Assignment Two: Searching and Hashing

Aidan Carr

Aidan.Carr1@Marist.edu

October 27, 2023

1 Introduction

1.1 The Basics of Searching

Uh oh! You lost your wallet! It's not going to find itself, or reappear in your hands, you have to look for it. Or, search for it.

When searching, there are a few main components: the group of things, what you are looking for, and the location for that thing. For example, when looking for a lost wallet, you search your house. You check the kitchen cabinets, in a bedroom dresser, in the bathroom medicine cabinet, and eventually you should find it, maybe under the couch.

In this assignment, we will be looking for magic items using computer science. Each magic item is stored in a unique, numbered location called an index. We will have a collection, or an array, that stores all 666 of these magic items. Like our locations around the house, each one has the possibility to store the particular magic item (or wallet) that we are looking for.

Using different searching methods, we can see how to store and find our target in a faster, more efficient manner.

1.2 Assignment Framework

Starting off in the main function of our code in Figure 1.1, we'll set an array called `magicItems[]` to take all the lines from magicitems.txt on line 3. We will then shuffle the array and pick the first `_SUB_ITEMS` items. In this Assignment the constant `_SUB_ITEMS` will be 42. We will use this subarray of items to test or different search methods.

To see how `setMagicItemsArray()`, `shuffle()`, and `mergeSort()` work, check out Assignment 1

```
    //put 666 magic items into magicItems array
    string magicItems[_NUM_OF_ITEMS];
    setMagicItemsArray(magicItems);

    //Shuffle the magic items, store the names the first 42
    string randomItems[_SUB_ITEMS];
    shuffle(magicItems);
    for (int i = 0; i < _SUB_ITEMS; i++){
        randomItems[i] = magicItems[i];
    }

    //Sort the magic items
    mergeSort(magicItems, _NUM_OF_ITEMS);
```

Figure 1.1: Setting up our arrays in C++

1.3 Comparisons functions

In searching, we are constantly comparing values to see if we have reached our target or if we are heading in the right direction. When comparing numbers, we use operators like $=$, $<$, and $>$. When comparing strings, however, we will have to use custom functions.

Figure 1.2 shows an `isEqual()` function. Given two strings, we check to see if the lengths are equal. If not, the string can't be equal, so we return false on line 8. Next, on line 12, we check every letter index and compare each character to each other. Once one letter is off, we return false. If the function successfully makes it through each character on both strings, it returns true.

```
bool isEqual(string first, string second){

    //find correct length to avoid out of bound error when comparing
    //different lengths means definetly not equal
    int length1 = first.length();
    int length2 = second.length();
    if (length1 != length2){
        return false;
    }

    //compare letter by letter until an unequal character string is found
    for (int i = 0; i < length1; i++){
        if (first.at(i) != second.at(i)){
            return false;
        }
    }

    //if the two words have passed its equal
    return true;
}
```

Figure 1.2: Comparing string for equality in C++

To see an in-depth look at the `isLessThan()` function, check out Assignment 1.

2 Linear Search

2.1 Understanding Linear Search

Linear search is the most basic version of searching. Given an array of magic items, we will go one by one down the line of items and check if the item in that location is equal to our target item. If not, move on to the next item. If we reach the end of the array and the item has not been found, the location is -1.

In Figure 2.1, when searching for 39, we check element number 0. 39 does not equal 13 so we move on to element 1. 39 does not equal 9, and we move on. Eventually, at element 4, we find that 39 does equal 39.

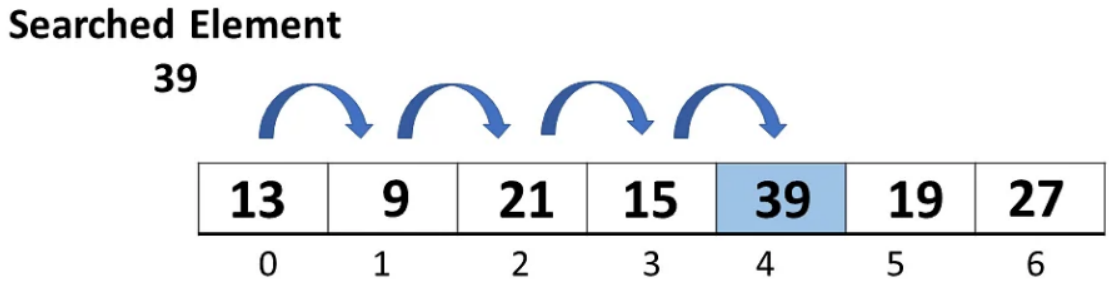

Figure 2.1: Example of a Linear Search of numbers[1]

2.2 Coding Linear Search

Let's look at the code for linearly searching an array of items in Figure 2.2.

```
1 int linearSearch(string items[], string target){
2     for (int i = 0; i < _NUM_OF_ITEMS; i++){
3         _comparisons ++;
4         if (isEqual(items[i], target)){
5             return i;
6         }
7     }
8     return -1;
9 }
```

Figure 2.2: Linear Search in C++

On line 2 we are looping through every element of the given array. And then, on line 4 and 5, we return the index if we have found an exact string match. If we loop through the entire array without any luck we return -1 on line 8, signifying there was no index associated with our target. Easy!

[1]Image: https://staragile.com/blog/linear-search.

2.3 Testing Linear Search

Going back to the main function, we'll use our subset of magic items and search for each one in the array of 666 items.

```
1     std::cout << "\nLINEAR SEARCH" << "\n" << std::endl;
2     int linearComparisons = 0;
3     string item;
4     int index;
5     for (int i = 0; i < _SUB_ITEMS; i++){
6
7         _comparisons = 0;
8         item = randomItems[i];
9         index = linearSearch(magicItems, item);
10        linearComparisons += _comparisons;
11
12        //print num of comparisons for the item
13        std::cout << item << "\n\tComparisons: " << _comparisons << std::endl;
14    }
15
16    //calculate avg comparisons, round 2 decimal place, print
17    float avgLinearComparisons = (float) linearComparisons / _SUB_ITEMS;
18    avgLinearComparisons = (int) ((avgLinearComparisons + 0.005) * 100) / 100.0;
19    std::cout << "\nAverage Linear Search Comparisons: "<< avgLinearComparisons << "\n" <<
      std::endl;
```

Figure 2.3: Linear Search Test in C++

In Figure 2.3, we first loop through every item in the sub array of 42 items on line 5. We reset the comparisons to 0, and using linear search, we find the index of our first sub item `randomItems[1]` on line 9. We'll add it to the total number of comparisons for linear search, then repeat for the other 41 items. After all is complete, the code will print the average number of linear search comparisons for 42 items on lines 17 through 19.

Knowing this search in linear search, We should expect the function to have a complexity of:

$$O(n)$$

To explain why, look at line 7 of Figure 2.2. Here, we counting a comparison for every loop of size n on line 5. We only loop through once so we get O(n). Let's see what an example output would look like in Figure 2.4

Item	Comparisons	Item	Comparisons
Cloak of resistance +5	144	Orb of storms	406
Boots of elvenkind	77	Sceptre of Illusions	506
Ring of Defense	456	War Hammer +3/+2	647
Large Shield	341	Wizards Wardrobe	663
Robe of bones	477	Vampire Cone	635
Dream Weaver	188	Attenuation field	37
Strand of prayer beads, greater	564	Long Staff	348
Sack of Plunder	497	Manual of bodily health +3	362
Seeds of Plenty	510	Tower Shield	621
War Drum	645	Cap	112
Arrow of Wormwood	32	Boots of Running	78
Eversmoking bottle	212	Malhrek's Staff of Flame	354
Horseshoes of Water-Striding	317	Short Bow	523
Manual of quickness of action +5	374	Razor Leaf	448
Boat, folding	67	snowghost	540
Tome of understanding +1	611	Stone of alarm	560
Staff of Divine Winds	552	Dust of tracelessness	196
Club	148	Dimensional shackles 28,000 gp	177
Manual of quickness of action +1	370	Short Battle Bow	522
Elixir of hiding	203	Ranger's Sword	447
Banded mail of grounding	48	Mongoose Ring	387

Figure 2.4: The 42 items found with Linear Search and their comparisons

With 42 items, our average number of comparisons comes to 366.71.

This number comes from O(666). If there are 666 items, every search has an expected amount of comparisons to be 666/2 or around 333. Our number is close to this expected value which means we are right on the money!

3 Binary Search

3.1 Understanding Binary Search

Binary search is a slightly more complicated version of searching, but it is much much quicker. Think about how we can efficiently find a word in a dictionary like one in Figure 3.1. You start by opening to the middle page. If your word comes later, you open to the middle of the top half. You constantly divide your range in half until, eventually, the word is found.

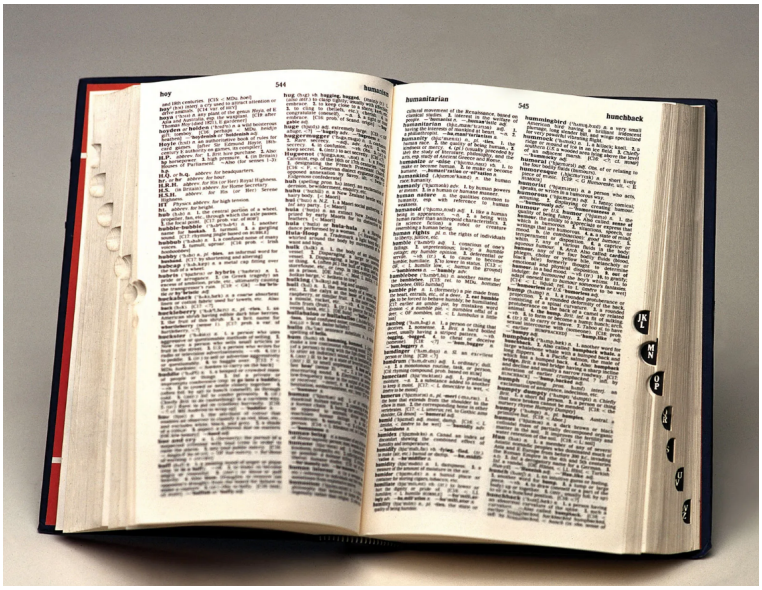

Figure 3.1: A dictionary... duh[2]

3.2 Coding Binary Search

Let's look at the code for using binary search to find a magic item in an array in Figure 3.2.

On line 2 and 3 we set the low and high indexes. This is the range of where we will be looking; to start the range is the whole array. We create a middle value `mid` and check to see the range works on line 6. We loop until either finding the target or shrink to range too far.

We calculate the middle index on line 7 and count a comparison. If the element at `mid` equals our target, we can return our position. If our target falls earlier on line 12, we make the new range only cover the former half of the array. If our target falls after this `mid` element on line 15, we make the new range the latter half of the array. Then, we search again.

If we have searched and not found it, we jump out of the while loop and return -1 at line 20.

[2]Image: https://www.britannica.com/summary/dictionary

```
int binarySearch(string items[], string target){
    int low = 0;               //inclusive
    int high = _NUM_OF_ITEMS; //exclusive
    int mid;

    while (low < high){
        mid = floor((low+high)/2);
        _comparisons ++;
        if (isEqual(target, items[mid])){
            return mid;
        }
        else if (isLessThan(target, items[mid])){
            high = mid;
        }
        else {
            low = mid + 1;
        }
    }

    return -1;
}
```

Figure 3.2: Binary Search in C++

3.3 Testing Binary Search

Going back to the main function again, we will use binary search to look for each item of our subset in the total 666 magic items.

```
    std::cout << "\nBINARY SEARCH" << "\n" << std::endl;
    int binaryComparisons = 0;
    for (int i = 0; i < _SUB_ITEMS; i++){

        _comparisons = 0;
        item = randomItems[i];
        index = binarySearch(magicItems, item);
        binaryComparisons += _comparisons;

        //print num of comparisons for the item
        std::cout << item << "\n\tComparisons: " << _comparisons << std::endl;
    }

    //calculate avg comparisons, round 2 decimal place, print
    float avgBinaryComparisons = (float) binaryComparisons / _SUB_ITEMS;
    avgBinaryComparisons = (int) ((avgBinaryComparisons + 0.005) * 100) / 100.0;
    std::cout << "\nAverage Binary Search Comparisons: "<< avgBinaryComparisons << "\n" <<
    std::endl;
```

Figure 3.3: Binary Search Test in C++

In Figure 3.3, we first loop through every item in the sub array of 42 items on line 3. We reset the comparisons to 0, and using binary search, we find the index of our first sub item `randomItems[1]` on line 7. We'll add it to the total number of comparisons for binary search, then repeat for the other 41 items. After all is complete, the code will print the average number of binary search comparisons for 42 items on lines 15 through 17.

This binary search function will have a complexity of:

$$O(log(n))$$

To explain why, look at line 7 of Figure 3.2. Here, we are dividing the range in half every time. This `mid` value will eventually be used for a new beginning or end of the range. We loop through until we have divided the array in half and get to size 0.

An example output of binary search comparisons can be seen in Figure 3.4

Item	Comparisons	Item	Comparisons
Cloak of resistance +5	9	Orb of storms	8
Boots of elvenkind	8	Sceptre of Illusions	9
Ring of Defense	10	War Hammer +3/+2	5
Large Shield	10	Wizards Wardrobe	10
Robe of bones	10	Vampire Cone	8
Dream Weaver	5	Attenuation field	7
Strand of prayer beads, greater	5	Long Staff	8
Sack of Plunder	10	Manual of bodily health +3	10
Seeds of Plenty	8	Tower Shield	7
War Drum	8	Cap	10
Arrow of Wormwood	6	Boots of Running	9
Eversmoking bottle	8	Malhrek's Staff of Flame	9
Horseshoes of Water-Striding	8	Short Bow	10
Manual of quickness of action +5	8	Razor Leaf	8
Boat, folding	10	snowghost	9
Tome of understanding +1	7	Stone of alarm	10
Staff of Divine Winds	8	Dust of tracelessness	9
Club	10	Dimensional shackles 28,000 gp	9
Manual of quickness of action +1	9	Short Battle Bow	5
Elixir of hiding	9	Ranger's Sword	9
Banded mail of grounding	7	Mongoose Ring	6

Figure 3.4: The 42 items found with Binary Search and their comparisons

With 42 items, our average number of comparisons comes to 8.29.

This number comes from O(log(666)). If there are 666 items, every search has an expected amount of comparisons to be log(666) or around 9.38. Our number is close and even a bit less than this expected value because sometimes the search function finds our target before reaching a range of 1. Cool beans!

4 Hashing

4.1 Understanding Hashing

Hashing is one way to store variables into a large array. Look at Figure 4.1 to see how names are stored in a hash map. We start with a given value, like "James" and hash it. Hashing can be done in a number of ways, but it somehow converts a string to a number. We can count the number of letters, or in this assignment, add up the ASCII values of each character in the string.

One thing to note is that the hashing function must produce a number that will fit within our array. This green storage array has indexes 00 through 06, so the hash function must produce a number in this range.

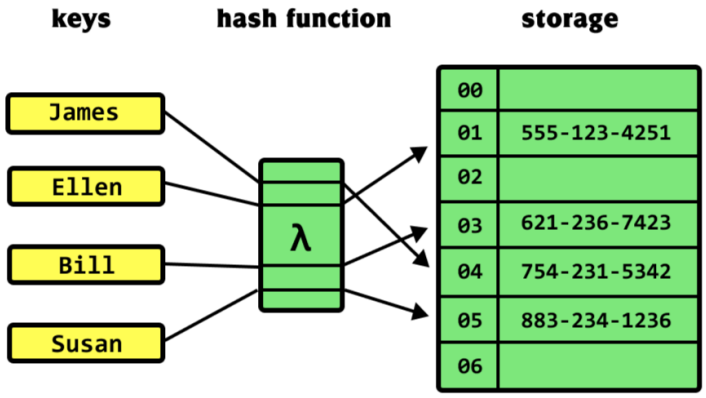

Figure 4.1: Storing names in a Hash Map[3]

A problem occurs when more than one string gets mapped to the same location. We do not want to override our data, so we have to store the value in what is called a chain.

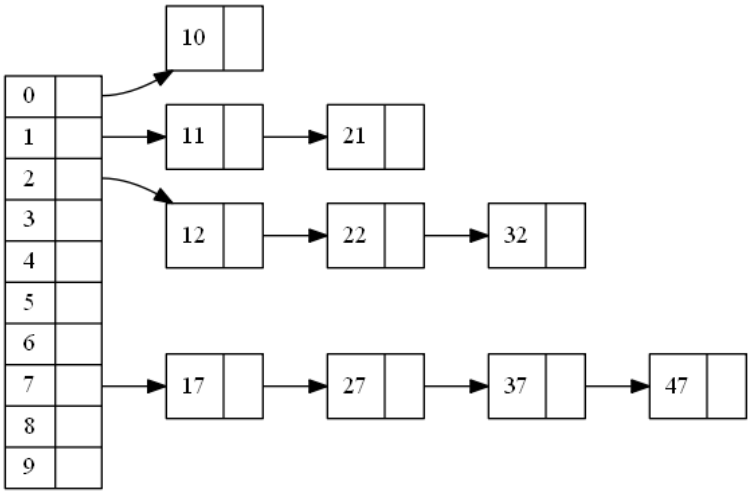

Figure 4.2: Hash Map with chaining[4]

[3]Image: https://khalilstemmler.com/blogs/data-structures-algorithms/hash-tables/
[4]Image: https://jasonlue.github.io/algo/2019/08/27/clustered-hashing-basic-operations.html

Figure 4.2 on the previous page shows what our hash array looks like with chaining. We use nodes and linked lists (see Assignment 1) to store every one of our values.

4.2 Coding with Hash Tables

The first step in hashing is have a function that can turn a string into a number. Figure 4.3 takes in a string and does just that. We go letter by letter on line 5, and increase `asciiTotal` by however much each character value is in ASCII. We then return this total, modulus `_HASH_TABLE_SIZE`, so it can stay within our range of hash array locations. For this assignment, the hash table size is 250. Because this is smaller than 666, we must prepare for chaining.

```
1 int makeHashCode(string item){
2
3     int asciiTotal = 0;
4     //Add up the ASCII values of all chars in the item
5     for (int i = 0; i < item.length(); i++){
6         asciiTotal += item[i];
7     }
8     return asciiTotal % _HASH_TABLE_SIZE;
9 }
```

Figure 4.3: Make Hash Code given a string in C++

Figure 4.4 shows how a given string is put into a hash array. We first get the index by putting our string into `makeHashCode()` on line 4. With this information, we create a Node with the name of the string on lines 7 and 8. To put it into our hash table, we just make this Node's next value what used to be the first array in that index, then make this node that first pointer. We are adding to the front of the chain.

```
1  void insert(Node* hashTable[], string item){
2
3      //find the hash location
4      int index = makeHashCode(item);
5
6      //create a node
7      Node* myNode = new Node();
8      myNode->itemName = item;
9
10     //place this node in front of previous node at hash index
11     myNode->next = hashTable[index];
12     hashTable[index] = myNode;
13 }
```

Figure 4.4: Insert string into Hash Array in C++

```
1  int hashSearch(Node* hashTable[], string item){
2
3      //find the hash location
4      int index = makeHashCode(item);
5
6      _comparisons ++;
7
8      if (hashTable[index] == nullptr){
9          return -1;
10     }
11     else if (isEqual(item, hashTable[index]->itemName)){
12         return index;
13     }
14
15     //check the chain
16     else {
17         Node* temp = new Node();
18         temp = hashTable[index]->next;
19
20         //is next spot availible? if not, move to next node
21         //+1 comparison for every
22         while (_comparisons ++ && temp != NULL){
23             if (isEqual(temp->itemName, item)){
24                 return index;
25             }
26             temp = temp->next;
27         }
28         //not found in chain
29         return -1;
30     }
31 }
```

Figure 4.5: Hash Search in C++

Above, in Figure 4.5, our goal is to find the location of the given string. Right away, we find the hash code on line 4. First, we check if the hash code index has a Node pointer in it. If it does not, return -1. If there is something there, we check this node on line 11. If this first node in the chain is our target, return the index. If not, move on down throughout the chain starting at line 16.

Here, we create a temporary Node pointer `temp`. We check to see if there is a second node in the chain on line 22. If there is, we compare the names, return index if we found our target. If it is not found, we move to the next node in the chain on line 26. We repeat until we have reached the end of the chain. If the target hasn't been found, return -1 on line 29.

Notice that our comparisons come on line 6, to count the initial target check on line 11. Then, we count on every while loop on line 22. This counter will take, on average, the length of the chain. Remember this idea when thinking about complexity in the next section.

Back to the main function in Figure 4.6, a Hash Array of node pointer is created. We initialize each element as a null pointer on lines 3 and 4. Then, on lines 8 and 9, we add all 666 items into the hash array.

```
1     std::cout << "\nHASH TABLE" << "\n" << std::endl;
2     Node* magicHash[_HASH_TABLE_SIZE];
3     for (int i = 0; i < _HASH_TABLE_SIZE; i++){
4         magicHash[i] = nullptr;
5     }
6
7     //load HASH TABLE with all 666 items
8     for (int i = 0; i < _NUM_OF_ITEMS; i++){
9         insert(magicHash, magicItems[i]);
10    }
```

Figure 4.6: Populating the Hash Array in C++

4.3 Testing Hash Tables

Time to test it out! Let's go through each item in of sub array of 42 items on line 2 in Figure 4.7. We will hash search for the index on line 6 and add up the comparisons. Repeating this 42 times, we will get our average number of comparisons for our special magic items.

```
1     int hashComparisons = 0;
2     for (int i = 0; i < _SUB_ITEMS; i++){
3
4         _comparisons = 0;
5         item = randomItems[i];
6         index = hashSearch(magicHash, item);
7         hashComparisons += _comparisons;
8
9         //print num of comparisons for the item
10        std::cout << item << "\n\tComparisons: " << _comparisons << std::endl;
11    }
12
13    //calculate avg comparisons (get and chaining), round 2 decimal place, print
14    float avgHashComparisons = (float) hashComparisons / _SUB_ITEMS;
15    avgHashComparisons = (int) ((avgHashComparisons + 0.005) * 100) / 100.0;
16    std::cout << "\nAverage Hash Search Comparisons: "<< avgHashComparisons << "\n" << std::
      endl;
```

Figure 4.7: Hashing Test in C++

Figure 4.8 below shows the number of comparisons for each magic item in our sub array. Notice how these numbers are much smaller compared to the other search methods. While some items required five or six comparisons, many only require one or two!

Item	Comparisons	Item	Comparisons
Cloak of resistance +5	3	Orb of storms	2
Boots of elvenkind	3	Sceptre of Illusions	1
Ring of Defense	1	War Hammer +3/+2	1
Large Shield	2	Wizards Wardrobe	1
Robe of bones	2	Vampire Cone	1
Dream Weaver	4	Attenuation field	5
Strand of prayer beads, greater	2	Long Staff	3
Sack of Plunder	3	Manual of bodily health +3	1
Seeds of Plenty	1	Tower Shield	1
War Drum	1	Cap	3
Arrow of Wormwood	6	Boots of Running	2
Eversmoking bottle	3	Malhrek's Staff of Flame	2
Horseshoes of Water-Striding	1	Short Bow	2
Manual of quickness of action +5	1	Razor Leaf	1
Boat, folding	2	snowghost	1
Tome of understanding +1	1	Stone of alarm	1
Staff of Divine Winds	1	Dust of tracelessness	3
Club	5	Dimensional shackles 28,000 gp	5
Manual of quickness of action +1	2	Short Battle Bow	2
Elixir of hiding	3	Ranger's Sword	2
Banded mail of grounding	6	Mongoose Ring	1

Figure 4.8: The 42 items found with Hash Search and their comparisons

With this data, we get an average number of comparisons to be 2.24 per item.

Recall Figure 4.5 where every look up included a search, and a look through the chain length. This complexity can be calculated by adding these two numbers together.

The average chain length, also known as the load factor, can be determined by taking the total number of items and divided it by the number of spots available in the hash table:

$$LoadFactor = \alpha = \frac{n}{size}$$

And this number gives us the complexity of a hash search:

$$O(1 + \alpha)$$