1. Main Concept of Hash Map

Hashing is the process of converting data of any size (like a string, number, or object) into a fixed-length integer value. This integer value is called a **"hash value,"** "hash code," or simply "hash."

The primary data structure that uses this concept is called a **Hash Table**.

1.1. Components of a Hash System

A hash-based search system consists of three main components:

- **Hash Table:** This is fundamentally an array (or a `std::vector`). Each "row" or "slot" in this array is called a **bucket**. The size of this table (`m`) is a key factor in its performance.
- **Key:** This is the data you want to store or look up. The key is fed into the hash function to find its proper location. For example, in a phone book, the ***name*** is the key.
- **Hash Function:** This is the "engine" or mathematical function that takes your **key** and computes an **index** (a row number from `0` to `m-1`) within the array where your data should be stored.

The main goal of hashing is to achieve extremely fast data access. In the ideal case, the hash function provides a unique index for every key, allowing insertion, search, and deletion operations to be performed in constant time, or `O(1)`.

Hash Map Analogy:

Imagine a large file cabinet (**Hash Table**) with 100 drawers numbered 0-99. To store a document for "Budi," you don't search one by one. You use a rule (**Hash Function**), for example, "Take the first letter ('B' -> 2), store it in drawer #2." When you need to find "Budi," you compute the same rule and immediately jump to drawer #2.

1.2. Good Hash Function Properties

The **Hash Function** is the most critical part of the Hash Table. A good function must have the following properties:

- **Deterministic:** The same input (key) must *always* produce the same output (hash value).
- **Efficient:** The hash function must be fast to compute (ideally `O(1)`).
- **Uniform Distribution:** This is the most important. The hash function must spread keys as evenly as possible across all slots (indices) in the table. This minimizes collisions.

If the hash function doesn't distribute data well (e.g., it hashes all names starting with 'A', 'B', and 'C' to the same slot), it will cause **collisions**, and the performance will degrade significantly.

1.3. Common Hash Function Methods

There are several methods to design a hash function, each with different properties. The goal is always to create a **"well-distributed" hash** that avoids clustering keys in the same buckets.

1.3.1. Division Method

This is the most common and simplest method. It takes the key (which must be a numeric value or convertible to one) and finds the remainder after dividing by the table size.

- **Formula:** `h(key) = key % tableSize`
- **Pros:** It is extremely fast, involving only a single modular arithmetic operation.
- **Cons:** The performance is highly dependent on the choice of `tableSize`. If `tableSize` is poorly chosen (e.g., a power of 10 or 2), and the input keys have a pattern (e.g., all keys are even), this can lead to massive collisions. This is why it is strongly recommended to make `tableSize` a **prime number** that is not close to a power of 2

1.3.2. Multiplication Method

This method is a popular choice because it is far less sensitive to the `tableSize` (which can be any value, often a power of 2 for efficiency).

- **Formula:** `h(key) = floor( tableSize * ( (key * A) % 1 ) )`
- **Explanation:**
  - **`key * A`:** The key is multiplied by a constant A (where $0 < A < 1$). A common and effective value for A is related to the golden ratio.
  - **`( ... ) % 1:`** This operation extracts only the fractional part of the result. This fractional part is highly mixed and influenced by all digits of the key.
  - **`tableSize * ( ... )`:** This fractional part is then scaled up to the range of the table.
  - **`floor( ... )`:** The integer part is taken as the final index.
- **Pros:** This method effectively scrambles the input bits, meaning that similar keys (like 100 and 101) are likely to be mapped to very different indices. The choice of `tableSize` is not critical.

1.3.3. Mid-Square Method

This method is particularly effective at breaking up patterns in keys that are sequential or have similar prefixes/suffixes.

- **Concept:** The key is first squared. The resulting number is often very large. A set of digits is then extracted from the middle of this squared result to be used as the hash index.
- **Why it works:** The middle digits of a squared number are influenced by all the digits of the original key. Changes in either the beginning or end of the key will cause significant changes in the middle digits, effectively "folding" the key in on itself and scrambling any existing patterns.
- **Pros:** Good at breaking up non-random sequences in input keys.

1.3.4. Folding Method

This method is excellent for hashing keys that are very long, such as account numbers, ISBNs, or long strings, where other methods might lead to overflow or only use a portion of the key.

- **Concept:** The key is divided into several equally-sized parts. These parts are then combined using a simple operation.
- **Operations:**
  - **Addition:** All parts are added together. The final result may need an extra modulo (`% tableSize`) if it exceeds the table size.
  - **XOR:** All parts are combined using the bitwise XOR operation. This is very fast and naturally stays within bounds if the parts are sized correctly.
- **Pros:** It ensures that all parts of a long key (beginning, middle, and end) contribute to the final hash value, making it highly distributive for long, structured keys.

Revision #4
Created 2025-11-04 15:20:20 UTC by RE
Updated 2025-11-04 15:50:46 UTC by RE