Assignment Three

Josh Seligman

joshua.seligman1@marist.edu

October 22, 2022

1 Linear Search

1.1 The Algorithm

Linear search is a searching algorithm that walks through an array and continues on until either it finds the target element or reaches the end of the array. As shown in Algorithm 1, the function has to compare the target value with every element in the array until the condition in the while loop becomes false. Since the entire array is being searched, no assumptions have to be made about the initial status of the array, which means that the array does not have to be sorted or in any particular order prior to running the search.

Algorithm 1 Linear Search Algorithm

```
1: procedure LinearSearch(arr, target)
2:     i ← 0    // Start at the beginning of the array
3:     while i < len(arr) && arr[i] != target do    // Search through the entire array or until the target is found
4:         i++
5:     end while
6:     if i == len(arr) then
7:         i = −1    // Set i to -1 to note that the target is not in the array
8:     end if
9:     return i
10: end procedure
```

1.2 Asymptotic Analysis and Comparisons

As mentioned in Section 1.1, performing a linear search requires going through each element until the target is found or until the end of the array is reached. The performance of linear search varies drastically on the position of the target element relative to the front of the array. For instance, linear search will be really fast when the target is near the front as the algorithm terminates upon finding the element. However, if the target is near the end of the array, the algorithm will have to iterate through nearly the entire array. Thus, for a randomly sorted list, the average expected case will be $\frac{n}{2}$ iterations. Also, at the worst case, the number of

iterations will be equal to the number of elements in the array. Therefore, as its name implies, linear search runs in linear time $O(n)$. Table 3.1 shows the average number of comparisons made for 42 of the 666 magic items, which for the case of linear search, was 341.67. This lines up with the expected case for the searching algorithm of $\frac{n}{2}$ as, for a list of 666 items, about 333 will have to be searched and compared with before the target value is found.

2 Binary Search

2.1 The Algorithm

Binary search is a searching algorithm that takes an already sorted list and progressively cuts it in half until there is only 1 element left, which is the one that is being searched for. As shown in Algorithm 2, each recursive call on lines 8 and 10 makes the problem smaller by moving the start or stop limits to a single side of the midpoint, which effectively cuts the array in half at each level of the recursion tree. Also, Figure 2.1 illustrates how binary search divides the array in half to eventually find the target element.

Algorithm 2 Binary Search Algorithm

```
1:  procedure BinarySearch(arr, target, start, stop)
2:      out ← −1    // Assume the element is not found, by setting the default output to -1
3:      if start <= stop then    // Working in a valid range
4:          mid ← ⌊(start+stop)/2⌋    // Get the middle of the range
5:          if target == arr[mid] then
6:              out ← mid    // Target found at position mid
7:          else if target < arr[mid] then    // Target is in bottom half of the array
8:              out ← BinarySearch(arr, target, start, mid − 1)    // Do binary search on lower half of array
9:          else    // Target is in top half of the array
10:             out ← BinarySearch(arr, target, mid + 1, stop)    // Do binary search on top half of array
11:         end if
12:     end if
13:     return out
14: end procedure
```

0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

Figure 2.1: A visualization of the binary search algorithm. The blue shaded areas are the parts of the array being considered at each step of the recursion tree until there is only the target element left.

2.2 Asymptotic Analysis and Comparisons

Binary search, similar to merge sort and quicksort, divides the array in half at each step of the recursion tree. This makes the recurrence relation of binary search to be $T(n) = T(\frac{n}{2}) + C$, where $T(\frac{n}{2})$ is the time to perform binary search on the half of the array and C is the constant time to perform the comparisons. Thus, since the array is being divided in half until the size of the array being considered is 1, the number of times binary search will be recursively called at worst case will be $log_2 n$ times, which makes it run in $O(log_2 n)$ time.

$O(log_2 n)$ is a huge improvement over $O(n)$ for linear search. However, the one tradeoff is that binary search needs to have an already sorted array. Thus, if the array is not sorted, an additional $n * log_2 n$ time will need

to be added to put the array in a state to use binary search. Therefore, linear search may be the better option for one and done search operations, but binary search with a sort will catch up as the number of search operations increases. The mathematical equation to determine this point is $x * n > x * log_2 n + n * log_2 n$, where x is the number of searching operations that have to be done on the unsorted array. For an array of 666 sorted items, binary search required an avereage of 8.57 comparisons to retrieve 42 random items, which is around $n * log_2 n = 9.37$, or the number of levels in the recursion tree.

2.3 Hashing with Chaining

A hash table is a data structure that stores data in an array and uses a hash function to determine the position in the array to store the data. Hash tables have 2 main functions: get and put, and both of these are provided in Algorithms 3 and 4, respectively. Since an array has a fixed size, there are 2 initial problems with hash tables. The first is that the output of a hash function may not be directly translatable to an array index. As displayed on line 19 of Listing 3, once the sum of the ordinal codes is calculated, modulus division is performed to get the hash code to be on the range $[0, size - 1]$, which are all valid indices within the array. The other issue is caused when 2 different elements return the same hash code, which is called a collision. A hash table may implement a method called probing by searching for an empty space within the array to put the value, but is restricted by the size of the created array. Chaining is the other way to resolve this problem of duplicate hash codes. Rather than searching for an empty spot in the array, chaining stares a linked list at each element in the array, and the value that is being put into the hash table can be added to the linked list at the index generated by the hash function. Additionally, unlike probing, chaining is not limited to the size of the array being used as a linked list is only limited by the size of memory. See Figure 2.2 for a visual understanding of hashing with chaining.

Algorithm 3 Hash Table Get Function

```
1:  procedure GET(value)
2:      hashCode ← hash(value)    // Generate the hash code for the value
3:      cur ← table[hashCode]    // Start iterating at the head of the linked list at the index the hash function
    returned
4:      found = false    // Assume the vaule is not found
5:      while !found && cur != null do
6:          if cur.data == value then
7:              found = true    // The value was found
8:          end if
9:          cur = cur.next    // Move to the next element in the linked list
10:     end while
11:     return found
12: end procedure
```

Algorithm 4 Hash Table Put Function

```
1:  procedure PUT(value)
2:      hashCode ← hash(value)    // Generate the hash code for the value
3:      curHead ← table[hashCode]    // Going to work with the head of the list
4:      newHead = newNode(value)    // Create the new node
5:      if curHead != null then
6:          newHead.next = curHead    // Place the new node ahead of the current head
7:      end if
8:      table[hashCode] = newHead    // Update the table entry to point to the new head
9:  end procedure
```

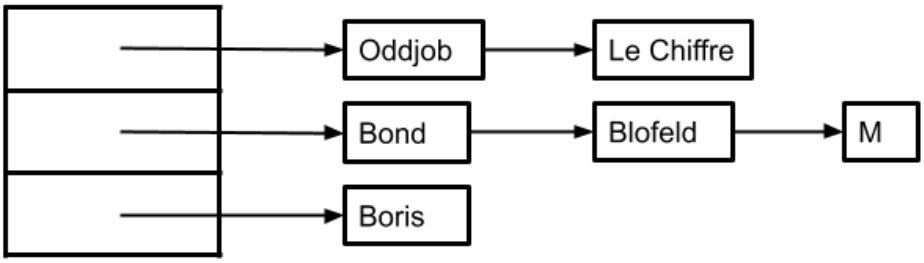

Figure 2.2: Example hash table of James Bond characters that uses chaining. The hash function used is the number of characters (excluding spaces) mod the size of the array.

2.4 Asymptotic Analysis and Comparisons

The functions of a hash table are extremely efficient and do not completely rely on the number of elements stored in the hash table. First, the get function for a hash table computes the hash code, which is just a mathematical formula, and then accesses the element based on the output of the hash function. However, depending on the method being used for collisions, the time to access the element varies. This variation is called the load factor and can be noted with α. In the case of chaining, as described in Algorithm 3 and demonstrated in Listing 4 on lines 16-29, the linked list stored at the index of the array has to be iterated over. Assuming the hash function produces evenly distributed hash codes, the number of elements that have to be iterated through will on average be $\frac{n}{size\ of\ array}$, which is the value for α. Therefore, the runtime for the hash table get function is $O(1) + \alpha$. Table 3.1 also highlights the average number of comparisons needed for performing the hash table get function. For a random selection of 42 of the 666 magic items, it took an average of 3.19 comparisons to retrieve the item, which is a drastic improvement over both binary search and linear search.

The C++ implementation for the put function for hash tables is shown in Listing 5 and is a bit simpler compared to the get function. After computing the hash code on line 3, the put function creates a new node and inserts it at the front of the list. Just like a stack, since the front of the list is being used, there is no need to traverse through the entire list. Therefore, the put function runs in constant time $O(1)$.

3 Appendix

3.1 Comparisons Table

Search Name	Comparisons
Linear Search	341.67
Binary Search	8.57
Hashing	3.19

Table 3.1: A table of the number of comparisons made and time to get an item from an array or hash table.

3.2 Linear Search

```
1 int linearSearch(StringArr* data, std::string target, int* comparisons) {
2     // Start with the first element in the array
3     int i = 0;
```

```
4
5       // Iterate through the array, looking for the target
6       while (i < data->length && data->arr[i].compare(target) != 0) {
7           if (comparisons != nullptr) {
8               // Increment the number of comparisons
9               (*comparisons)++;
10          }
11          i++;
12      }
13
14      // Default to -1 as the output if the target is not in the array
15      int out = -1;
16
17      // Add a comparison and set the index because we found the element
18      if (i < data->length) {
19          (*comparisons)++;
20          out = i;
21      }
22
23      // Return the position of the target element
24      return out;
25  }
```

Listing 1: Linear Search (C++)

3.3 Binary Search

```
1   int binarySearch(StringArr* data, std::string target, int* comparisons) {
2       // Call the helper function to perform the binary search, starting off with the entire
            array
3       return binarySearchHelper(data, target, 0, data->length - 1, comparisons);
4   }
5
6   int binarySearchHelper(StringArr* data, std::string target, int start, int stop, int*
        comparisons) {
7       // Assume nothing is found, which initializes it to 0
8       int out = -1;
9
10      // Only search if start <= stop. Otherwise, the element doesn't exist ond out is already
             set to -1
11      if (start <= stop) {
12          // Do integer division to get the midpoint of the array we are considering
13          int mid = (start + stop) / 2;
14
15          // Make the comparison once up top
16          int compStr = target.compare(data->arr[mid]);
17          // Increment the number of comparisons
18          if (comparisons != nullptr) {
19              (*comparisons)++;
20          }
21
22          if (compStr == 0) {
23              // The index was found, which is the midpoint
24              out = mid;
25          } else if (compStr < 0) { // The target is in the first half of the array
26              // Run the binary search on the first half of the array
27              out = binarySearchHelper(data, target, start, mid - 1, comparisons);
28          } else { // The target is in the second half of the array
29              // Run the binary search on the second half of the array
30              out = binarySearchHelper(data, target, mid + 1, stop, comparisons);
31          }
32      }
33
34      // Return the index of the element
35      return out;
```

```
}
```

Listing 2: Binary Search (C++)

3.4 Hashing with Chaining

```
    // Keep a runnning sum of the ordinal codes seen so far
    int letterTotal = 0;

    // Iterate through each character in the word
    for (int i = 0; i < value.length(); i++) {
        char character = value[i];
        if (character >= 'a' && character <= 'z') {
            // Adjust the character to make it uppercase by taking the difference between
            // the start of the lowercase letters and the start of the uppercase letters
            character -= 'a' - 'A';
        }

        // Add the ordinal value to the running sum
        letterTotal += character;
    }

    // Hash code is the sum of the ordinal values mod the size of the array
    int hashCode = letterTotal % size;
    return hashCode;
}
```

Listing 3: Simple Hash Function

```
bool HashTable::get(std::string value, int* comparisons) {
    // Increment the number of comparisons off the bat
    if (comparisons != nullptr) {
        (*comparisons)++;
    }

    // Compute the hash function
    int index = hash(value);

    // Assume the value isn't in the table
    bool found = false;

    // Start at the head of the list
    Node<std::string>* cur = table[index];

    while (!found && cur != nullptr) {
        // Determine if the value was found
        if (cur->data.compare(value) == 0) {
            found = true;
        } else {
            // Move to the next element
            cur = cur->next;
        }

        // A comparison was made
        if (comparisons != nullptr) {
            (*comparisons)++;
        }
    }
    return found;
}
```

Listing 4: Hash Table Get with Chaining

```
void HashTable::put(std::string value) {
```

```
    // Generate the hash
    int index = hash(value);

    // Create a new node for the table
    Node<std::string>* newNode = new Node<std::string>(value);

    // If the table entry is not empty, set the next pointer of the new node
    if (table[index] != nullptr) {
        newNode->next = table[index];
    }

    // Place the new node at the head of the list
    table[index] = newNode;
}
```

Listing 5: Hash Table Put with Chaining