Assignment Two

Christian Sarmiento

christian.sarmiento1@marist.edu

November 2, 2024

1 Introduction

Assignment Two focuses on creating a program that searches for 42 items in a bigger array of 666 magic items stored in string format (magicitems.txt) using Linear Search, Binary Search, and Hashing. For each of the searches, the comparisons used to find each item is kept track of and as well the average number of comparisons for each of the processes. This program was developed using C++.

2 Searching

For both of the searching algorithms, 42 items are randomly sampled from the bigger array of 666 items using this random selection function:

```
1     std::vector<std::string> getSearchItems(std::vector<std::string> items) {
2
3     // Variables
4     int randomPos = 0;
5     std::vector<std::string> randomSelections;
6     srand((unsigned) time(NULL));  // setting seed value
7
8     // Get 42 entries
9     for (int i=0; i < 42; i++) {
10
11        // Select a random position in the array
12        randomPos = rand() % items.size();
13
14        // Save a random entry
15        randomSelections.push_back(items[randomPos]);
16
17    } // for i
18
19    return randomSelections;
20
21 } // getSearchItems()
```

This function will use the current time as the seed value (Line 6) to be able to select a random position in the bigger array. The for loop then iterates 42 times and selects a random position, using a modulo operator to

ensure that the value stays within the range of the array (Line 12). After a position is selected, the position is used to select the magic item and is pushed back into the vector that stores the random items to be searched (Line 15). This function defines and populates the array within the scope of the function, thus causing the full array of random items to be returned to main (Line 19).

After the items are randomly selected, the bigger array of magic items is then sorted using QuickSort. This is done before both of the search algorithms as per assignment requirements.

2.1 Linear Search - $O(n)$

For each of the 42 random items, Linear Search is performed using this algorithm:

```
1  int linearSearch(std::string target, std::vector<std::string> items) {
2
3      // Variables
4      bool found = false;
5      int numComparisons = 0;
6      int i = 0;
7
8      // Search through the list
9      while ((found != true) && (i < items.size())) {
10
11         numComparisons++;
12         if (items[i] == target)
13             found = true;
14
15         else
16             i++;
17
18     } // while
19
20     // Output results
21     if (found)
22         std::cout << "Linear Search found " << target << "." << std::endl;
23
24     else
25         std::cout << "Linear Search could not Find " << target << "." << std::endl;
26
27     return numComparisons;
28
29 } // linearSearch()
```

This algorithm performs in $O(n)$ time due to it utilizing only one loop (Lines 9 - 18) to go through the each of the magic items in the bigger list to find the target item. Utilizing a while loop instead of a for loop adds a layer of efficiency where we can do some control flow to avoid going through the entirety of the list if we find the target item before the list is exhausted. This is done with a "found" flag, initialized at Line 4 and only updated at Line 13 if the item is found. The flag is also used to dictate what output the user sees dependent on whether or not we found the item.

2.2 Binary Search - $O(log_2 n)$

For each of the same random 42 items, Binary Search is performed using this iterative algorithm:

```
1  int binarySearch(std::string target, std::vector<std::string> items, int startPos, int
       endPos) {
2
3      // Variables
4      int low = startPos;
5      int high = endPos;
6      int mid = 0;
7      int numComparisons = 0;
8      bool found = false;
9
10     // Search for the value
11     while (!found && low <= high) {
12
13         // Get the midpoint
14         mid = low + (high - low) / 2;
15
16         // Compare
17         numComparisons++;
18         if (target == items[mid])
19             found = true;
20
21         else if (target < items[mid])
22             high = mid - 1;
23
24         else
25             low = mid + 1;
26
27     } // while
28
29     // Print results
30     if (found)
31         std::cout << "Binary Search found " << target << "." << std::endl;
32
33     else
34         std::cout << "Binary Search could not find " << target << "." << std::endl;
35
36     return numComparisons;
37
38 } // binarySearch()
```

This algorithm performs in $O(log_2 n)$ time due to the algorithm cutting the available options of where the target would be in half on each iteration. This is specified in the algorithm where if the target is not found, the low or high position of the available list is adjusted depending if the target is lower or greater than the current midpoint of the available list (Lines 21-22, 24-25). The current midpoint is always the first thing calculated on each iteration (Line 14) since that is what dictates our conditions for the rest of the algorithm. A while loop is also used here since we want to stop once the target is found and also because the values for our positions are dictated by division and not by increments.

Although there is a recursive version of this algorithm which achieves the same results, I personally had trouble writing that algorithm to have only one return while also outputting the proper responses to the user in an orderly fashion. Because of the recursion, I would have output statements that would occur prematurely. For example, if we were to look for an item, we would get a bunch of output statements saying that the item was not found before the statement saying that it was found. I tried patching this with a found flag but due to recursion, it becomes messy to accurately keep track of the state of that flag. For the purposes of clean output and an easier implementation of using only one return, I opted for the iterative version of Binary Search.

2.3 Hashing

For our hash table, I made a hash table class to deal with the different aspects of the table. Here are the constructor definitions:

```
1  /**
2   * Null Constructor for hash table class
3   */
4  HashTable::HashTable() {
5
6      HASH_TABLE_SIZE = 250;
7      myTable.resize(HASH_TABLE_SIZE, nullptr);
8
9  } // Null Constructor
10
11 /**
12  * Full Constructor for hash table class
13  *
14  * @param tableSize size for the hash table
15  */
16 HashTable::HashTable(int tableSize) {
17
18     HASH_TABLE_SIZE = tableSize;
19     myTable.resize(HASH_TABLE_SIZE, nullptr);
20
21 } // Full Constructor
22
23 /**
24  * Deconstructor method to remove nodes in the hash table from memory.
25 */
26 HashTable::~HashTable() {
27
28     // Variables
29     NodeLinkedList* curr = nullptr;
30     NodeLinkedList* temp = nullptr;
31
32     // Iterate through the indexes
33     for (int i=0; i < HASH_TABLE_SIZE; i++) {
34
35         // Set current to the head at the index
36         curr = myTable[i];
37
38         // Go through the chain at the index and remove it from memory
39         while (curr) {
40
41             temp = curr;
42             curr = curr->getNext();
43             delete temp;
44
45         } // while
46
47     } // for
48
49 } // Deconstructor
```

The null constructor automatically defines the hash table to be of size 250 per assignment requirements (Lines 6-7). For software craftsmanship purposes, the full constructor allows you to change this value to whatever is needed. The deconstructor method ensures that all nodes used for chaining are deleted once the code leaves scope to ensure proper memory management.

The class also contains a method that makes the hash table index for the value passed:

```
1  int HashTable::makeHashCode(std::string item) {
2
3      // Transform item to UPPER CASE.
4      transform(item.begin(), item.end(), item.begin(), [](unsigned char c) {
5
6      return std::toupper(c);
7
8      });
9
10     // Iterate over all letters in the item string, totalling their ASCII values.
11     int letterTotal = 0;
12     for (int i = 0; i < item.length(); i++) {
13       letterTotal = letterTotal + (int)item[i];
14     }
15
16     // Scale letterTotal to fit in HASH_TABLE_SIZE.
17     int hashCode = letterTotal % HASH_TABLE_SIZE;  // % is the "mod" operator
18     // TODO: Experiment with letterTotal * 2, 3, 5, 50, etc.
19
20     return hashCode;
21
22 } // makeHashCode()
```

This method, written by Dr. Labouseur, takes in a string item and creates a hash code based on the ASCII values of each character in the string. I had to change the code a little bit to use a lambda function (Lines 4-8) since it is an entire string being passsed. The original code gave me errors which required this to be done.

The next method in the class is the put() method, which puts an item into the hash table:

```
1  void HashTable::put(const std::string& item) {
2
3      // Variables
4      int hashKey = 0;
5      NodeLinkedList* newNode = new NodeLinkedList();
6
7      if (newNode) {
8
9          // Make key with the hash function
10         hashKey = makeHashCode(item);
11
12         // Make node with the item
13         newNode->setData(item);
14
15         // Add to the table
16         newNode->setNext(myTable[hashKey]);
17         myTable[hashKey] = newNode;
18
19     } // if
20
21     else
22         std::cerr << "Memory allocation failed for new node" << std::endl;
23
24 } // put()
```

The main work of the function is wrapped in an if statement to ensure that the node to be added was initialized properly (Lines 5, 7). The makeHashCode() function is used to make the key for the value to be inserted (Line 10). The new node that was initialized is then populated with data at Line 13 and then the node is inserted into the hash table at the position provided by the hash key (Lines 16-17). This implementation treats the chaining like a stack for each chain, reducing the complexity for adding to the chain if there is

more than one value at the index. This method operates in constant time since there are no loops involved, just a few variable assignments.

Lastly, we have a get() method that searches for an item in the hash table:

```
1  int HashTable::get(const std::string& item) {
2
3      // Variables
4      int numComparisons = 0;
5      int index = 0;
6      NodeLinkedList* curr = nullptr;
7      bool found = false;
8
9      // Get index where item is stored
10     index = makeHashCode(item);
11
12     // Get to the node where it is stored
13     curr = myTable[index];
14
15     // Go through chain at the index
16     while (!found && curr) {
17
18         // Compare
19         numComparisons++;
20         if (curr->getData() == item) {
21
22             std::cout << "Found " << item << " in the hash table." << std::endl;
23             found = true;
24
25         } // if
26
27         curr = curr->getNext();
28
29     } // while
30
31     // Let user know if the item was never found
32     if (!found)
33         std::cout << "Cound not find " << item << " in the hash table." << std::endl;
34
35     return numComparisons;
36
37 } // get()
```

The method first gets the index at where the value could be stored at (Line 10). We then direct our attention to the head node at that index (Line 13). Using a while loop (Line 16), we get the data in the current node and compare the item there to the target (Line 20). If found, we let the user know and set our found flag to true in order to stop the loop (Lines 22-23). Regardless if it was found or not, we move to the next node in case we did not find it (Line 27). After the while loop, if the target was never found, we let the user know (Lines 32-33). The number of comparisons used to look for the target is returned (Line 35). This get() method operates in $O(n)$ time since it is using a while loop to iterate through the chain that exists at a given index to find the target.

3 Results

As previously stated, each item had the number of comparisons noted and then averaged out for each of the searches. For hashing, we also include the number of get operations used. Let $E(x)$ be the expected value for comparisons. In other words, $E(x) = \frac{0(x)}{2}$, where $O(x)$ is the time-complexity operation for the given method. Please note that for Hashing, our load factor, α, is $\alpha = \frac{n}{s}$ where n = the total number of items in the hash table and s = the size of the hash table. For this assignment, $\alpha = \frac{666}{250} = 2.66$.

For each test in the table, there are three separate runs of the program, meaning that those values are the average comparisons for that run. The final average is an average of those averages.

Search Method	$E(x)$	Test 1	Test 2	Test 3	Final Average
Linear Search - $O(n)$	333	357.05	372.36	356.10	361.84
Binary Search - $O(log_2 n)$	9.38	8.38	8.14	8.60	8.37
Hashing - $O(1 + \alpha)$	3.66	3.38	3.50	2.95	3.28

The searching algorithms work as expected given our expected values and our final averages. Hashing of course works the fastest due to the indexing that hash tables make use of. Our load factor α tells us that we expect an average chain length of about 2.66, meaning that on average, we should not expect to be going through chains in a given index more than that value. This is faster than Binary Search and way faster than Linear Search, showing why hash tables are a very effective and efficient option for data storage in a lot of cases. If a hash table does not suit the problem at hand, Binary Search is also a great option with an average of 8.37 comparisons to find a given value. Linear Search is painfully brute force and that number only gets worse with a greater n size. Moral of the story: use Linear Search if you absolutely need to/if its more efficient than sorting a list to do Binary Search. Or just use a hash table, always the answer to the problems of life.

4 References

- SimpliLearn - What Is the SetPrecision Function in C++ and Examples
- StackOverflow - Recursive Binary Search Function
- Labouseur.com - Algorithms Slides
- ChatGPT - Personal Chat for C++ Syntax & Help