
Assignment Two – Searching and Hashing

Tyler DeLorey
tyler.delorey1@marist.edu

November 1st, 2024

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 2 |
| 1.1 | Changes from First Assignment | 2 |
| 1.2 | Choosing 42 Random Items | 3 |
| 1.3 | Sorting | 4 |
| 2 | Linear Search | 4 |
| 2.1 | Introduction | 4 |
| 2.2 | Implementing Linear Search | 4 |
| 2.3 | Printing Average for Linear Search | 5 |
| 2.4 | Results for Linear Search | 6 |
| 3 | Binary Search | 7 |
| 3.1 | Introduction | 7 |
| 3.2 | Implementing Binary Search | 8 |
| 3.3 | Printing Average for Binary Search | 10 |
| 3.4 | Results for Binary Search | 10 |
| 4 | Hash Table | 11 |
| 4.1 | Introduction | 11 |
| 4.2 | hashIndex | 13 |
| 4.3 | hashPut | 14 |
| 4.4 | hashGet | 15 |
| 4.5 | hashUnload | 16 |
| 4.6 | Printing Average for Hash Table Search | 17 |
| 4.7 | Results for Hash Table Search | 17 |
| 4.8 | Adjusting Size of Hash Table | 18 |
| 5 | Conclusion | 19 |

1 Introduction

1.1 Changes from First Assignment

Before I explain what I did for this assignment, I want to explain the differences I made between this assignment and the first assignment. Like before, I used C++. I used some code from the last assignment in this one, including Makefile, Node.h, some functions in Sorts.h, and some sections of code from main.cpp. All of Makefile is the same as previously, except for the compiler flags. I added a flag "-Wno-c++11-extensions" to suppress warnings dealing with C++11 extensions, as shown below in Figure 1.1.

```
4 # Compiler flags
5 CXXFLAGS = -Wall -g -Wno-c++11-extensions
```

Figure 1.1: Makefile Flag Changes

Of course, I also had to change the header files to match the files used for this assignment, as shown below in Figure 1.2. These were all the changes made in Makefile.

```
13 # Header files
14 HEADERS = Sort.h Search.h Hash.h Node.h
```

Figure 1.2: Makefile Header File Changes

Node.h remains almost the exact same as it was in the previous assignment. However, I had to change the type of data that was stored. Before, each Node had a character stored as its data because I was dealing with pushing and enqueueing each character onto a Stack and Queue respectively. Now, for hashing with chaining (which will be explained more in Section 4), I need to have the entire magic item name as the data, instead of each of its letters as its own Node.

```
13 // Holds a string as the Node's data
14 string data;

20 Node(string value)
21 {
22     this->data = value;
23     this->next = nullptr;
24 }
```

Figure 1.3: Node.h File Changes

The functions that I used in Sort.h that came from Sorts.h from Assignment 1 were shuffle, partition, and quickSort. As I will explain in the following subsections, I needed a way to both shuffle and sort the array, so I took my implementations of shuffle and quickSort and copied them over to here. Absolutely no code was changed when copying over, except I renamed the "quickSort" function to "sort" just to make it easier to read.

The code I copied from main.cpp were things like file reading and using the vector data type in order to store the magic items. Things I added to main.cpp will be addressed further through this entire document.

1.2 Choosing 42 Random Items

For my implementation of linear and binary search, I only wanted to search for just 42 items out of the 666 in the magic items array. I wanted to choose those 42 items randomly in a very straightforward way. Figure 1.4 below shows how I managed to do this.

```
17     const int ITEM_COUNT = 42;
18     string randomItems[ITEM_COUNT];

44     // Shuffles and adds first ITEM_COUNT (42) items to an array
45     // These items are random since the array was shuffled
46     shuffle(magicItems);
47     for (int i = 0; i < ITEM_COUNT; i++)
48     {
49         randomItems[i] = magicItems[i];
50     }
51
52     // Sort array
53     sort(magicItems);
```

Figure 1.4: Randomly Choosing 42 Items (main.cpp)

In order to do this, I called the shuffle function from Sort.h to shuffle the magic items array (Line 46). Then, I just took the first forty two items in the magicItems array and put them in another array called randomItems (Lines 47-50). I used a constant called ITEM_COUNT to store the number 42 (Line 17). This constant also defines the size of the randomItems array (Line 18). Changing the value of this constant changes how many values will be run through the searches. Also, it is important to notice how the magicItems array uses the vector datatype (from Assignment 1) and the randomItems uses a normal C++ array. I did this because the size of the randomItems array is known before its declaration, while the size of the magicItems vector isn't technically known beforehand because the values are being read from a file. It doesn't actually change anything technical in the code because both of them are treated as arrays.

1.3 Sorting

To perform a binary search, the entire array needs to be sorted. The sort that I use for the array is Quick Sort, as explained earlier. I chose this sort over the others because it has a time complexity of $O(n * \log(n))$, which is faster than Selection Sort and Insertion Sort, which have time complexities of $O(n^2)$. The reason why I chose Quick Sort over Merge Sort is because Quick Sort is an in-place sort, meaning the algorithm doesn't need any more than a constant extra amount of space to perform. Above in Figure 1.4 shows when I called the sort function to sort the magicItems array (Line 53).

2 Linear Search

2.1 Introduction

Linear Search is a type of search that goes through a collection of items one by one until finally it finds or it is not. In the case of this assignment, the item will always be found, but in general, if the item isn't found, the search algorithm will iterate through the entire array. Figure 2.1 below shows the linear search function being called in main.

```
55 // Perform linear search to find each random item
56 cout << "LINEAR SEARCH:" << endl;
57 for (int i = 0; i < ITEM_COUNT; i++)
58 {
59     comparisonNum = linearSearch(magicItems, randomItems[i]);
60     cout << "Comparisons to find \"" << randomItems[i] << "\":
    " << comparisonNum << endl;
61     avg += comparisonNum;
62 }
```

Figure 2.1: Iterating Array and Linear Searches (main.cpp)

The search starts out by iterating through the randomItems array (Line 57). Then, each iteration, the linear search function is called to find each random item (randomItems[i]) in the magicItems array (Line 59). Then, the number of comparisons to find the random item is printed (Line 60) and the variable to calculate the average is incremented by the number of comparisons (Line 61). The average count will be further discussed later in this section.

2.2 Implementing Linear Search

My implementation of linear search is found in the Search.h file. Figure 2.2 below shows how the linear search was implemented in my program.

```

10 // Linear search to find the 42 items in the entire array
11 // Returns comparison count
12 int linearSearch(vector<string> myItems, string target)
13 {
14     int comparisonNum = 0;
15
16     int size = myItems.size();
17
18     // Finding target in the myItems array
19     for (int i = 0; i < size; i++)
20     {
21         comparisonNum++;
22
23         if (myItems[i] == target)
24         {
25             break;
26         }
27     }
28
29     return comparisonNum;
30 }

```

Figure 2.2: Implementing Linear Search (Search.h)

The function aims to find the target string in the myItems array. The function starts off by initializing the comparison number (Line 14) and the size of the array (Line 16). The function then loops through the array one by one in order to find the target item (Line 19). It starts at the beginning of the array, at index 0, and keeps iterating throughout the array until it finds the item or until the last index and the item isn't found (but like mentioned earlier, because of how the program is set up, the item will always be found, however the program in theory handles the cases where it isn't found). If the item at that index in the array is the target item, the program will break (Lines 23-26) and then return the comparison number (Line 29).

2.3 Printing Average for Linear Search

To get accurate results from the linear search, I found the average amount of comparisons for the linear searches I performed on the 42 items. Figure 2.3 below shows how I accomplished this.

```

4 #include <iomanip>
21 float avg = 0.00f;
64 // Find average comparisons for Linear Search
65 avg = avg / ITEM_COUNT;
66 cout << endl << "Average Number of Comparisons for Linear
Search: " << fixed << setprecision(2) << avg << endl << endl;

```

Figure 2.3: Finding Average Count for Linear Search (main.cpp)

By the end of the loop shown in Figure 2.1 of Section 2.1 (Lines 57-62), the "avg" variable will be the sum of all comparisons that the linear search performed on the 42 random items, due to its incrementation every iteration on Line 61. In order to find the average from this sum, this number must be divided by the number of inputs. In this case, there were 42 inputs because the linear search was executed 42 times, once on each item. This division operation is shown on Line 65 (back in Figure 2.3). Line 66 prints out the average number of comparisons that the linear search performs.

Because I want the overall average to print up to two decimal places, I had to use the fixed manipulator and the setprecision function, which are available in the iomanip C++ library (Line 4). Used together allow the programmer to set the decimal precision used for an output. The fixed manipulator ensures that the output uses fixed-point notation, meaning the number of digits after the decimal point is fixed. This is different from floating-point notation, where the decimal point can "float" depending on the magnitude of the number. The setprecision function is set to 2 because I want the number to display up to 2 decimal places. Since it is fixed-point notation, it will always display 2 decimal places no matter the magnitude of the number.

Notice how the "avg" variable is declared and initialized as a float (Line 21). This is because the final average count will show precision up to 2 decimal places, so an integer would not be a valid data type to store this information.

2.4 Results for Linear Search

The table below shows the results of 10 runs of the program. Each result in the table is the average comparison count of 42 results, so there are technically 420 total linear searches in the entire table.

| Linear Search Results |
|-----------------------|
| 296.79 |
| 363.05 |
| 303.12 |
| 373.31 |
| 303.64 |
| 362.74 |
| 361.33 |
| 357.60 |
| 317.19 |
| 350.93 |

Table 1: Linear Search for an Array of Size 666

To actual use this data and interpret it into something that it actually meaningful, it is important to the note the time complexity of linear search. Linear

search is $O(n)$ since the worst case scenario iterates through the entire array once if the target element is at the last index, or it's not in the array at all. In this case, $n = 666$ because there are 666 items in the entire magic items array. The best case of linear search is $O(1)$ since there is a chance where the first item that is looked at is the target item (the item at index 0 is the target item). The average case of linear search is $O(n/2)$ because on average, the item will be found towards the middle of the array. However, the average case time complexity is technically also $O(n)$ because asymptotic notation ignores constants.

When looking at the table results, I will be using $O(n/2)$ to get more accurate results. In this case, as mentioned previously, $n = 666$, so the average result should be around 333. In the table of the linear search results, all of those values are very close to 333, proving that the average case for linear search is $O(n/2)$. This can be further proven by taking the average of all the averages, which is 338.97. This number is extremely close to 333. Most likely, if this process keeps continuing, and more linear searches are completed, it will get closer and closer to the number 333, or $n/2$.

3 Binary Search

3.1 Introduction

Binary Search is a type of search that divides a sorted array or collection of items into halves repeatedly until eventually the item is found, or it is not. In the case of this assignment, like with linear search, the item will always be found. Figure 3.1 below shows the binary search function being called in main.

```
68 // Perform binary search to find each random item
69 avg = 0;
70
71 cout << "BINARY SEARCH:" << endl;
72 for (int i = 0; i < ITEM_COUNT; i++)
73 {
74     comparisonNum = binarySearch(magicItems, randomItems[i]);
75     cout << "Comparisons to find \"" << randomItems[i] << "\":
76     " << comparisonNum << endl;
77     avg += comparisonNum;
78 }
```

Figure 3.1: Iterating Array and Binary Searches (main.cpp)

The code in main is very similar to the code that was used for Linear Search (view Figure 2.1 to see the comparison). The only aspect added to the Binary Search code is setting the "avg" variable back to 0 (Line 69). Because this takes place after the linear search code, the average count needs to be reset. Then, like linear search, it starts out by iterating through the randomItems array (Line

72). Each iteration, the binary search function is called to find each random item (randomItems[i]) in the magicItems array (Line 74). Then, the number of comparisons to find the random item is printed (Line 75) and the variable to calculate the average is incremented by the number of comparisons (Line 76). It is important to notice that I am using the same "random" items as I did for the linear search. The randomItems array has not been altered at all.

3.2 Implementing Binary Search

My implementation of binary search is found in the Search.h file. Figure 3.2 below shows how the binary search was implemented in my program.

```
32 // Binary search to find the 42 items in the entire array
33 // Returns comparison count
34 int binarySearch(vector<string> myItems, string target, int
    comparisonNum = 0, int begin = 0, int end = INT_MIN)
35 {
36     // If end parameter wasn't clarified, set it to the last index
    // of the array (myItems[end] returns last element)
37     // This means the function is dealing with the entire list for
    // the binary search
38     if (end == INT_MIN)
39     {
40         end = myItems.size() - 1;
41     }
42     int returnValue = 0;
43     int midPoint = int((begin + end)/2);
44
45     comparisonNum++;
46
47     // If the array has no more items to search OR it was found,
    // the function finished
48     if ((begin > end) || (myItems[midPoint] == target))
49     {
50         returnValue = comparisonNum;
51     }
52     // If target is smaller than the midpoint (in ASCII terms)
53     else if (target < myItems[midPoint])
54     {
55         returnValue = binarySearch(myItems, target, comparisonNum,
        begin, midPoint - 1);
56     }
57     // If target is larger than the midpoint (in ASCII terms)
58     else
59     {
60         returnValue = binarySearch(myItems, target, comparisonNum,
        midPoint + 1, end);
61     }
62
63     return returnValue;
64 }
```

Figure 3.2: Implementing Binary Search (Search.h)

The function aims to find the target string in the `myItems` array using recursion.

I used a very similar implementation regarding the parameters of this function like I did for the `mergeSort` and `quickSort` functions in Assignment 1. I didn't want to have any extra parameters in the original call of the function in `main.cpp` (Line 74 of Figure 3.1 shows the program calling `binarySearch(magicItems, randomItems[i])` with no extra parameters). I wanted to do this to increase readability for the main function, instead of also including the comparison count, the begin index, and the end index parameters in the function call. To do this, I had to use default parameters. A problem came with this because I couldn't set the default parameter for the variable "end" to `myItems.size() - 1` because an error would occur stating that `myItems` may not appear in that context. Like before, I think the error occurs because `myItems` wouldn't be accessible whenever the function was declared because it isn't static. Setting the "end" variable to `INT_MIN` as a default parameter seemed to do the trick again. All I had to do was check if that variable was equal to `INT_MIN`, and if it was, it would be changed to equal `myItems.size() - 1` (Lines 38-41 of Figure 3.2) because that's the last index of the whole array.

Now, moving on to the actual binary search. To ensure the function has only one exit point, I created a variable `returnValue` to hold the return value (Line 42). There is only one return to this function, located on Line 63. To divide the array into halves, I calculated the middle index for the current range of indices (`myItems[begin] ... myItems[end]`) being searched (Line 43). Each recursive call will do one comparison, so the comparison count is incremented (Line 45). Then, there is a block of if-else statements:

- The first if statement checks whether or not the calculated midpoint is the target item, meaning it has been found, or if there are no more items to search, meaning the target is not in the array (Line 48). The second case shouldn't occur in this program since I'm only searching for values known to be in the list. In either case, the function sets `returnValue` to the comparison count (Line 50) and eventually returns this value (Line 63).
- The second if statement handles the case where the target item has a smaller ASCII value than the midpoint (Line 53). Because the array is already sorted, this means there's no need to search the upper half of the array. The function recursively calls itself, narrowing the search to the lower half from the beginning index up to, but not including, the midpoint (Line 55). The returned comparison count is passed back up to ensure each call in the recursion gets the correct count (Line 63).
- The else statement runs only if the target item has a larger ASCII value than the midpoint (Line 58). Here, there's no need to search the lower half of the array. The function recursively calls itself to search the upper half, from the midpoint + 1 to the end index, again excluding the midpoint

(Line 60). As with the previous cases, the comparison count returned from this function call is passed back up on Line 63, allowing the recursion to keep track of the correct comparison count.

3.3 Printing Average for Binary Search

To get accurate results from the binary search, I found the average amount of comparisons for the binary searches I performed on the 42 items. Figure 3.3 below shows how I accomplished this.

```

79 // Find average comparisons for Binary Search
80 avg = avg / ITEM_COUNT;
81 cout << endl << "Average Number of Comparisons for Binary
    Search: " << fixed << setprecision(2) << avg << endl << endl;

```

Figure 3.3: Finding Average Count for Binary Search (main.cpp)

The functionality of this section of the program is the exact same as linear search. The "avg" variable will be the sum of all comparisons that the binary search performed on the 42 random items, due to its incrementation every iteration on Line 76 of Figure 3.1. Then, to find the average, the sum is divided by the ITEM_COUNT (Line 80). Like linear search, I had to use the fixed manipulator and the setprecision function to print the overall average up to two decimal places (Line 81).

3.4 Results for Binary Search

The table below shows the results of 10 runs of the program. Each result in the table is the average comparison count of 42 results, so there are technically 420 total binary searches in the entire table. Also, note that the binary search function will be executed more than 420 times since the function uses recursion to divide the array into halves.

| Binary Search Results |
|-----------------------|
| 8.45 |
| 8.52 |
| 8.69 |
| 8.40 |
| 8.17 |
| 8.64 |
| 8.76 |
| 8.62 |
| 8.69 |
| 8.50 |

Table 2: Binary Search for an Array of Size 666

It is important to note the time complexity of binary search. Binary search is $O(\log(n))$. This is because every iteration, the array is being divided into half. Dividing an input size of n like this results in logarithmic time, just like how Merge Sort and Quick Sort both do dividing, and a portion of their time complexity consists of $\log(n)$ (there is no conquering like there is in Merge Sort or Quick Sort, which is why binary search time complexity is just $O(\log(n))$ and not $O(n * \log(n))$ like those sorts). More specifically, the time complexity for binary search should be $O(\log_2 n)$ since it is being divided into halves (2 halves, hence the log is base 2). However, since asymptotic notation ignores constants, it is just $O(\log(n))$. In this case, $n = 666$ because there are 666 items in the entire magic items array.

The best case of binary search is $O(1)$ since there is a chance where the first item that is looked at is the target item (the item at the mid point is the target item). The average case of binary search is also approximately $O(\log(n))$ because it is very similar to the worst case scenario. The search may need to go on until there is just one element left.

As mentioned previously, $n = 666$, so the average result should be around $\log_2 666 \approx 9.38$. In the table of the binary search results, all of those values are close to 9.38, and they are all actually smaller. The reason why the calculated result isn't 100% accurate is because it is difficult to calculate the average case for binary search because the results can vary so much.

4 Hash Table

4.1 Introduction

The next type of search completed in this assignment uses a hash table. A hash table is a data structure mostly used for fast lookup times. An input is passed through a hashing function that returns a (mostly) unique index. This index is used to store the item. When searching for an item in a hash table, all that needs to be done is put that input into the hashing function, and the value will be found (or not) in close to constant time (time complexity of the hash table will be explained later).

My implementation of a hash table is shown below in Figure 4.1.

```
7 #include "./Node.h"
8
9 using namespace std;
10
11 const int HASH_TABLE_SIZE = 250;
12 Node *hashTable[HASH_TABLE_SIZE];
```

Figure 4.1: Hash Table (Hash.h)

My hash table is an array of pointers to Nodes (Line 12). The array is of size 250, as determined by the constant set on Line 11. The Node class is the exact same as the one from the previous assignment, and like I explained in Section 1.1, the only thing that is different is the data type being stored is now a string instead of a single character.

My implementation of a hash table uses hashing with chaining to handle collisions. If two separate inputs get passed into the hash function and they return the same index (which does happen), there needs to be some way to handle the collision, otherwise data will be overwritten and lost. One way to handle collision is hashing with probing. This is when the collision object is moved to the next index in the array. This can cause even more problems since the array has limited space and there could be some item that needs to go in that next index, and the problem may keep repeating itself. This is why I implemented hashing with chaining. When two or more items are assigned the same index, they're stored at a linked list at that index. More about the linked lists will be explained with the hashPut function.

Below is how I implemented the hash table functions in main.

```
83 // Loading hash table (defined in Hash.h) with magicItems array
84 hashPut(magicItems);
85
86 // Search for the 42 items using the hash table
87 avg = 0;
88 cout << "HASH TABLE SEARCH:" << endl;
89 for (int i = 0; i < ITEM_COUNT; i++)
90 {
91     comparisonNum = hashGet(randomItems[i]);
92     cout << "Comparisons to find \"" << randomItems[i] << "\" :
93     " << comparisonNum << endl;
94     avg += comparisonNum;
95 }
```

Figure 4.2: Hash Table Searches and Iterating Array (main.cpp)

I'm not going to explain this section of the program in extreme detail since most of it is extremely similar to both the linear search and the binary search. However, there are a couple differences. The most notable one is that the hash table in Hash.h needs to be loaded with the data from the magicItems array. I do this by calling the hashPut function I created (Line 84), which will be explained in detail in Section 4.3. Next, the average variable is reset (Line 87). Then with each iteration of the loop, the hash search function called hashGet is called to find each random item (randomItems[i]) in the hash table (Line 91). Then, the number of comparisons to find the random item is printed (Line 92) and the variable to calculate the average is incremented by the number of comparisons it took (Line 93). Like before, these are the same "random" items used for both previous searches.

4.2 hashIndex

The first function to explain in Hash.h is the hash index function because it's the backbone for the functionality of the hash table. This function takes an input and correlates it with some index value. Below is my implementation of the function.

```
14 // Finds index for current item
15 int hashIndex(string item)
16 {
17     // Unsigned cannot represent negative numbers, fixes overflow
18     unsigned int total = 0;
19
20     // Calculates index based on adding ASCII values of characters
21     // in the string
22     // Multiplies by 13 each time to reduce collision chance
23     for (int i = 0, wordSize = item.size(); i < wordSize; i++)
24     {
25         total = total * 13 + item[i];
26     }
27     return total % HASH_TABLE_SIZE;
28 }
```

Figure 4.3: hashIndex (Hash.h)

The function begins by declaring and initializing a variable named total (Line 18). The value of this variable may get to be very large, so I made it unsigned so it could store even more values since it doesn't need space for negative numbers. My hash function calculates the index based on adding the ASCII value of each character in the string (Lines 22-25). Something unique I added to this function is multiplying the total by 13 every iteration (Line 24). Multiplying by a prime number helps introduce the idea of the values becoming more spread out and "random". This helps reduce the chances of collision.

After the total is calculated, it has to correspond with an index in the hash table. In order to make sure the total is not larger than the amount of indices in the table (which it most definitely will be), I use the modulo operator. Doing the total % HASH_TABLE_SIZE operation (Line 27) makes sure that the value returned from the function will be from 0 - HASH_TABLE_SIZE - 1. In this case, since the hash table size is 250, the range of possible values returned would be from 0-249, which correspond with the bounds of the hash table.

Increasing the size of the hash table will definitely reduce the amount of collisions of indices in the hash table, however it would require a lot more space. There is a trade-off between space and time when it comes to algorithms, especially when talking about hash tables.

4.3 hashPut

The hash put function is the function that actually loads the hash table with the magic items. This was the function called in main (Line 84 in Figure 4.2). Below is my implementation of this function.

```
30 // Loads hash table with the magic items
31 void hashPut(vector<string> myItems)
32 {
33     int size = myItems.size();
34     int value = 0;
35     string curItem;
36
37     for (int i = 0; i < size; i++)
38     {
39         curItem = myItems[i];
40         value = hashIndex(curItem);
41
42         Node* newItem = new Node(curItem);
43         newItem->next = hashTable[value];
44         hashTable[value] = newItem;
45     }
46 }
```

Figure 4.4: hashPut (Hash.h)

The function begins by initializing a size variable to be the size of the myItems (magicItems) array (Line 33). It also declares and initializes a variable called "value" that will eventually store the index of the hash table the item will be put into (Line 34). A variable called "curItem" is also declared, which will store the name of the current item being worked with in the loop (Line 35). The loop at Line 37 loops through the entire array. Each iteration, curItem will be set to the current item (Line 39) and the item will be passed through the hashIndex function explained back in Section 4.2. The index returned from that function will be set in the "value" variable (Line 40).

Now it is time to deal with the linked list at that specific index. Each index in the hash table array has a linked list. My implementation of the linked list within the index has the new Node be inserted at the "head" of the list. First, the new Node was created with its data being the name of the current item being dealt with (Line 42). Then, the new Nodes next pointer will be set to the current Node at that correct index (43). This is essentially the former "head" of the linked list. Lastly, the new Node will be set to that index of the hash table (Line 44), making the new Node the "head" of the linked list. The reason why I decided to insert the new Node at the beginning of the linked list is because I wanted it to run in constant time, or $O(1)$. Inserting the new Node at the end of the linked list (without a pointer to the "tail") would be $O(n)$, where n represents the amount of Nodes in the linked list at that index in the hash table. Once the loop is complete, all the magic items will be in the hash table.

4.4 hashGet

The hashGet is the function that searches through the hash table for the target item. Below is my implementation of this search function.

```
48 // Find target item in the hash table
49 // Returns number of comparisons
50 int hashGet(string target)
51 {
52     int comparisonNum = 0;
53     int value = hashIndex(target);
54
55     comparisonNum++;
56
57     // Iterates through the linked list at that index
58     for (Node *ptr = hashTable[value]; ptr != NULL; ptr = ptr->next
59     )
60     {
61         comparisonNum++;
62         if (ptr->data == target)
63         {
64             break;
65         }
66     }
67     return comparisonNum;
68 }
```

Figure 4.5: hashGet (Hash.h)

The function starts off by declaring and initializing the comparison number variable (Line 52). Then, exactly like the hashPut function, a variable called "value" stores the index of the hash table the item should be in (Line 53). On this line, the target item is passed through the hashIndex function. So, technically, if the item was in the hash table, it would be at that index returned by the hashIndex function. This means there is no reason to search any other index of the hash table. This also counts as a comparison for the comparison count, so that value is incremented (Line 55).

Because now the program only looks at that calculated index in the hash table, it will only iterate through the linked list at the index (Line 58-65). Each iteration of this loop increases the comparison count (Line 60). Then, the program checks if the current Node's data is the target item (Line 61). If it is, the search is successful and then the program breaks out of the loop (Line 63) and returns the comparison number (Line 67). In the case of this program, like explained a thousand times beforehand, the target item will always be found because of how it was set up. However, in a case where it isn't found, the loop will end once the pointer becomes NULL.

4.5 hashUnload

C++ doesn't have any automatic garbage collection. This makes it very easy for memory leaks to occur. Because of this, there needs to be some sort of way that the hash table can be unloaded since new objects (Nodes) were being created that were never deleted. First of all, the unload function call is at the end of main, as shown below in Figure 4.6.

```
100 // Unload hash table
101 hashUnload();
```

Figure 4.6: hashUnload Function Call (main.cpp)

Now that the function call is established, it is time to dive into the specifics of how this function works. The implementations of the hashUnload function and its helper function (hashUnloadIndex) are shown below in Figure 4.7

```
70 // Unloads each index of hash table (including all linked Nodes at
    that index)
71 void hashUnloadIndex(Node *hash)
72 {
73     if (hash == NULL)
74     {
75         return;
76     }
77     hashUnloadIndex(hash->next);
78     delete(hash);
79 }
80
81 // Unloads hash table to avoid memory leaks
82 void hashUnload()
83 {
84     for (int i = 0; i < HASH_TABLE_SIZE; i++)
85     {
86         if (hashTable[i] != NULL)
87         {
88             hashUnloadIndex(hashTable[i]);
89         }
90     }
91 }
```

Figure 4.7: hashUnload and hashUnloadIndex (Hash.h)

The hashUnload function iterates through the hash table, one index at a time (Lines 84-90). If there is at least one Node at the current index (Line 86), then that Node, and all other Nodes that may be in the linked list at the index, need to be deleted. This is where the hashUnloadIndex function is called (Line 88).

The hashUnloadIndex actually does the deleting from memory. This is also a recursive function. If the given parameter Node is NULL, then it returns back to

the hashUnload function to continue its iterations (Lines 73-76). If it's not, the function will be called again and again until it reaches the "tail" of the linked list (Line 77). Once it reaches the tail, it will delete the Node and then the Node before it, and the one before that, etc (Line 78). It starts by deleting the Node at the end of the linked list since the Node's next pointers would be pointing at garbage values if we delete the first Node in the linked list. Therefore, the other Nodes would be lost in memory.

The time complexity of the hash table search will all be explained in Section 4.7, but since the unload function is complete separate, I would like to explain that here. The time complexity of the unload function is $O(n)$, where n represents the amount of Nodes in the hash table. In this case, $n = 666$. This is because the program has to visit every Node in order to delete it, which makes the time complexity of the unload $O(n)$.

4.6 Printing Average for Hash Table Search

To get accurate results from the hash table searches, I found the average amount of comparisons for the table searches I performed on the 42 items. Figure 4.8 below shows how I accomplished this.

```
96 // Find average comparisons for Hash table Search
97 avg = avg / ITEM_COUNT;
98 cout << endl << "Average Number of Comparisons for Hash Table
    Search: " << fixed << setprecision(2) << avg << endl << endl;
```

Figure 4.8: Finding Average Count for Hash Table Search (main.cpp)

The functionality of this section of the program is the exact same as linear search and binary search. The "avg" variable will be the sum of all comparisons that the search performed on the 42 random items, due to its incrementation every iteration on Line 93 of Figure 4.2. Then, to find the average, the sum is divided by the ITEM_COUNT (Line 97). Again, I had to use the fixed manipulator and the setprecision function to print the overall average up to two decimal places (Line 98).

4.7 Results for Hash Table Search

The table below shows the results of 10 runs of the program. Each result in the table is the average comparison count of 42 results, so there are technically 420 total hash "gets" in the entire table.

| Hash Table Search Results |
|---------------------------|
| 3.38 |
| 3.05 |
| 3.02 |
| 3.10 |
| 3.57 |
| 3.17 |
| 3.24 |
| 3.57 |
| 3.29 |
| 3.29 |

Table 3: Hash Table Search for an Array of Size 666 and Hash Table Size of 250

It is important to note the time complexity of hash table lookup. As eluded to earlier, hash table searching is extremely close to constant time, which is $O(1)$. However, the time complexity of hash table lookup is noted at $O(1 + \alpha)$. The 1 represents the "get" comparison, meaning when the hashIndex function is executed to find the correct index to search through. The α , or alpha, represents the average chain length, also referred to as the load factor. The average chain length can be calculated by dividing the number of items, or Nodes, in the hash table by the size of the hash table.

For the case of this assignment, the hash table is of size 250, and there are 666 Nodes in the hash table. To find the average chain length, the equation is $666/250 = 2.664$. Now obviously this is just an estimate since it is the AVERAGE chain length. All that needs to be done now is to plug in that number for α in the time complexity, so the time complexity of this specific hash table search is $O(1 + 2.664) = O(3.664)$. All of the values in the table above are very close to 3.664, thus proving that this is the average case.

4.8 Adjusting Size of Hash Table

What if the size of the hash table was changed? How would this affect the time complexity of the algorithm? The solution to solve these questions is pretty easy to implement, considering I had the HASH_TABLE_SIZE constant, which I could change on each run of the program. Here are some comparison number results I got when changing the hash table size.

| Table size = 25 | Table size = 250 | Table size = 2500 |
|-----------------|------------------|-------------------|
| 15.90 | 3.55 | 2.05 |
| 14.07 | 3.57 | 2.17 |
| 14.45 | 3.17 | 2.07 |
| 14.12 | 3.14 | 2.24 |
| 15.83 | 3.76 | 2.10 |

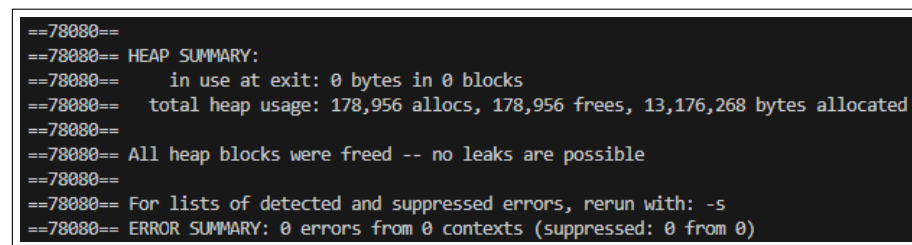
Table 4: Hash Table Comparison Results when Changing Table Size

It is obvious that there is a relationship between the table size of the number of comparisons to do the hash table lookup. To further prove this, I can use the equal mentioned in the last subsection, $O(1 + \alpha)$.

- For when the table size is 25, α would be equal to $666/25 = 26.64$, and $26.64 + 1 = 27.64$, meaning the average time complexity is 27.64. The reason why I think this average value is slightly off from the values in the table is because of the hashing function I used to find the correct index. Either way, having less buckets in the hash table means there will be more comparisons on average.
- For when the table is 250, as proven in the last subsection, the average time complexity would be 3.664. This is obviously a better result than when the hash table is smaller, but it seems to be worse than the results when the hash table is larger.
- For when the table size is 2500, α would be equal to $666/2500 = 0.2664$, and $0.2664 + 1 = 1.2664$, meaning the average time complexity is 1.2664. However, because of my implementation of the Hash.h functions, the minimum amount of comparisons to find an item will always be 2. There must be 1 to find the correct index (the get), and another to see if the head of the linked list at the calculated index is the target item.

Like hinted at earlier, though, this is a case of prioritizing time or space. Allocating more space to store a larger hash table means that the lookup times will be faster. However, if the programmer doesn't want to use that much space, they don't have to.

5 Conclusion



```
==78080==
==78080== HEAP SUMMARY:
==78080==    in use at exit: 0 bytes in 0 blocks
==78080==   total heap usage: 178,956 allocs, 178,956 frees, 13,176,268 bytes allocated
==78080==
==78080== All heap blocks were freed -- no leaks are possible
==78080==
==78080== For lists of detected and suppressed errors, rerun with: -s
==78080== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 5.1: Valgrind Output (running command "valgrind ./main")

In conclusion, this assignment was another great learning experience for me. Implementing linear and binary search was a fun and relaxing experience. Implementing the hash table, especially with unloading it, took me a bit longer to do, but it was still fun. Speaking of unloading the hash table, I used Valgrind a lot to see if the unloading functions were working properly. As seen in Figure 5.1, there were no memory leaks at all, meaning the Nodes were successfully deleted. Valgrind was, again, very useful for me to use. I'm having a great time working in C++.