CMPT 435 - Fall 2024 - Dr. Labouseur

Assignment Two – Searching & Hashing

Ryan Munger
Ryan.Munger1@marist.edu

October 7, 2024

1 Introduction

1.1 Goals

Assignment 2 instructed us to implement sequential/linear search, binary search, and a hash table. We took a random sample of magic items from the full list to demonstrate searching and hash look-ups.

1.2 Write-up Format

In this report I will describe the logic being presented. Below the text explanation, relevant code will follow in both C++ and Ada.

1.3 Limerick of Luck

After years of searching
Interviewees found an idol perching.
    The ultimate guess for any question,
    Answering ”Hash Table” is rarely a transgression,
Bolstering job-seekers with no besmirching.

Why did they close Hash Street? - *It had too many collisions...*

---

2 Searching

2.1 Sequential/Linear Search

Selection/Linear search is a very simple algorithm. We simply iterate through the array looking for the target. As soon as we find it, we return the index we found it at. If we don't find it, we can return -1 to indicate this. This algorithm is $O(n)$ time as we need to iterate through the length of the array to find our target. On average, it will take us $\frac{n}{2}$ comparisons since sometimes we will find our target early and sometimes late. So, for 666 magic items, it should take us an average of 333 comparisons.

```cpp
template <typename T>
int sequentialSearch(const vector<T>& arr, const T& target) {
    for(size_t i = 0; i < arr.size(); ++i) {
        if(arr[i] == target){return static_cast<int>(i);}
    }
    return -1;
}
```

Notice I did not count comparisons directly in this function. This is because since we are linearly searching the array item by item, the number of comparisons is exactly the index we found the item at + 1 (Since indexing starts at 0 - we need 1 comparison even if it is element 0). If we did not find it, then the amount of comparisons is the length of the array!

2.2 Binary Search

Binary search is much more efficient than sequential search. For this algorithm, the trade-off is that the array must be sorted. To find the target, we use the fact that the array is sorted to our advantage. We know that a splice of the array after a certain value contains only values greater than it and one before a certain value only contains values less than it. In each iteration of binary search, the search space is halved, resulting in a logarithmic reduction in the number of comparisons. We check half of the half of the half... This leaves binary search at $0(log(n))$. For our array of 666 magic items, we can expect it to take $\log_2(666)$ comparisons, or 9.38. It is base 2 as we are halving.

1. Find the midpoint of the array.
2. If the midpoint is the target, return the midpoint index.
3. If the target is greater than the midpoint, search the half after the midpoint.
4. If the target is less than the midpoint, search the half before the midpoint.
5. Continue calculating midpoints on the smaller halves until we find the element, or our left and right indexes hit each other.

```cpp
template <typename T>
int binarySearch(const std::vector<T>& arr, const T& target, int&
    comparisons) {
    int left = 0;
    int right = arr.size() - 1;

    // if left > right we did not find it
    while (left <= right) {
        int mid = left + (right - left) / 2;

        comparisons++;
        if (arr[mid] == target) {
            return mid;
        }
        else if (arr[mid] < target) {
            left = mid + 1; // search right half
        }
        else {
            right = mid - 1; // search left half
        }
    }
    return -1;
}
```

I considered implementing **EVIL** Binary Search like we saw in class - *"Yeah man, its in the array..."*

2.3 Searches in Action

Random Sample
To test our searching algorithms, we are taking a sample of 42 unique items from the list of Magic Items (length 666). I re-used code from assignment1 to read in the items and sort the array (using my merge sort if you were curious). I then used the c++ sampler to create my 42 item subarray. This was tricky to do, but after seeing how concise this function was (available in c++ 17 or later) on stack overflow, I had to understand and implement it. Interestingly, even though it samples the elements uniquely & randomly, it places them into the new subarray in the relative order they originally were in! Another good way to take the sample would be to shuffle the array and take the first 42 elements, but shuffling the entire array for that is not as efficient.

```cpp

    /* taken from stack overflow. Used ChatGPT to find out I needed
     c++ version 17.
    sample() takes range of elements and number to select, selects
    randomly without repetition.
    selected elements are added to the randomSample container with
    back_inserter().
    Rand generator (mt19937) ensures sample is random. After some
    ChatGPT and google, I found out that
    the reason the new selection is sorted is because sample()
    maintains the relative ordering of the elements it selected. */
```

```
    vector<string> randomSample;
    const int sample_size = 42;
    sample(magicItems.begin(), magicItems.end(), back_inserter(randomSample), sample_size, mt19937{random_device{}()});
```

Testing the Searches

Using the sample, we found each of the 42 elements in the original array of 666 items. I then took an average of the trials (to two decimal places).
Since the sample is in order as well, the amount of sequential comparisons we need to do increases with each element! **Code:**

```
    // since the random sample is in order (relative to the sorted array), as we progress through the sample, comparisons will always increase!
    int totalComparisons = 0;
    int foundIdx;
    cout << "\nSequential/Linear Search:\n" << endl;
    for (string item : randomSample) {
        foundIdx = sequentialSearch(magicItems, item);
        if(foundIdx != -1){
            cout << "\"" << item << "\" was found in magicItems at index: " << foundIdx
                << ". It took " << foundIdx + 1 << " Comparisons." << endl;
        } else {
            cout << "\"" << item << "\" was not found in magicItems. Comparisons: " << magicItems.size() << endl;
            foundIdx = magicItems.size() - 1; // since we are adding one later
        }
        totalComparisons += foundIdx + 1;
    }
    cout << "\nSequential/Linear search took an average of "
        << fixed << setprecision(2) // Set fixed-point notation and precision
        << static_cast<double>(totalComparisons) / randomSample.size() // cast double so we don't lose our decimal accuracy
        << " comparisons to find each element." << endl;

    int comparisons = 0;
    totalComparisons = 0;
    cout << "\n\nBinary Search:\n" << endl;
    for (string item : randomSample) {
        foundIdx = binarySearch(magicItems, item, comparisons);
        if(foundIdx != -1){
            cout << "\"" << item << "\" was found in magicItems at index: " << foundIdx
                << ". It took " << comparisons << " Comparisons." << endl;
        } else {
            cout << "\"" << item << "\" was not found in magicItems. Comparisons: " << comparisons << endl;
        }
        totalComparisons += comparisons;
        comparisons = 0;
    }
```

```cpp
    cout << "\nBinary search took an average of "
        << fixed << setprecision(2)
        << static_cast<double>(totalComparisons) / randomSample.size()
        << " comparisons to find each element." << endl;
```

Output:

```
Sequential/Linear Search:

"Amulet of mighty fists +3" was found in magicItems at index: 13. It took 14 Comparisons.
"Amulet of mighty fists +4" was found in magicItems at index: 14. It took 15 Comparisons.
"Amulet of Proof Against Turning" was found in magicItems at index: 20. It took 21 Comparisons.
"Apparatus of the crab" was found in magicItems at index: 26. It took 27 Comparisons.

...

"Universal solvent" was found in magicItems at index: 632. It took 633 Comparisons.
"Whisper Blade" was found in magicItems at index: 654. It took 655 Comparisons.

Sequential/Linear search took an average of 362.71 comparisons to find each element.


Binary Search:

"Amulet of mighty fists +3" was found in magicItems at index: 13. It took 10 Comparisons.
"Amulet of mighty fists +4" was found in magicItems at index: 14. It took 7 Comparisons.
"Amulet of Proof Against Turning" was found in magicItems at index: 20. It took 9 Comparisons.
"Apparatus of the crab" was found in magicItems at index: 26. It took 8 Comparisons.

...

"Universal solvent" was found in magicItems at index: 632. It took 9 Comparisons.
"Whisper Blade" was found in magicItems at index: 654. It took 10 Comparisons.

Binary search took an average of 8.69 comparisons to find each element.
```

Even though the code gave us the average of the 42, I went ahead and ran it 20 times, averaging the averages.

Search	Time Complexity	Expected	Actual
Sequential Search	$O(n)$	333	338.13
Binary Search	$O(log(n))$	9.38	8.77

Our searching algorithms are performing just as we expected!

3 Hashing