Hashing

Absolutely! Here's a **detailed explanation of all hashing concepts** from your `Unit-02.pptx`, structured clearly for understanding and revision:

🔐 HASHING — Introduction

Hashing is a technique used to map large data sets to a smaller table of fixed size using a mathematical formula called a **hash function**.

It is commonly used for:

- Fast data retrieval
- Storing key-value pairs (like in dictionaries)
- Implementing sets, hash tables, and symbol tables

🔧 Key Components of Hashing

Component	Description
Key	The input data to be stored or searched (e.g., roll number, name).
Hash Function	A function that maps the key to an index in the hash table.
Hash Table	An array-like structure that stores values based on the index returned by the hash function.

📍 HASH FUNCTION

A **Hash Function** takes a key and returns a hash index (a number representing the position in the hash table).

Goals of a good hash function:

- Uniform distribution of values
- Fast computation
- Minimize collisions (same index for multiple keys)

📐 TYPES OF HASH FUNCTIONS

1. 🔢 Division Method

- Formula: `h(k) = k mod m`
- `k` = key, `m` = size of hash table (preferably prime)

Example:
`k = 23`, `m = 10` → `h(23) = 23 % 10 = 3`

Pros: Simple and fast
Cons: Poor distribution if `m` is not prime

---

2. ✖️ Multiplication Method

- Formula: `h(k) = ⌊m × (k × A mod 1)⌋`
- `A` = constant between 0 and 1, `mod 1` means the fractional part

Example:
`k = 1234`, `A = 0.618`, `m = 1000`
Multiply → 1234 × 0.618 = 762.612
Take fraction → 0.612 × 1000 = 612
Final hash = 612

Pros: Better distribution than division
Cons: Slightly more complex to compute

---

3. ⬜ Mid-Square Method

- Square the key, extract middle digits

Steps:

1. Square the key
2. Take middle digits as the hash index

Example:
`Key = 123` → `123² = 15129` → middle digits = `512` → hash = 512

Pros: Good distribution
Cons: Costly (multiplication and digit extraction)

---

4. 📐 Folding Method

- Break key into equal parts, sum the parts, take mod

Steps:

1. Split key (e.g., 123456 into 123 and 456)
2. Add: 123 + 456 = 579

3. Hash = 579 mod m

Pros: Easy to implement
Cons: Depends on partitioning scheme

💥 COLLISIONS

A **collision** happens when two different keys hash to the same index.

🛠️ COLLISION HANDLING TECHNIQUES

A. **Open Hashing (Separate Chaining / Closed Addressing)**

- Use **linked lists** at each index of the hash table
- All keys that hash to the same index are stored in the list

Example:
`h(k) = k mod 5`, keys = `12, 22, 15, 25`
All give remainder `2`, stored at index 2 using a linked list

Pros:

- Simple to implement
- Unlimited number of keys per index

Cons:

- Extra space for lists
- Slower lookup in long chains

B. **Closed Hashing (Open Addressing)**

No extra structures — all elements are stored directly in the array. If a slot is occupied, the algorithm probes for another empty one.

1. 🔁 **Linear Probing**

- Check next slot: `(hash + i) % table_size`

Example:
Hash table size = 5, key = 50 → `50 mod 5 = 0`
If slot 0 is full, check 1, 2, 3, ...

Pros: Simple
Cons: Clustering — multiple items form a block

---

2. ■ Quadratic Probing

- Checks farther slots with increasing gaps:
  `(hash + i²) % table_size`

Example:
Insert 22, 30, 50 in table size 7 with `f(i) = i²`

- `22 % 7 = 1` → slot 1
- `30 % 7 = 2` → slot 2
- `50 % 7 = 1` → slot 1 occupied → try (1 + 1² = 2), then (1 + 2² = 5)

Pros: Less clustering than linear
Cons: Can skip over available slots

---

3. 🔁 Double Hashing

Uses **two hash functions** to calculate probe sequence:
`(h1(k) + i * h2(k)) % table_size`

Example:
Table size = 7, keys = 27, 43, 692, 72
`h1(k) = k % 7`, `h2(k) = 1 + (k % 5)`

- 27 → h1 = 6, h2 = 3 → index = (6 + i×3) % 7

Pros: Best distribution
Cons: Complex logic

⚙️ Summary

Concept	Description
Hashing	Fast data lookup using keys and hash functions
Hash Function	Converts key to index
Division Method	`key % tableSize`, simple
Multiplication	`⌊m(kA mod 1)⌋`, better distribution
Mid-Square	Use middle digits of squared key
Folding	Add parts of key
Collision	Two keys map to same index
Separate Chaining	Linked lists for each slot
Linear Probing	Next slot → index+1

Concept	Description
Quadratic Probing	index+i²
Double Hashing	Uses second hash function to compute jump

Great! You've corrected the input. Let's **recalculate** the **Linear Probing insertion** and then find the **average number of successful and unsuccessful probes**, step-by-step.

📜 Given:

- **Table Size**: 11 slots → `T[1] to T[11]`
- **Records** and their **hash addresses (H(k))**:

Record	A	B	C	D	E	X	Y	Z
H(k)	4	8	2	11	4	11	5	1

🔁 Step 1: Insert Using **Linear Probing**

Start inserting based on H(k). If the target index is full, move to the next index (`+1`), wrapping around circularly until an empty slot is found.

Insert Order	Record	H(k)	Final Position	Probes
1	A	4	4	1
2	B	8	8	1
3	C	2	2	1
4	D	11	11	1
5	E	4	5 (4 → occupied)	2
6	X	11	1 (11 → occupied, 1 free)	2
7	Y	5	6 (5 occupied, 6 free)	2
8	Z	1	3 (1,2,3rd slot free after 1)	3

🔢 Hash Table after Insertion

Index	Record
1	X
2	C

Index	Record
3	Z
4	A
5	E
6	Y
7	-
8	B
9	-
10	-
11	D

📊 Step 2: Count Probes

✅ Successful Probes

Use the same number of probes taken while inserting each record.

Record	Probes
A	1
B	1
C	1
D	1
E	2
X	2
Y	2
Z	3

Total Successful Probes = 1 + 1 + 1 + 1 + 2 + 2 + 2 + 3 = **13**

Average Successful Probes = 13 / 8 = **1.625**

❌ Unsuccessful Probes (Linear Probing Formula)

For **unsuccessful search**, average number of probes ≈

$$\frac{1 + \frac{1}{1-\alpha}}{2}$$

Where:

- $\alpha = \frac{n}{m} = \frac{8}{11} \approx 0.727$

$$\Rightarrow \text{Average Unsuccessful Probes} \approx \frac{1 + \frac{1}{1-0.727}}{2} = \frac{1 + 3.66}{2} = \frac{4.66}{2} \approx \boxed{2.33}$$

✅ Final Answer:

- **Average Successful Probes** = **1.625**
- **Average Unsuccessful Probes** = **2.33**

Let's now apply **chaining (separate chaining / closed addressing)** using the same table setup.

📌 Problem Setup (Revisited)

- **Table Size**: 11 → `T[1] to T[11]`
- **Records** and their hash addresses:

Record	A	B	C	D	E	X	Y	Z
H(k)	4	8	2	11	4	11	5	1

🔁 Step 1: Insert Using **Chaining**

In separate chaining:

- Each index of the hash table points to a **linked list**
- Multiple records that hash to the same index are stored in the list

Let's insert the records into chains at their hash index.

Table After Chaining Insertion:

Index	Chain (Linked List)
1	Z
2	C
3	—
4	A → E
5	Y
6	—
7	—
8	B
9	—
10	—
11	D → X

📊 Step 2: Count Probes

✅ Successful Search Probes (on average)

We compute probes per element:

- In each chain, record position in the list = probe count
- Assume the element is found during sequential search

Probe Count for Each Record:

Index	Chain	Records with Probes
1	Z	Z(1)
2	C	C(1)
4	A → E	A(1), E(2)
5	Y	Y(1)
8	B	B(1)
11	D → X	D(1), X(2)

→ Total Probes = 1 (Z) + 1 (C) + 1 (A) + 2 (E) + 1 (Y) + 1 (B) + 1 (D) + 2 (X) = **10**

→ Average = 10 / 8 = **1.25**

❌ Unsuccessful Search Probes

To search unsuccessfully at index `i`, you must traverse the **entire chain** at that index.

So, we sum lengths of chains at all indices, divide by total number of slots (11):

Index	Chain Length
1	1
2	1
3	0
4	2
5	1
6	0
7	0
8	1
9	0
10	0
11	2

→ Total comparisons = 1 + 1 + 0 + 2 + 1 + 0 + 0 + 1 + 0 + 0 + 2 = **8**
→ Average = 8 / 11 = **0.727**

✅ Final Result (Separate Chaining)

- **Average Successful Probes** = **1.25**
- **Average Unsuccessful Probes** = **0.727**

💡 **Observation**:
Chaining gives better average-case performance (especially for unsuccessful searches) when the load factor is moderate.
Let me know if you'd like to **visualize this as a diagram** or compare with **quadratic probing** or **double hashing** next!