Assignment Three – LaTeX Sorts

Daniel Bilotto
danielbilotto1@marist.edu

November 6, 2021

1 Search Class

This class is the main class for the lab. It includes methods for both linear and binary search. As well as the method ”main”.

2 Main Method

So jumping right into main, the first thing main does is initialize all of the variables and objects it needs which includes hash maps, nodes, and such. After that it will make a array called list and fill it with the items from magicitems.txt. Then it will make another array BEFORE sorting the array and that will be our 42 random items that we will use to test with later. After that it will run and print out the comparisons for linear and binary search as well as hash maps, more on that below.

```
public static void main(String[] args)
  {
    String fileName = null;
      String line = null;
      int size = 666;
      String[] list = new String[size];
    String[] ft = new String[42];
    float bcompare = 0;
    float hcompare = 0;

    HashMap hMap = new HashMap();

    try
      {
        fileName = "magicitems.txt";

        File theFile = new File(fileName);

        Scanner input = new Scanner(theFile);

        for (int i = 0; i<size; i++)
        {
          line = input.nextLine().toLowerCase();
          list[i] = line;
        }

        input.close();
        keyboard.close();
      }//try

      catch(Exception ex)
      {
        System.out.println("Oops, something went wrong!");
      }//catch

      //this sends back a message if something goes wrong in
    importing the text into the array from magic items

    for (int k = 0; k < ft.length; k++)
    {
      line = list[k];
        ft[k] = line;
    }

    QuickSort.sort(list, 0, list.length -1);

    System.out.println("linear search: " + linear(list, ft));
    for (int j = 0; j < ft.length; j++)
    {
      binary(list, ft, j);
      bcompare += binary(list, ft, j);
    }
    System.out.println("binary search: " + bcompare/ft.length);



        for (int w = 0; w < size; w++)
    {
      hMap.add(list[w]);
    }

        for (int p = 0; p < ft.length; p++)
        {
          String want = list[p];
          hcompare += hMap.lookup(want);
        }

        System.out.println("Hash Map: " + hcompare);


  }//main
```

3 Linear Search Method

This is my linear search function. It is super simple, all it does is makes a link list (in this case a queue) and runs through the list checking every item with the item I tell it too from the 42 item list. Notice how I'm passing two arrays, the first array is the whole 666 items and the second is the 42 items and in this class from lines 13 to 22 I loop to the size of the second array for the 42 items. Linear search has a complexity is between Big O of 1 in best case scenarios and Big O of n in worst case. Best case is would be if it had to search a small amount. And worst case if it had to search the whole list every time.

```
1  public static float linear (String[] items, String[] ft)
2    {
3      QueueBilotto que = new QueueBilotto();
4      float compare = 0;
5
6      String want = null;
7      for (int i = 0; i < items.length; i++)
8          {
9            que.enqueue(items[i]);
10         }
11
12
13     for (int k = 0; k < ft.length; k++)
14     {
15       NodeBilotto curr = que.head;
16       want = ft[k];
17       while ((curr != null) && (curr.getData() != want))
18       {
19         curr = curr.getNext();
20         compare++;
21       }
22     }//for
23
24     return (compare/ft.length);
25   }//linear
```

4 Binary Search Method

This is the method for binary search. It also is pretty simple, I'm passing it the two arrays (already explained in linear search) and a integer called times (more on that in a second). Basically there are 3 integers called low, med and high. As shown by line 12, when low is lesser or equal to high, mid will be the sum of the two divided by 2. From there it will start to compare, so if mid is less than the item I want, then the number low will go up and vice versa if the item I'm looking for is alphabetically higher then high will go down. Basically it's p[laying a giant game of higher or lower (which is why the data needs to be sorted!!!). It splits the table in high and checks to see if that half way point is high or lower alphabetically. The complexity is best case Big O of 1 (constant time) or worst case Big O of log n. That would only happen if it gets super unlucky and like picks the worst number each and every time.

```java
public static float binary(String[] items, String[] ft, int times)
  {
    int low = 0;
    int high = items.length;
    int mid;
    String want;
    float compare = 0;

      want = ft[times];
      while (low <= high)
      {
        mid = (low + high) / 2;

        if (items[mid].compareToIgnoreCase(want) < 0)
        {
          low = mid + 1;
          compare++;
        }
        else if (items[mid].compareToIgnoreCase(want) > 0)
        {
          high = mid - 1;
          compare++;

        }
        else
        {
          return compare;
        }
      }//while

    return -1;
  }//binary
```

5 HashMap Class

Okay this is one of the bigger classes. The first method ”makeHashCode” is the one provided by our amazing professor. Basically what that function does is takes the length of each item and mods it by the set length of the map in this case 250. This gives us an index for each item and tells us where the clashing is happening. So we can take this number and as said before use it as a index. For example lines 47 to 50 and 69 to the end. I use the number as an index so I can tell my program to work there. Speaking of which let's talk about the add method. All it does is sees if there is a item already in a part of the table and if there is, follow that path until it finds room. You can see that in the while loop on line 53 with its conditions. The look up function does something similar. Since we know that the map is already populated we don't need to bother with checking if there is a null. All we need to do is go through the map until we find the item we are looping for! The while loop on line 72 shows that. The complexity for this is on average Big O of 1 and at worst case Big O of n. Because it all it has to do is go to the right point in the array and then go down the link list associated with it.

```java
public class HashMap
{
  public final static int size = 250;
  NodeBilotto item = new NodeBilotto();
  NodeBilotto[] table = new NodeBilotto[size];

  HashMap()
  {
    for (int i = 0; i < size; i++)
    {
      table[i] = null;
    }
  }//hashmap


  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.
        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) % size;  // % is the "mod"
     operator
        // TODO: Experiment with letterTotal * 2, 3, 5, 50, etc.

        return hashCode;
     }



  public void add(String key)
  {
    int hash = makeHashCode(key);
    if (table[hash] == null)
      table[hash] = new NodeBilotto(key);
    else
    {
      NodeBilotto entry = table[hash];
      while (entry.getNext() != null && entry.getData() != key)
      {
        entry = entry.getNext();

      }

      if (entry.getData() != key)
      {
        entry.setNext(new NodeBilotto(key));

      }
    }
  }//add

  public int lookup(String item)
  {
    int hash = makeHashCode(item);
    int compare = 0;
    NodeBilotto entry = table[hash];
    while (entry.getData() != item)
    {
      entry = entry.getNext();
      compare++;
    }
    return compare;
  }//lookup

}//hashmap
```

6 Node Class

This is a copy paste from the previous labs!

```java

public class NodeBilotto
{
  private String myData;
  private NodeBilotto myNext;

  public NodeBilotto ()
  {
    myData = null;
    myNext = null;
  }//NodeBilotto

  public NodeBilotto (String newData)
  {
    myData = newData;
    myNext = null;
  }//NodeBilotto

  public String getData()
  {
    return myData;
  }//getData

  public void setData (String newData)
  {
    myData = newData;
  }//setData

  public NodeBilotto getNext()
  {
    return myNext;
  }//getNext

  public void setNext(NodeBilotto newNext)
  {
    myNext = newNext;
  }//setNext


}//node
```

7 QuickSort Class

This is a copy paste from the previous labs! But the cool thing is that binary search only works on sorted data because you need to play higher or lower which can only happen if the data is sorted.

```

public class QuickSort
{
  static int compare = 0; //take out?

  public static int sort(String[] items, int begin, int end)
  {
    if (begin < end)
    {
      int partitionInd = partition(items, begin, end);

      sort(items, begin, partitionInd - 1);
      sort(items, partitionInd + 1, end);
    }

    return compare;
  }

  public static int partition(String[] array, int begin, int end)
  {
    String pivot = array[end];
    int i = (begin - 1);

    for (int j = begin; j < end; j++)
    {
      if (array[j].compareTo(pivot) < 0)
      {
        i++;
        String swap = array[i];
        array[i] = array[j];
        array[j] = swap;
      }
      compare++;
    }
```

```java

    String swapTwo = array[i + 1];
    array[i+1] = array[end];
    array[end] = swapTwo;

    return i + 1;
  }//partition
}
```

8 Queue Class

This is a copy paste from the previous labs!

```java

public class QueueBilotto {

  public NodeBilotto head;

  private NodeBilotto tail;

  public QueueBilotto()
  {
    head = null;
    tail = null;
  }//Queue

  public boolean isEmpty()
  {
    return (head == null);
  }//is empty


  public void enqueue(String item)
  {
    NodeBilotto oldTail = tail;
    tail = new NodeBilotto();
    tail.setData(item);
    tail.setNext(null);
    if (isEmpty())
    {
      head = tail;
    }//if
    else
    {
      oldTail.setNext(tail);
    }//else

  }//enqueue


  public String dequeue()
  {
    String item = null;
    if (!isEmpty())
    {
      item = head.getData();
      head = head.getNext();
    }//if

    return item;
  }//dequeue

}//queue
```

Search Name or Hashing	Complexity	number of compares
Linear	O(1) to O(n)	333.666
Binary	O(1) to O(log n)	7.5
Hashing	O(1) to O(n)	5