UNIT-III

- File Organization:
- Indexed sequential access files;
- implementation using B & B++ trees,
- hashing,
- hashing functions,
- collision resolution,
- extendible hashing,
- dynamic hashing approach implementation and performance

What is File Organization?

- **File Organization** in DBMS means **the way records (data) are stored in a file on disk**.
- It defines **how data is arranged, accessed, and updated**.
- Good file organization improves **speed** of data retrieval and **efficient storage**.

File Organization Types

File Organization = The way data records are stored on storage media.

Main types:

Types of File Organization

1. **Heap (or Pile)File Organization**
2. **Sequential File Organization**
3. **Indexed Sequential File Organization (ISAM)**
4. **Direct (or Hashed) File Organization**

1. Heap (or Pile) File Organization

•Records are **stored wherever space is available — no order at all**.

•Also called **Unordered File Organization**.

⚙️ **Working**

•New records simply appended (add) to the end of file or in any free block.

•Searching = scan entire file (slow).

📝 **Diagram**

[RecC] [RecA] [RecB] [RecE] [RecD] (no order)

Advantage: Simple, fast insertion.

Disadvantage: Very slow searching and deletion.

2.Sequential File Organization means storing records **one after another** in a specific sequence — either **in the order they were entered** or **sorted by a key field** (like Roll No or Employee ID).

Emp_ID	Name	Salary
101	Aditi	50000
102	Rajesh	48000
103	Neha	52000

- Stored sequentially by **Emp_ID**.
- To find Emp_ID = 103, DBMS scans record 101 → 102 → 103.

Advantages

- **Simple** to implement.
- **Fast** for sequential access (batch operations).

Disadvantages

- **Slow random access** — must scan sequentially.
- **Costly insertions/deletions** — may need reorganization.

Indexed Sequential File Organization

- Records (data) are **stored in order** (sequentially) based on a **key field** (like Roll No or Account No).
- But there is also an **index** — a small table that helps **find records faster**.

It's a **combination** of:

- **Sequential file** (records arranged in order)
- **Index file** (used for fast searching)

RollNo	Name	Marks
101	Ravi	80
102	Sita	90
103	Aman	85
104	Neha	95

Records are **stored in order of RollNo**.
An **index** file keeps key pointers, e.g.:

RollNo	Address (Pointer)
101	001
103	003

Now, if you want to find RollNo 103:

- Instead of reading all records one by one,
- The system **uses the index** to jump directly to record 103

Advantages

- **Faster access** than pure sequential files (thanks to index).
- Allows both **sequential** and **direct/random access**.

Disadvantages

- **Extra storage** needed for the index.
- **Maintenance overhead** (index must be updated on insertion/deletion).

4.In **Direct (or Hash) File Organization**, a **hashing function** is used to calculate the **address (or location)** of a record directly from its key field.

- No sequential search
- No index table
- Directly compute storage location

This is the **fastest method** for random/direct access.

Emp_ID	Name	Hash = Emp_ID % 10	Stored in Bucket (Address)
101	Aditi	1	1
104	Raj	4	4
117	Neha	7	7
125	Amit	5	5

How it Works:

If you want to **find Emp_ID = 117**,

The system does:

```
h(117) = 117 % 10 = 7
```

So it **directly goes to bucket 7** and retrieves Neha's record — **no searching, no index**.

Collision:
Sometimes two keys give the **same hash value**.
Example:

Emp_ID = 105 → 105 % 10 = 5
Emp_ID = 125 → 125 % 10 = 5

Both go to bucket 5 → this is called a **collision**.

Solution: We use methods like:
- **Chaining (overflow list)** → store both records in a linked list.
- **Open addressing** → find the next empty bucket.

Advantages
- **Very fast random access** (direct calculation of address).
- No need to scan sequentially or maintain index.

Disadvantages
- **Collisions** occur when multiple keys map to the same bucket.
- Not good for **sequential processing** (no order).

Feature	Heap (Pile)	Sequential	Indexed Sequential	Direct / Hash
Order of Records	No order	In sequence (by key)	In sequence + Index	Address by Hash Function
Insertion	Fast (append anywhere)	Costly (maintain order)	Moderate (with overflow)	Fast (direct address)
Search Speed	Slow (scan all)	Moderate (scan in order)	Fast (via index)	Very fast (hash lookup)
Random Access	Poor	Poor	Good	Excellent
Best For	Temporary/log data	Batch processing	Mixed access	Real-time lookups

B-Tree (Balanced Tree)

Definition

•A **B-Tree** is a **self-balancing search tree**.

•All leaves are at the **same level**.

•Nodes can have **multiple keys** and **multiple children**.

•Used to store **index keys** so that searches, insertions, and deletions are efficient.

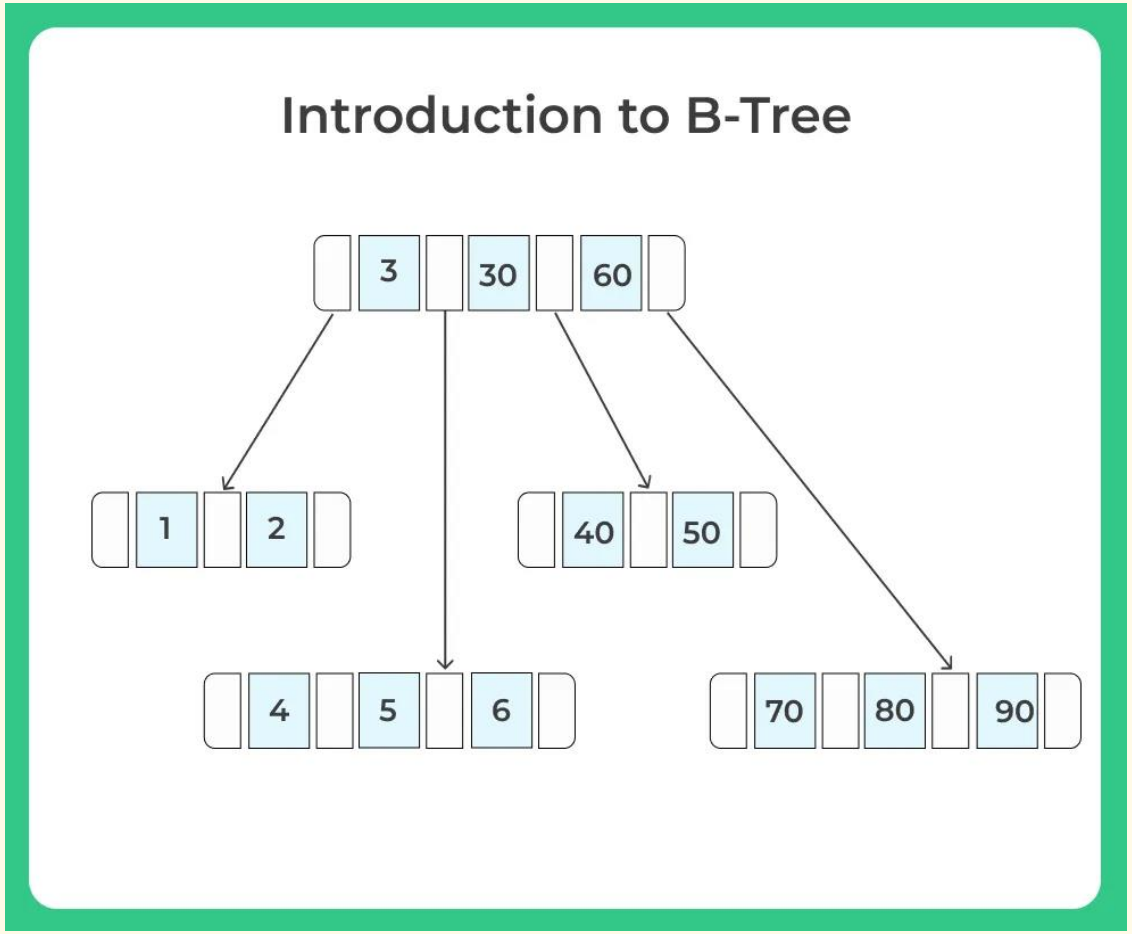
Structure of a B-Tree Node

•Contains **keys** (sorted)

•Contains **pointers** to child nodes

•Root node may also contain keys and pointers

Example (order = 3):



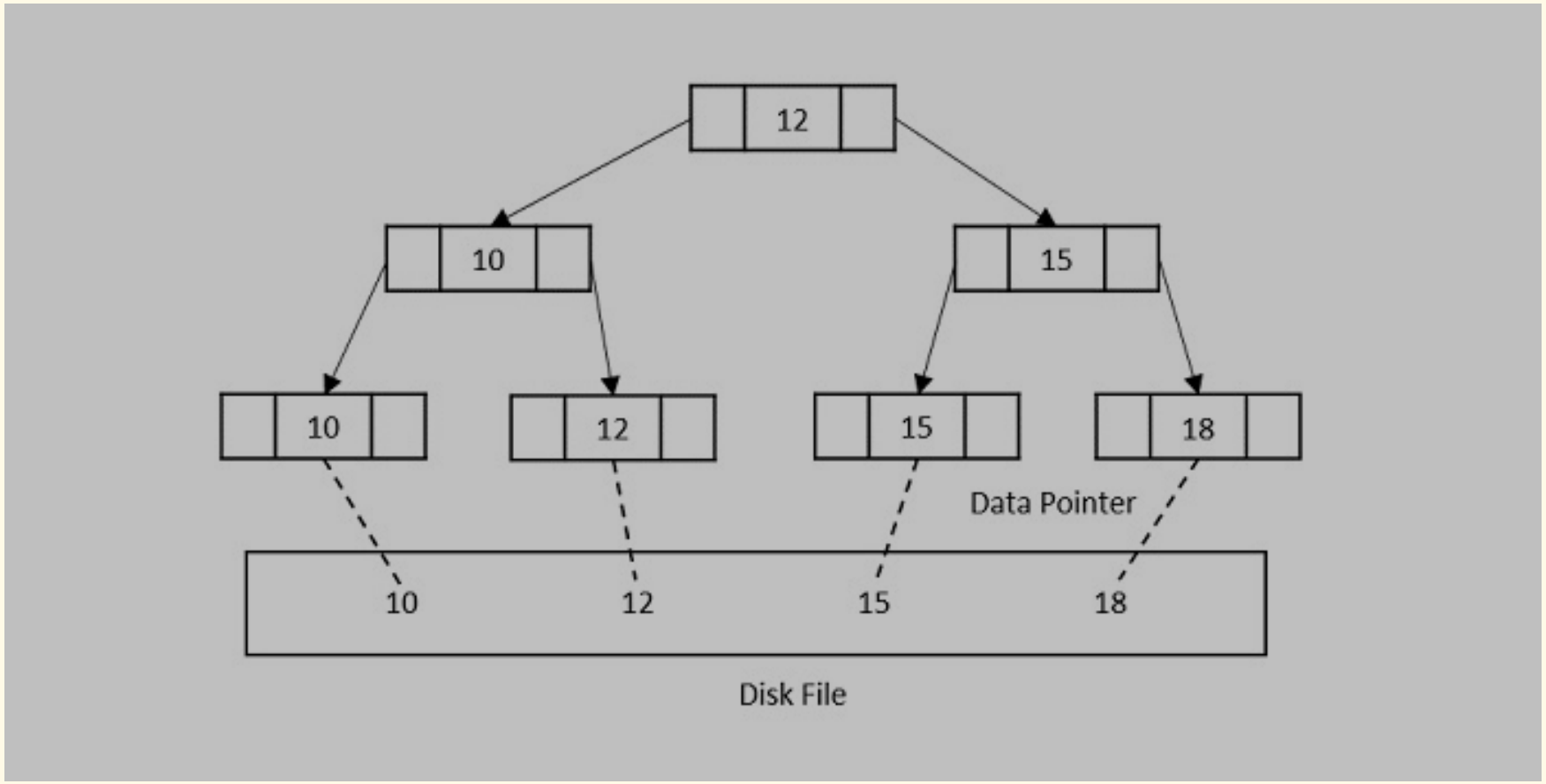
- The top row (root) has **3 keys**: **3, 30, 60**.
- Each key divides the tree into ranges.
- Numbers **less than 3** go to the first child.
- Numbers **between 3 and 30** go to the second child.
- Numbers **between 30 and 60** go to the third child.
- Numbers **greater than 60** go to the fourth child.
- **Children nodes** (lower rows) store the actual data:
- First child → 1, 2
- Second child → 4, 5, 6
- Third child → 40, 50
- Fourth child → 70, 80, 90

B++ Tree

- A **B++ Tree** is a **variant of the B-Tree(Improved version of B-Tree)**.
- In a B++ Tree, **all data (records)** are stored **only at the leaf level (last level)**.
- Internal nodes store only **keys (index)**, not actual data pointers.
- **Leaf nodes are linked** to make sequential access easy.

Structure of a B++ Tree Node

- Internal nodes = index keys only
- Leaf nodes = actual data pointers
- Leaves linked as a **linked list**



Insertion operation

The insertion to a B+ tree starts at a leaf node.

Step 1 − Calculate the maximum and minimum number of keys to be added onto the B+ tree node.

Insert 1, 2, 3, 4, 5, 6, 7 into a B+ Tree with order 4

Order = 4

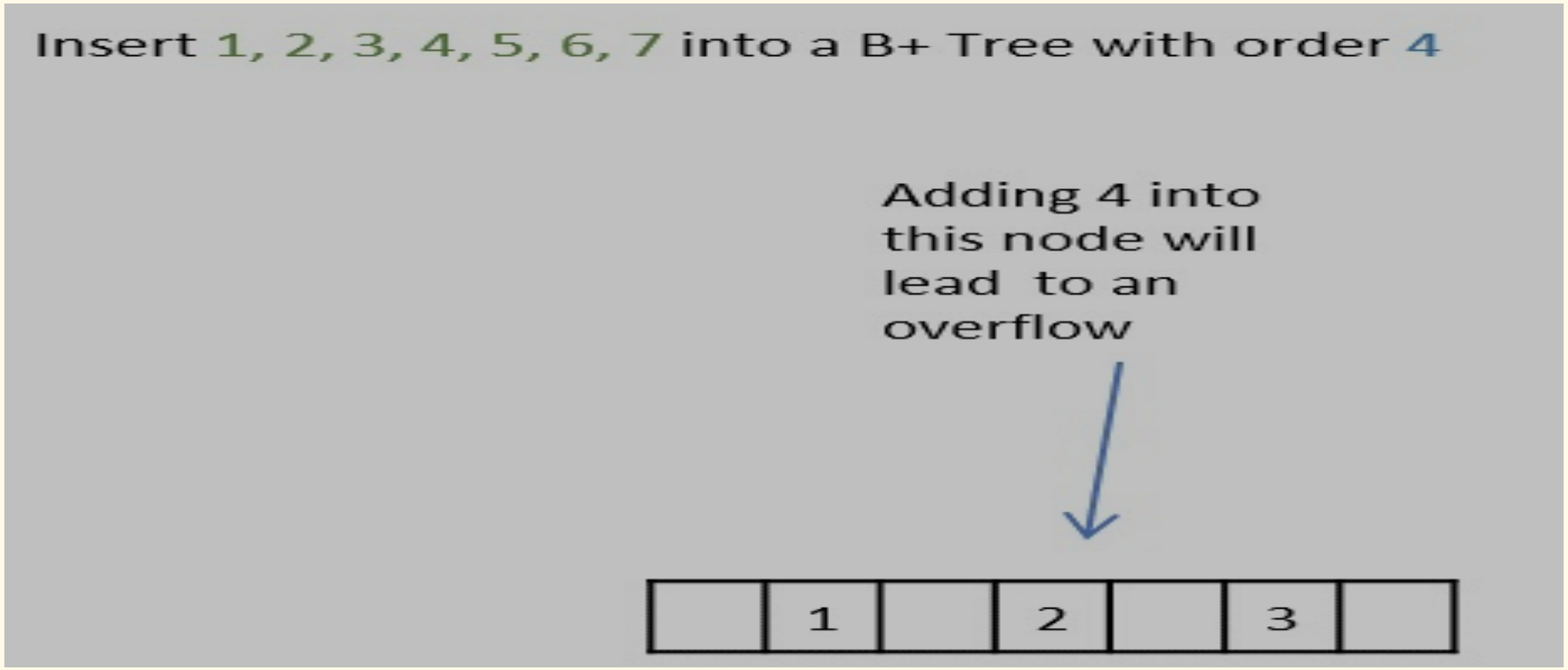
Maximum Children (m) = 4

Minimum Children ($\lceil \frac{m}{2} \rceil$) = 2

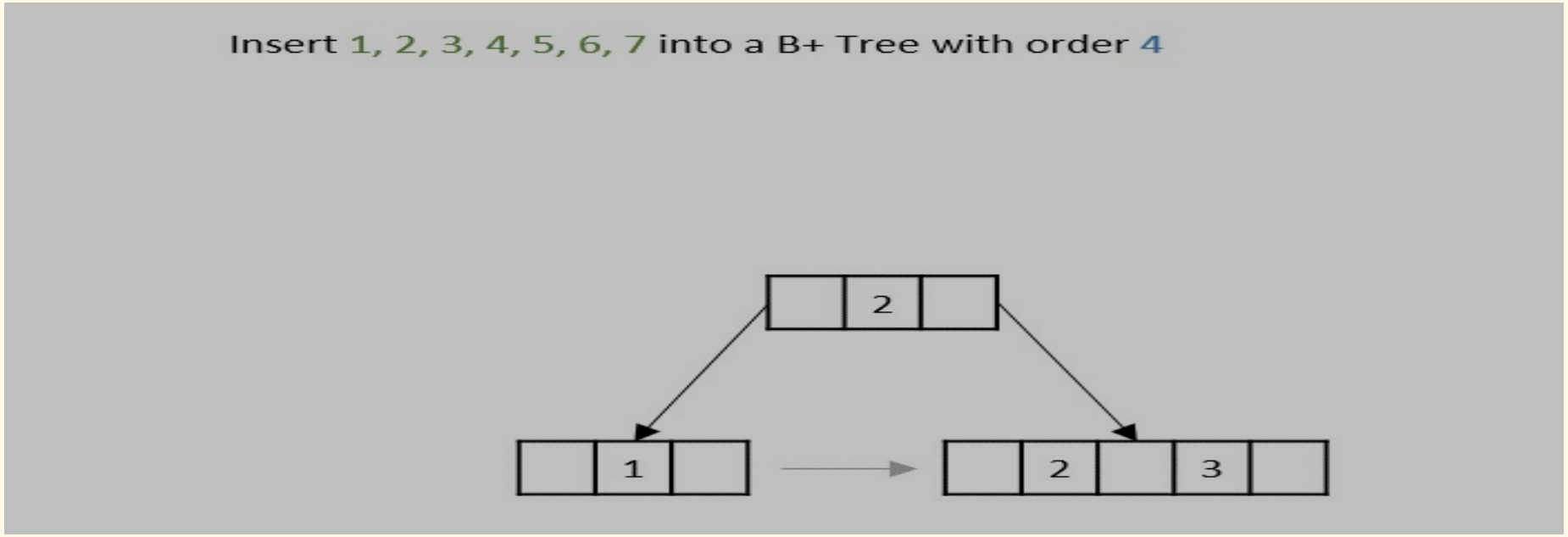
Maximum Keys (m - 1) = 3

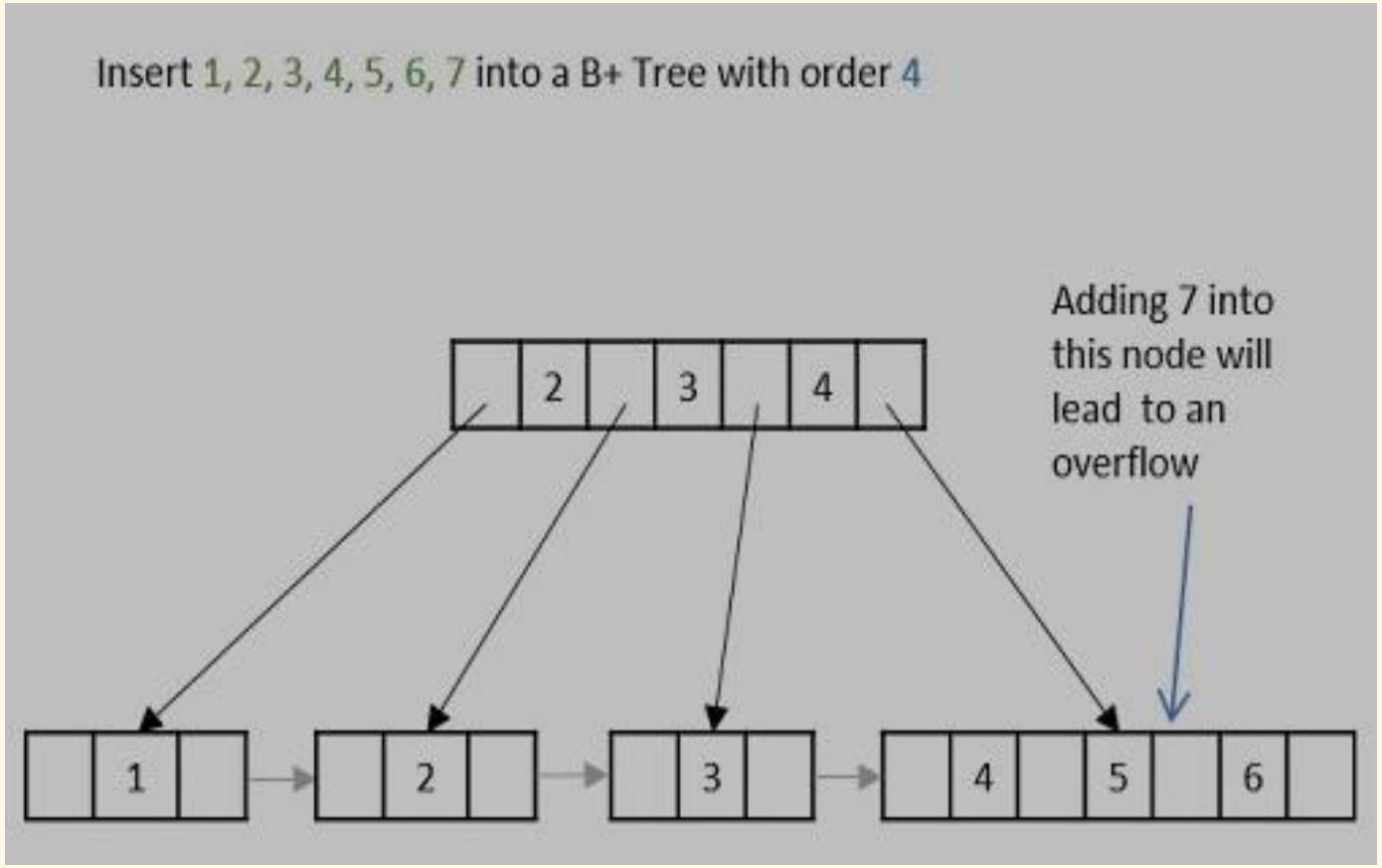
Minimum Keys ($\lceil \frac{m-1}{2} \rceil$) = 1

Step 2 − Insert the elements one by one accordingly into a leaf node until it exceeds the maximum key number.



Step 3 – The node is split into half where the left child consists of minimum number of keys and the remaining keys are stored in the right child.

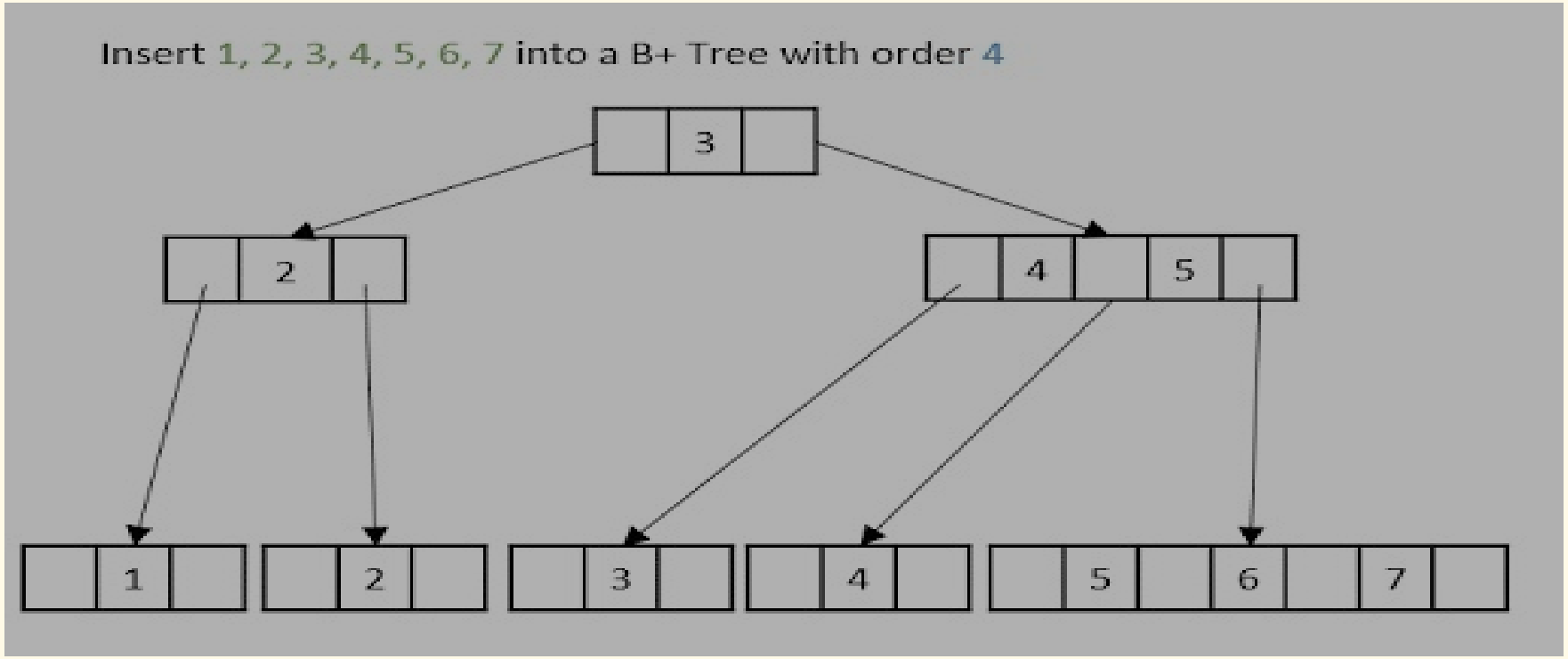




Step 4 If an internal node also gets more keys than allowed, then we divide it into two equal parts.

- The **left part** keeps the smaller keys.
- The **right part** keeps the larger keys.
- The **smallest key** from the right part becomes the **parent key** (moves up to the next level).

Step 5 If both leaf and internal nodes are full, split them the same way, and move the smallest key from the right part to the parent node.



Feature	B-Tree	B++ Tree
Data storage	Both internal & leaf nodes	Only leaf nodes
Internal nodes	Keys + data pointers	Keys only
Sequential access	Slower	Faster (linked leaves)
Range queries	Less efficient	Very efficient
Index size	Larger	Smaller

Hash function

A **hash function** is a function that takes an input (called a **key**) and converts it into a fixed-size number (called a **hash value** or **hash address**) which is used to store or find the data in a hash table.

Simple Example:

If table size = 10 and key = 123

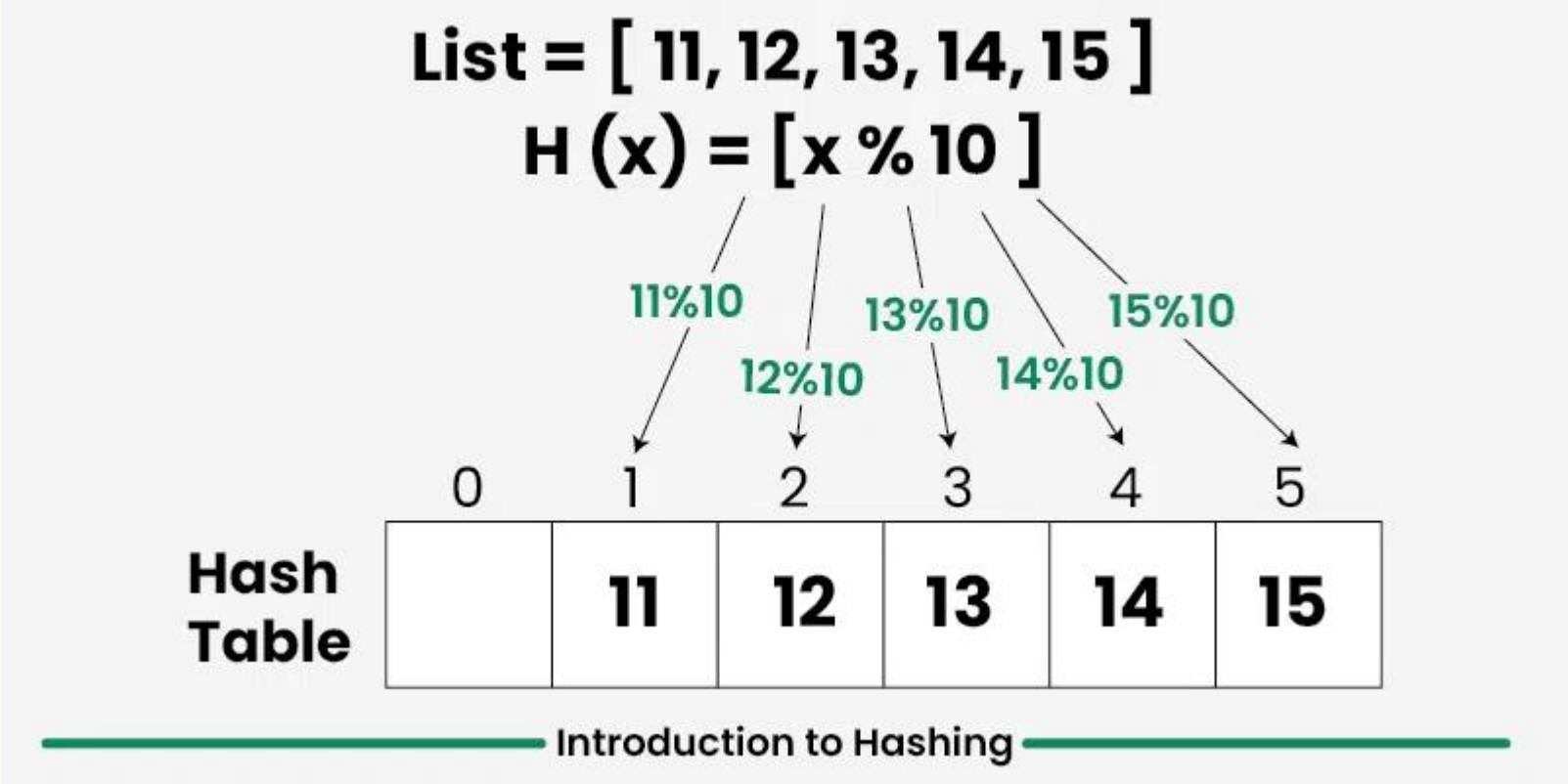
- Hash Function: `h(k) = k mod 10`
- `h(123) = 123 mod 10 = 3`

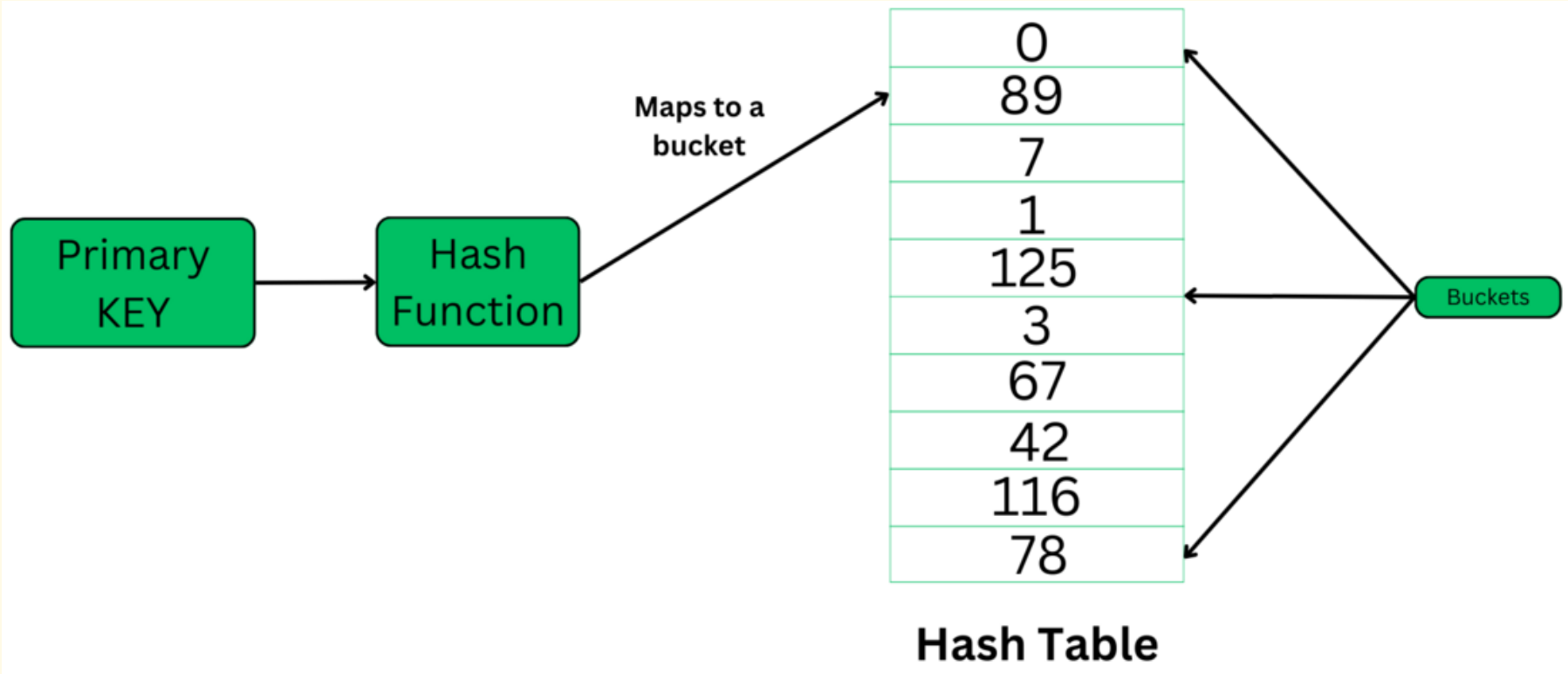
✅ Data stored at index 3 in the table.

```
10)123(12
   10
   ---
    23
    20
    --
     3 R
```

Types of Hash Functions

- **Division (Modulo) Method**
- **Multiplication Method.**
- **Mid-Square Method**
- **Folding Method**





1.Division (Modulo) Method

1. `h(k) = k mod m`
2. Take remainder after dividing key by table size.
3. Simple and fast.

Example
Division (Modulo) Method
Formula: `h(k) = k mod m`
•Example: Table size **m = 10**
•Key **k = 123** → `123 mod 10 = 3`
✅ **Hash address = 3**

2.Multiplication Method

1. Multiply key by a constant A (0 < A < 1), take fractional part × table size.
2. Reduces clustering.

Example

Formula: $h(k) = \lfloor m \times (k \times A \bmod 1) \rfloor$ (A between 0 and 1)

•Example: Table size **m = 10**, A = 0.618

•Key **k = 123**

- $123 \times 0.618 = 76.114$
- Fractional part = 0.114
- $10 \times 0.114 = 1.14$ → **1** (floor)

✅ **Hash address = 1**

3. Mid-Square Method

1. Square the key and take the middle digits as the hash value.
2. Gives more uniform distribution

Example

Idea: Square the key and take the middle digits.

•Example: Key **k = 123**

- $123^2 = 15129$
- Take middle two digits “51”

  ✅ **Hash address = 51 (or 1 if table size small)**

Types of Hash Functions

4.Folding Method

1. Break the key into parts and add them together to form the hash value.
2. Works well for large keys.

Example

Idea: Split key into equal parts and add them.

•Example: Key **k = 123456**, table size m = 100

- Split into parts: 12, 34, 56
- Add: 12 + 34 + 56 = 102
- `102 mod 100 = 2`

  ✅ **Hash address = 2**

Types of Hashing Approaches

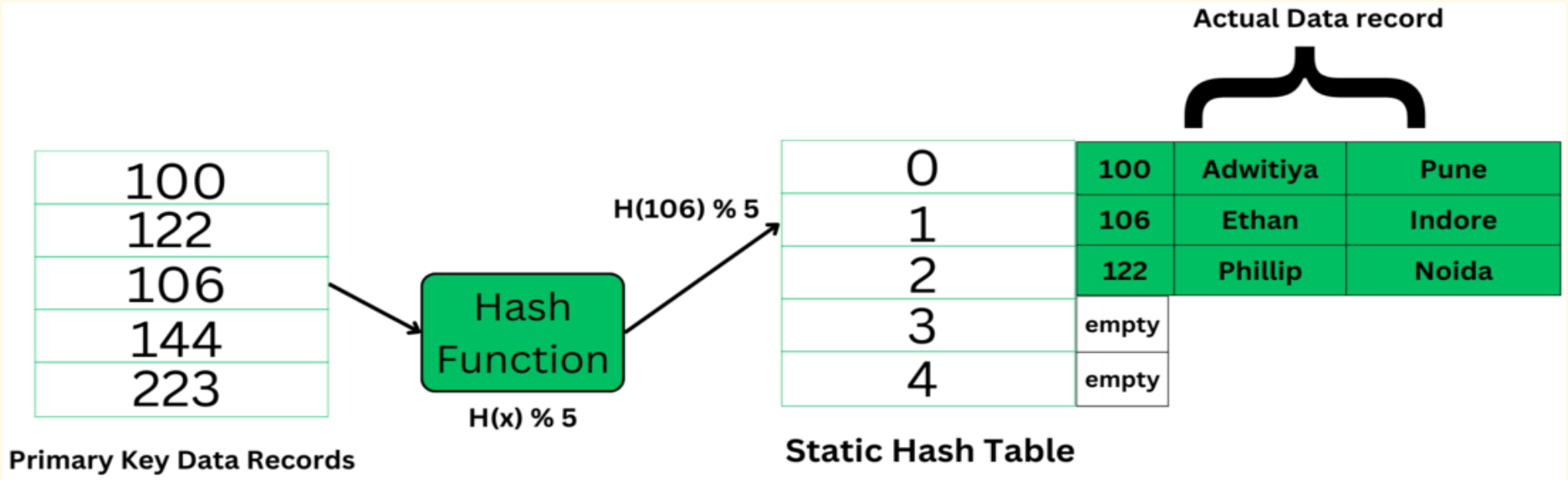
- Static Hashing
- Dynamic Hashing

1.Static Hashing

In static hashing, the hash function always generates the same bucket's address.

For example, if we have a data record for employee_id = 107, the hash function is mod-5 which is - H(x) % 5, where x = id.

H(106) % 5 = 1.



Types of Operations in Static Hashing

Static hashing keeps a fixed-size hash table. The main operations are:

1.Insertion

- Use the hash function to compute the address for the new key.
- Place the record at that address.
- If a collision occurs, use overflow area or chaining.
- ✅ *Example:* Key 123 → `123 mod 10 = 3` → store at index 3.

2.Search (Retrieval)

- Compute the hash address using the hash function.
- Go to that address and fetch the record.
- ✅ *Example:* To find key 123 → compute `123 mod 10 = 3` → check index 3.

3.Deletion

•Compute the hash address of the key.

•Remove the record from that location (or mark it deleted).

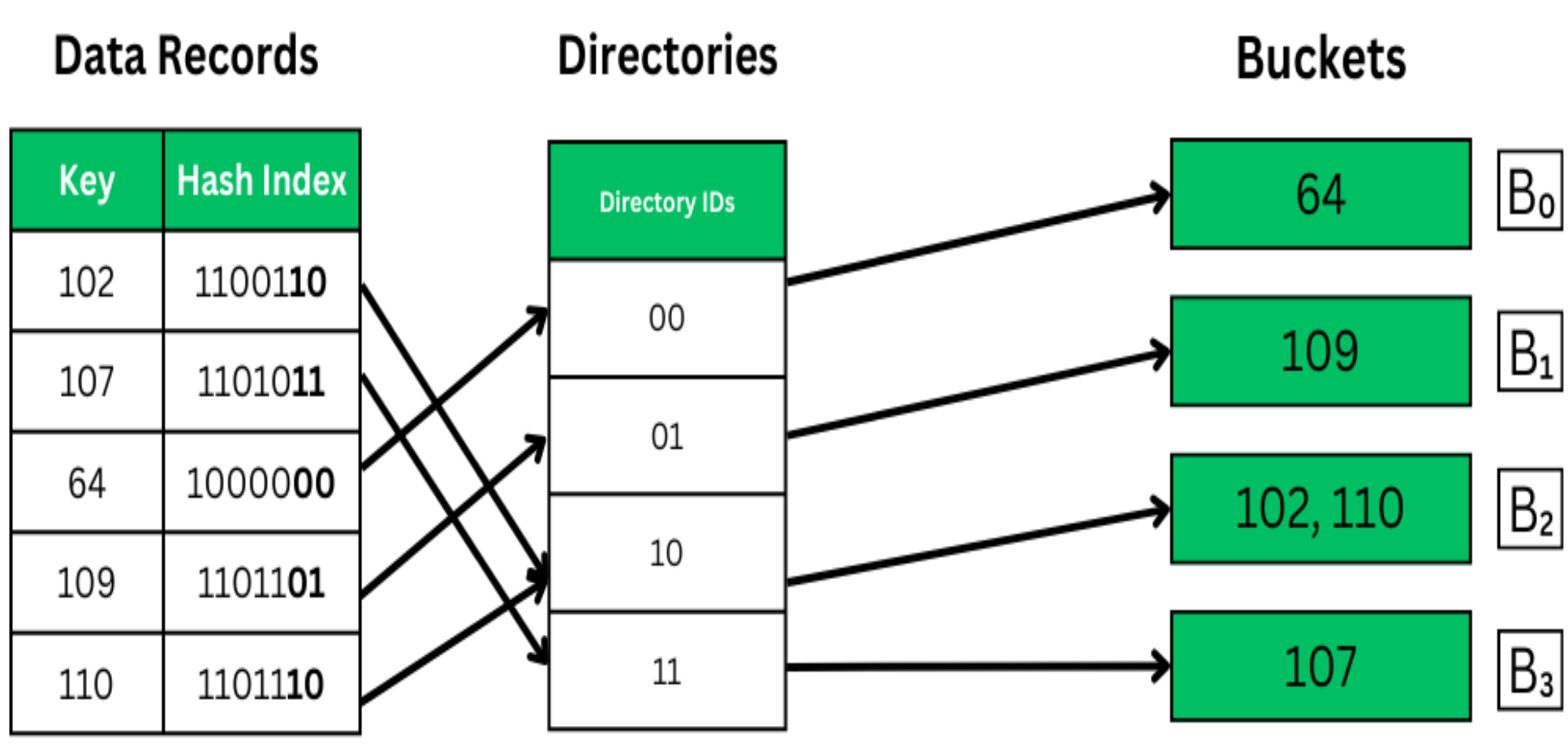
• ✅ *Example:* Key 123 → compute index 3 → delete the entry there.

2.Dynamic Hashing

Dynamic hashing is also known as extendible hashing, used to handle database that frequently changes data sets. This method offers us a way to add and remove data buckets on demand dynamically.

Working of Dynamic Hashing

Example: If global depth: k = 2, the keys will be mapped accordingly to the hash index. K bits starting from LSB will be taken to map a key to the buckets. That leaves us with the following 4 possibilities: 00, 11, 10, 01.



Dynamic Hashing Approach

1.Extendible Hashing –Extendible Hashing uses a **directory of pointers** to buckets.

- Each directory entry corresponds to a binary prefix of the hash value.
- When a bucket overflows, it **splits**, and sometimes the **directory doubles**.
- This allows **fast searching and dynamic growth**.

Initial Setup

- Directory has **1 bit** (2 entries: 0 and 1).
- Hash function: use binary of `h(k) mod 4`.
- Buckets:
  - Bucket 0: keys ending in 0
  - Bucket 1: keys ending in 1

Insert Keys

Let's insert keys: **1, 2, 3, 4**.

- 1 → binary `01` → directory 1 → bucket 1
- 2 → binary `10` → directory 0 → bucket 0
- 3 → binary `11` → directory 1 → bucket 1
- 4 → binary `100` → directory 0 → bucket 0 (overflow now)

Bucket Split

•Bucket 0 overflows → split into two buckets (00 and 10).

•Directory now has **2 bits** (4 entries: 00, 01, 10, 11).

Redistribute keys:

•2 → 10 → bucket 10

•4 → 100 → bucket 00

Directory now points to:

•00 → bucket A

•01 → bucket 1

•10 → bucket B

•11 → bucket 1

0	0	0
0	0	1
0	1	0
0 1 1	1 0 0	1 0 1

Insert More Keys

Insert **5 (101)** → goes to bucket 01

Insert **6 (110)** → goes to bucket 10

Insert **7 (111)** → goes to bucket 11

2.Linear Hashing

- Splits **one bucket at a time** gradually instead of doubling the directory.
- Reduces sudden large changes in memory size.
- Directory is not needed separately; uses a level and a next bucket pointer.
- ✅ *Example:* Each insertion may trigger a single bucket split, not the whole table.

Initial: a mod b= remainder when a/b
h0(k) = k mod 2
Buckets:
[0] →
[1] →
Step 2:
Insert 2,3,4:
Insert 2 → 2 mod 2= 2/2=remainder=0= 0 → goes to bucket 0
Insert 3→ 3 mod 2=2/3=remainder=1= 1→ goes to bucket 1
Insert 4 → 4 mod 2=2/4=remainder=0= 0 → bucket 0 again
[0] → 2,4 (overflow if only 1 record allowed per bucket)
[1] → 3

Split bucket 0 (use h1(k)=k mod 4):
New bucket [2] created
Redistribute:
2 mod 4 =4/2=2→goes to bucket [2]
4 mod 4 = 4/4=remainder=0=0→ stays in bucket [0]
[0] → 4
[1] → 3
[2] → 2

Insert 7,8: 7mod2=1 → goes to bucket [1] ,
8mod2 =0→ goes to bucket [0]
[0] → 4,8 (overflow again)
[1] → 3,7
[2] → 2

In **Hashing**, hash functions were used to generate hash values.
The hash value is used to create an index for the keys in the hash table.
The hash function may return the same hash value for two or more keys.
When two or more keys have the same hash value, a **collision** happens.
To handle this collision, we use **Collision Resolution Techniques**.

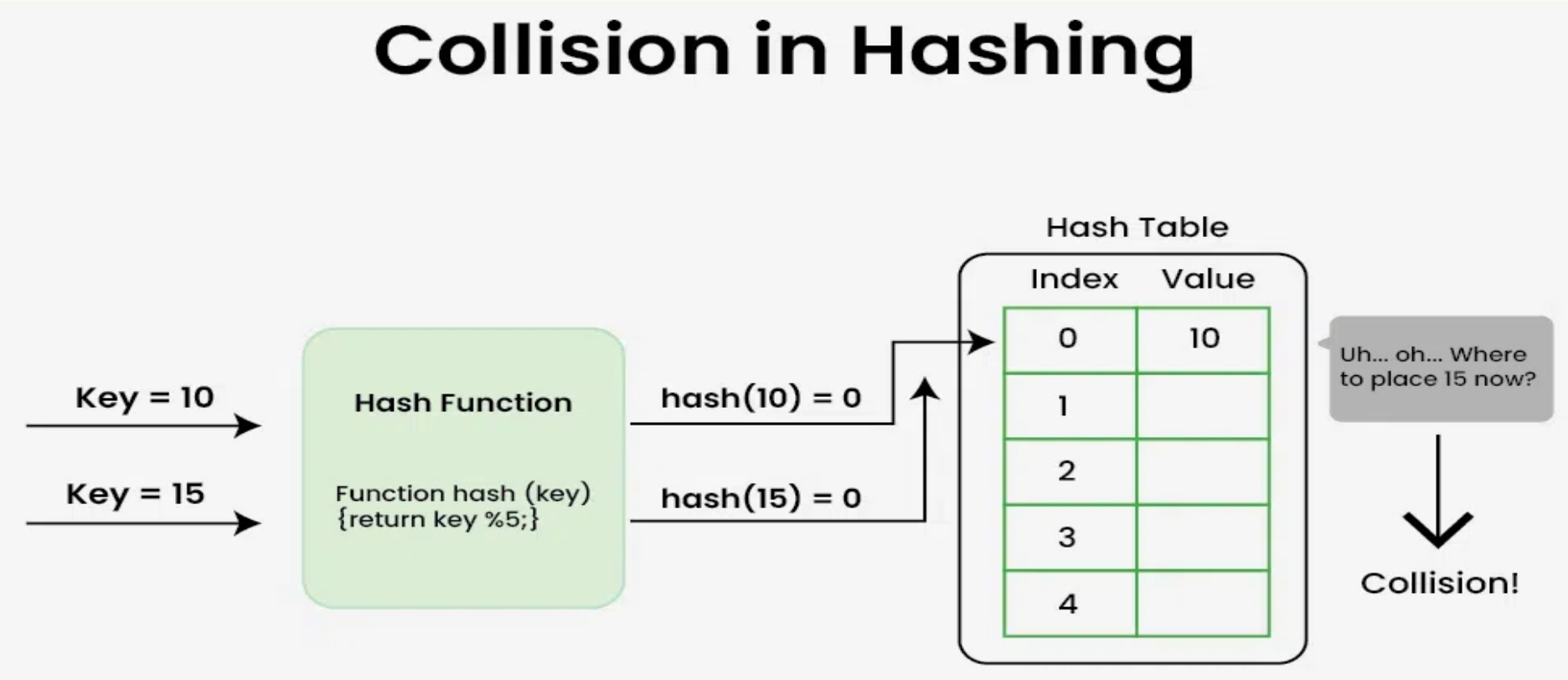
Collision Resolution Techniques

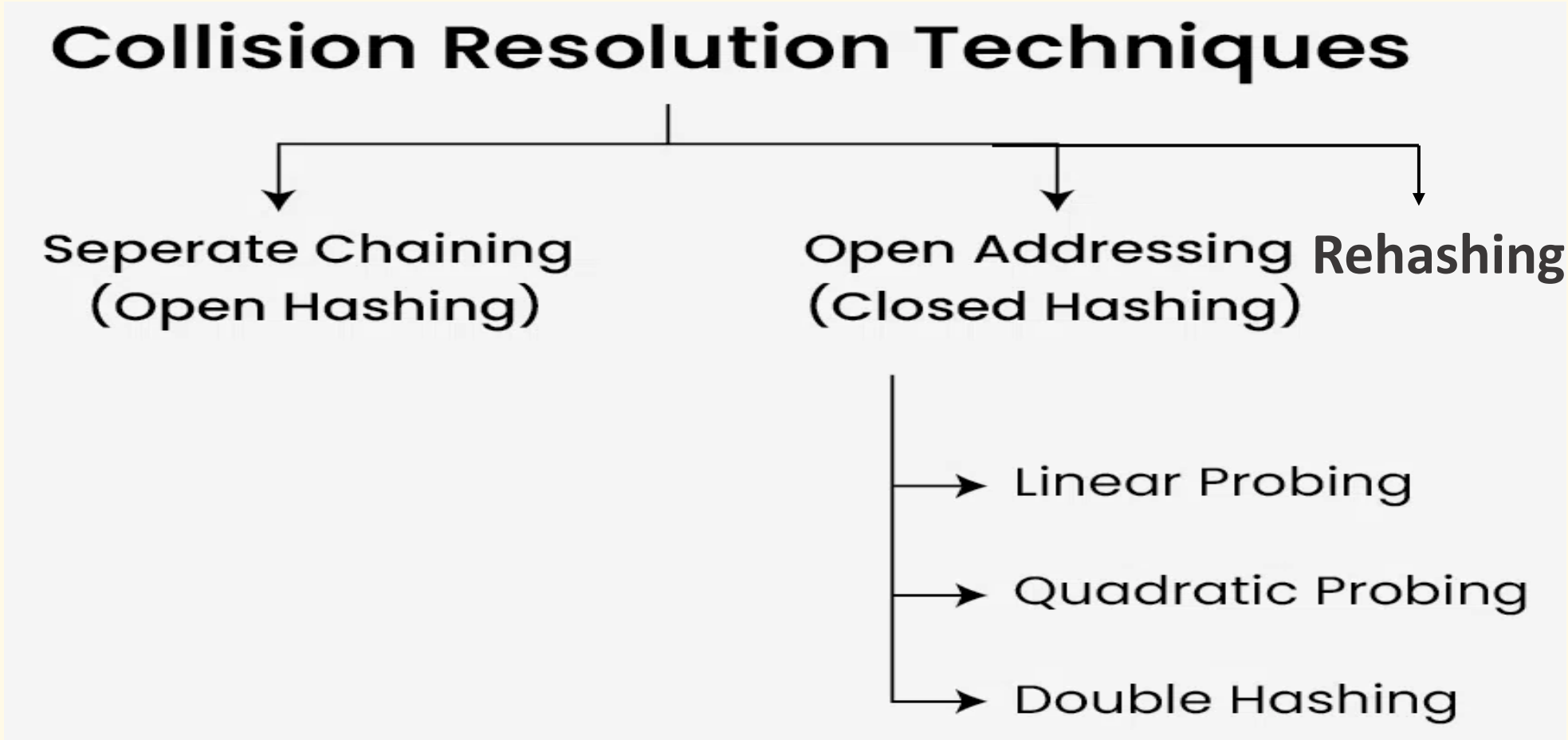
There are mainly two methods to handle collision:

1.Separate Chaining

2.Open Addressing

Collision Resolution Techniques





2. Chaining (Separate Chaining)

Chaining is a mechanism in which the hash table is implemented using an array of type nodes, where each bucket is of node type and can contain a long chain of linked lists to store the data records.

Key	Hash Index (key % 5)	Inserted At
10	0	Bucket 0 → [10]
15	0	Bucket 0 → [10 → 15]
20	0	Bucket 0 → [10 → 15 → 20]
25	0	Bucket 0 → [10 → 15 → 20 → 25]
30	0	Bucket 0 → [10 → 15 → 20 → 25 → 30]
11	1	Bucket 1 → [11]

Given:

- Hash table size = **5**
- Hash function: h(key) = key % 5
- Keys to insert: **10, 15, 20, 25, 30, 11**

Final Hash Table (with separate chaining):

Index	Linked List (Bucket)
0	10 → 15 → 20 → 25 → 30
1	11
2	--
3	--
4	--

3. Open Addressing (Closed Hashing)

This is also called closed hashing this aims to solve the problem of collision by looking out for the next empty slot available which can store data. It uses techniques like **linear probing**, **quadratic probing**, **double hashing,** etc.

Example:

•Hash table size = **7**

•Hash function: h(key) = key % 7

•Collision resolution: **Linear Probing**

Insert the keys: 50, 700, 76, 85, 92, 73

Key	Hash (key % 7)	Insert At	Collision?	Final Position (after probing)
50	50 % 7 = 1	1	No	1
700	700 % 7 = 0	0	No	0
76	76 % 7 = 6	6	No	6
85	85 % 7 = 1	1	Yes	2 (next slot)
92	92 % 7 = 1	1	Yes	3 (after 1 and 2 are filled)
73	73 % 7 = 3	3	Yes	4 (next slot after 3)

Example:

•Original table size = 5

•Keys = {12, 18, 13}

•Hash function: `h(k) = k mod 5`

Positions:

•12 → 2

•18 → 3

•13 → 3 (collision)

Load factor high → Rehash to size 10

New hash: `h'(k) = k mod 10`

•12 → 2

•18 → 8

•13 → 3

✅ Now fewer collisions.

Rehashing

Rehashing means **changing the size of a hash table and re-computing (rehashing) the positions of all existing keys** using a new hash function.

•It's done when the hash table becomes **too full** (many collisions) or **too empty** after deletions.