Search Algorithm Analysis

Ryan Davis

Ryan.Davis3@Marist.edu

November 2, 2024

This document presents an analysis of linear, binary, and hash table search algorithms implemented in Java. Each algorithm is implemented with detailed explanations provided, focusing on the underlying logic and performance implications.

Contents

1 Introduction

This document details implementations and analyses of three fundamental search algorithms: linear search, binary search, and hash table search with chaining. We focus on measuring the number of comparisons required by each algorithm, analyzing the asymptotic running times, and comparing their performance across different test cases.

2 Binary and Linear Search Listings and Explanations

In this section, we provide code listings and detailed explanations for linear, binary, and hash table search algorithms, highlighting key logic and performance characteristics.

2.1 Linear Search

```java
1 public static int linearSearch(List<String> arr, String searchItem, int[] comparisonCount) {
2     for (int i = 0; i < arr.size(); i++) {
3         comparisonCount[0]++;
4         if (arr.get(i).equals(searchItem)) {
5             return i;
6         }
7     }
8     return -1;
9 }
```

Listing 1: Linear Search Implementation

Explanation:

- **Line 2**: The method iterates through each element in the array from start to end.
- **Line 3**: For each element, it increments the comparison counter.
- **Line 4**: If the current element matches the search item, the method returns the index of the found item.
- **Line 6**: If no match is found after iterating through the array, the method returns `-1`.

This method sequentially checks each element, leading to an average time complexity of $\mathcal{O}(n)$.

2.2 Binary Search

```java
public static int binarySearch(List<String> arr, String searchItem, int[] comparisonCount) {
    int left = 0;
    int right = arr.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        comparisonCount[0]++;
        if (arr.get(mid).equals(searchItem)) {
            return mid;
        }
        if (arr.get(mid).compareTo(searchItem) < 0) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}
```

Listing 2: Binary Search Implementation

Explanation:

- **Lines 2-3**: Initializes pointers for the left and right bounds of the search interval.
- **Line 5**: Calculates the midpoint of the current interval to divide the search space.
- **Line 6**: Increments the comparison counter for each iteration of the loop.
- **Line 7**: If the midpoint element matches the search item, the index is returned.
- **Line 9**: If the midpoint element is less than the search item, the left pointer is adjusted to search the right half.
- **Line 11**: Otherwise, the right pointer is adjusted to search the left half.

Binary search achieves logarithmic time complexity, $\mathcal{O}(\log n)$, by halving the search space on each comparison. This effectively means each comparison rules out half of the possible data.

3 Introduction to Hashing

Hashing is a technique for mapping data to a fixed-size table (hash table) using a hash function. In essence, hashing transforms data (such as a string) into a unique numeric representation, called a hash code, which is used as the index for storing the data in a table. The goal is to achieve fast data retrieval by leveraging these hash codes.

3.1 How Hashing Works

Hash functions convert input data to hash codes by performing mathematical operations. Ideally, a good hash function distributes data evenly across the hash table to minimize collisions, where two inputs yield the same hash code. Collisions are resolved by techniques such as:

- **Chaining:** Each bucket in the hash table contains a linked list of entries. If multiple items hash to the same index, they are stored in a chain within that bucket.
- **Open Addressing:** When a collision occurs, the hash table searches for an empty slot based on a defined probe sequence.

In this assignment, we use chaining to handle collisions, and a custom hash function called ‘makeHashCode‘ to generate hash codes.

3.2 Hash Table Creation and Item Storage

In this section, we analyze the construction of our hash table and the specific choices made in defining each part of the hash function.

```java
private static final int HASH_TABLE_SIZE = 250;
private static List<List<String>> hashTable = new ArrayList<>(HASH_TABLE_SIZE);

static {
    for (int i = 0; i < HASH_TABLE_SIZE; i++) {
        hashTable.add(new LinkedList<>());
    }
}

public static void loadHashTable(List<String> items) {
    for (String item : items) {
        int hashCode = makeHashCode(item);
        hashTable.get(hashCode).add(item);
    }
}
```

Listing 3: Hash Table Creation and Item Insertion

Explanation:

- **Lines 1-2**: The hash table is initialized as an ‘ArrayList‘ containing 250 ‘LinkedList‘ buckets to store items with potential collisions.
- **Lines 4-6**: Each bucket is instantiated as an empty ‘LinkedList‘ to handle collisions through chaining.
- **Lines 8-12**: The ‘loadHashTable‘ method takes a list of items and assigns each item to a bucket based on its hash code.
- **Line 10**: The ‘makeHashCode‘ method generates a hash code for each item, ensuring it falls within the range of available buckets.
- **Line 11**: Each item is added to the corresponding bucket using chaining to handle any collisions that might occur.

This setup provides efficient data retrieval by distributing items across buckets, with chaining to resolve hash collisions.

3.3 Hash Code Generation - makeHashCode Function

```
1 private static int makeHashCode(String str) {
2     str = str.toUpperCase();
3     int letterTotal = 0;
4     for (int i = 0; i < str.length(); i++) {
5         letterTotal += str.charAt(i);
6     }
7     return letterTotal % HASH_TABLE_SIZE;
8 }
```

Listing 4: Hash Code Generation Function

Explanation:

- **Line 2**: Converts the input string to uppercase to ensure case insensitivity, so that, for example, "Bond" and bond" will produce the same hash code.
- **Line 3**: Initializes ‘letterTotal‘ to zero, which will store the cumulative ASCII values of the characters in the string.
- **Line 4**: Iterates over each character in the string:
- **Line 5*: Adds the ASCII value of each character to ‘letterTotal‘, effectively creating a unique numerical signature based on the sum of character codes.
- **Line 6**: Returns the computed hash code by taking `letterTotal` modulo `HASH_TABLE_SIZE`. This ensures that the hash code falls within the valid range of indices for the hash table.

The ‘makeHashCode‘ function is designed to distribute items evenly across the hash table by mapping each string to a unique integer based on its ASCII values. The modulo operation limits the hash code to fit within the predefined table size, helping to minimize collisions and improve lookup efficiency.

3.4 Retrieve from Hash Table

```
public static int retrieveFromHashTable(String item) {
    int hashCode = makeHashCode(item);
    List<String> bucket = hashTable.get(hashCode);
```

```
 4     int comparisons = 0;
 5
 6     for (String element : bucket) {
 7         comparisons++;
 8         if (element.equals(item)) {
 9             break;
10         }
11     }
12
13     return comparisons;
14 }
```

Listing 5: Hash Table Retrieval with Chaining

Explanation:

- **Line 2**: Calculates the hash code for the item to determine its bucket.
- **Line 3**: Retrieves the appropriate bucket for the item using the hash code.
- **Line 4**: Initializes the comparison counter for counting comparisons within the bucket.
- **Lines 6-9**: Iterates through each element in the bucket:
  - **Line 7**: Increments the comparison counter for each element checked.
  - **Line 8**: Breaks out of the loop if the item is found.
- **Line 11**: Returns the total number of comparisons needed to find the item within the bucket.

This method leverages chaining within the hash table, where each bucket stores a list of elements. The retrieval operation iterates through the bucket only when collisions occur, for each collison the time complexity decays further away from $\mathcal{O}(1)$ complexity.

4 Results

The following table summarizes the average number of comparisons for linear, binary, and hash table searches across multiple test runs on a set of 42 specific items

Table 4.1: Average Comparisons for Linear, Binary, and Hash Table Searches

Run	**Linear Search**	**Binary Search**	**Hash Table Search**
1	319.31	8.33	2.57
2	369.83	8.60	2.38
3	352.36	8.40	2.57
Average	347.17	8.44	2.51

Observations:

- Linear search required significantly more comparisons, consistent with its $\mathcal{O}(n)$ complexity.
- Binary search performed fewer comparisons, aligning with its $\mathcal{O}(\log n)$ complexity.
- Hash table search consistently had the lowest comparison count, reflecting its average-case $\mathcal{O}(1)$ complexity.

5 Asymptotic Analysis

This section provides the theoretical time complexities and asymptotic analyses for each search algorithm, explaining their expected efficiencies and bounding behaviors.

5.1 Linear Search

- **Best Case ($\mathcal{O}(1)$):** The search item is located at the first index.
- **Worst Case ($\mathcal{O}(n)$):** The item is at the end or not present.

Explanation: Linear search examines each element until a match is found or the list ends, resulting in a time complexity of $\mathcal{O}(n)$ on average.

5.2 Binary Search

- **Best Case ($\mathcal{O}(1)$):** The search item is at the middle.
- **Worst Case ($\mathcal{O}(\log n)$):** The item is at one of the ends.

Explanation: Binary search reduces the search space by half on each iteration, achieving logarithmic time complexity on sorted data.

5.3 Hash Table Search with Chaining

- **Best and Average Case ($\mathcal{O}(1)$):** With an effective hash function and a uniform distribution of items across buckets, the search item is likely to be found directly in its bucket, yielding an average time complexity of $\mathcal{O}(1)$.
- **Worst Case ($\mathcal{O}(n)$):** In the event of excessive collisions (e.g., many items hashing to the same bucket), all items in the bucket would need to be searched linearly, resulting in a worst-case time complexity of $\mathcal{O}(n)$.

Explanation: The work factor, or average time complexity, for hash table operations like insertion and retrieval generally depends on two main factors: the load factor and the collision resolution strategy.

- **Load Factor:** The load factor is defined as $\alpha = \frac{n}{k}$, where n is the total number of items and k is the number of buckets in the hash table. A load factor close to or below 1 (i.e., $n \leq k$) helps ensure that items are spread evenly across buckets, keeping operations close to $\mathcal{O}(1)$ on average. As the load factor increases (e.g., if n significantly exceeds k), the likelihood of collisions rises, which can increase the work factor.
- **Chaining and Collision Resolution:** Since this hash table uses chaining as a collision resolution strategy, each bucket holds a linked list of items that hash to the same index. In the average case, with a low load factor and a well-designed hash function, the number of items per bucket remains low, resulting in $\mathcal{O}(1)$ complexity. However, as collisions accumulate (for example, if the load factor grows much larger than 1), the time complexity can approach $\mathcal{O}(n/k)$ per bucket, where n is the total number of items and k is the number of buckets.

In summary, under optimal conditions with a low load factor and uniform item distribution, the average time complexity for search operations in this hash table is approximately $\mathcal{O}(1)$. However, in the worst case, where multiple items hash to the same bucket, the complexity can degrade to $\mathcal{O}(n)$. Ensuring a balanced load factor and efficient hash function helps maintain the performance benefits of hash tables.

6 Conclusion

The experimental results confirm the theoretical expectations for the time complexities of linear, binary, and hash table search algorithms. Linear search's time complexity of $\mathcal{O}(n)$ is reflected in its higher comparison count, particularly as the dataset size increases. Binary search, with its $\mathcal{O}(\log n)$ complexity, showed a significant reduction in comparisons on sorted data, demonstrating its efficiency for ordered data structures. Hash table search, which relies on hashing and chaining, consistently achieved the lowest comparison counts, averaging $\mathcal{O}(n/k)$. The effectiveness of hash table searches depends largely on the design of the hash function and the load factor, defined as the ratio of items to buckets. In this implementation, a fixed hash table size of 250 provided effective performance for the dataset in this assignment. However, had we used a size of 666 the hash table would likely have $\mathcal{O}(1)$ complexity.

Summary of Findings

Overall, this assignment demonstrates the trade-offs among different search algorithms:

- **Linear Search** is straightforward and effective for unsorted or small datasets but has a higher time complexity ($\mathcal{O}(n)$) compared to other methods.
- **Binary Search** is highly efficient for sorted datasets, achieving logarithmic time complexity ($\mathcal{O}(\log n)$), but requires pre-sorted data for effective performance.
- **Hash Table Search** provides the best average-case performance with near $\mathcal{O}(1)$ complexity, especially when the hash table is appropriately sized, and the hash function is well-designed. Managing the load factor by adjusting the table size relative to the number of items helps maintain this optimal performance.

In summary, hash tables offer the most efficient search method among the three analyzed here, provided that collisions are minimized through effective load factor management and hash function design.

References

[1] Oracle, *Java Platform, Standard Edition Documentation*, 2023.

[2] Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C., *Introduction to Algorithms*, Third Edition, MIT Press, 2009.