Assignment Two

Collin Drake
Collin.Drake1@Marist.edu

October 27, 2023

1 Linear Search

1.1 The Algorithm

The linear, or sequential, search algorithm is an algorithm used to search a collection of elements until it reaches the end of the collection or the desired object is found. This algorithm works by simply iterating one by one through each element and comparing it to the desired target. If the element is found then the search stops, otherwise, the search continues until it reaches the end of the array. The time complexity of linear search is linear time, O(n), meaning that as the input size grows, the time it takes to search through the collection of items grows linearly.

Linear Search

```
//linear search
float Searching:: linearSearch(vector<string>& magicItems, vector<string>& randomItems){
    //total comparisons
    int totlinComparisons = 0;

    //iterate through both vectors
    for(int i = 0; i < randomItems.size(); i++){
        //comparisons for each magic item
        int lComparisons = 0;
        string current = randomItems[i];
        for(int j = 0 ; j < magicItems.size(); j++){
            //compare each element in magic items to current element in random items
            lComparisons++;
            if(current.compare(magicItems[j]) == 0){
                //if found break loop
                break;
            }
        }
        //add to total
        totlinComparisons += lComparisons;
    }
    //find everage and round
    return (static_cast<float>(totlinComparisons) / static_cast<float>(randomItems.size()));
}
```

1.2 Asymptotic Analysis

In our example, given two different-sized collections of items, one larger and one smaller, when using linear search to look through the larger collection for each element in the smaller collection the time complexity is O(m * n). In this case, m stands for the size of the first collection while n stands for the size of the second, each is represented differently. In most use cases where linear search has a best-case running time of O(1), constant time, or worst-case of O(n), linear time, there is one element to search for in a collection of items. However, in our case, there are two collections of items that are of different lengths, denoting that there is a nested loop, which results in a quadratic time complexity of O(m * n).

2 Binary Search

2.1 The Algorithm

The algorithm binary search is a searching algorithm used to search a sorted collection of items by dividing the searchable collection with each comparison. Binary search works by checking if the target element is the middle object, greater than the middle object, or less than it. With each comparison, the searchable collection is reduced until the target is found or the entire array is searched. Generally, the binary search time complexity is O(log n), logarithmic time.

Binary Search

```
//binary search
float Searching:: binarySearch(vector<string>& magicItems, vector<string>& randomItems){
    //keep track of total comparisons
    int totbinComparisons = 0;

    //iterate through each element in random items
    for(int i = 0; i < randomItems.size(); i++){
        //comparisons for each magic item
        //current is the target
        //create start and end index variables
        int bComparisons = 0;
        string target = randomItems[i];
        int start = 0;
        int end = magicItems.size() - 1;

        while(start <= end){
            bComparisons++;
            //find middle index
            int middle = (start + end) / 2;

            //check if target is middle
            if(magicItems[middle] == target){
                break;
            }
            //if target is greater than middle, ignore left half
            if(magicItems[middle].compare(target) < 0){
                start = middle + 1;

            }
            //if target is less than middle, ignore right half
            else{
                end = middle - 1;
            }
        }
        //add to total
        totbinComparisons += bComparisons;
    }
    //find average and round
    return (static_cast<float>(totbinComparisons) /
    static_cast<float>(randomItems.size()));
}
```

2.2 Asymptotic Analysis

In our case given two collections of items, one larger, sorted array and one smaller array, when using binary search to look through the bigger collection for each target element in the smaller collection the time complexity is O(m * log n). Usually, the binary search algorithm has a best-case time complexity of O(1), constant time, or worst-case time complexity of O(log n), logarithmic time, but in our case, it is slower because we are given two collections to search through, m for the size of the first and n for the size of the second.

3 Hash table

3.1 The Data Structure

A hash table is a data structure that can be imagined as a vertical array that stores elements in key-value pairs. To store these pairs hash tables use hash functions, which determine the position a value is stored based on the hash of the given input. When a hash function produces the same hash for multiple different values this is what's called a hash collision. Hash collisions are resolved by hash tables that use chaining. Chaining in a hash table is when each index within the array contains its own linked list, allowing multiple duplicate hashes to be stored within the same index. There are two fundamental operations that make up a hash table: put and get.

constructor

```
//hash table constructor
HashTable:: HashTable(){
    this->hashTableSize = 250;

    //set each node to null and allocate memory for hash table
    hashTable.resize(hashTableSize, nullptr);
}
```

3.1.1 Put

The put function of a hash table adds the input value to an index within the table. When the put function is called the given value is hashed and mapped to a specific index within the array. If there are multiple other nodes within the mapped index, the new node is placed at the head of that index's linked list, and its next pointer is directed toward the previous head. Otherwise, if there are no collisions at that index the new node simply becomes the head of that index.

Put

```
//adds a value to the hash table
void HashTable:: put(string input){
    //hash and find the index of input
    int hash = hashFunction(input);

    //create a new node with the given input
    Node* newNode = new Node(input);

    //check if hashed index contains other values (collision)
    if(hashTable[hash] == nullptr){
        //if there is no collision set new node to head
        hashTable[hash] = newNode;
    }
    else{
        //set next pointer to current head and set the head as the new node
        newNode->next = hashTable[hash];
        hashTable[hash] = newNode;
    }
}
```

3.1.2 Get

The get function of a hash table simply returns the value hashed by a given key. When the get function is called the key is hashed to find the target index, walking through its linked list, and returning either the value that matches the given key, or null if the value does not exist. Furthermore, the get function simply returns a value that is matched to its input key and does not remove anything from the hash table.

Get

```
//finds the given value in the hash table and returns the comparison count
int HashTable:: get(string key){
    //count comparisons for each get call
    int comparisons = 1;
    //hash and find the index of the target
    int hash = hashFunction(key);

    //temp node to iterate through linked list
    Node* current = hashTable[hash];

    //check if index is populated
    if(hashTable[hash] == nullptr){
        //throw exception if index is empty
        throw invalid_argument("This value is not in the hash table");
    }
    //if index is populated iterate through nodes
    else{
        //walk down list untl you find the value
        while(current != nullptr){
            comparisons++;
            if(current->val == key){
                break;
            }
            else{
                //if the current node is not the target move to the next
                current = current->next;
            }
        }
    }
    return comparisons;
}
```

3.1.3 hashing

The hash function of a hash table calculates the hash value of a given key to find its index. It does this by summing all of the ASCII values of each letter within the given input and finding its remainder when dividing it by the size of the hash table. This number is the index in which the value will be stored.

Hash Function

```
//hashing function to find the hash code for the given input
int HashTable:: hashFunction(string input){
    //sum of ascii values
    int letterTotal = 0;

    //finds the ascii value of each letter in the input
    for(int i = 0; i < input.length(); i++){
        char currentLetter = input[i];
        int asciiVal = int(currentLetter);
        letterTotal += asciiVal;
    }
    //find the hash code for the input and return it
    return (letterTotal % hashTableSize);
}
```

3.2 Asymptotic Analysis

Given a hash table with chaining, there are two fundamental operations that can be performed: put and get. Both functions can be characterized by their time complexities O(1) and O(1 + α). When it comes to adding or "putting" an element in the hash table the worst-case time complexity would be constant time or O(1), as the new node will become the head of the hashed index regardless of collision. Getting an element from a hash table would have a time complexity of constant time plus alpha, O(1 + α). This is because when receiving or "getting" an element from a hash table if the element is the head of the index it is a constant time operation, however, if the element is further down that index's linked list the time complexity grows at a larger rate. Furthermore, in a hash table, values are not distributed equally meaning, each index's retrieval will differentiate based on its size or number of elements needed to be iterated through. Therefore, the average index's elements that need to be iterated through is represented as the load factor or α.

4 Appendix

4.1 Comparisons and Time Complexity

Linear Search	333.50	$O(n)$
Binary Search	7.95	$O(logn)$
Hashing With Chaining	3.38	$O(1 + \alpha)$

Searching cpp

```
//This file creates the search classes for linear and binary search
#include "Searching.hpp"

#include <vector>
#include <string>

//linear search
float Searching:: linearSearch(vector<string>& magicItems, vector<string>& randomItems){
    //total comparisons
    int totlinComparisons = 0;

    //iterate through both vectors
    for(int i = 0; i < randomItems.size(); i++){
        //comparisons for each magic item
        int lComparisons = 0;
        string current = randomItems[i];
        for(int j = 0 ; j < magicItems.size(); j++){
            //compare each element in magic items to current element in random items
            lComparisons++;
            if(current.compare(magicItems[j]) == 0){
                //if found break loop
                break;
            }
        }
        //add to total
        totlinComparisons += lComparisons;
    }
    //find everage and round
    return (static_cast<float>(totlinComparisons) /
    static_cast<float>(randomItems.size()));
}

//binary search
float Searching:: binarySearch(vector<string>& magicItems, vector<string>& randomItems){
    //keep track of total comparisons
    int totbinComparisons = 0;

    //iterate through each element in random items
    for(int i = 0; i < randomItems.size(); i++){
        //comparisons for each magic item
        //current is the target
        //create start and end index variables
        int bComparisons = 0;
        string target = randomItems[i];
        int start = 0;
        int end = magicItems.size() - 1;

        while(start <= end){
            bComparisons++;
            //find middle index
            int middle = (start + end) / 2;

            //check if target is middle
```

```
            if(magicItems[middle] == target){
                break;
            }
            //if target is greater than middle, ignore left half
            if(magicItems[middle].compare(target) < 0){
                start = middle + 1;

            }
            //if target is less than middle, ignore right half
            else{
                end = middle - 1;
            }
        }
        //add to total
        totbinComparisons += bComparisons;
    }
    //find average and round
    return (static_cast<float>(totbinComparisons) /
    static_cast<float>(randomItems.size()));
}
```

Hashing cpp

```
//this file creates the hashing and chaining functions for a hash table
#include "Hashing.hpp"

#include <vector>
#include <string>
#include <stdexcept>

//hash table constructor
HashTable:: HashTable(){
    this->hashTableSize = 250;

    //set each node to null and allocate memory for hash table
    hashTable.resize(hashTableSize, nullptr);
}

//adds a value to the hash table
void HashTable:: put(string input){
    //hash and find the index of input
    int hash = hashFunction(input);

    //create a new node with the given input
    Node* newNode = new Node(input);

    //check if hashed index contains other values (collision)
    if(hashTable[hash] == nullptr){
        //if there is no collision set new node to head
        hashTable[hash] = newNode;
    }
    else{
        //set next pointer to current head and set the head as the new node
        newNode->next = hashTable[hash];
        hashTable[hash] = newNode;
    }
}

//finds the given value in the hash table and returns the comparison count
int HashTable:: get(string key){
    //count comparisons for each get call
    int comparisons = 1;

```

```
    //hash and find the index of the target
    int hash = hashFunction(key);

    //temp node to iterate through linked list
    Node* current = hashTable[hash];

    //check if index is populated
    if(hashTable[hash] == nullptr){
        //throw exception if index is empty
        throw invalid_argument("This value is not in the hash table");
    }
    //if index is populated iterate through nodes
    else{
        //walk down list untl you find the value
        while(current != nullptr){
            comparisons++;
            if(current->val == key){
                break;
            }
            else{
                //if the current node is not the target move to the next
                current = current->next;
            }
        }
    }
    return comparisons;
}

//hashing function to find the hash code for the given input
int HashTable:: hashFunction(string input){
    //sum of ascii values
    int letterTotal = 0;

    //finds the ascii value of each letter in the input
    for(int i = 0; i < input.length(); i++){
        char currentLetter = input[i];
        int asciiVal = int(currentLetter);
        letterTotal += asciiVal;
    }
    //find the hash code for the input and return it
    return (letterTotal % hashTableSize);
}
```

5 References

5.1 Links

Below are the resources I have used to create simple, readable, and beautiful code.

- This page helped me with float casting when rounding: stackoverflow.com
- This stack post helped me round my answers to two decimal places: stackoverflow.com
- The textbook helped me with basic algorithm and data structure definitions: Algorithms textbook
- Your website helped me form and articulate descriptions for each data structure and algorithm used: Labouseur.com
- This Java source code helped me write the hashing and chaining for my program: Hashing.java
- This video helped me understand hash tables and how they operate: Data Structures: Hash Tables
- A stack overflow post that helped me iterate through a linked list: stackoverflow.com
- Quick lookup to find ASCII values from characters in C++: ASCII from Char
- Used this line of code to set the size and allocate null pointers to a vector: Resize Vector