Hashing

A Guide to Hashing Collision Resolution

Hello! Welcome to your complete guide on handling hash collisions. Just like in cooking, where you might have a favorite burner on the stove, in hashing, different data items (your ingredients) might want the same spot in your hash table (the stove). When this happens, it's called a **collision**.

The techniques we'll explore are part of a strategy called **Open Addressing**. The core idea is simple: if a spot is taken, we find another open spot in a systematic way. Let's dive into two classic "recipes" for doing this!

1. Linear Probing (The "One-by-One" Method)

Linear probing is the most straightforward approach. If a key's designated spot is taken, you simply check the next slot, then the one after that, and so on, until you find an empty one. It's like checking your stove burners in order: 1, 2, 3, 4... until you find one that's free.

Syntax (The Formula)

The position for the i-th attempt (or "probe") is calculated as:

$h(k,i)=(h'(k)+i)(mod m)$

- `h'(k)`: The initial hash value for your key `k`.
- `i`: The probe number, starting at `0` (for the first try), then `1`, `2`, etc.
- `m`: The size of the hash table.

Benefits & Drawbacks

- **Benefit (Simplicity & Cache-Friendliness):** It's incredibly easy to implement and can be fast because checking adjacent memory locations works well with modern computer caches.
- **Drawback (Primary Clustering):** Its major weakness is that filled slots tend to clump together into long chains. This creates a "traffic jam" that can significantly slow down finding, inserting, or deleting elements.

Step-by-Step Example

Let's see it in action with a practical example.

- **Table Size (`m`):** `7` (Indices 0 through 6)
- **Keys to Insert:** `[15, 22, 8, 29]`
- **Hash Function (`h1`):** `key % 7`
- **Initial Table:** `[ , , , , , , , ]`

Step 1: Insert `15`

- `h1(15) = 15 % 7 = 1`.
- Index `1` is empty. Place `15` there.
- **Table:** `[ , 15, , , , , , ]`

Step 2: Insert `22`

- `h1(22) = 22 % 7 = 1`.
- **Collision!** Index `1` is taken. We probe.
- Probe `i=1`: `(1 + 1) % 7 = 2`. Index `2` is empty. Place `22` there.
- **Table:** `[ , 15, 22, , , , , ]`

Step 3: Insert `8`

- `h1(8) = 8 % 7 = 1`.
- **Collision!** Index `1` is taken.
- Probe `i=1`: `(1 + 1) % 7 = 2`. Index `2` is also taken.
- Probe `i=2`: `(1 + 2) % 7 = 3`. Index `3` is empty. Place `8` there.
- **Table:** `[ , 15, 22, 8, , , , ]`

Step 4: Insert `29`

- `h1(29) = 29 % 7 = 1`.
- **Collision!** Index `1` is taken.
- Probe `i=1`: `(1 + 1) % 7 = 2`. Taken.
- Probe `i=2`: `(1 + 2) % 7 = 3`. Taken.
- Probe `i=3`: `(1 + 3) % 7 = 4`. Index `4` is empty. Place `29` there.
- **Final Table:** `[ , 15, 22, 8, 29, , , ]`

You can clearly see the **cluster** forming from index 1 to 4.

2. Double Hashing (The "Custom Step" Method)

Double hashing is a more sophisticated experiment. Instead of a fixed step of `+1`, it uses a **second hash function** to determine a unique jump size for each key. This smart approach helps scatter the keys more evenly, avoiding the "traffic jams" of linear probing.

Syntax (The Formula)

The position for the i-th probe is calculated using two hash functions:

$h(k,i) = (h1(k) + i \cdot h2(k)) \pmod m$

- `h1(k)` : The primary hash function.
- `h2(k)` : The second hash function that calculates the step size.
- `i` : The probe number (`0, 1, 2, ...`).
- `m` : The table size.

Important Rule: The value of `h2(k)` should never be 0.

Benefits & Drawbacks

- **Benefit (Eliminates Clustering):** This is its superpower! By using key-specific jump sizes, it effectively prevents clusters from forming, leading to much better performance. 🚀
- **Drawback (Complexity & Overhead):** It's a bit more complex to implement and requires the extra computation of a second hash function, which can add a small amount of overhead.

Step-by-Step Example

Let's use the same setup but with our new technique.

- **Table Size (`m`):** `7`
- **Keys to Insert:** `[15, 22, 8, 29]`
- **Primary Hash (`h1`):** `key % 7`
- **Second Hash (`h2`):** `5 - (key % 5)`
- **Initial Table:** `[ , , , , , , ]`

Step 1: Insert `15`

- `h1(15) = 15 % 7 = 1` .
- Index `1` is empty. Place `15` there.
- **Table:** `[ , 15, , , , , ]`

Step 2: Insert `22`

- `h1(22) = 22 % 7 = 1` .
- **Collision!** Calculate the step size.
- Step size `h2(22) = 5 - (22 % 5) = 5 - 2 = 3` .

- Probe `i=1`: `(1 + 1 * 3) % 7 = 4`. Index `4` is empty. Place `22` there.
- **Table:** `[ , 15, , , 22, , ]`

Step 3: Insert `8`

- `h1(8) = 8 % 7 = 1`.
- **Collision!** Calculate the step size.
- Step size `h2(8) = 5 - (8 % 5) = 5 - 3 = 2`.
- Probe `i=1`: `(1 + 1 * 2) % 7 = 3`. Index `3` is empty. Place `8` there.
- **Table:** `[ , 15, , 8, 22, , ]`

Step 4: Insert `29`

- `h1(29) = 29 % 7 = 1`.
- **Collision!** Calculate the step size.
- Step size `h2(29) = 5 - (29 % 5) = 5 - 4 = 1`.
- Probe `i=1`: `(1 + 1 * 1) % 7 = 2`. Index `2` is empty. Place `29` there.
- **Final Table:** `[ , 15, 29, 8, 22, , ]`

Notice how beautifully the keys are distributed! No clustering at all. This is why double hashing is often the preferred method in practice.