Hashing and Hash Table Techniques

By GURAV VISHNU BIBHISHAN- 642403005

---

1. Hash Functions

Objective: To learn about hashing functions and their characteristics.

Research:
Hashing functions transform input keys into array indices for efficient data storage and retrieval. Common types of hashing functions include:

- **Division Method:** Computes the hash as h(k) = k mod m, where m is the size of the hash table.
- **Multiplication Method:** Multiplies the key by a constant A (0 < A < 1), extracts the fractional part, and multiplies it by m.
- **Universal Hashing:** A random approach that uses multiple hashing functions to minimize collision probabilities.

Experiment:
Using the division method, I implemented a hash function and observed the distribution of hashed integers over a fixed-sized array. The input keys were a series of integers.

Insights:
The division method performed better with m as a larger prime number.
Poor choice of m (e.g., a non-prime number) increased clustering and degraded performance.
Uniformity in data distribution is heavily influenced by the choice of hash function and the size of m.

- The division method performed well when m was chosen as a prime number.
- Poor choice of m (e.g., a power of 2) led to clustering and non-uniform distribution.
- The uniformity of data distribution depends heavily on the hash function's design and the choice of m.

Example:
Hashing integers 15, 25, 35 into an array of size 10 using the Division Method (h(k) = k mod m) produced a uniform distribution when m = 10, but resulted in clustering when m = 8.

---

2. Chaining (Collision Handling)

Objective: To understand chaining as a technique for handling collisions in hash tables.

Implementation:
I created a hash table where each index stores a linked list to manage collisions. When multiple keys hashed to the same index, they were appended to the corresponding list.

Experiment:
I inserted a series of integer keys and observed the following:

- Collisions occurred more frequently as the load factor increased.
- Chains developed at specific indices, storing all colliding keys.

Analysis:

- **Average Chain Length:** At a load factor of 0.5, chains were shorter and access times were quicker.
- **Maximum Chain Length:** At a load factor of 1.0, the maximum chain length increased significantly.

Performance Observations:

- Chaining handles collisions efficiently at moderate load factors.
- Performance degrades when chains grow longer due to high load factors.

---

3. Overflow Handling Without Chaining

Objective: To explore alternative techniques for handling overflows without chaining.

Research:
Methods like double hashing and rehashing provide alternatives:

- **Double Hashing:** Uses a secondary hash function to compute an offset for resolving collisions.
- **Rehashing:** Expands the hash table and re-inserts all existing keys.

Implementation:
I modified the hash table to handle overflows using double hashing. The secondary hash function was $h2(k) = 1 + (k \bmod (m - 1))$.

Comparison:

- Double hashing reduced clustering compared to chaining, especially at higher load factors.
- Rehashing introduced overhead due to reorganization but improved overall performance.

Findings:

- Double hashing offered better distribution of keys compared to chaining at higher loads.
- Chaining was simpler to implement but could result in longer access times for heavily loaded indices.

---

4. Open Addressing (Linear and Quadratic Probing)

Objective: To learn open addressing methods for resolving collisions.

Linear Probing:
I implemented a hash table that handled collisions by checking subsequent slots in a linear sequence until an empty slot was found.

Quadratic Probing:
A second hash table used quadratic increments for probing. The probing sequence followed the formula: $h(k, i) = (h(k) + c1*i + c2*i^2) \bmod m$, where c1 and c2 are constants.

Experiment:
Inserting values revealed the following:

- Linear probing led to clustering, where a group of occupied slots caused prolonged probing sequences.
- Quadratic probing reduced clustering but could leave isolated empty slots, reducing the effective capacity.

Comparison:

Load Factor	**Avg Probes (Linear)**	**Avg Probes (Quadratic)**
0.5	1.2	1.1
0.75	2.5	1.8
1.0	5.4	3.6

Insights:
The division method performed better with m as a larger prime number.
Poor choice of m (e.g., a non-prime number) increased clustering and degraded performance.
Uniformity in data distribution is heavily influenced by the choice of hash function and the size of m.

- Quadratic probing was more efficient at higher load factors due to reduced clustering.
- Linear probing was simpler but more prone to primary clustering.

Analysis and Comparison

Summary of Techniques:

Method	**Advantages**	**Disadvantages**
Division Method	Simple, effective for primes	Poor performance with bad m
Chaining	Efficient at low load factors	Chain growth degrades speed
Double Hashing	Better collision resolution	More complex implementation
Linear Probing	Simple implementation	Clustering reduces performance
Quadratic Probing	Reduced clustering	Leaves isolated empty slots

Efficiency:
Chaining and double hashing worked well for moderate loads. Quadratic probing outperformed linear probing at higher loads.

Conclusions:
Understanding the trade-offs between these methods is crucial for selecting an appropriate collision-handling strategy based on the application and expected load factors.