
in the array of the item based on the hashing function, then go through the
queue until we get the item. So in reality the complexity is $O(1+chainLength)$
which is why our average number of comparisons is above 1.

3 Code Listings

3.1

Linear Search Code with Comments:

```
//Uses a given array and a specified target to go through array
//to search for the specified item until we find it or we reach
//the end while counting the comparisons.
//Will also print out if the item was found or not.
//Returns the compCount to be used in main to find the average.
public int linearSearch(String array[], String target)
{
int i = 0;
boolean found = false;
int compCount = 0;
int result = -1;
while (i < array.length && result != 0)
{
    compCount++;
    result = array[i].compareToIgnoreCase(target);
    if (result == 0)
    {
        System.out.println("The item, " + target + ","
            + " was found at index " + i);
        found = true;
    }//if
    i++;
}//while
if (found == false)
{
    System.out.println("The item, " + target + ", was not found");
}
System.out.println("Linear Search had " + compCount + " comparisons.");
System.out.println("");

return compCount;
}//linearSearch
```

3.2

Binary Search Code with Comments:

```
//Binary Search is similar to Merge Sort's Divide step
//without the Conquer Step
public class BinarySearchVissicchio {

boolean found = false;
int compCount = 0;

void binarySearch(String array[], String target, int left, int right)
{
//will allow us to divide using recusrsion until left is greater than right
if (left <= right)
{
    compCount++;
    /*
    mid represents the middle of the array and is the indicator to know if
    searching for the target was successful
    */
    int mid = (right + left)/2;
    int result = array[mid].compareToIgnoreCase(target);
    //if array[mid] = target, the target was found and the search is over
    if (result == 0)
    {
        System.out.println("The item, " + target + ", was found at index "
            + mid);
        found = true;
    }
    //the target is to the left of mid
    //split off to the left of mid and continue the search using recursion
    if (result > 0)
    {
        binarySearch(array, target, left, mid - 1);
    }
    //the target is to the right of mid
    //split off to the right of mid and continue the search using recursion
    else
    {
        binarySearch(array, target, mid + 1, right);
    }
}//if
//Will print out if the item was not found
if (found == false)
{
```

```
    System.out.println("The item, " + target + ", was not found");
    /*
    found is set to true so that the string is not repeatedly printed
    every step of recursion where the item is not found
    */
    found = true;
}

    }//binarySearch

    //will print out all of the comparisons found through binary search
    //(should be smaller than linear search)
    public int binaryComp(String array[], String target, int left, int right)
    {
        compCount = 0;
        binarySearch(array, target, left, right);
        System.out.println("Binary Search had " + compCount + " comparisons.");
        System.out.println("");
        //return the compCount to be used to find the average comparisons
        return compCount;
    }//binaryComp

}//BinarySearchVissicchio
```

3.3

Hash Table Insert with Comments:

```
public void insertHash(String newItem)
{
//find the index for the item to be inserted to based on
//the hashing function
int index = hash.makeHashCode(newItem);
//the index is null, declare it as a new queue
//then enqueue the newItem into it
if (hashTable[index] == null)
{
    hashTable[index] = new QueueVissicchio();
    hashTable[index].enqueue(newItem);
}
//The queue already exists in this index so enqueue the newItem
//into it to create chaining which avoids collisions
else
{
    hashTable[index].enqueue(newItem);
```

```
}
}//insertHash

//NOTE: I used a queue to implement chaining within my hashTable
//making it into an array of queues.
```

3.4

Hash Table Get with Comments:

```
public int getHash(String target)
{
boolean found = false;
int compCount = 0;
int result = -1;
//find the index where the target item is based on the hashing function
int index = hash.makeHashCode(target);

//this means that the index is null and does not have a
//queue therefore the target item is not found
if (hashTable[index] == null)
{
    compCount++;
    found = false;
}
else
{
    result = hashTable[index].dequeue().compareToIgnoreCase(target);
    //Go through the queue within the index until the target
    //item is found or the queue is empty
    while (!hashTable[index].isEmpty() && result != 0)
    {
        compCount++;
        result = hashTable[index].dequeue().compareToIgnoreCase(target);
    }
    if (result == 0)
    {
        compCount++;
        found = true;
        System.out.println("The item, " + target + " was found.");
    }
    else
    {
        compCount++;
        found = false;
```

```
    }
}
if (found == false)
{
    System.out.println("The item, " + target + " was  not found.");
}

//print the number of comparisons
System.out.println("Hash search had " + compCount + " comparisons.");
System.out.println("");

//return the compCount to be used to find the average comparisons
return compCount;
}//searchHash

//NOTE: Because dequeue is being used the array must be emptied and
//reinserted for each get
```

3.5

How did I choose the 42 random items to search?

```
/*
Making use of KnuthShuffle to randomly take 42 items from the temp
array and place them into a new array of 42 items to use for searching
while still keeping the original array sorted to be used for binary search.
*/

for (i = 0; i < tempArray.length; i++)
{
        tempArray[i] = items[i];
}//for
knuth.shuffle(tempArray);
for (i = 0; i < randoItems.length; i++)
{
        randoItems[i] = tempArray[i];
}//for
```