CMPT 435 - Fall 2020 - Dr. Labouseur

Assignment Three – Searching and Hashing
Linear And Binary Search and an implementation of a Hash Table

Sam Alcosser
Samuel.Alcosser1@Marist.edu

October 29, 2020

Contents

1 Introduction

This document explains the implementations of linear search, binary search, and hash tables using C++. The organization of each section will generally follow the format of a brief explanation of the function, the code itself, and an explanation of the code with references to line numbers.

2 the `randPick()` Function

Most of the algorithms used in this program are reliant on the `randPick()` function, the routine used to generate the list of random elements.

Code Analysis

```cpp
1  oid Search::randPick(std::string mainArr[], std::string chosen[], int len, int lenOfChosen)
2  {
3  
4          int countTries = 0;
5  
6          for (int i = 0; i < lenOfChosen; i++)
7          {
8  
9                  bool same = true;
10                 while (same)
11                 {
12                         std::random_device dev;
13                         std::mt19937 rng(dev());
14                         std::uniform_int_distribution<std::mt19937::result_type> dist6(0, len);
15 
16                         std::string attempt = mainArr[dist6(rng)];
17 
18                         bool foundSame = false;
19                         for (int j = 0; j < lenOfChosen; j++)
20                         {
21 
22                                 if (chosen[j] == attempt)
23 
24                                 {
25 
26                                         countTries++;
27                                         foundSame = true;
28                                         break;
29                                 }
30                         }
31 
32                         if (foundSame)
33                         {
34 
35                                  countTries++;
36                                  continue;
37                          }
38
39                          else
40                          {
41
42                                  countTries++;
43                                  chosen[i] = attempt;
44                                  same = false;
45                          }
46                  }
47          }
48
49          return;
50  }
```

1. Begin a loop that will run an amount of times defined by the `lenOfChosen` variable, as was passed to the function.(line 6)
2. After initializing a boolean variable called `same` to true, start a while loop that will complete once the variable is set to false. This will allow us to continue trying to generate random elements until an original element is found.(lines 9,10)
3. Generate a random number between 0 and the length of the main array as defined by the `len` variable that was passed into the function. Retrieve the element at that index from array `mainArr[]`(lines 12-14).
4. Set a flag variable called `foundSame` to false.(line 18)
5. Check if the element selected exists in the list of already chosen elements by looping through and checking each element with a loop.(lines 19-22)
6. If the current element matches a previous one, set the `foundSame` flag, and break out of the for loop. With this the algorithm will stop the search.(lines 26-28)
7. If the `foundSame` flag was set, reset the while loop and try generating a new number all over again.(lines 32-37)
8. If there was not a same element, add that new item to the list of items and set the `same` flag to false. Now, the function will break out of the outer while loop.(lines 39-45)

3 Searching

3.1 Linear Search

The linear search algorithm is a brute force algorithm used to retrieve data from a set of data. The algorithm simply starts at the beginning of the set, and tries each item until it finds what it is looking for.

Code Analysis

```
 1 int Search::linSearch(std::string mainArr[], int len, std::string chosen[], int lenOfChosen)
 2 {
 3         std::cout << "---Linear Search---" << std::endl;
 4         int tCount = 0;
 5 
 6         for (int i = 0; i < lenOfChosen; i++)
 7         {
 8                 int count = 0;
 9                 for (int j = 0; j < len; j++)
10                 {
11                         if (mainArr[j] != chosen[i])
12                         {
13                                 count++;
14                         }
15                         else
16                         {
17                                 tCount += ++count;
18                                 std::cout << "Found it at index " << j << " after " << count << " checks." << st
19 
20                                 break;
21                         }
22                 }
23         }
24 
25         int avg = (int)(tCount / lenOfChosen);
26         return avg;
27 }
```

As seen above, this algorithm really is quite simple.

1. Begin a loop from zero to the length, `lenOfChosen`, of the list of items `chosen[]`,we would like to retrieve. (line 6)
2. After initializing our temporary counter variable `count`, begin another loop from zero to the length, `len`, of the array `mainArr[]` which we are searching in. (line 9)
3. If the item at the index defined by the inner loop does not match the item at the index defined by the outer loop (if the current item in the main array

does not match the item we are searching for) increment our temporary count variable by one. (lines 11-14).

4. Otherwise, meaning the items do match, increment the counter once more. Even though the item was found, this still counts as a comparison. Also, add the temporary counter `count` to the sum of `tCount`, which holds the total count of comparisons.(line 17)

5. Since the item was found, end the inner for loop, and move onto the next item. (line 20)

6. Once all items have been found, compute the average by dividing `tCount`, the total comparisons by `lenOfChosen`, the number of items searched for and return it from the function. (lines 25,26)

3.2 Binary Search

Binary search is more complicated than linear search, but it is also much more efficient. To begin, binary search needs a sorted array. Without the array being sorted, the algorithm cannot function correctly. It is not shown within this document in full, as the purpose of this document is to explain searching and hashing, but the Merge Sort algorithm is used to sort the items prior to searching.

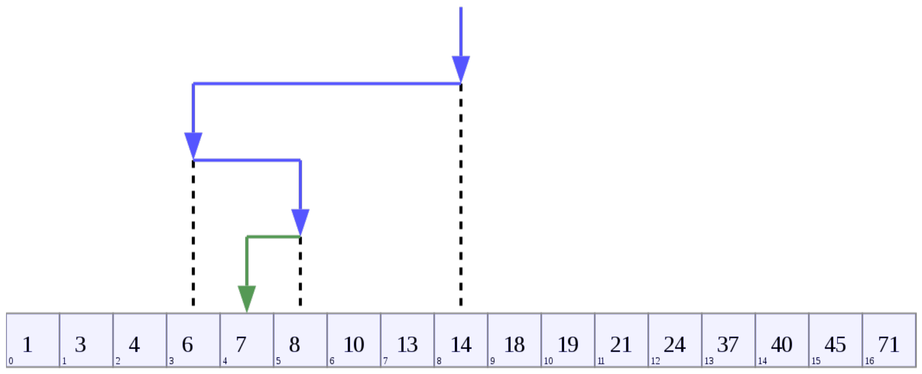

Figure 1

Once the array is sorted, binary search starts from the midpoint of the array, and depending on whether or not that midpoint value is greater than or less than the item being searched for, it will then switch to checking the lower half of the array or the higher half of the array. Once again, the same process will happen at that half of the array, and will continue to split the array and search until the item is found.

Code Analysis

```
 1  int Search::binSearch(std::string mainArr[], int len, std::string chosen[], int lenOfChosen)
 2  {
 3          std::cout << "---Binary Search---" << std::endl;
 4          int tCount = 0;
 5
 6          for (int x = 0; x < lenOfChosen; x++)
 7          {
 8
 9                  int count = 0;
10                  int low = 0;
11                  int high = len - 1;
12                  bool found = false;
13                  while (!found)
14                  {
15
16                          int mid = (int)floor((low + high) / 2);
17                          if (chosen[x] < mainArr[mid])
18                          {
19                                  count++;
20                                  high = mid;
21                          }
22                          else if (chosen[x] > mainArr[mid])
23                          {
24                                  count++;
25                                  low = mid++;
26                          }
27                          else
28                          {
29
30                                  tCount += ++count;
31                                  std::cout << "found it at index " << high << " after " << count << " tries." << 
32                                  found = true;
33                          }
34                  }
35          }
36          int avg = (int)(tCount / lenOfChosen);
37          return avg;
38  }
```

1. First, start a loop that will iterate through to the end of the length of the list of items to be retrieved(`lenOfChosen`).(line 16)

2. Next, set up the variables to be used in the algorithm such as `count`, `low`, `high`, and `found`. `count` and `low` are initialized to zero, `high` is initialized to `len-1`, so that it can reach the top, and `found` is a flag variable that will be used to determine whether or not the item has been found. For this reason, it is initialized as `false`.(lines 9-12)

3. Start a while loop which will continue until the `found` variable is set to `true`. Keep in mind, this is operating inside of the outer for loop which iterates for each item in the list of things to find.(line 13)
4. Get the midpoint of the array by simply taking the floor of `(low + high) / 2)` (line 16).
5. Depending on whether or not the current item we are searching for, `chosen[x]`, is less than or greater than the item at the midpoint `mainArr[mid]`, assign `high` to the midpoint `mid` or assign `low` to the index right after `mid`. This way, depending on whether or not we are too high or too low, we adjust the bounds of our search accordingly (lines 17-26).
6. On the off chance that our `chosen[x]` is neither greater or less than our midpoint item `mainArr[mid]` we can assume that the two are equal, and therefore we have found the item. Just as we have with the previous two cases, increment our `count` variable to show that we made a comparison. Ensure that the now total amound of counts for this round are added to the sum variable `tCount`. Set the `found` flag to `true` so it breaks from the while loop.(lines 27-33)
7. Once all items have been found, compute the average number of comparisons to reach the item by dividing the total comparisons `tCound` by the number of items we searched for `lenOfChosen`. Finally, return this average out of the function(lines 36,37).

3.3 Asymptotic Analysis and Comparison of Linear and Binary Search

Search Algorithm	Averagage Comparisons
Linear Search	332
Binary Search	8

Figure 2

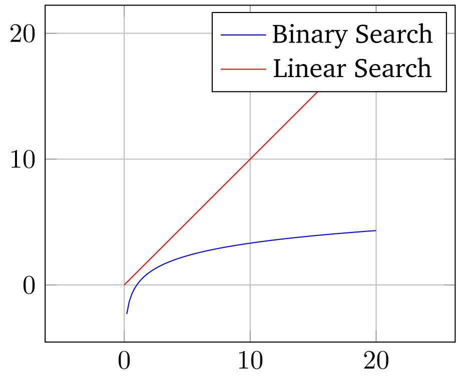

Figure 3

It is clear that binary search is a more efficient algorithm than linear search. The key reason being that linear search is a brute force algorithm with complexity of $O(n)$, while binary search is a more complicated yet more efficient algorithm with complexity of $O(log_2(n))$. Linear search simply is checking every single item until it finds a match. The time it takes to find that item is somewhat random. For this reason, with a set size of 666 items, we can expect that on average it will take 333 tries to get the right item, which is incredibly close to the actual result.

Binary search, however, is very stable. The reason being that, the only processing being done, is splitting the set into smaller and smaller pieces until it is small enough to know exactly where the item is. This splitting process is what gives Binary Search it's $O(log_2(n))$ complexity (see figure 3).

4 Hashing and Hash Tables

A Brief Explanation

There are many complex ways to implement hashing using things such as one way hashes, nonces, public and private keys, and other things. Although, hashing can also be implemented quite simply. Hashing is the process of taking data, processing it in a certain way that returns a value that is not the same as the original data, and placing the data in a table in accordance to where it the value belongs. This certain way of processing is called a hash function.

As long as this hash function is known, and constant, the data can be retrieved from the table fairly quickly. Say for example the algorithm is simply the length of the string multiplied by six, we can just apply the same algorithm to the data we want to find and we will know where to look. Because of the likely chance that multiple data points will return the same hash value, we link together all of the like data points in lists called chains. Then, all the function would have to do is iterate over that list until it finds the correct data point.

4.1 Overview of Functionality

Being that a hash table is a conceptual thing, as opposed to the search functions which are static members of a class, it is useful to review the header file for the `HashTable` class.

```
#pragma once
#include "Node.h"
#include "Queue.h"
class HashTable {
public:
        static const int HASH_TABLE_SIZE = 250;
        Queue table[HASH_TABLE_SIZE];
        static int getHash(std::string item);
        void placeHashed(std::string item);
        void visualize();
        int getHashed(std::string items[], int chosenLen);
};
```

1. `HASH_TABLE_SIZE` stores the defined length for the hash table.
2. The line `Queue table[HASH_TABLE_SIZE];` shows that the table itself is made up of `queues`, a concept previously explored in a prior set of documentation.
3. the `getHash()` function was made static so that a developer could use the hashing function without needing to create a hash table.
4. `placeHashed()` simply takes in an item and places it in an array. This function does not take an array, as other than in the case of this demonstration, I did not believe that the table would be loaded all at once.
5. `visualize()` simply allows the developer to see the data in the hash table in a simplified histogram view.
6. `getHashed()` is used to retrieve items from the hash table.

4.2 Hash Table Functions

4.2.1 `getHash()`: Getting the Hash Value

As explained above, this function is used to apply the hashing algorithm to a piece of data, in this case a string, and return a hash value to be used as the index in the hash table.

Code Analysis

```
1  int HashTable::getHash(std::string item)
2  {
3          std::transform(item.begin(), item.end(), item.begin(), ::toupper);
4          int sLen = item.length();
5          int letterTotal = 0;
6          for (int i = 0; i < sLen; i++)
7          {
8                  char thisLetter = item[i];
9                  int thisValue = (int)thisLetter;
10                 letterTotal += thisValue;
11         }
12         int hashCode = (letterTotal * 1) % HASH_TABLE_SIZE;
13
14         return hashCode;
15 }
```

1. Normalize the string by setting the whole string to upper case.(line 3)
2. For each character in the string, (line 6) take the integer value of that character (line 9) and add it to the `letterTotal` variable created previously(line 10).

3. Take the modulus of the `letterTotal` mod `HASH_TABLE_SIZE`, the size of the table. Return this value. (lines 12,14)

4.2.2 `placeHashed()`: Putting in the Data

Since the process of finding the hash value of an item is already defined, the `placeHashed()` function simply calls that function, and places it in the correct chain, or queue accordingly.

Code Analysis

```
1 void HashTable::placeHashed(std::string item)
2 {
3         int hash = getHash(item);
4         table[hash].enQueue(item);
5 }
```

1. Get the hash value of the item by calling `getHash()` on the item.(line 3)

2. Place the item in the `queue` defined by its hash value.(line 4)

4.2.3 `getHashed()`: Retrieving the data

In order to retrieve the data, as covered previously, there is a very simple process. All the algorithm must do is apply the same hashing algorithm to the data that is desired to be retrieved, and search for it at the queue at the index defined by that hash value.

```
int HashTable::getHashed(std::string items[], int chosenLen)
{
        std::cout << "--Retrieving hashed Data--" << std::endl;
        int tCount = 0;

        for (int i = 0; i < chosenLen; i++)
        {
                int count = 0;

                int hashLoc = getHash(items[i]);

                Queue chain = table[hashLoc];

                Node *n = chain.head;

                if (n->data == "")
                {
                        std::cout << "If you're seeing this, the list is empty." << std::endl;
                }
                else
                {
                        while (n->data != items[i])
                        {
```

```
24                  count++;
25                  n = n->next;
26              }
27
28              std::cout << "Found it at place " << ++count << " in the chain." << std::endl;
29              tCount += count; //take off the pre increment if uncommenting above line
30          }
31      }
32      int avg = tCount / chosenLen;
33      return avg;
34  }
```

Code Analysis

1. For each item in the chosen items list (line 6) start by initializing the temporary `count` variable, and getting the hash value of the current items being looked for.
2. Ensure that the queue is not empty before proceeding. (line 18)
3. Iterate over the queue until the desired item is found, adding one to the `count` variable each time. (lines 22-26)
4. Announce the place in the chain, and add on the temporary `count` to the `tCount`, which holds the total count.
5. Compute and return the average number of comparisons to get to the desired elements, including the correct one, by dividing the `tCount` by the amount of items searched for.(line 32)

4.2.4 `visualize()`: Seeing What Is In The Table

This function iterates through each `queue` in the table, adding an asterisk for every element it sees. With the careful placement of some `std::cout << std::endls`, it creates a histogram of the data.

Code Analysis

```
1   void HashTable::visualize()
2   {
3       std::cout << "---visualizing your data---" << std::endl;
4
5       for (int i = 0; i < HASH_TABLE_SIZE; i++)
6       { //getting the data
7
8           if (table[i].head == nullptr)
9           {
10              std::cout << i << "|" << std::endl;
11
12
13                  }
14                  else
15                  {
16                          std::cout << i << "|";
17                          Node *start = table[i].head;
18                          while (start != nullptr)
19                          {
20                                  std::cout << "*";
21                                  start = start->next;
22                          }
23                          std::cout << std::endl;
24                  }
25          }
26  }
```

1. For each `queue` in the table (line 5), iterate over the `queue`, or chain.
2. Check that the chain is populated, and if it is not, print a blank ”bar”.(lines 8-13)
3. If the chain is not empty, begin by placing the ”x axis” line. Then, grab the head pointer of the chain (lines 16,17)
4. Iterate over the chain, adding an asterisk for every `node` passed, and continuing until the last element is found (lines 18-22)
5. End the current ”bar” and continue to the next chain.

4.3 Asymptotic Running Time Analysis Of Hash Table Functions

Generally speaking, searching in a hash table has complexity of $O(1)$. The reason being that it is very unlikely that two items will have the same hash value. Also, there is the simple fact that by knowing the hash value, you have a very good idea of where the item is, it’s just a matter of getting to your element in the array. Here, the running time turns into a linear search, which is $O(n)$. To find the complexity we start by computing our load factor, and adding in the 1 for the constant time to hash and move to the right place. With this, we see that our complexity is $O(1 + 1.67)$, or $O(2.67)$.

As a side note, getting the hash value and placing the hash value are also constant time operations. Getting the hash value will always take the same amount of time, and all `placeHashed()` is doing is `enQueue()`ing that element, which is also constant time.

5 `Main.cpp`: Putting it All Together

Lastly, There is the implementation of all of these algorithms. Due to the fact that it has been covered multiple times in previous documents, the process of setting up the array from the text file will not be covered here. The only modification is that once the array is loaded, `mergeSort()` is called on the array in order to sort it.

5.1 `hashTheTable()`: Loading The Table In One Batch

This function simply iterates over the main array of elements, applys the hashing algorithm, and places it accordingly using the `placeHashed()` function all in one step. Then once all is completed, `visualize()` the data.

```cpp
void hashTheTable(std::string arr[], int len, HashTable * tab) {
        //HashTable* tab = new HashTable();
        for (int i = 0; i < len; i++) {
                tab->placeHashed(arr[i]);
        }
        tab->visualize();

}
```

5.2 `main()`: Running the Code

Here is where all of the pieces come together and are ran by the program.

Code Analysis

```cpp
1  int main()
2  {
3          setup();
4
5
6          std::string items[CHOSEN_LEN];
7          Search::randPick(arr, items,LEN,  42);
8          int linAvg = Search::linSearch(arr, LEN, items, CHOSEN_LEN);
9          int binAvg = Search::binSearch(arr, LEN, items, 42);
10         std::cout << "Linear Search:||: Average comparisons to get a hit: " << linAvg << std::endl;
11         std::cout << "Binary Search:||: Average comparisons to get a hit: " << binAvg  << std::endl;
12         HashTable * hTable = new HashTable();
13         hashTheTable(arr, LEN, hTable);
14         int tCount = hTable->getHashed(items, CHOSEN_LEN);
15         std::cout << "Average " << tCount << " comparisons to retrieve each item." << std::endl;
16 }
```

1. set up the table.(line 3)
2. Initiate the items array, and populate it with random items using the `randPick` function.(lines 9, 10)
3. Run linear and binary search, and print out the average comparisons for each. (lines 11-14)
4. Create the hash table. (line 15)

5. run the `hashTheTable()` function on the hash table to load it with data. (line 16)
6. Retrieve the list of random items from the main list, and print the average comparisons to find each one (lines 17-18)

6 Results and Conclusion

The results of these functions have been listed throughout this document, although here is a consise summation of the data collected along with a graph of all of the complexities.

Search Algorithm	Aveargage Comparisons	Complexity/ $O()$
Linear Search	332	$O(n)$
Binary Search	8	$O(log_2(n))$
getHashed()	2	$O(2.67)$
placeHashed() & getHash()	-	$O(1)$

Figure 4

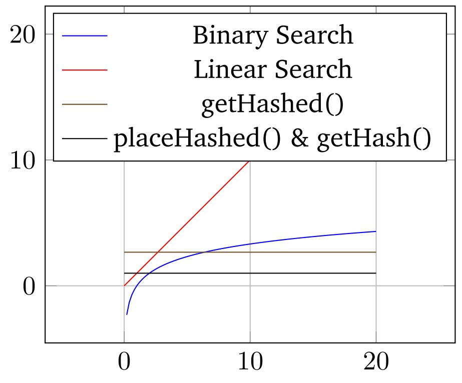

Figure 5

Closing Thoughts

At the start, I was sure that this assignment would be no trouble at all. Simple arrays, simple enough functions, what could go wrong? But yet again, the concept of pointers and references, and view access rights caused me to completely give up on my efforts. Thankfully, I remembered that at the point of my previous commit, I was not having this error, so I hedged my bets, downloaded the previous commits version of the repository, and started over. With this, along with using g++ to debug, the problem was solved in no time.

as a wise person once said, " give a person a program and they'll be frustrated for a day, teach a person to program and they'll be frustrated for the rest of their life.". {S.A.}

References

Williams Jr., L. F. (2020, October 20). Binary search algorithm. Retrieved October 29, 2020, from https://en.wikipedia.org/wiki/Binary_search_algorithm