CMPT 435 - Fall 2023 - Dr. Labouseur

Assignment Two – LaTeX Data

Patrick Tyler
Patrick.Tyler1@marist.edu

October 27, 2023

1 Linear and Binary Searches

1.1 Linear Search

A linear search is one of the most flexible search algorithms. The only requirement of this search is that the collection of elements can be traversed and checked for equality. This algorithm simply goes through each element and checks for equality. This algorithm works in linear time, $O(n)$, because, in the worse case, it checks every element for equality.

The following code implementation loops through an array keeping track of each iteration and returning once the element is found or all items are traversed.

```
39 int getLinearSearchComparisons(string* items, string item, int
       length=DEVILS_NUMBER) {
40     // could just use one variable for count and i
41     //  index plus 1 could be count
42     //  instead of having two variables
43     // using two for clarity
44     int comparisonCount = 0;
45     for(int i=0; i < length; i++){
46         comparisonCount++;
47         if(items[i] == item){
48             break;
49         }
50     }
51     return comparisonCount;
52 }
```

1.2 Binary Search

For a binary search to be applicable, the collection must be sorted (therefore of comparable elements) and random access. An array of sorted elements is the most common data type in which a binary search can be done on. An array is random access because an element can be accessed at any time with its given index.

There's no joke about binary searches that is funny. Not even *a bit.*

This algorithm works with the following process: (1) picking the midpoint of its bounds; (2) check equality between midpoint value and target if equal, element is found at search is complete; (3) check whether midpoint value is greater/ lesser than target then set lower / upper bound to midpoint -1 / +1 and repeat to step 1 if the upper bound does is not equal or lower than the lower bound. How the bounds change also depends on if the array is in ascending or descending order. The following is a code example of binary search on an ascending array.

```
55 int getBinarySearchComparisons(string* items, string item, int
       length=DEVILS_NUMBER) {
56     int comparisonCount = 0;
57     int istart = 0;
58     int iend = length - 1;
59     int imiddle = (istart + iend) / 2;
60     while(istart <= iend) {
61         comparisonCount++;
62         if(items[imiddle] == item) {
63             break;
64         }
65         if(items[imiddle] > item) {
66             // item is to the left of the middle value
67             iend = imiddle - 1;
68         } else {
69             // item is to the right of the middle value
70             istart = imiddle + 1;
71         }
```

2 Hash Table Search

2.1 Overview

For a hash table search to be implemented the data has to be stored in well... a hash table. A hash table maps a value to a key with a hash function. In an array implementation the key would be an index. A hash function should be able to hash an indefinite number of inputs depending only constrained by the data type itself. Ideally this mapping is 1 to 1, but practically mapping an indefinite number of inputs to a key also requires an indefinite amount of

space to store the value at each given key. Thus, there may be collisions where values map to the same key; this will have to be dealt with through storing a collection at each key which can dynamically increase in size such as a linked list. The search process of a hash table is hashing the target value then going to its collection and finding or not finding (usually through a linear search) the target in the chain.

2.2 Absurdity of a Perfect Hash Function

A perfect hash function maps every valid value as a unique key. As an exercise, let's see how many keys are needed to create a perfect hash function for 8 bit ascii strings. Each ascii character has 2^8 or 256 different possibilities. So, if the hash function only hashes a single ascii character it will need 256 different keys. If the valid input is extended each additional character will multiply the number of different keys needed by 256 because there will be an additional 256 valid strings for each already existing combination. To be able to hash just 16 character strings there will need to be 256^{16} or 2^{128} keys. Each key will need 1 bit to store whether it exists in the hash table (it will not need to store the string itself because it is unique). In total, this hash table will be 2^{128} bits. This is about $4 * 10^{16}$ *zettabytes*. The amount of storage – *not even working memory* – in all of the world's computers is a rounding error compared to this number. So it is safe to say: prefect hashes are impractical for hash tables.

Guy: *Hey girl, are you are perfect hashing function?*
Girl: *What?*
Guy: *Because you're impossible to find and entirely unique.*
Girl: *...*

2.3 Sample Hash Function

The following is an example of an imperfect hash that has hashSize different keys (250 for later examples). This maps any string into an integer from 0 to hashSize-1. It does this through summing the ascii values (treating all alphabetical characters as uppercase) and multiplying by a large prime and the finding the reminder after dividing by the hashSize. This modulo operation ensures any arbitrarily large product is converted to an integer in the correct bounds. Multiplication by a large prime is done to spread the results across the keys. This is a common technique as many hashing methods have trends such that if the input values also have trends it can result in clustered keys.

```
143         int hash(string value){
144             for (char& c : value) {
145                 // independent of casing
146                 c = std::toupper(c);
147             }
148             // ascii sum of characters
149             int asciiTotal = 0;
150             for(char c: value){
151                 asciiTotal += toascii(c);
152             }
153 
154             // multiply the asciiTotal by a large prime number to
155             //  spread the ascii totals
156             return (asciiTotal * hashPrime) % hashSize;
157         }
```

2.4 Sample Add Implementation

A hash table can be implemented with an array which is the same length as the amount of keys. In this implementation both an array for the heads of the linked list and the tails is created to ensure constant time $O(1)$ adding as well as maintaining relative insertion order even if the chain of the a specific key grows indefinitely large. Other implementations of adding to a hash table may have a factor of the chain length of each index; a possible advantage could be ensuring each value is only stored once. This factor is often called the load factor or α; it can be calculated by dividing the amount of elements added to the hash table by the amount of keys. In the this case adding would be $O(1 + \alpha)$.

The following code adds to the linked list at the tail much like an enqueue from a queue.

```
123 class HashTable {
124     private:
125         // Pointer to array of Nodes
126         Node<string>** heads;
127         Node<string>** tails;
128         int hashSize;
129         int hashPrime;
```

```
159         void add(string value){
160             // create new node
161             auto newNode = new Node<string>;
162             newNode->data = value;
163             newNode->next = nullptr;
164 
165             // hash node value to find which tail to add it to
166             int hashValue = hash(value);
167             Node<string>*& tail = tails[hashValue];
168             if(tail == nullptr){
169                 // case this hash value has no elements in it
170                 Node<string>*& head = heads[hashValue];
171                 head = newNode;
172                 tail = newNode;
173             } else{
174                 tail->next = newNode;
175                 tail = newNode;
176             }
177         }
178 
179         int getComparisons(string value){
180             int hashValue = hash(value);
181 
182             int comparisonCount = 1;
183             Node<string>*& traversingNode = heads[hashValue];
184             // although i'm not counting this comparison for
    elements that are here
185             //  it feels weird if it were to return 0 even though
    the program checked head
186             if(traversingNode == nullptr) { return comparisonCount;
     }
187 
188             while((traversingNode->data != value)){
189                 traversingNode = traversingNode->next;
190                 if(traversingNode == nullptr) { break; }
191                 comparisonCount++;
192             }
193 
194             return comparisonCount;
195         }
```

3 Analysis of Searches

Linear Search $O(n)$

Comparisons	Name
575	Sword of the Kauhns
65	Blood Spear +2
234	Fire Stones
370	Manual of quickness of action +3
465	Ring of Telepathic Bonds
224	Feather token, fan
423	Pipes of haunting
380	Metal-morphic Hammer
233	Fire Javelin
162	Cup of Change
96	Bracers of armor +6
66	Bloodstone Ring
294	Helm
316	Hunter's Bow
253	Gems of Darkness
570	Sword of Life
187	Dream Weaver
31	Armatha's long sword
395	Necklace of fireballs type III
435	Potion of Liquefaction
620	Troll Jar
581	Teleport Ribbon
442	Quicksilver Amulet
421	Phylactery of faithfulness
177	Dior Droid
101	Breast plate of the champion
225	Feather token, swan boat
364	Manual of gainful exercise +2
326	Ioun stone, orange
602	Tome of leadership and influence +3
569	Sword of Kings
40	Backhand
14	Amulet of mighty fists +2
94	Bracers of armor +2
237	Flail of Armor Disruption
606	Tome of understanding +2
116	Casters Aid
413	Pearl of power, 9th-level spell
300	Helm of underwater action
213	Exploding Caltrops
58	Belt of Keeping
654	Wings of flying
Average $\approx$ 308.26	

Binary Search $O(log_2 n)$

Comparisons	Name
9	Sword of the Kauhns
9	Blood Spear +2
9	Fire Stones
10	Manual of quickness of action +3
8	Ring of Telepathic Bonds
10	Feather token, fan
8	Pipes of haunting
9	Metal-morphic Hammer
7	Fire Javelin
10	Cup of Change
9	Bracers of armor +6
10	Bloodstone Ring
9	Helm
10	Hunter's Bow
10	Gems of Darkness
9	Sword of Life
9	Dream Weaver
9	Armatha's long sword
5	Necklace of fireballs type III
9	Potion of Liquefaction
9	Troll Jar
9	Teleport Ribbon
7	Quicksilver Amulet
7	Phylactery of faithfulness
9	Dior Droid
9	Breast plate of the champion
8	Feather token, swan boat
9	Manual of gainful exercise +2
10	Ioun stone, orange
9	Tome of leadership and influence +3
8	Sword of Kings
10	Backhand
10	Amulet of mighty fists +2
9	Bracers of armor +2
10	Flail of Armor Disruption
9	Wings of flying
8	Tome of understanding +2
9	Casters Aid
8	Pearl of power, 9th-level spell
10	Helm of underwater action
9	Exploding Caltrops
10	Belt of Keeping
Average = 8.90	

Hash Table Search $O(1+\alpha)$

Comparisons	Name
3	**Sword of the Kauhns**
1	**Blood Spear +2**
3	**Fire Stones**
3	**Manual of quickness of action +3**
2	**Ring of Telepathic Bonds**
1	**Feather token, fan**
6	**Pipes of haunting**
1	**Metal-morphic Hammer**
3	**Fire Javelin**
3	**Cup of Change**
1	**Bracers of armor +6**
2	**Bloodstone Ring**
2	**Helm**
3	**Hunter's Bow**
1	**Gems of Darkness**
2	**Sword of Life**
3	**Dream Weaver**
1	**Armatha's long sword**
3	**Necklace of fireballs type III**
3	**Potion of Liquefaction**
4	**Troll Jar**
5	**Teleport Ribbon**
2	**Quicksilver Amulet**
2	**Phylactery of faithfulness**
2	**Dior Droid**
1	**Breast plate of the champion**
3	**Feather token, swan boat**
5	**Manual of gainful exercise +2**
1	**Ioun stone, orange**
2	**Tome of leadership and influence +3**
2	**Sword of Kings**
1	**Backhand**
1	**Amulet of mighty fists +2**
1	**Bracers of armor +2**
1	**Flail of Armor Disruption**
2	**Tome of understanding +2**
1	**Casters Aid**
3	**Pearl of power, 9th-level spell**
2	**Helm of underwater action**
1	**Exploding Caltrops**
1	**Belt of Keeping**
2	**Wings of flying**
Average $\approx$ 2.19	

The comparison average for each of the searches agrees with its respective time complexity for input size n of 666. For linear search's complexity, $O(n)$, implies the comparisons will grow linearly with the input size. This is true as the average is 308.26 or about half of the 666 length input. Binary search's complexity, $O(log_2 n)$, implies the comparisons will grow logarithmically with the input size. This is true as the average is 8.90 which is a little less than $\lceil log_2 666 \rceil$. The ceiling is taken because the comparison start counting after the first equality check of midpoint; if the comparisons were counted only for greater or greater than or less than comparisons, the rare case in which the target is found on the first equality check, it would count 0 comparisons which seems nonsensical. Regardless of how comparisons, the trend will still be logarithmic. Finally, the hash table's complexity, $O(1 + \alpha)$, implies the comparisons will grow with the load factor. The load factor for an input size of 666 and a key amount of 250 is $666/250 \approx 2.66$. The load factor and average comparisons are about the same which is expected.

Each search method has advantages and drawbacks: linear search has low overhead and a lot of flexibility with the data structure the collection is stored in; binary search only works on sorted, random access collections, but the overhead is low and the comparisons scale logarithmically; hash tables have relatively more overhead (especially if operations such as iteration are needed), but they can significantly reduce the amount of comparisons by the load factor.

Girl: *I think we need to take some space from each other.*
Guy: *Great! Then we'll spend more time together.*
Girl: *What?*
Guy: *It's a classic tradeoff: space versus time. "Taking space" will allow us to spend more time together.*
** 10 second pause **
Girl: *I've been cheating on you for 5 years.*

---

4 Display Listings

Javascript developers frontend developers seem to want a framework or library for everything. Here is an example of a very small package for padding many javascript developers used as a dependency and it ended up breaking their website: `https://qz.com/646467/how-one-programmer-broke-the-internet-by-deleting-a-tiny-piece-of-code.`

```
80 string rightPad(int num, int size){
81     string padded = to_string(num);
82     int curLength = padded.length();
83     while(curLength < size){
84         padded += " ";
85         curLength++;
86     }
87     return padded;
88 }
```

What is the difference between a javascript developer and an Amazon warehouse employee?
Nothing they're both just package managers.

The rest of the display code is self-documenting and can be found in the online resources then assignment 2 then main.cpp.