Hash Functions

1. Division Method

- **Technique**: The hash function is `h(k) = k mod m`, where `k` is the key, and `m` is the table size.
- **Uniformity**: The division method works best when `m` is a prime number. If `m` shares factors with the keys, clustering may happen.
- **Example**:
  - With a table size of 11 (a prime number) and keys `{20, 50, 73, 19, 6, 77, 10, 55, 44, 92, 36, 25}`, keys are hashed to:
    - `20 mod 11 = 9`
    - `50 mod 11 = 6`
    - `73 mod 11 = 7`
  - Keys are distributed across the hash table but can still exhibit clustering if certain keys produce the same remainder.

2. Multiplication Method

- **Technique**: The hash function is `h(k) = ⌊m * (k * A mod 1)⌋`, where `A` is typically a fractional constant close to `(√5 - 1) / 2`, and `m` is the table size.
- **Uniformity**: The multiplication method is less dependent on table size and offers uniform distribution by spreading values across available slots.
- **Example**:
  - With a fractional constant `A ≈ 0.618033`, the same keys as above might hash to:
    - `20 * A mod 1 = 0.3606, 11 * 0.3606 ≈ 4`
    - `50 * A mod 1 = 0.9017, 11 * 0.9017 ≈ 9`
    - `73 * A mod 1 = 0.1584, 11 * 0.1584 ≈ 1`
  - This approach typically prevents clustering by distributing keys evenly, regardless of their factors.

3. Universal Hashing

- **Technique**: A randomly selected hash function minimizes predictable clustering with `h(k) = ((a * k + b) mod p) mod m`, where a and b are constants, and p is a prime larger than `m`.
- **Uniformity**: Universal hashing ensures that each key has an equal probability of mapping to each index, making it ideal for reducing clustering.
- **Example**:
  - With random values `a = 3`, `b = 7`, and `p = 13`, the keys above might hash to:

        - `((3 * 20 + 7) mod 13) mod 11 = 4`
        - `((3 * 50 + 7) mod 13) mod 11 = 6`
        - `((3 * 73 + 7) mod 13) mod 11 = 1`
    - This approach provides a nearly uniform spread of keys across the hash table, ideal for cases where collision reduction is critical.

Chaining (Collision Handling)

For a table size of 11, insert keys: `{20, 19, 6, 10, 55, 44, 92, 36, 25}`.

Load Factor Calculation:

- **Number of keys** = 9
- **Load Factor** = 9 / 11 ≈ 0.82

Index	Keys	Chain length
0		0
1		0
2	92, 36	2
3		0
4		0
5	55, 44, 25	3
6	6	1
7		0
8		0
9	19	1
10	20,10	2

Calculations:

- **Total Chains** = 5 (only indices with chains are counted)
- **Total Length of Chains** = 2 + 3 + 1 + 1 + 2 = 9
- **Average Chain Length** = Total Length of Chains / Total Chains = 9 / 5 = 1.8
- **Maximum Chain Length** = 3 (at index 5)

Overflow Handling Without Chaining

Aspect	Double Hashing	Chaining
Method	Resolves collisions using a secondary hash function.	Stores colliding keys in a linked list at each index.
Collision Handling	Probes sequential indices until an empty slot is found, which can lead to clustering.	Handles collisions flexibly, as each index has its own linked list.
Space Efficiency	Fixed table size; collisions increase probes but do not require additional storage for chains.	Requires additional memory for linked lists, especially with more collisions.
Probing	Increased probes as load factor approaches 1.0.	Chains grow dynamically, so no additional probing needed.
Average Probe Count	Increases with higher load factors; minimal at low loads (e.g., ≤ 0.75).	N/A (Chaining doesn't rely on probing).
Average Chain Length	N/A (No chains are created in double hashing).	Chain length grows with load factor but remains consistent.
Efficiency	Slows down with high load factors due to clustering.	Consistent efficiency even at high load factors, as lists grow independently.
Best Use Cases	Small datasets, lower load factors (≤ 0.75).	Larger datasets, higher load factors, flexible memory availability.
Load Factor Impact	Performance degrades noticeably as load factor approaches 1.0 due to increased probe count.	Performance remains stable even at higher load factors due to dynamic chaining.

Open Addressing (Linear and Quadratic Probing)

Aspect	Linear Probing	Quadratic Probing
Collision Handling	Collisions result in checking consecutive slots.	Collisions result in checking slots with quadratic increments (i^2).
Probe Count	Number of probes increases linearly as the table gets more filled.	Number of probes increases more gradually due to quadratic increments.
Clustering	Tends to cause primary clustering (keys hash to adjacent spots).	Reduces primary clustering by spreading out the probes more evenly.
Performance with High Load Factor	Performance decreases significantly as the table gets more filled due to clustering.	Performance is more stable as the probing space is more spread out.
Efficiency	Less efficient when the table load factor is high due to increased probes.	More efficient in handling collisions and maintaining performance at higher load factors.
Table Utilization	May result in under-utilization of table slots because of consecutive collisions.	Utilizes table slots more effectively as probing is spread.
Complexity	Simpler implementation and concept.	Slightly more complex due to quadratic calculation.
Overflow Handling	Becomes inefficient as table fills, leading to more probes and slower performance.	Handles overflow more gracefully, but performance still degrades at high load factors.
Scalability	Scalability issues arise at higher load factors, causing excessive probe counts.	Scalability is better at higher load factors, with a more even distribution of probes.

Key	Linear Probing Probes	Quadratic Probing Probes
20	1	1
30	1	1
40	1	1
50	1	1
22	1	2
42	1	2
53	1	3
66	1	3
77	2	4
18	1	2

Conclusion

The choice of hashing method and collision handling strategy significantly impacts hash table performance:

- **Division and Multiplication methods** are simple and effective, but Universal Hashing provides better security and clustering resistance.
- **Chaining** is efficient at managing collisions in dynamic memory settings, while **Open Addressing** (especially with quadratic probing) is efficient when memory is limited but becomes less performant at high load factors.

Using a combination of an effective hash function and the appropriate collision-handling strategy ensures balanced performance based on data distribution and memory constraints.