Analysis of a Hash Table with Chaining

Abhishek Vishwakarma

June 2025

Abstract

This report provides a detailed analysis of a Hash Table implemented with chaining for collision resolution. We describe an implementation that supports dynamic resizing and uses a universal hash function to ensure robust performance. The core of this report is a theoretical analysis of the expected time complexities for insert, search, and delete operations, with a specific focus on the role of the load factor. Finally, we discuss key strategies for maintaining efficiency in a practical hash table implementation.

Contents

1 Implementation Overview

A hash table was implemented to support efficient key-value storage and retrieval. The design addresses the core challenges of hashing through the following means:

1. **Collision Resolution via Chaining:** The fundamental data structure is an array of buckets. When a collision occurs (i.e., multiple keys hash to the same array index), the key-value pairs are stored in a linked list at that index. Each node in the list contains a key, a value, and a pointer to the next node.

2. **Universal Hash Function:** To minimize the probability of collisions regardless of the input data, a hash function from a universal family was chosen. The specific function used is of the form:

$$h_{a,b}(k) = ((ak + b) \pmod{p}) \pmod{m}$$

   where p is a large prime number, a and b are randomly chosen integers, and m is the capacity of the hash table. This method provides strong probabilistic guarantees against clustering and worst-case performance.

3. **Core Operations:** The implementation efficiently supports the three fundamental operations:

   - **Insert:** A key-value pair is added. If the key already exists in the chain, its value is updated. Otherwise, a new node is added to the head of the linked list.
   - **Search:** The value associated with a given key is retrieved by hashing the key and traversing the corresponding chain.
   - **Delete:** A key-value pair is removed by locating the node in its chain and updating the linked list pointers.

2 Analysis of Expected Time Complexity

The efficiency of a hash table with chaining is analyzed under the assumption of **simple uniform hashing**, where any given key has an equal probability of hashing into any of the m slots. The key performance metric is the **load factor**, α, defined as:

$$\alpha = \frac{n}{m}$$

where n is the number of elements in the table and m is the number of slots (buckets). The load factor represents the average length of a chain.

2.1 Expected Time for Operations

The expected time complexity for the main operations is directly dependent on the load factor α.

Search An unsuccessful search requires traversing the entire chain corresponding to a key's hash value. A successful search requires traversing, on average, half the chain. In both cases, the expected work is proportional to the average chain length. The '1' represents the constant-time work of computing the hash.

$$\text{Expected Search Time} = O(1 + \alpha)$$

Insert An insertion first requires a search to check if the key already exists. This takes $O(1 + \alpha)$. If the key is new, adding it to the head of a linked list is an $O(1)$ operation.

$$\text{Expected Insert Time} = O(1 + \alpha)$$

Delete A deletion requires searching for the element, which takes $O(1 + \alpha)$. Once found, removing it from the linked list is an $O(1)$ operation.

$$\text{Expected Delete Time} = O(1 + \alpha)$$

2.2 The Role of the Load Factor

The analysis clearly shows that if the load factor α is maintained as a small constant (e.g., $\alpha < 1$), the expected time for all operations is effectively $O(1)$. This is the ideal state for a hash table. However, if α is allowed to grow unchecked, the chains become long, and the performance degrades linearly, approaching $O(n)$ in the worst case where all keys hash to the same slot.

3 Strategies for Maintaining Efficiency

To ensure the hash table delivers on its promise of constant-time operations, it is critical to manage the load factor and minimize collisions.

3.1 Dynamic Resizing

The most effective strategy to manage the load factor is dynamic resizing. The implementation monitors the load factor before every insertion. If it exceeds a set threshold (e.g., $\alpha \geq 0.7$), the following steps are taken:

1. A new, larger array is allocated (typically double the original capacity).
2. A new hash function (with new random a and b values) is chosen for the new capacity.
3. Every key from the old table is re-hashed and inserted into the new table.

Although this resizing operation is costly ($O(n+m)$), its cost is **amortized** over many insertions. This means the average cost per insertion remains $O(1)$, ensuring the hash table scales efficiently.

3.2 Collision Minimization

Using a high-quality hash function is paramount. A simple ‘mod‘ function can be easily defeated by patterned input, leading to many collisions. A universal hash function, as implemented, provides a probabilistic safeguard against this, ensuring that keys are distributed as evenly as possible across the available slots. This keeps the chain lengths short and uniform, which is essential for consistent performance.