Assignment2

Sorin Macaluso

October 2023

1 Note

All code with comments is at the bottom in the appendix. The cout's in the original code had to be commented out or turned into a descriptive comment to explain what happened. All double quotes or single quotes where taken off and I added a comment with the data type next to it.

2 Main

The main section is where I do the read in of all the magic items form the magicitems.txt file. This is the same design as the first assignment. Then I call a Quick Sort to be done so that I can have the array sorted for when the binary search and the hash map are ran. I also create a array of 42 items, these are testing values. Then I call the first sort, Linear Sort, and make sure to print out the average of the total comparisons. Linear search will automatically print out the comparisons for the 42 different values that my sorts while be checking for. Then I do the same for Binary search, output the average in main while Binary search will output the individual amount of comparisons. Then HashTable is called, in this function the hash map will be made and will hash all 666 elements in the magicitems.txt file. Then will be searched for the 42 different chosen values.

3 Sort

```
1  int Partition(string* pivotArray, int start, int end){
2      string pivotValue = pivotArray[end];
3
4      int place = start;
5
6       for(int i  = start; i < end; i++){
7
8
9          if(pivotArray[i] < pivotValue){
10
11             swap(pivotArray[place], pivotArray[i]);
12             place++;
13         }
14     }
15
16     swap(pivotArray[place], pivotArray[end]);
17
18     return place;
19
20 }
21
22
23 void QuickSort(string* quickSortArray, int start, int end){
24
25     if(start < end){
26
27         int pivotPoint = Partition(quickSortArray, start, end);
28
```

```
29          QuickSort(quickSortArray, start, pivotPoint-1);
30          QuickSort(quickSortArray, pivotPoint+1, end);
31
32      }
33
34  }
```

The above code is a re-use of the QuickSort algorithm form Assignment1. I choose to use this sorting algorithm for the sort because I believe that it was the one that it able to do the sorting in the fastest time. While it did take more comparisons than Merge sort, I feel that QuickSort is still much faster if I where to give it a pivot value that would help to optimize the algorithm a bit more. For this I left the pivot value the same as Assignment1 to avoid any extra bugs while doing the lab.

It is the same time complexity as well, O(n $*$ log(n)), this is because as we are sorting the array we are splitting the array into log(n) parts but having to sort the array n time as each level of splitting. It does this through recursion, where we first have to find the pivot value and the pass back the function from the start to right before the pivot value and, from right after the pivot value to the end. We do not include the pivot value because everything will be sorted around the pivot making it sorted.

4 Linear Search

4.1 Linear Search Code

```
1
2       string key = null; // was just two double quotes
3       int linearSearchTotal = 0;
4
5       int comparisons = 0;
6
7       for(int i = 0; i < 42; i++){
8           key = values[i];
9           comparisons = 0;
10
11          for(int k = 0; k < size; k++){
12              if(key == magicIteams[k]){
13
14                  //cout the message number of comparisons needed
15                  //for that specific word
16
17                  linearSearchTotal = linearSearchTotal + comparisons;
18              }else{
19                  comparisons++;
20              }
21          }
22      }
23
24      return linearSearchTotal;
```

This is a implementation of a linear search. What the linear search will do is go through the whole array of n size and once the key is found in the array that is being searched then the found message will be outputted. This is represented by the for loop on line 11. If the key is found the total comparisons will be updated to add the new comparison amount to the original total to get the new total. Else if the key is not found then the comparisons amount is increased by one and then the loop goes again. The time complexity to find the key with this method is the 0(n). Since we will at worst have to check the key on every item in the searched array.

This may seem confusing since there are two for loops in the above code. But since we have a different array that hold the values of the 42 different keys, we would need another for loop to loop through the other array and give the amount of comparisons for each one of the 42 different keys. So it really will only compare the key to the searched array size or n. The average amount of searches for linear search is 360.43.

4.2 Linear Search Table

Search	Word	Comparison	Run Time
Linear	"Bag of holding type I"	41	0(n)
Linear	"Shield - Stone Warden"	512	0(n)
Linear	"Mace"	349	0(n)
Linear	"Sword of Darkness"	565	0(n)
Linear	"Statuette of Nightveil"	549	0(n)
Linear	"Portable Portal"	429	0(n)
Linear	"Deck of Cards"	173	0(n)
Linear	"Brazier of commanding fire elementals"	98	0(n)
Linear	"Manual of gainful exercise +1"	362	0(n)
Linear	"Silversheen"	524	0(n)
Linear	"Mood Ring"	385	0(n)
Linear	"Eyes of doom"	215	0(n)
Linear	"Amulet of Deception"	6	0(n)
Linear	"Elixir of love"	202	0(n)
Linear	"Wind fan"	651	0(n)
Linear	"Sable"	491	0(n)
Linear	"Kite Shield"	335	0(n)
Linear	"Wand of Yellow Stones"	637	0(n)
Linear	"Death Rod"	172	0(n)
Linear	"Censer of controlling air elementals"	116	0(n)
Linear	"Robe"	471	0(n)
Linear	"War Hammer"	639	0(n)
Linear	"Taco cat"	577	0(n)
Linear	"Crown"	153	0(n)
Linear	"Hectorius's Twin Rings"	292	0(n)
Linear	"Pearl of power, 9th-level spell"	412	0(n)
Linear	"Cloak"	131	0(n)
Linear	"Bracers of armor +7"	96	0(n)
Linear	"Potion of Infestation"	433	0(n)
Linear	"Torch Ring"	610	0(n)
Linear	"Circlet of persuasion"	127	0(n)
Linear	"Restorative ointment"	446	0(n)
Linear	"Cloak of resistance +2"	141	0(n)
Linear	"Fire Stones"	233	0(n)
Linear	"Sovereign glue"	539	0(n)
Linear	"Vambraces of Unarmed Prowess"	627	0(n)
Linear	"Nigrals Book of Amassment"	397	0(n)
Linear	"Manual of gainful exercise +3"	364	0(n)
Linear	"Pearl of power, 7th-level spell"	410	0(n)
Linear	"The Eye of Iaxolok"	584	0(n)
Linear	"Candle of truth"	108	0(n)
Linear	"Soap of Cleanliness"	536	0(n)

The table above shows that for each key value the search does not take more than n where n is 666. So 666 is the largest amount of comparisons we could possibly have. The largest amount of comparisons is 651 "Wind fan". Which since the array is sort makes sense since W is a the end of the alphabet. So the comparison amount would be closer to the size of the searched array.

5 Binary Search

5.1 Binary Search Code

```
1
2     int start = 0;
3     int end = 666;
4     int comparisons = 0;
5     string key = null; // was just two double quotes
6     int binarySearchTotal = 0;
7
8     for(int i = 0; i < 42; i++){
9         key = values[i];
10        comparisons = 0;
11        start = 0;
12        end = 666;
13
14        while (start < end){
15
16            int mid = (start + end)/2;
17
18            if (key == magicIteams[mid]){
19
20                //cout the message number of comparisons needed
21                //for that specific word
22
23                binarySearchTotal = binarySearchTotal + comparisons;
24
25                break;
26            }else if(key < magicIteams[mid]){
27                end = mid;
28                comparisons++;
29            }else{
30                start = mid + 1;
31                comparisons++;
32            }
33        }
34    }
35
36    return binarySearchTotal;
```

This is a implementation of a binary search. A binary search acts a lot like quick and merge sort from Assignemt1. This is due to the fact that binary search uses the divide part of the divide and conquer method that merge and quick sort used. The only extra thing that binary search needs to work is that the array has to be sorted in order. Binary search will split the array in half little by little until it finds the key in the array. As seen on line 16. It does this by starting at the middle of the array and then going to the left or the right of the middle depending on if the key is less than or greater than the value you are on, and then repeat. This part is handled at line 18 to line 32. First we check if the key is the match, if so output message, if not then check if the key is less than the thing at the middle, if so the end is the middle now, if not the key is bigger so the start becomes one after the middle. Goes to the middle of the new section and checks if it should go left or right.

Since Binary search is so similar to Merge and Quick sort you can expect that the time complexity of the search will also be similar. This assumption is correct! For binary search the search will only at worst O(log(n)) amount of time. Unlike Merge and Quick sort there is no n in the time complexity since we are not splitting and sorting, we are just searching by splitting in half so at most we will split log(n) times. The average amount of searches for binary search is 7.64. Which is smaller than log(n) which proves its correct.

5.2 Binary Search Table

Search	Word	Comparison	Run Time
Binary	"Bag of holding type I"	3	0(log(n))
Binary	"Shield - Stone Warden"	9	0(log(n))
Binary	"Mace"	6	0(log(n))
Binary	"Sword of Darkness"	8	0(log(n))
Binary	"Statuette of Nightveil"	9	0(log(n))
Binary	"Portable Portal"	9	0(log(n))
Binary	"Deck of Cards"	9	0(log(n))
Binary	"Brazier of commanding fire elementals"	8	0(log(n))
Binary	"Manual of gainful exercise +1"	8	0(log(n))
Binary	"Silversheen"	7	0(log(n))
Binary	"Mood Ring"	8	0(log(n))
Binary	"Eyes of doom"	9	0(log(n))
Binary	"Amulet of Deception"	9	0(log(n))
Binary	"Elixir of love"	8	0(log(n))
Binary	"Wind fan"	6	0(log(n))
Binary	"Sable"	9	0(log(n))
Binary	"Kite Shield"	8	0(log(n))
Binary	"Wand of Yellow Stones"	9	0(log(n))
Binary	"Death Rod"	6	0(log(n))
Binary	"Censer of controlling air elementals"	9	0(log(n))
Binary	"Robe"	9	0(log(n))
Binary	"War Hammer"	7	0(log(n))
Binary	"Taco cat"	8	0(log(n))
Binary	"Crown"	8	0(log(n))
Binary	"Hectorius's Twin Rings"	3	0(log(n))
Binary	"Pearl of power, 9th-level spell"	6	0(log(n))
Binary	"Cloak"	6	0(log(n))
Binary	"Bracers of armor +7"	8	0(log(n))
Binary	"Potion of Infestation"	6	0(log(n))
Binary	"Torch Ring"	6	0(log(n))
Binary	"Circlet of persuasion"	8	0(log(n))
Binary	"Restorative ointment"	8	0(log(n))
Binary	"Cloak of resistance +2"	6	0(log(n))
Binary	"Fire Stones"	9	0(log(n))
Binary	"Sovereign glue"	8	0(log(n))
Binary	"Vambraces of Unarmed Prowess"	8	0(log(n))
Binary	"Nigrals Book of Amassment"	9	0(log(n))
Binary	"Manual of gainful exercise +3"	8	0(log(n))
Binary	"Pearl of power, 7th-level spell"	7	0(log(n))
Binary	"The Eye of Iaxolok"	9	0(log(n))
Binary	"Candle of truth"	9	0(log(n))
Binary	"Soap of Cleanliness"	8	0(log(n))

The table above shows that for each key value the search does not take more than log(n) where n is 666. This equates to about 9 and the largest amount of comparisons is 9 which is shown on 14 different key values.

6 HasMap & Hashing

6.1 Hashing

```
1     const int HASH_TABLE_SIZE = 250;
2
3     for(int k = 0; k < word.length(); k++){
4         word.at(k) = toupper(word.at(k));
5     }
```

```
6
7       string HashString = word;
8       int length = word.length();
9       int letterTotal = 0;
10
11      character to the total of the whole string
12      for (int j = 0; j < length; j++) {
13          char thisLetter = HashString.at(j);
14          int thisValue = (int)thisLetter;
15          letterTotal = letterTotal + thisValue;
16      }
17
18      int hashCode = (letterTotal * 1) % HASH_TABLE_SIZE;
19
20      return hashCode;
```

This is the hashing function that was used to determine the place in the hash table that the string value would be placed. The function is passed the string value to work on. To go through the code the first line is a constant value, `HASH_TABLE_SIZE`, that is the length of the hash table. Then there is a for loop that goes through the whole string and add the ascii values for the string together. Then to make sure that the total number will always fit in the hash table size, we get the remainder of the total divided by the size, or constant value `HASH_TABLE_SIZE`. We use that remainder division value as the placement of where the string will go in the hash map.

6.2 HashMap

```
1       Node* HashMap[250] = { nullptr };
2       int hashTotal = 0;
3
4       for(int k = 0; k < 666; k++){
5           int place = HashValue(HashingArray[k]);
6
7           Node* newNode = new Node;
8           newNode->data = HashingArray[k];
9           newNode->link = nullptr;
10
11          if (HashMap[place] == nullptr) {
12
13              HashMap[place] = newNode;
14
15          } else {
16
17              newNode->link = HashMap[place];
18              HashMap[place] = newNode;
19          }
20
21      }
22
23      for(int k = 0; k < 42; k++){
24          int comparisons = 0;
25          int valuePlace = HashValue(values[k]);
26          string key = values[k];
27
28          hashmap
29          Node* temp = new Node;
30          temp->link = HashMap[valuePlace]->link;
31          temp->data = HashMap[valuePlace]->data;
32
33
34          while(temp != 0x0){
35              if(temp->data == key){
```

```
                hashTotal = hashTotal + comparisons;
                break;
            }else{
                comparisons++;
                temp =  temp->link;
            }
        }

        delete temp;

    }

    return hashTotal;
```

For the hash map itself I make a array of 250 elements and assign everything to be a null pointer at the start of the link list. Then for all 666 values in the magicitems.txt file I send the string to the HashValue function mentioned above to find what value I need to place the string in the hash map. Once I find this value it can be added in constant time, or 0(1) since the index for where the string needs to go in the array is already found. I re-use the node class from the Assignment1 to create the lists in hash map to store values if there are two values that go too the same hashing value. This action is called hashing with chaining, where I chain the like values together. The next for loop goes through the 42 items in the value array to check how many comparisons are needed in order to find the item. First I grab the hash value of the value that I am currently searching for in the array of 42. Then go to that spot in the hash map, buy asking for the HashValue again, and while the value does not equal 0 or null then go down the linked list and check to see if you found the value, if not then continue to move down the linked list count the comparisons until the key is found. This is of time complexity $0(1 + \alpha)$ where α is the average size of the linked list. This is also know as the load factor, or the size of array n / hash map size, this comes out to 2.66 or about 3 in size. Once done then return the comparison amount. We add α to 1 for search since we can get to the place in the array in constant time but have to traverse the linked list at the index about α amount. The linked list are going to be very small since the string values will be well spread apart, the average amount of comparisons is 2.29. As shown below on the table the largest comparison amount is 6, proving that the linked list are not that large. This specific linked list would be the outlier.

6.3 HashMap Table

Search	Word	Comparison	Run Time
Hash-Map	"Bag of holding type I"	4	$0(1 + \alpha)$
Hash-Map	"Shield - Stone Warden"	1	$0(1 + \alpha)$
Hash-Map	"Mace"	2	$0(1 + \alpha)$
Hash-Map	"Sword of Darkness"	1	$0(1 + \alpha)$
Hash-Map	"Statuette of Nightveil"	1	$0(1 + \alpha)$
Hash-Map	"Portable Portal"	2	$0(1 + \alpha)$
Hash-Map	"Deck of Cards"	3	$0(1 + \alpha)$
Hash-Map	"Brazier of commanding fire elementals"	4	$0(n + \alpha)$
Hash-Map	"Manual of gainful exercise +1"	2	$0(1 + \alpha)$
Hash-Map	"Silversheen"	2	$0(1 + \alpha)$
Hash-Map	"Mood Ring"	2	$0(1 + \alpha)$
Hash-Map	"Eyes of doom"	2	$0(1 + \alpha)$
Hash-Map	"Amulet of Deception"	4	$0(1 + \alpha)$
Hash-Map	"Elixir of love"	4	$0(1 + \alpha)$
Hash-Map	"Wind fan"	1	$0(1 + \alpha)$
Hash-Map	"Sable"	1	$0(1 + \alpha)$
Hash-Map	"Kite Shield"	3	$0(1 + \alpha)$
Hash-Map	"Wand of Yellow Stones"	1	$0(1 + \alpha)$
Hash-Map	"Death Rod"	2	$0(1 + \alpha)$
Hash-Map	"Censer of controlling air elementals"	7	$0(1 + \alpha)$
Hash-Map	"Robe"	1	$0(1 + \alpha)$
Hash-Map	"War Hammer"	1	$0(1 + \alpha)$
Hash-Map	"Taco cat"	1	$0(1 + \alpha)$

Search	Word	Comparison	Run Time
Hash-Map	"Crown"	4	$0(1+\alpha)$
Hash-Map	"Hectorius's Twin Rings"	1	$0(1+\alpha)$
Hash-Map	"Pearl of power, 9th-level spell"	1	$0(1+\alpha)$
Hash-Map	"Cloak"	5	$0(1+\alpha)$
Hash-Map	"Bracers of armor +7"	3	$0(1+\alpha)$
Hash-Map	"Potion of Infestation"	1	$0(1+\alpha)$
Hash-Map	"Torch Ring"	1	$0(1+\alpha)$
Hash-Map	"Circlet of persuasion"	3	$0(1+\alpha)$
Hash-Map	"Restorative ointment"	1	$0(1+\alpha)$
Hash-Map	"Cloak of resistance +2"	5	$0(1+\alpha)$
Hash-Map	"Fire Stones"	2	$0(1+\alpha)$
Hash-Map	"Sovereign glue"	3	$0(1+\alpha)$
Hash-Map	"Vambraces of Unarmed Prowess"	1	$0(1+\alpha)$
Hash-Map	"Nigrals Book of Amassment"	1	$0(1+\alpha)$
Hash-Map	"Manual of gainful exercise +3"	2	$0(1+\alpha)$
Hash-Map	"Pearl of power, 7th-level spell"	1	$0(1+\alpha)$
Hash-Map	"The Eye of Iaxolok"	1	$0(1+\alpha)$
Hash-Map	"Candle of truth"	5	$0(1+\alpha)$
Hash-Map	"Soap of Cleanliness"	3	$0(1+\alpha)$

The table above shows a new type of time complexity that we have not seen before. The average size of each of the linked list is represented as α. This is also called the load factor it is found by taking the size n and dividing it by the size of the hash map. For this assignment the load factor is 2.66 or 666/250. This is added to 1 since we get to the linked list in constant time but have to search through that linked list to find the key value. Table has run time of $0(1+\alpha)$ since the table shows the amount of comparisons for the search of one of the 42 keys.

7 References

1. Slide 10 was used to make sure that the liner search was performed correctly.
   Link: Alan Labouseur

2. Slide 28 was used to make sure that the binary search was performed correctly.
   Link: Alan Labouseur

3. Translated from Java to C++ to be used for the hash function for the hash map.
   Link: Alan Labouseur

4. This article give extra details on real world applications of a hash map, besides job interviews. It also lists and explains some of the advantages of used a hash map. This is used in line 4 of appendix section 8.8.
   Link: Geeks For Geeks

5. Just some simple syntax on how to make a array of pointer values. This is used in line.
   Link: Geeks For Geeks

8 Appendix

8.1 Sort

```
int Partition(string* pivotArray, int start, int end){
    //create the pivot value at what is at the middle of the parted array
    string pivotValue = pivotArray[end];

    //to keep track of what place you are at
    int place = start;

    //goes from the start of the sub array to the end
    for(int i  = start; i < end; i++){

```

```
11
12          //checks to see if the ith value is less than the end
13          if(pivotArray[i] < pivotValue){
14
15              //if so swap and make place move over
16              swap(pivotArray[place], pivotArray[i]);
17              place++;
18          }
19      }
20
21      //swap the place with the end to make sure all is well
22      swap(pivotArray[place], pivotArray[end]);
23
24      //return the place you are at to become the new end for the rest
25      return place;
26
27  }
28
29
30  //takes in a array uses quick sort to sort it
31  void QuickSort(string* quickSortArray, int start, int end){
32
33      //creates the condition to make it start calling
34      if(start < end){
35
36          //creates a accurate pivotPoint (middle)
37          int pivotPoint = Partition(quickSortArray, start, end);
38
39          //sets up the recursion
40          QuickSort(quickSortArray, start, pivotPoint-1);
41          QuickSort(quickSortArray, pivotPoint+1, end);
42
43      }
44
45  }
```

8.2 Insert & Binary

8.3 Insert

```
1   int linearSearch(string* magicIteams, string* values, int size){
2       //used to check
3       string key = null; // was just two double quotes
4       //use to find the total value
5       int linearSearchTotal = 0;
6
7       //counts the number of comparisons
8       int comparisons = 0;
9
10      //goes through the 42 different things in the arrya
11      for(int i = 0; i < 42; i++){
12          //Makes the key the item in the array you are looking for
13          key = values[i];
14          comparisons = 0;
15
16          //goes througt the whole array looking for the key value
17          for(int k = 0; k < size; k++){
18              if(key == magicIteams[k]){
19                  //when key is found prints out the number of comparisons
```

```
                //cout the message number of comparisons needed
                //for that specific word
                //add the value together plus the amount
                //of comaprisons to get the total
                linearSearchTotal = linearSearchTotal + comparisons;
            }else{
                comparisons++;
            }
        }
    }

    //returns the total to be used to find the average in main.cpp
    return linearSearchTotal;
}
```

8.4 Binary

```
int binarySearch(string* magicIteams, string* values, int size){
    //low and high to split
    int start = 0;
    int end = 666;
    //smae values for comparisons and the key
    int comparisons = 0;
    string key = null; // was just two double quotes
    //to find the total number of comparisons
    int binarySearchTotal = 0;

    //for loop of the 42 items in the key array
    for(int i = 0; i < 42; i++){
        key = values[i];
        comparisons = 0;
        start = 0;
        end = 666;


        //only split while the low is lesser than high so that you dont go infinite
        while (start < end){

            //finds the mid
            int mid = (start + end)/2;

            //if its found will print out the number of comparisons
            if (key == magicIteams[mid]){
                //when key is found prints out the number of comparisons
                //cout the message number of comparisons needed
                //for that specific word
                //add the value together plus the amount
                //of comaprisons to get the total
                binarySearchTotal = binarySearchTotal + comparisons;
                //once foudn breaks out the loop to not cause extra stuff to be done
                break;
            }else if(key < magicIteams[mid]){
                //if not high equal mid making the new high the mid (if less)
                end = mid;
                comparisons++;
            }else{
                //if it greater or a bigger value amkes the low the midd plus one
                start = mid + 1;
                comparisons++;
```

```
            }
        }

    }

    //returns the total to be use to find the average in main.cpp
    return binarySearchTotal;
}
```

8.5 HashMap & Hashing

8.6 Hashing

```
    int HashValue(string word){

    //const for the has table size
    const int HASH_TABLE_SIZE = 250;

    //makes everything upper case
    for(int k = 0; k < word.length(); k++){
        word.at(k) = toupper(word.at(k));
    }

    //what we are hashing
    string HashString = word;
    //the length of the word we are hashing
    int length = word.length();
    //this will become the hash
    int letterTotal = 0;

    //hash goes through till the length of the string and adds the int value (ascii) of the character to the total of the whole string
    for (int j = 0; j < length; j++) {
        char thisLetter = HashString.at(j);
        int thisValue = (int)thisLetter;
        letterTotal = letterTotal + thisValue;
    }

    //makes the value not a float in case and mod divided by the size to make sure it will be a index of the array
    int hashCode = (letterTotal * 1) % HASH_TABLE_SIZE;

    //returns the hash code
    return hashCode;

}
```

8.7 Node

```
struct Node{
    string data;
    Node* link;
};
```

8.8 HashMap

```
int HashTable(string* HashingArray, string* values){

    //creates the hash map
    Node* HashMap[250] = { nullptr };
    int hashTotal = 0;

    for(int k = 0; k < 666; k++){
        //finds the hash value and assign it to place
        int place = HashValue(HashingArray[k]);

        //makes a new node and assigns the data to the magicitems[k]
        //and the link to null
        //since it will be the first thing
        Node* newNode = new Node;
        newNode->data = HashingArray[k];
        newNode->link = nullptr;

        //gets checks if place has a nullptr
        if (HashMap[place] == nullptr) {

            //then create the newNode since this will be the first one
            HashMap[place] = newNode;

        } else {

            //if not make the link the place and the head place is the newNode
            newNode->link = HashMap[place];
            HashMap[place] = newNode;
        }

    }


    //goes through the the 42 different values and finds them
    for(int k = 0; k < 42; k++){
        //to get the num of comparisons
        int comparisons = 0;
        //what the hash function will get it too and then that will
        //equal the place it is on the array
        int valuePlace = HashValue(values[k]);
        //the key we are looking for
        string key = values[k];

        //will create a temp node so that we can travers the linked lists inside fo
    the hashmap
        Node* temp = new Node;
        temp->link = HashMap[valuePlace]->link;
        temp->data = HashMap[valuePlace]->data;


        //does this while the linked list does not equal 0x0 or the end
        while(temp != 0x0){
            //if it finds the end then prints out the message and adds
            //to the total to be used later and breaks out the loop
            if(temp->data == key){
                //when key is found prints out the number of comparisons
                //cout the message number of comparisons needed
                //for that specific word
                //add the value together plus the amount
                //of comaprisons to get the total
```

```
60                  hashTotal = hashTotal + comparisons;
61                  break;
62              }else{
63                  //if not add to comparison and then move along the linked list
64                  comparisons++;
65                  temp =  temp->link;
66              }
67          }
68
69          //delets the temp value once done
70          delete temp;
71
72      }
73
74      //return the total to be used to find the average
75      return hashTotal;
76  }
```