Hashing in Data Structure

Hash Tables and Hashing

- The implementation of hash tables is called **hashing**.
- **Hashing** is a technique used for performing:
  - Insertions
  - Deletions
  - Searches

  in **constant average time** (i.e., $O(1)$).

Limitation

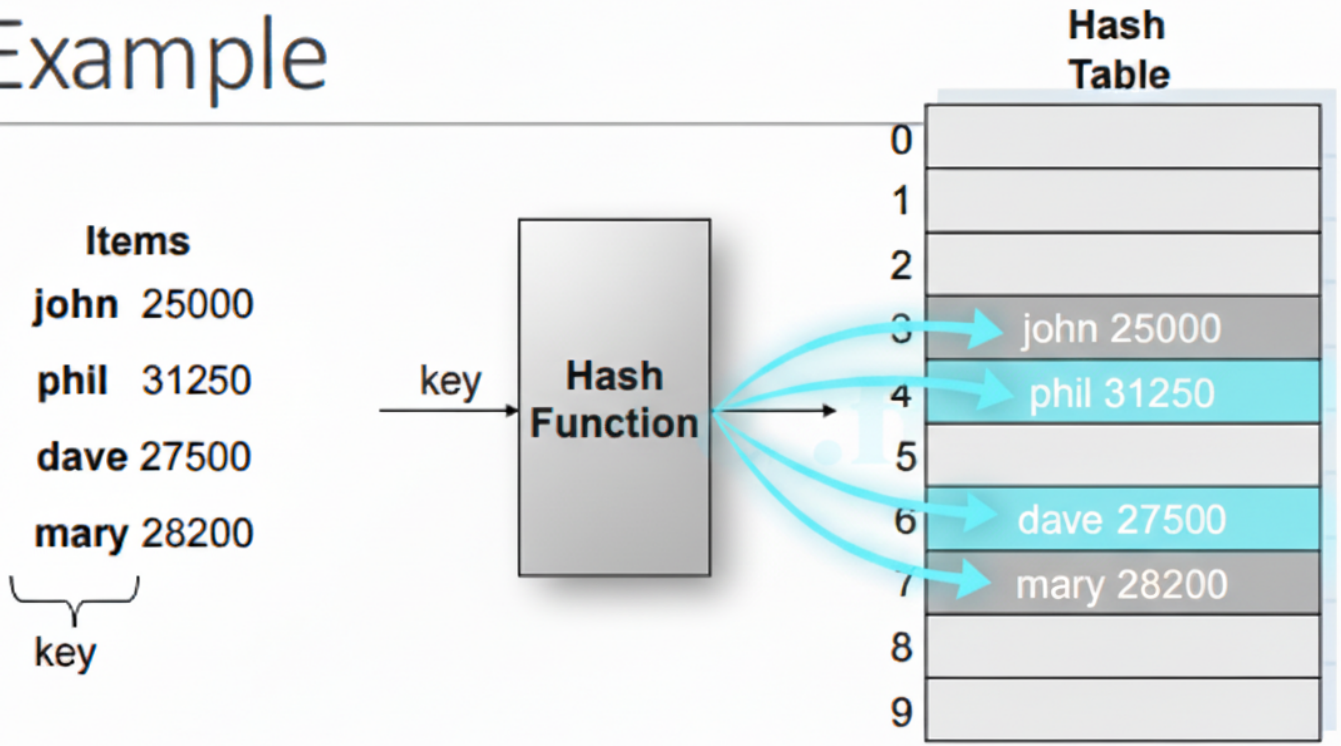
This data structure, however, is **not efficient** in operations that require any ordering information among the elements, such as:

- findMin
- findMax
- Printing the entire table in sorted order.

General Idea of Hash Tables

- The ideal hash table structure is merely an **array of some fixed size**, containing the items.
- A stored item needs to have a data member, called **key**, that will be used in computing the index value for the item.
  - Key could be an **integer**, a **string**, etc.
  - e.g. a name or Id that is a part of a large employee structure.
- The size of the array is TableSize.
- The items that are stored in the hash table are indexed by values from 0 to TableSize – 1.
- Each key is mapped into some number in the range 0 to TableSize – 1.
- The mapping is called a **hash function**.

Example


Items
john 25000
phil 31250
dave 27500
mary 28200
key
key
Hash Function
Hash Table
0
1
2
3
4
5
6
7
8
9
john 25000
phil 31250
dave 27500
mary 28200

What is a Hash Function?

Definition

Hash function: A function that governs the mapping of key values to indices (slots) of a hash table.

Key Requirements

The hash function must be:

- **Simple, easy, and quick** to compute.
- Designed to distribute the keys **evenly** among the cells.

Key Properties of Hash Functions

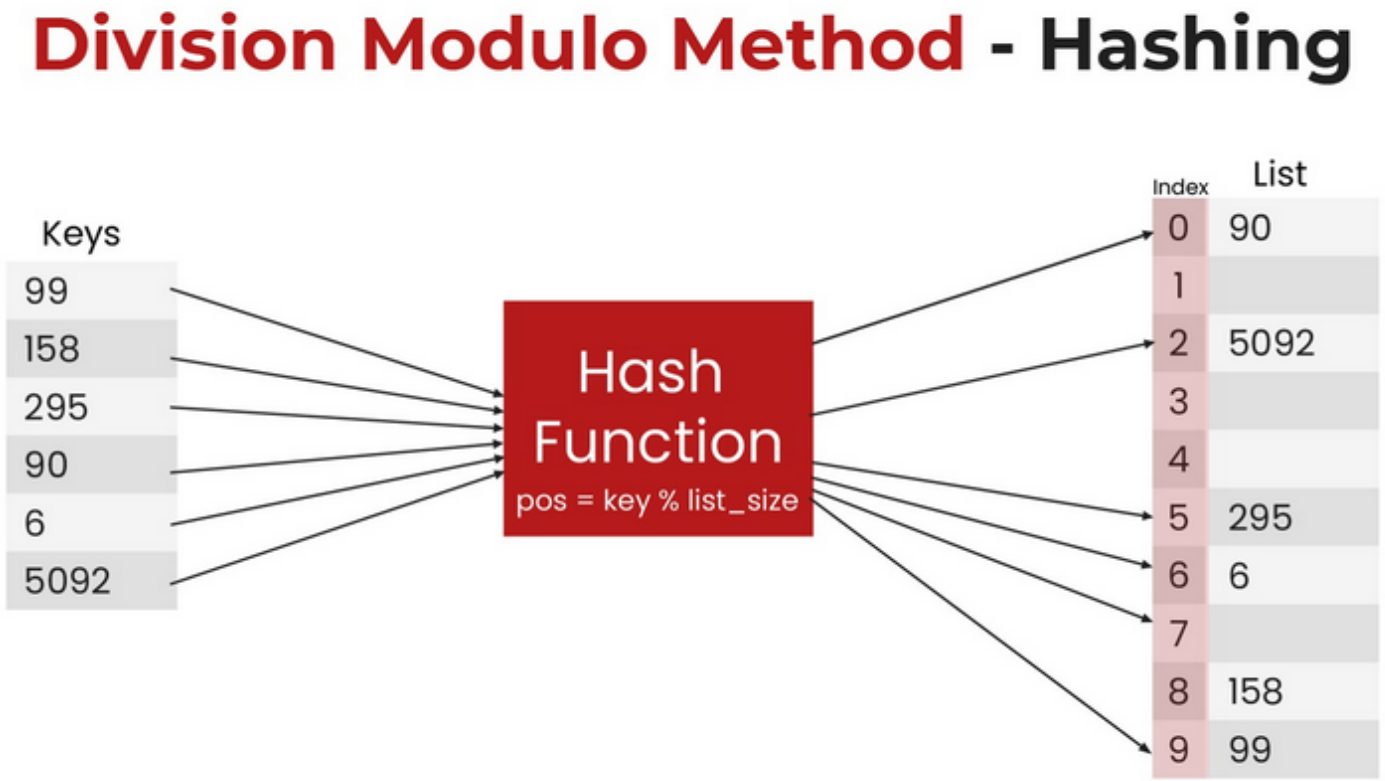
- **Deterministic:** A hash function must consistently produce the same output for the same input.
- **Fixed Output Size:** The output of a hash function should have a fixed size, regardless of the size of the input.
- **Efficiency:** The hash function should be able to process input quickly.
- **Uniformity:** The hash function should distribute the hash values uniformly across the output space to avoid clustering.
- **Pre-image Resistance:** It should be computationally infeasible to reverse the hash function, i.e., to find the original input given a hash value.
- **Collision Resistance:** It should be difficult to find two different inputs that produce the same hash value.
  ssem
- **Avalanche Effect:** A small change in the input should produce a significantly different hash value.

Common Hashing Methods

Four Basic Hashing Techniques

1. **Division Method**
   - Uses the modulo operator (key $\pmod{M}$) to calculate the index.
2. **Multiplication Method**
   - Multiplies the key by a constant A $(0 < A < 1)$, takes the fractional part, and then multiplies by the table size M.
3. **Mid-Square Method**
   - Squares the key and extracts the middle digits of the result as the index.
4. **Folding Method**
   - Divides the key into several parts, adds or XORs the parts together, and then uses a suitable index calculation method on the result.

Division method



Multiplication method

- **Deterministic**: Same input $\rightarrow$ same output.
- **Fixed Output Size**: Output size is constant, regardless of input size.
- **Efficiency**: Must process input quickly.
- **Uniformity**: Distribute hashes uniformly to avoid clustering.
- **Pre-image Resistance**: Hard to reverse (find input from hash).
- **Collision Resistance**: Hard to find two inputs with the same hash.

Division Method Disadvantage

The division method's performance is heavily dependent on:

- The **size of the table** (m).
- The **choice of m** significantly affects the hash function's performance.

Multiplication Method Note

The multiplication method hash function does not depend on the table size (m) as much as the division method hash function.

Example of a Bad Choice for A

- If $m = 100$ and $A = 1/3$, then
- for $k = 10$, $h(k) = 33$,
- for $k = 11$, $h(k) = 66$,
- And for $k = 12$, $h(k) = 99$.
- This is not a good choice of A, since we'll have only three values of $h(k) \ldots$

Optimal Choice of A

- The optimal choice of A depends on the keys themselves.
- Knuth claims that A near the golden ratio conjugate is likely to be a good choice:

$$A \approx \frac{\sqrt{5}-1}{2} = 0.6180339887\ldots$$

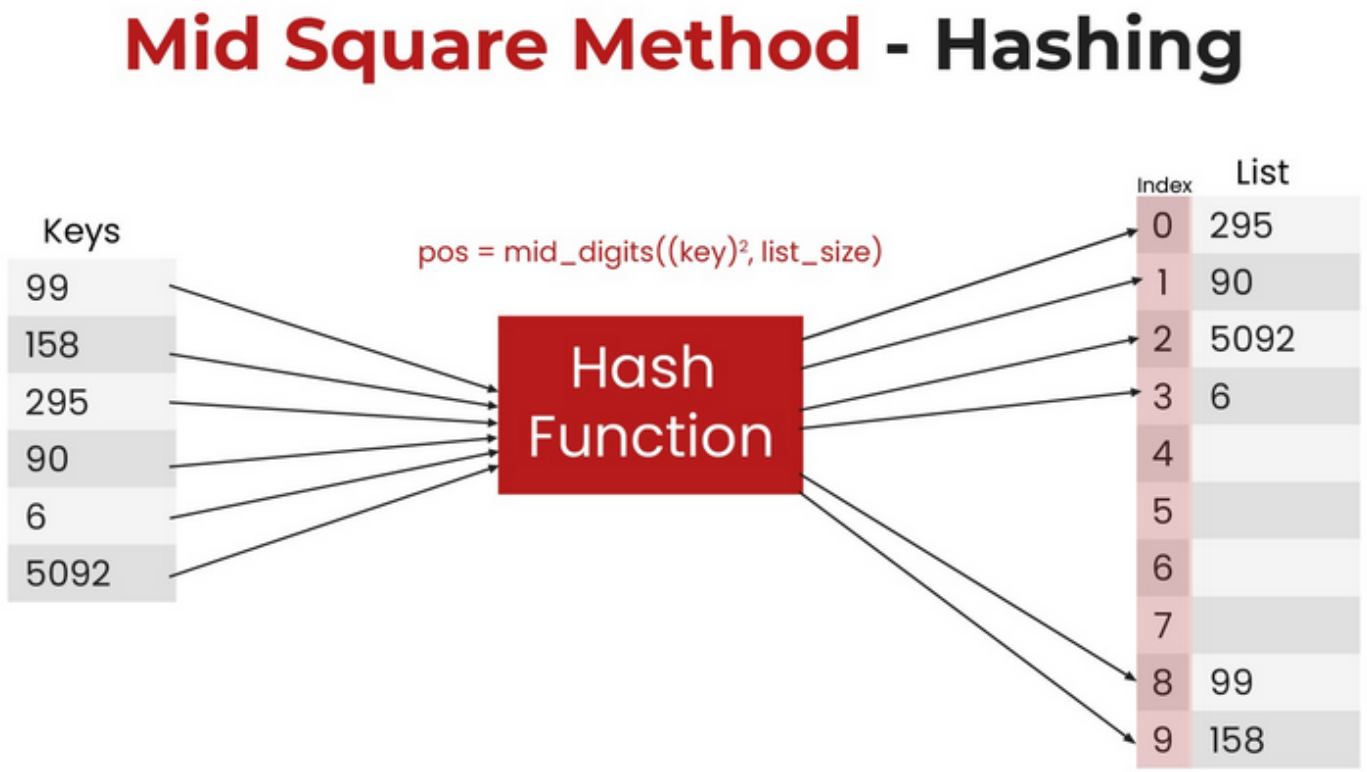
The multiplication method

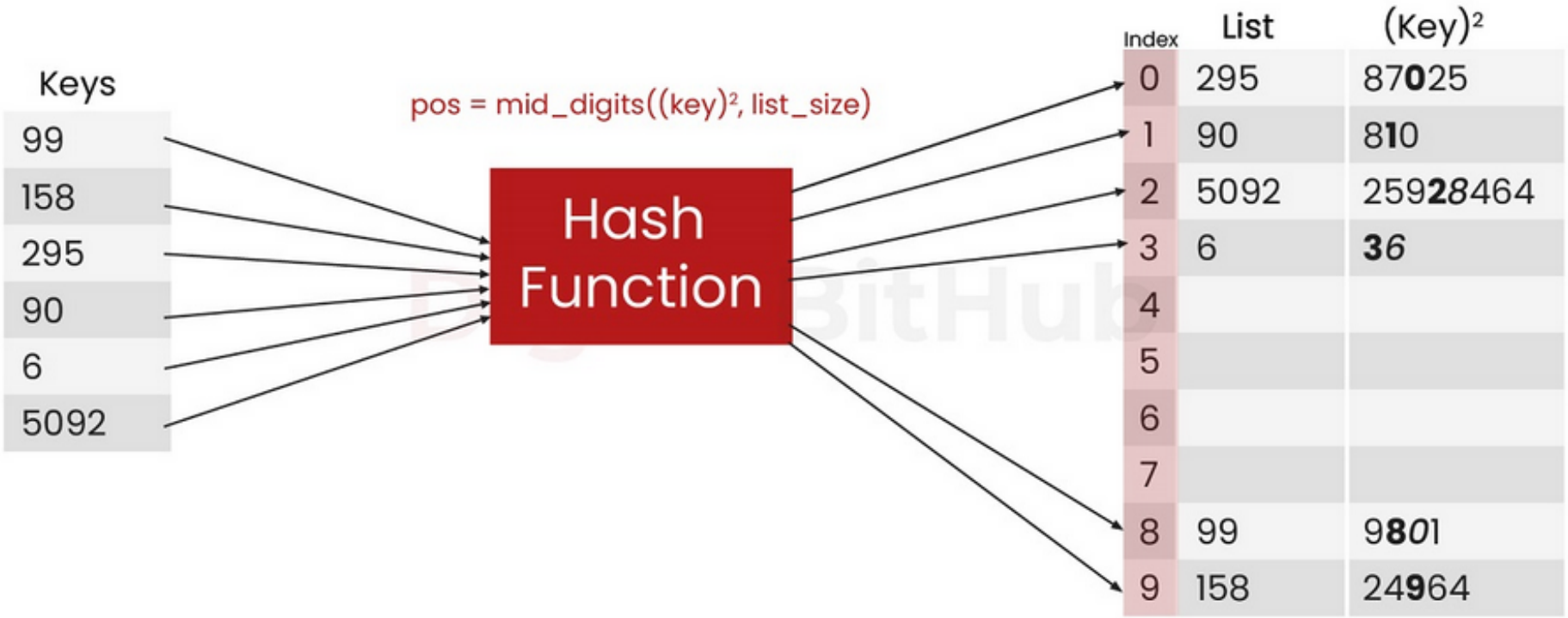
- A good choice of A, example:
  - if m = 1000
  - and $A \approx (\sqrt{5} - 1)/2 = 0.6180339887\ldots$, then
  - for k=61, h(k)=700,
  - for k=62, h(k)=318,
  - For k=63, h(k)=936
  - And for k=64, h(k)=554.

Mid-Square Method

- **Mid-square:** $h(k) =$ middle digits of k^2.
- Sometimes it is impractical to use a table whose size is a prime number.
- This hash function can be used to reduce the probability of collisions when table size $N = b^e$ (a power of some base b).
- **Example 1:** Table size is a power of 10.
  - $H(4150130) = 21526\ \mathbf{4436}\ 17100$
  - $H(415013034) = 526447\ \mathbf{3522}\ 151420$
  - $H(1150130) = 13454\ \mathbf{2361}\ 7100$
- **Example 2:** Table size is a power of 2 (binary keys).
  - $h(1001) = 10\ \mathbf{100}\ 01$
  - $h(1011) = 11\ \mathbf{110}\ 01$
  - $h(1101) = 101\ \mathbf{010}\ 01$

Mid-Square method




Keys
99
158
295
90
6
5092
pos = mid_digits((key)², list_size)
Hash Function
Index
List
(Key)²
0
295
87025
1
90
810
2
5092
25928464
3
6
36
4
5
6
7
8
99
9801
9
158
24964

Digit Folding Method

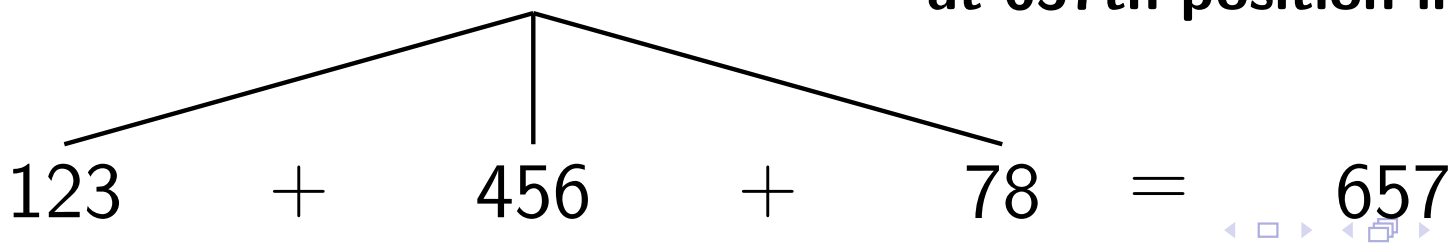
- Here, the key is divided into separate parts. These separated parts are then combined using simple operations to produce a hash.
- **Example:** Consider a record of key 12345678.
  - Divide this into parts: 123, 456, 78.
  - Combine these parts by performing an add operation:

$$h(\text{key}) = h(12345678) = 123 + 456 + 78$$
$$= 657$$

Hash Function: Digit Folding Method

Key = 12345678

Place the record of key 12345678 at 657th position in the hash table



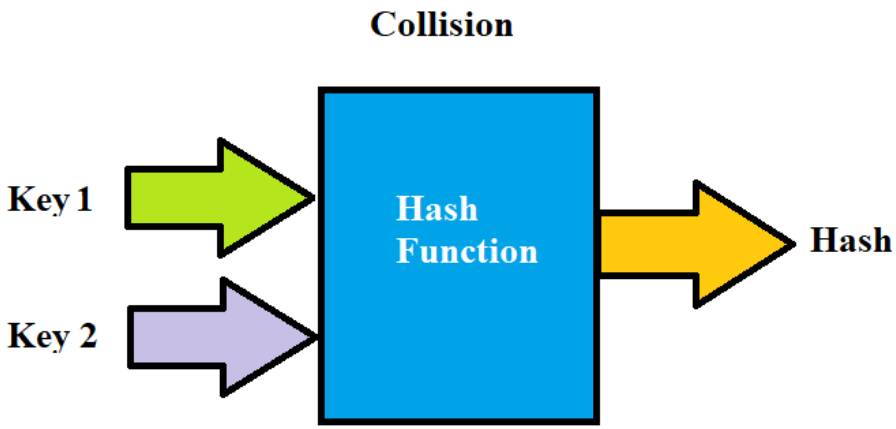
Collisions

Definition

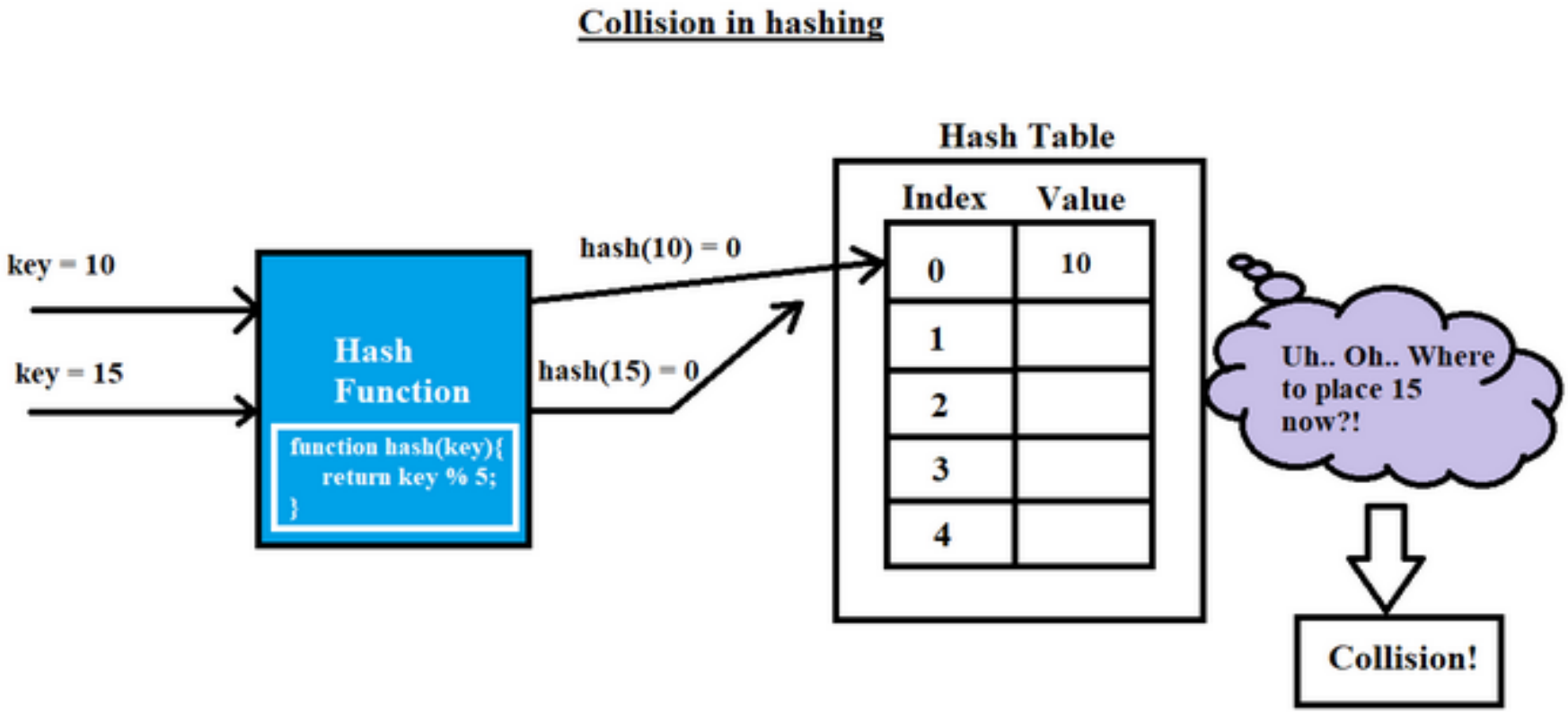
The phenomenon where **two keys generate the same hash value** from a given hash function is called a collision.

A good hash function should ensure that collisions are minimum.

Collision Illustration



Collision

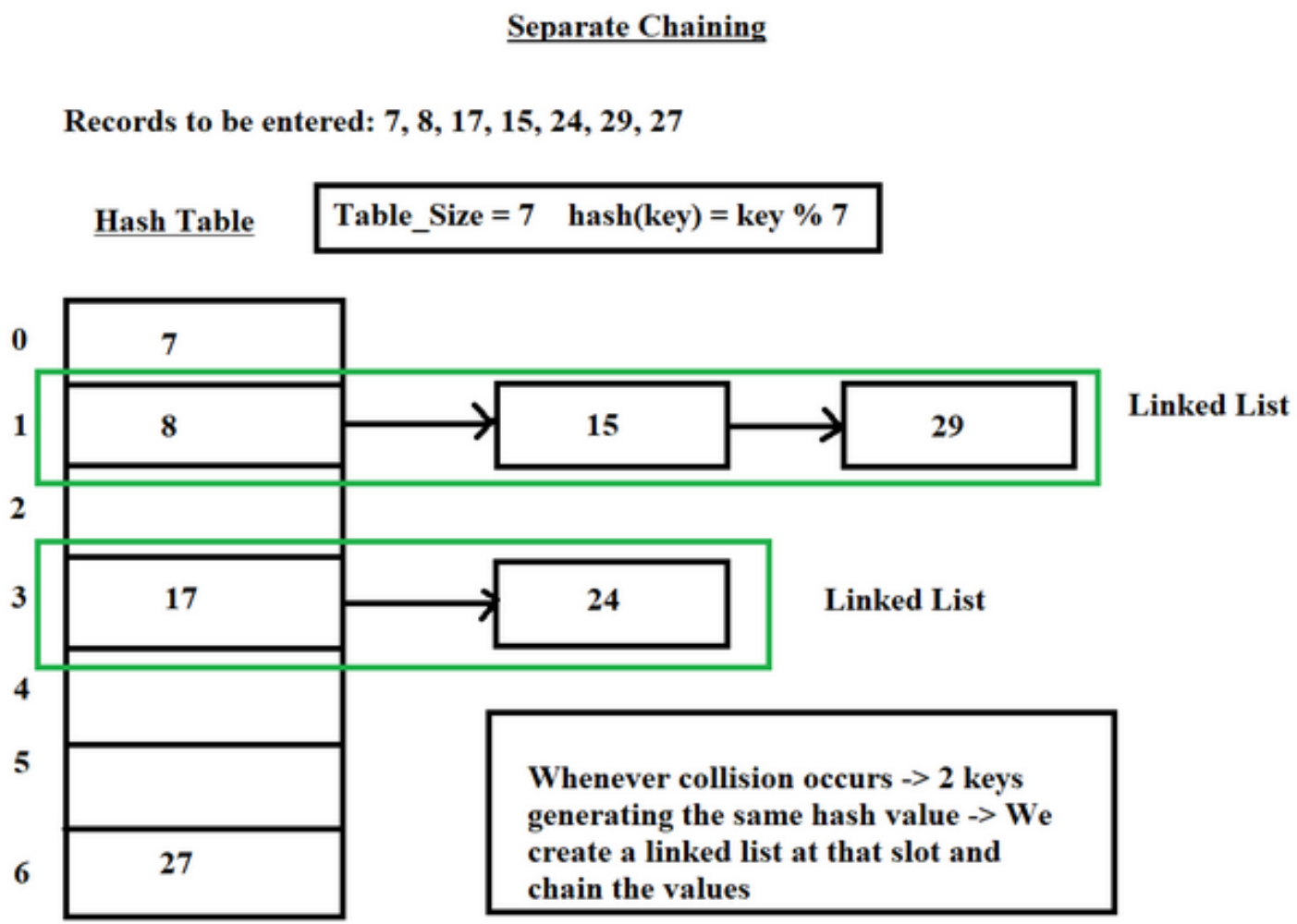

Collision in hashing
Hash Table
Index
Value
key = 10
key = 15
Hash Function
function hash(key){
return key % 5;
}
hash(10) = 0
hash(15) = 0
0
10
1
2
3
4
Uh.. Oh.. Where to place 15 now?!
Collision!

Collision Resolution Techniques

Two Main Categories for Handling Collisions

1. **Open Addressing**
   - When a collision occurs, the system probes for an alternative empty slot within the hash table itself.
   - Examples: Linear Probing, Quadratic Probing, Double Hashing.
   - Data is stored directly in the hash table.
2. **Closed Addressing (Separate Chaining)**
   - Each slot in the hash table points to a data structure (e.g., a linked list or another small hash table) that holds all keys hashing to that slot.
   - Elements with the same hash value are stored externally to the main table.

Separate chaining



Advantages of Separate Chaining:

- This technique is very simple in terms of implementation.
- We can guarantee that the insert operation always occurs in $\mathbf{O(1)}$ time complexity as linked lists allows insertion in constant time.
- We need not worry about hash table getting filled up. We can always add any number elements to the chain whenever needed.
- This method is less sensitive or not very much dependent on the hash function or the load factors.
- Generally this method is used when we do not know exactly how many and how frequently the keys would be inserted or deleted.

Disadvantages of Separate Chaining:

- Chaining uses extra memory space for creating and maintaining links.
- It might happen that some parts of the hash table will never be used. This technically contributes to wastage of space.
- In the worst case scenario, it might happen that all the keys to be inserted belong to a single bucket. This would result in a linked list structure and the search time would be $\mathbf{O(n)}$.
- Chaining cache performance is not that great since we are storing keys in the form of linked list. Open addressing techniques has better cache performance as all the keys are guaranteed to be stored in the same table. We will explore open addressing techniques in the next section.

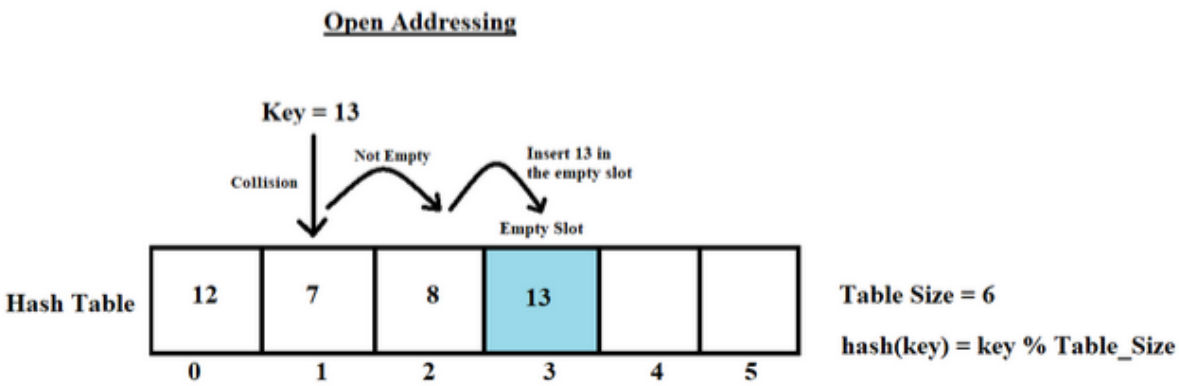
Linear Probing:

- In this, we linearly probe for the next free slot in the hash table. Generally, gap between two probes is taken as 1.
- Consider `hash(key)` be the slot index computed using hash function and `table_size` represent the hash table size. Suppose `hash(key)` index has a value occupied already, then:
  We check if (hash(key) + 1) If ((hash(key) + 1) then we check for ((hash(key) + 2) If ((hash(key) + 2) then we try ((hash(key) + 3) : : and so on until we find the next available free slot
- When performing search operation, the array is scanned linearly in the same sequence until we find the target element or an empty slot is found. Empty slot indicates that there is no such key present in the table.

Open Addressing-Linear probing

- In this technique, we ensure that all records are stored in the hash table itself. The size of the table must be greater than or equal to the total number of keys available. In case the array gets filled up, we increase the size of table by copying old data whenever needed. How do we handle the following operations in this techniques? Let's see below:
  - **Insert(key):** When we try to insert a key to the bucket which is already occupied, we keep probing the hash table until an empty slot is found. Once we find the empty slot, we insert `key` into that slot.

Open Addressing Example



Open Addressing: Quadratic Probing

Probing Sequence

Let hash(key) be the initial slot index and table_size be the size of the table.

// i = 0 (Initial check) slot = (hash(key) + 0*0)
// i = 1 (First probe) slot = (hash(key) + 1*1)
// i = 2 (Second probe) slot = (hash(key) + 2*2)
// i = 3 (Third probe) slot = (hash(key) + 3*3)
// and so on until we find the next available empty slot

Quadratic Probing