Notes on Hashing and Collision

1. Introduction to Hashing

Hashing is a technique used in computing to uniquely identify a specific object from a group of similar objects.
It involves transforming an input (or 'key') into a fixed-size value, usually a number, which represents the original string of characters in a condensed format.

Hashing is especially useful in scenarios where rapid access to data is required. It plays a fundamental role in implementing efficient data structures like hash tables, hash maps, and dictionaries.
Hashing is widely used in database indexing, caches, and cryptography.

An ideal hash function distributes keys uniformly across the hash table, thereby minimizing the number of collisions and ensuring constant time complexity (O(1)) for search, insert, and delete operations.

2. Need for Hashing

With the explosion of data on the internet and within systems, traditional data structures like arrays and linked lists become inefficient due to their search times (O(n) or O(log n)).
Although arrays allow constant time for insertion and access by index, searching for a value takes linear time unless the array is sorted.

Hashing allows for:
- Constant time complexity for searching and insertion.
- Efficient memory usage.
- Faster access patterns.

Hash tables use a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.

3. Components of Hashing

1. Key: The data to be stored or searched (e.g., strings, numbers).
2. Hash Function: Converts the key into an index.
3. Hash Table: An array where the data is stored at the index computed by the hash function.

Each key is mapped to an index using the hash function. The key-value pair is stored at that index

in the hash table.

4. How Hashing Works (with Example)

Suppose we want to store the strings {"ab", "cd", "efg"}. Assigning character values: a=1, b=2, c=3, ..., z=26:

1. "ab" = 1+2 = 3
2. "cd" = 3+4 = 7
3. "efg" = 5+6+7 = 18

If table size is 7, hash function = sum % 7:

- "ab" → 3 % 7 = 3
- "cd" → 7 % 7 = 0
- "efg" → 18 % 7 = 4

So values are stored in slots 3, 0, and 4 respectively.

5. Hash Functions (Types and Examples)

Hash functions map data of arbitrary size to data of fixed size.

1. Division Method: h(k) = k mod m
   - Simple and fast.
   - M should be a prime number for better distribution.

2. Mid Square Method:
   - Square the key and extract the middle digits.
   - h(k) = middle digits of k^2

3. Folding Method:
   - Split the key into parts and sum them.
   - Works well with long numeric keys.

4. Multiplication Method:
   - h(k) = floor(m * (kA mod 1)), where 0 < A < 1.
   - Independent of table size structure.

6. Collisions in Hashing

A collision occurs when two keys produce the same index in a hash table.

Causes of Collisions:
- Poor hash function design.
- Small hash table size.
- Limited key space.

Example: "ab" and "ba" both may sum to the same value and hence same index.

Impact:
- Reduces performance.
- Makes search, insert, and delete less efficient.
- May require complex collision resolution.

7. Collision Resolution Techniques

There are two main strategies:

1. Separate Chaining:
   - Each slot has a linked list.
   - New values are appended to the list at the hashed index.
   - Easy to implement but uses extra memory.

2. Open Addressing:
   - All data is stored in the table.
   - If a slot is occupied, find the next available one.

   i) Linear Probing: Check next slot (index + 1).
   ii) Quadratic Probing: Use quadratic jumps (i^2).
   iii) Double Hashing: Use a second hash function.

Disadvantages of Linear Probing

1. **Primary Clustering**
   - Elements with nearby hash indices group together.
   - This leads to long clusters that degrade performance.
   - Future insertions and searches take longer due to these continuous occupied areas.
2. **Performance Degrades at High Load Factors**
   - As the table fills up, finding an empty slot takes longer.

- Insertions and lookups become inefficient.

3. **More Probes Required**
   - Especially if many elements are inserted or deleted, it may have to linearly check many slots.
4. **Deletion is Complex**
   - Deleting an element can break the probe sequence.
   - Special markers (like "deleted") must be used to maintain search integrity.

Disadvantages of Quadratic Probing

1. **Secondary Clustering**
   - While it avoids primary clustering, elements that hash to the same location follow the same quadratic probe path.
   - This still causes clustering, just less severe than linear.
2. **May Not Cover All Slots**
   - Doesn't always find an empty slot even if one exists.
   - This happens unless:
     - Table size is a **prime number**
     - Load factor is **< 0.5**
3. **More Complex Calculations**
   - Probing involves squares `(i²)`, slightly slower than linear `(i)`—though still fast.
4. **Harder to Implement Efficient Deletion**
   - Like linear probing, deletion requires special handling (tombstones or markers).

Why Quadratic Probing is Better Than Linear Probing

Feature	Linear Probing	Quadratic Probing
Probe Sequence	(h + 1) % m, (h + 2) % m, ...	(h + 1²) % m, (h + 2²) % m, ...
Clustering	Causes **primary clustering** (large groups of filled slots)	Reduces primary clustering
Search Spread	Probes consecutive slots	Probes farther apart, spreading out better
Risk	Higher chance of long probe chains	Lower chance due to dispersion
Performance Over Time	Degrades quickly as table fills	More stable even when moderately full
Full Coverage	Always finds a slot (if one exists)	May not find a slot unless table size is prime and load is low

Clustering Explained

- **Linear Probing:** If a few elements collide at the same index, they create a block. Future insertions also pile up there → **Primary Clustering**.
- **Quadratic Probing:** The next slots checked are farther away, reducing grouping.

When Quadratic Probing Can Fail

Quadratic probing can fail to examine all slots unless:

- Table size is a **prime number**
- **Load factor is low** (e.g. < 0.5)

When to Prefer Quadratic

Use quadratic probing when:

- trying to **reduce clustering**
- have a **reasonably sparse table**
- not overly worried about not covering 100% of slots

8. Load Factor and Rehashing

Load Factor = number of elements / size of table.

- A high load factor indicates a full table → leads to more collisions.
- Typical threshold: 0.75

Rehashing:
- Resize the table (usually double the size).
- Recalculate and reinsert all elements.
- Keeps collision low and efficiency high.

9. Applications of Hashing

- Database indexing.
- Cache design (key-value storage).
- Password storage (hashing with salt).
- Blockchain and cryptography.
- Pattern searching (Rabin-Karp algorithm).

10. Advantages and Disadvantages

Advantages:

- O(1) average-case performance.
- Fast insertion, deletion, and access.
- Useful in real-time systems.

Disadvantages:

- Collisions degrade performance.
- Requires careful hash function design.
- Poor performance in worst-case scenario (O(n)).