CMPT 435 - Fall 2024 - Dr. Labouseur

Assignment Two

Jayden Melendez
Jayden.Melendez1@Marist.edu

October 27, 2024

1 Linear Search

1.1 Code Implementation

```
// Method to do a linear search on an array of items
void Search::doLinearSearch(std::string *items, Stack &stack, int size) {
    int comparisonCount = 0; // Comparison counter for each search
    int totalComparisons = 0; // Search comparisons searches
    double averageComparisons = 0; // Hold the average number of comparisons
    // Search for each item in stack in items
    while (!stack.isEmpty()) {
        std::string item = stack.pop(); // Get next item
        bool found = false; // Default to not found
        // Iterate through items to search for item popped from stack
        for (int i = 0; i < size; i++) {
            comparisonCount++;
            if (items[i] == item) {
                found = true;
                break;
            }
        }
        // Print if an item is not found in items
        if (!found) {
            std::cout << "Item not found: " << item;
        }
        // Update the total number of comparisons
        totalComparisons = totalComparisons + comparisonCount;
        // Update comparison counter for the next item
        comparisonCount = 0;
    }
    // Compute the average number of searches
    std::cout << "---------------------------------------" << "\n";
    std::cout << "LINEAR SEARCH" << "\n" << "\n";
    averageComparisons = totalComparisons / 42.0;
    std::cout << std::fixed << std::setprecision(2);
    // Set precision to 2 decimal places
    std::cout << "Average Comparisons: " << averageComparisons << "\n";
    std::cout << "---------------------------------------" << "\n";
}
```

Figure 1: Header file for the Node class in a linked list.

1.2 Complexity Overview

The time complexity of linear search in the best, worst, and average cases is as follows:

- **Best Case**: $O(1)$ - when the target element is the first in the list.
- **Worst Case**: $O(n)$ - when the target element is the last in the list or not present.
- **Average Case**: $O(n/2)$, which simplifies to $O(n)$ because constants are ignored in complexity.

1.3 Complexity Explanation

Linear search, on average is $O(n/2)$ complexity. This is attributed to the fact that we are always searching through the entire array. The item is either in the first half or the second half of the array. Therefore, when there are multiple test cases that track the number of comparisons, the instances where the item was in the first half and the instances it was in the second half, average out. This averaging out is the reasoning why a calculated average number of comparisons is fairly close to the midpoint of the array. In the case of the magic items array, there are 666 items. In a perfect world, the average number of comparisons would be 333 because of 666/2. In the test cases shown in Table 1 below, the average number of comparisons is 318.39. If there were more test cases, this number would likely get closer to the midpoint of the array, 333.

1.4 Comparison Data

Iteration	**Number of Comparisons**
1	297.83
2	362.62
3	306.93
4	310.43
5	333.88
6	271.74
7	378.90
8	287.79
9	303.76
10	330.02
Total Comparisons	3183.90
Average Comparisons	318.39

Table 1: Number of comparisons for each iteration of Linear Search

1.5 Code Explanation

The **doLinearSearch**() method, found in Figure 1, is one of the many ways of creating a custom linear search algorithm. Its parameters include, taking in an array of item strings, a stack of item strings to search for, and the size of the array to maintain data integrity. One line 3, line 4, and line 5, variables for the number of comparisons, total number of comparisons, and average number of comparisons are defined respectively. In this context, comparisons are the number of elements iterated through to find the desired string from the stack. These variables will be used to determine the efficiency of the algorithm. On line 7, there is a while loop that continuously pops items from the stack. These items are fed into the algorithm and are individually searched for in the items array. The while loop continues until the stack is empty. On line 8, the item is popped. On line 9, a boolean, **found**, is defined and set to false. This variable will change to true when an items is found. On line 11, a for loop is used to iterate through the items array. The current index of the for loop and the current item popped from the stack are compared. In order to ensure an accurate count of comparisons, the comparison counter is incremented before the strings are compared in order to prevent a premature break from the loop that would skip over the final comparison. On line 19, if the item is not found, the method prints that there were no strings found in the items array that matched the item string popped from the stack. On line 23, total comparisons is incremented in order to keep track of how many comparisons total for all 42 items in the stack were compared. One line 25, the comparison counter is reset to 0 for the next item popped from the stack.

2 Binary Search

2.1 Code Implementation

```
// Method to do a binary search on an array of items
void Search::doBinarySearch(std::string *items, Stack &stack, int size) {
    double comparisonCount = 0; // Comparison counter
    double totalComparisons = 0; // Total comparisons
    double averageComparisons = 0; // Average number of comparisons
    int midpoint = 0;
    int startIndex = 0;
    int endIndex = 0;
    bool found = false;
    // Search for all items in the stack
    while(!stack.isEmpty()) {
        std::string item = stack.pop(); // Get the next item
        // Keep track of the focused portion of the array
        midpoint = size / 2;
        startIndex = 0;
        endIndex = size - 1;
        found = false; // By default, the item is not found
        comparisonCount = 0; // Counter to keep track of comparisons
        // Continuously halve the array
        while(!found && startIndex <= endIndex) {
            midpoint = startIndex + (endIndex - startIndex) / 2;
            comparisonCount++;
            if (item == items[midpoint]) { // Item was found
                found = true;
            } else if (item < items[midpoint]) { // Focus on the left
                endIndex = midpoint - 1;
            } else if (item > items[midpoint]) { // Focus on the right
                startIndex = midpoint + 1;
            }
        }
        totalComparisons += comparisonCount; // Update total comparisons
    }

    std::cout << "BINARY SEARCH" << "\n" << "\n";
    averageComparisons = totalComparisons / 42;
    // Set precision to 2 decimal places
    std::cout << std::fixed << std::setprecision(2);
    std::cout << "Average Comparisons: " << averageComparisons << "\n";
    std::cout << "-------------------------------------" << "\n";
}

}
```

Figure 2: Header file for the Node class in a linked list.

Complexity Overview

The time complexity of binary search in the best, worst, and average cases is as follows:

- **Best Case**: $O(1)$ - when the target element is the middle element of the array.
- **Worst Case**: $O(\log n)$ - as the search space is halved with each step.

- **Average Case**: $O(\log n)$ - as the algorithm consistently reduces the search space by half regardless of the target's position.

2.2 Complexity Explanation

Binary search, on average is $O(\log n)$ complexity. This is because the array of items is constantly halved until a match is found. This means that for all cases, the algorithm is always pruning half of the items in the array. The mathematical representation of this concept is $O(\log n)$. In the algorithm, the midpoint of the array is determined. Then, the item that is being searched for is being compared to the midpoint. If the item is greater than the midpoint, the algorithm only focuses on the second half of the array. If the item is less than the midpoint, the algorithm only focuses on the first half of the array. This process continues until the item is the midpoint or the array cannot be split anymore, meaning the item was not found. Binary search is only applicable if the array is sorted in order to sure the comparison of the current item and midpoint are representative of the data as a whole.

2.3 Comparison Data

Iteration	**Number of Comparisons**
1	8.55
2	8.21
3	8.69
4	8.45
5	8.50
6	8.67
7	8.50
8	8.71
9	8.43
10	8.43
Total Comparisons	85.14
Average Comparisons	8.51

Table 2: Number of comparisons for each iteration of Linear Search

2.4 Code Explanation

The code in Figure 2 is a custom representation of a binary search algorithm. The method is called **doBinarySearch**(). On lines 3-5, the variables for counting and calculating comparison values for complexity analysis are defined. Line 6 defines an integer that represents the midpoint of the array. On line 7, the start index that represents the starting point of the array is defined. Line 8 is the end index variable for the last relevant index of the array. The boolean, found, on line 9 is set to false until an item is found. This implementation doesn't change or add arrays. The start and end index variables are used to represent the focused part of the array after the current item is compared to the item at the midpoint. Within the first while loop on line 11 that encompasses the algorithm, the midpoint is always initialized to half the size of the array. The start index remains 0 to represent the first index. The end index is one less than the size of the array. The inner while loop is where the nitty gritty of the algorithm begins. It continues until the boolean, **found**, changes to true. This occurs on line 24, when the midpoint is equal to the value of the item that was popped from the stack. The while loop can also end if the start index is greater than the end index. This is an indication that on the final iteration, the item was never found to be at any of the midpoints. For each split of the array at the midpoint. On line 21, the midpoint is adjusted to be the value of the start plus the value of the end minus the start divided by 2. In a simple example, with the original values, the midpoint would be 333 because the start (0) plus the end (665) minus the start (0) divided by 2 is 333 on a scale of 0-665. The conditional logic starting on line 23 reads as follows. When the item and the

midpoint are equal, the item has been found, so the while loop ends and the method continues. When the item is less than the item at the midpoint, the end index is shifted to one less of the midpoint to represent the pruning of the second half of the array. When the item is greater than the item at the midpoint, the start index becomes one greater than the midpoint to represent pruning the first half of the array. This process continues until the condition in the while loop is no longer satisfied. The data from the test cases performed slightly better than the expected value. The complexity $O(\log n)$ with a base of 2 and with 666 items is approximately 9.38. The average number of comparisons for the test dat was 8.51. If there were more text cases, the average number of comparisons would begin creeping closer to the expected value.

3 HashMap

3.1 Code Implementation

```
// Get the index in HashTable for a string
int HashTable::makeHashCode(std::string str) {
    transform(str.begin(), str.end(), str.begin(), ::toupper);
    int length = str.length();
    int letterTotal = 0;
    // Iterate over all letters in the string, totalling their ASCII values.
    for (int i = 0; i < length; i++) {
        char thisLetter = str[i];
        int thisValue = (int)thisLetter;
        letterTotal = letterTotal + thisValue; }
    int hashCode = (letterTotal * 1) % HASH_TABLE_SIZE;
    return hashCode; }

void HashTable::populateHashTable(std::string item) {
    // Check to see if the index is empty
    int hashCode = makeHashCode(item);
    Node *node = new Node(item);
    if (buckets[hashCode] == nullptr) {
        buckets[hashCode] = node;
    } else {
        Node* current = buckets[hashCode];
        // Iterate through until the end of the list
        while (current->getNext() != nullptr) {
            current = current->getNext();
        }
        // Add the desired node to the end of the list
        current->setNext(node);

int HashTable::getValueFromHashTable(std::string item) {
    int hashCode = makeHashCode(item); // Get the hash code value
    int comparisons = 1; // Initialize to 1 to represent the first get in the array
    Node* current = buckets[hashCode];
    while(current->getData() != item && current != nullptr) {
        current = current->getNext();
        comparisons = comparisons + 1;
    }
    if (current == nullptr) {
        comparisons = -1;
    }
    return comparisons;
}
```

Figure 3: Hash Table functions

3.2 Complexity Overview

Complexity Explanation

The time complexity for common operations on a hash table, assuming a good hash function, is as follows:

- **Average Case**:
  - **Insertion**: $O(1)$

  - **Deletion**: $O(1)$
  - **Search**: $O(1)$

  Hash tables provide constant-time complexity on average for these operations because of direct index access using a hash function.

- **Worst Case**:
  - **Insertion**: $O(n)$
  - **Deletion**: $O(n)$
  - **Search**: $O(n)$

  In the worst case, all elements hash to the same index, causing a collision. This results in a linked list structure at that index, making the operations $O(n)$.

3.3 Complexity Explanation

Hash tables are a remarkable data structure that allows for constant or close to constant time for a large dataset. At its core, a hash table is an array of pointers pointing to objects that contain data. Each index of the array has a pointer. Each node following the first pointer has a pointer which creates a chain. This is also known as a linked list. Therefore, a hash table is an array of linked lists. If each index of the table, known as a bucket, had one item in it, the hash table would be representing a normal array. Insertion, deletion, and searching would all be constant time. Insertion and deletion are constant time because adding an item designates a space in memory for that item to be called by a program. Searching is constant time for a different reason. Hash tables have a unique property that makes them distinctly different from normal arrays. An item is placed in a bucket in the hash table based upon its hash value. This value can be determined in many different ways but, it should be a representation that would limit the number of collisions in the table. A collision is when two elements share the same hash. In the typical hash table approach, chaining solves this issue by appending each additional item with the same hash onto the linked list found at the bucket index. This characteristic of hash tables allows for a string to be converted into the hash that the table takes and access the index of the array from memory. This is because accessing data from an array index is constant time. Therefore, if each bucket has one item, retrieving an item will always be at constant time. When collision events happen and new nodes containing item strings are added to the chain at the bucket index, the hash table slowly begins to degrade. If a bucket index has 3 strings in 3 nodes chained together in order to add a new string in a node, the program would need to access the array index from the hashcode and then iterate through the list until it gets to the end. This becomes $O(n)$ complexity, where n is equal to 3. A good hash should distribute elements relatively evenly. When there is an even distribution of elements, the complexity degradation is minimal. Two or three elements at a bucket would not make a big difference in inserting, deleting, or searching because we are only ever concerned about the length of the linked list at one particular bucket index.

3.4 Comparison Data

Iteration	Number of Comparisons
1	1.98
2	2.19
3	2.21
4	2.17
5	2.40
6	2.17
7	2.48
8	2.31
9	2.45
10	2.52
Total Comparisons	22.88
Average Comparisons	2.28

Table 3: Number of comparisons for each iteration of a Hash Table

3.5 Code Explanation

There are three main components of the custom hash table. A hash table implementation needs a function that defines the hash value, populates the table with a value, and retrieves a value. The implementation above separates these functions into three different methods names **makeHashCode()**, **populateHashTable()**, and **getValueFromHashTable()**. The first method listed, **makeHashCode()**, is found on line 2. When this method is called, an integer is returned that will be the bucket the item will be placed into in the hash table. In order to maintain debugging consistency, the string is made into all uppercase on line 3. The length of the string is saved in an integer variable called length. The letterTotal variable is initialized to 0 and will be used to keep track of the sum of ASCII values for the string. The for loop on line 7 iterates through the string. On line 8, the current letter in the loop is saved in the thisLetter char. The value is saved in thisValue on line 9. On line 10, the letterTotal variable is incremented with the value of current character. When the loop terminates on line 11, the hashCode value is take and translated to fit into the size of the hash table. In the case of this testing, the hash table was set to a size of 250 to encompass 666 item strings.

The next method, populateHashTable(), takes in a string and places it into the hash table. On line 15 a variable, hashCode is defined that called the makeHashCode() method. The value returned will be the bucket index for the item string. The string will be saved into a node. The node is defined from a custom node class on line 16. The condition on line 17 checks to see if the bucket index is empty. If the bucket is empty, the node storing the string will be the head of the linked list. If the bucket has a a node, the item string at the first bucket index is put into a current variable. The current node variable represents the head of the list. The while loop on line 22 iterates through the linked list until a nullptr is detected while the getNext() method of the node class is invoked. This signals that the end of the list was found. When this happens, the node created with the new item string outside of the while loop is added to the end of the list with a pointer from the previous last element pointing into it.

The final method getValueFromHashTable() takes in an item string that will be used to compare against the hash table for searching. On line 30, the hashCode integer variable is the value returned from makeHash-Code() which is the bucket index the string will be found in. The comparisons variable is initialized to 1 on line 31 to represent the original get comparison when the algorithm navigates to the correct bucket index. A new node instance is created in the variable current to represent the head node object at the correct bucket that the item passed in will be linked to in a linked list. On line 33, a while loop begins that iterates through the linked list starting from the head, current variable. The while loop does not begin if current is a nullptr because this would mean that the bucket is empty and the string is definitely not in that index. On line 34, current is set to value of the next node. On line 35, the comparison counter is incremented. The loop ends when the item is found when invoking the getData() method on the current node object and its equal to

the value of the string passed into the method. If current is a null pointer, the method returns -1 to signify this. If the item was found, the number of comparisons is returned. The number of comparisons is 1 + the number of elements in the linked list starting from the bucket index.

4 Conclusion

This second assignment broadened my horizons on how the complexity of an algorithm can have such a drastic effect on its outcome. In terms of search, there are various methods that speed up how quickly a search algorithm can find an element. All of these algorithms have different weaknesses. The weaknesses of an algorithm are the reason why computer scientists and mathematicians were forced to discover so many. With this assignment, I have learned the importance of making an analysis to decide what algorithm to use. I also learned that the difficulty to write an algorithm does not always have to be complicated. The concept behind a hash table is not complicated. It is just an array of pointers. However, this simple modification to an already simple structure of an array can have drastic consequences. A hash table for searching is much more efficient than an array with linear search.

The continuation of the assignment with C++ has allowed me to continue refining my skills. I am becoming more comfortable with the common sytnax and structure of its files. I can myself using it in more of my person projects.