

# CSC10004: Data Structure and Algorithms

## Lecture 7: Hashing

Lecturer: Bùi Văn Thạch

TA: Ngô Đình Hy/Lê Thị Thu Hiền

[{bvthach,ndhy}@fit.hcmus.edu.vn](mailto:{bvthach,ndhy}@fit.hcmus.edu.vn), [lththien@hcmus.edu.vn](mailto:lththien@hcmus.edu.vn)

# Course topics

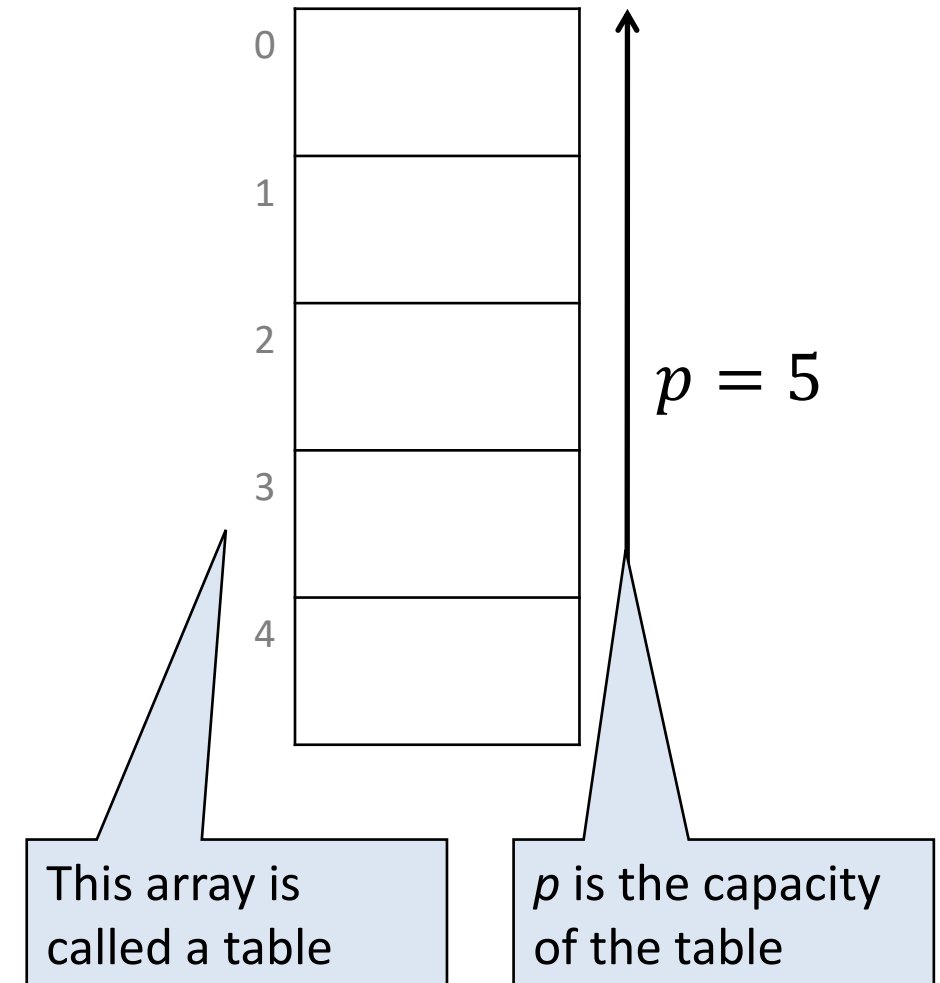
0. Introduction
1. Algorithm complexity analysis
2. Recurrences
3. Search
4. Sorting
5. Linked list
6. Stack, and Queue, Priority queue
7. Hashing
8. Tree
  1. Binary search tree (BST)
  2. AVL tree
8. Graph
  1. Graph representation
  2. Graph search
9. Algorithm designs
  1. Greedy algorithm
  2. Divide-and-Conquer
  3. Dynamic programming

# Goals

1. Understand the Purpose of Hashing.
2. Learn Hash Functions.
3. Handle Collisions Effectively.
4. Understand Hash Table Design Choices.

# Example

1. A dictionary that maps zip codes (keys) to neighborhood names (values) for 650 students K24 at HCMUS.
2. Zip codes are 5-digit numbers -- e.g., 13213
  1. Use a 100,000-element array with indices as keys?
  2. Possibly, but most of the space will be wasted:
  3. Only **650** students
  4. Only some **11,000** zip codes are in use
3. Use a much smaller  $p$ -element array (Here  $p = 5$ )
4. Reduce a key to an index in the range  $[0, p)$ 
  1. Here reduce a zip code to an index between 0 to 4
  2. Do zip code % 5
5. This is the first step towards **a hash table**.



# Summary

	Design	Run time					Space	Collision
		Computational cost	Search	Insert	Delete	Find_min		
Unsorted array	Det.		$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	
Sorted array by key			$O(\log n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	
Singly Linked list			$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	
Hashing	Rnd.		$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(p)$	YES

$$p < n$$

# What is Hashing?

1. Hashing is a technique that uses a **hash function** to convert large, variable-length data sets (known as **keys**) into **smaller, fixed-length** representations.
2. A hash table, or hash map, is a data structure that employs this hash function to quickly associate keys with values, enabling fast lookup and retrieval.
3. This approach is commonly used in various types of computer software, especially in associative arrays, database indexing, caching mechanisms, and set operations.

# Outline

1. Direct Addressing Table
2. Hash Table
3. Hash Functions
  - Good/bad/perfect/uniform hash function
4. Collision Resolution
  - Separate Chaining
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

# Outline

1. Direct Addressing Table
2. Hash Table
3. Hash Functions
  - Good/bad/perfect/uniform hash function
4. Collision Resolution
  - Separate Chaining
  - Linear Probing
  - Quadratic Probing
  - Double Hashing



# Student database problem

1. Retrieval: find(StudentID)
  - Find whether the student with StudentID exists.
2. Insertion: insert(StudentID)
  - Introduce a new student with StudentID
3. Deletion: delete(StudentID)
  - Remove the student with StudentID

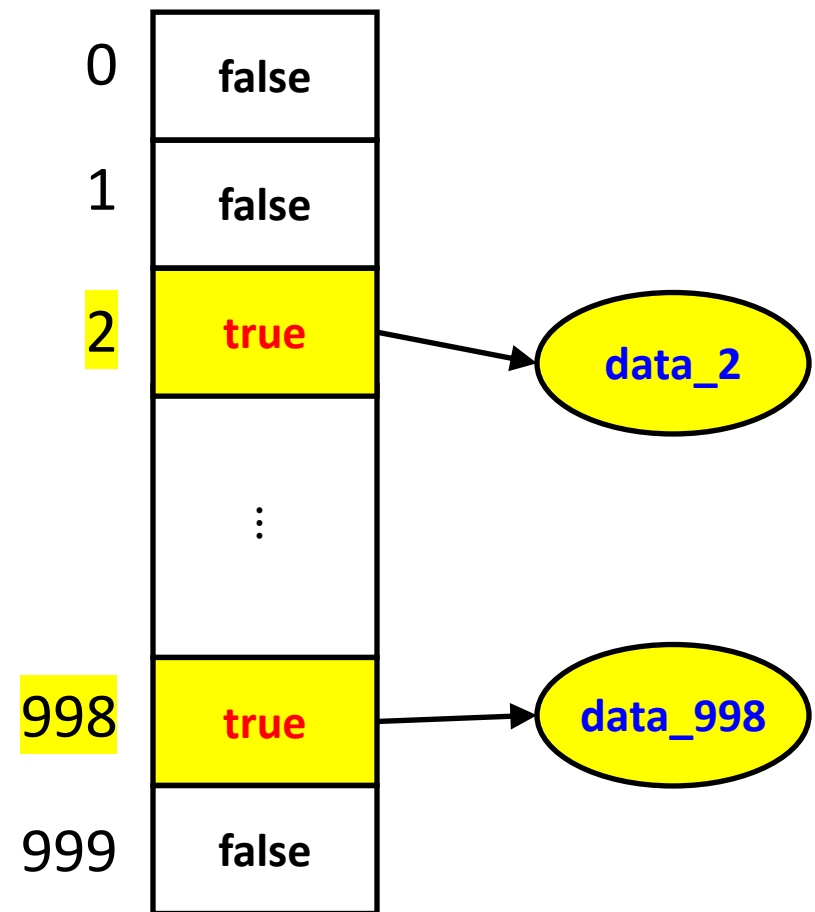
## Direct Addressing Table (1/2)

- Assume that student ID numbers are integers between 0 and 999, we can create an array with 1000 Boolean values.
- If **StudentID** exists, just set **position StudentID** to **true**.

0	false
1	false
2	true
	⋮
998	true
999	false

## Direct Addressing Table (2/2)

- If we need to store **additional information** about a student, we can use an array with 1000 slots, where each slot holds a **reference** to an object containing the student's details.
- **Note:** It may also be useful to store the key values—such as the student ID—alongside the data..



# Direct Addressing Table: Operations

1. **insert**(key, data)  
     $a[key] = \text{data}$       // where **a[]** is an array – the table
2. **delete**(key)  
     $a[key] = \text{null}$
3. **find**(key)  
    return  $a[key]$

## Direct Addressing Table: Limitations

1. Keys must be **non-negative integer values**.
  - What happens for key values 0812CNTN and CTT310?
2. Range of keys must be **small**.
3. Keys must be **dense**, i.e. not many gaps in the key values.
4. How to overcome these limitations?

# Outline

1. Direct Addressing Table
2. Hash Table
3. Hash Functions
  - Good/bad/perfect/uniform hash function
4. Collision Resolution
  - Separate Chaining
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

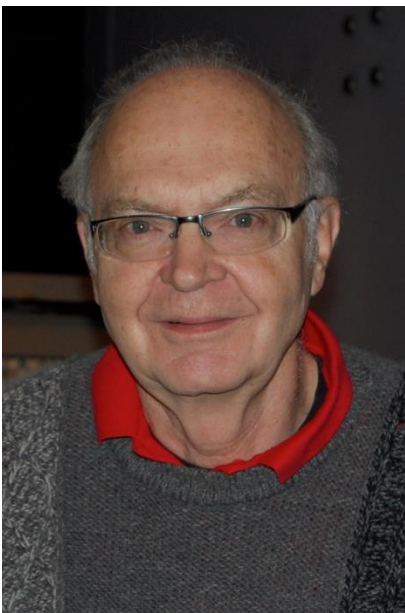
# History (1/2)

1. A hash table is an extension of the Direct Addressing Table designed to overcome its limitations.
2. The term "hash" is inspired by its everyday meaning—“to chop and mix.”
3. This metaphor fits well, as common hash functions, such as the *modulo* operation, effectively “chop” the input domain into smaller parts and “mix” them into the output range.



## History (2/2)

1. Donald Knuth points out that Hans Peter Luhn of IBM was likely the first to introduce the concept of hashing, as documented in **a memo** from January **1953**.
2. Later, Robert Morris helped popularize the term by using it in a *Communications of the ACM* survey paper, transitioning it from technical jargon to established terminology.



Donald Knuth  
(1938 - )



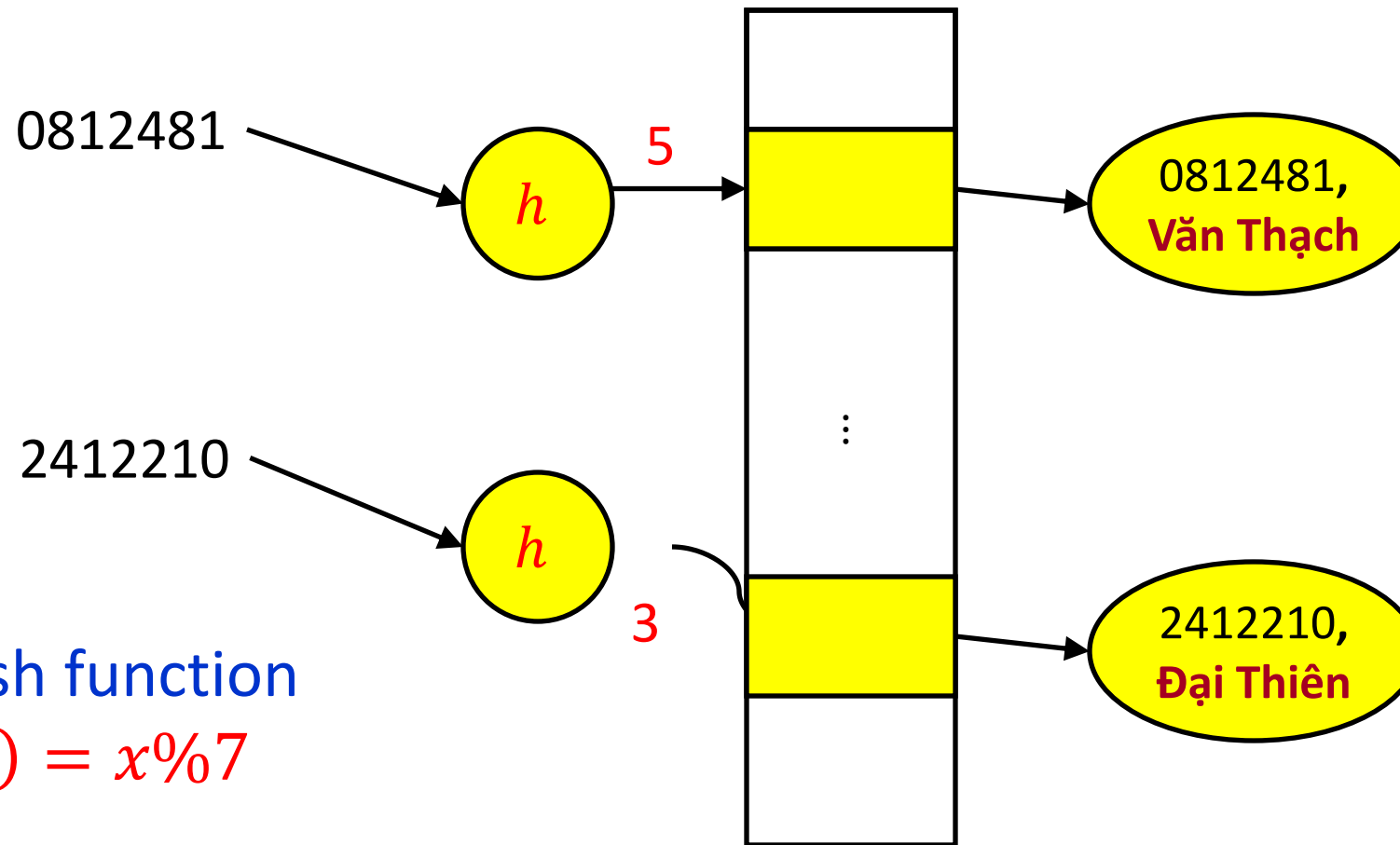
Hans Peter Luhn  
(1896 – 1964)



# Ideas

1. Map **large** integers to **smaller** integers.
2. Map **non-integer** keys to **integers**.

# Hash table



$h$  is a hash function

E.g.,  $h(x) = x \% 7$

**Note:** we must store the key values. **Why?**

**insert** (key, data)

$a[h(\text{key})] = \text{data}$  //  $h$  is a hash function and  $a[]$  is an array

**delete** (key)

$a[h(\text{key})] = \text{null}$

**find** (key)

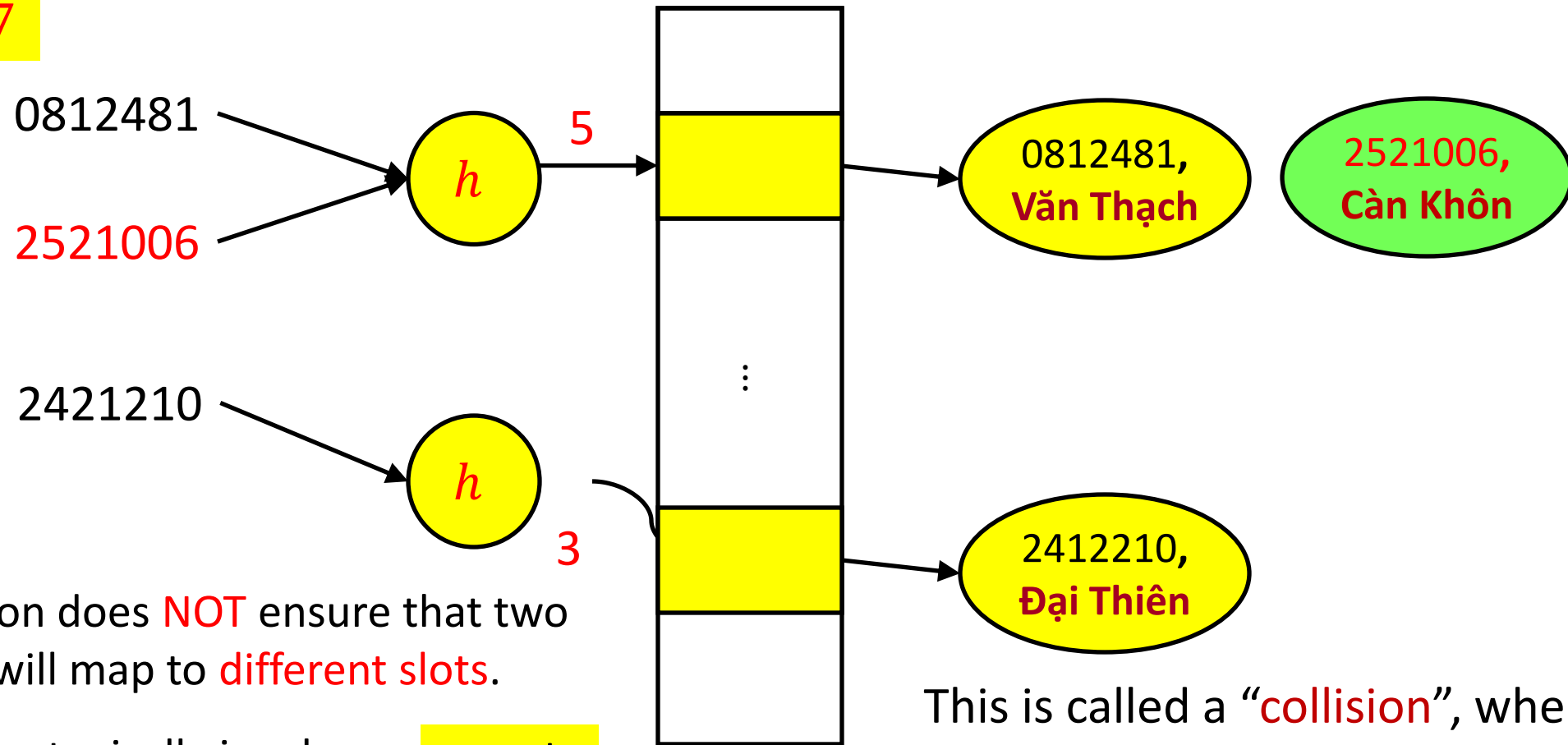
return  $a[h(\text{key})]$

However, this does **not**  
work for **all** cases!  
(Why?)

**key**='0812CNTN' or ' CTT310'?

# Collision: example

$$h(x) = x \% 7$$



A hash function does **NOT** ensure that two distinct keys will map to **different slots**.

In fact, hashing typically involves a **many-to-one** mapping, rather than a **one-to-one** correspondence.

This is called a “**collision**”, when two keys have the same hash value, i.e.,  $h(\text{ID1}) = h(\text{ID2})$ .

## Two Important Issues

- How to **hash**?
- How to **resolve collisions**?
- These are important issues that can affect the efficiency of hashing.

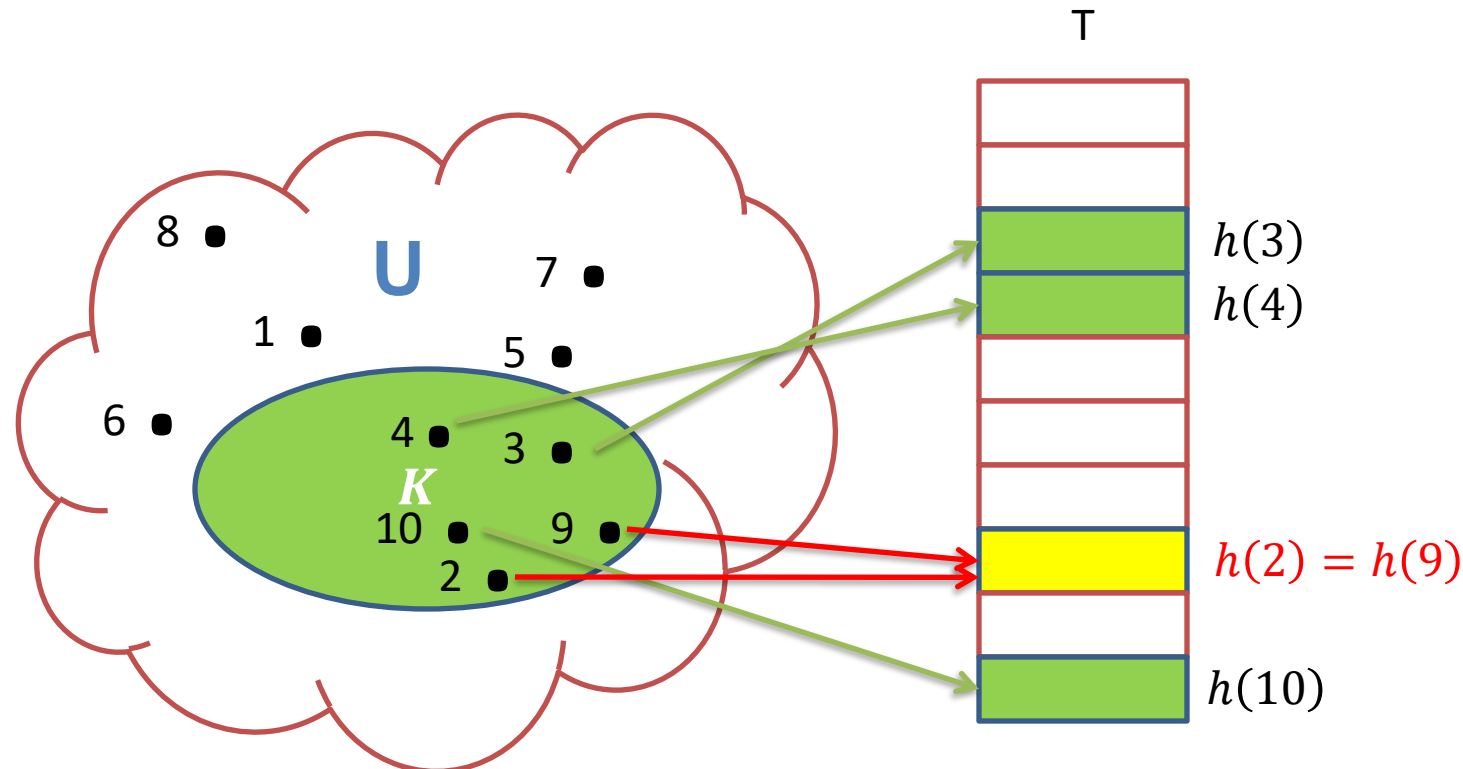
# Outline

1. Direct Addressing Table
2. Hash Table
3. Hash Functions
  - Good/bad/perfect/uniform hash function
4. Collision Resolution
  - Separate Chaining
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

# Collisions: definition

**Definition 1 (Collision).** Consider a hash function  $h: \mathcal{X} \rightarrow \{0,1\}^\ell$ . The output  $h(x)$  is called the hash of the input  $x$ . The function  $h$  contains a collision if there exist  $k_1 \neq k_2 \in \mathcal{X}$  such that:

$$h(k_1) = h(k_2).$$



## Collision: definition based on random hash

- Collision: given a hash function  $h$  and two ID1 and ID2, if  $h(\text{ID1}) = h(\text{ID2})$  then there is a collision.

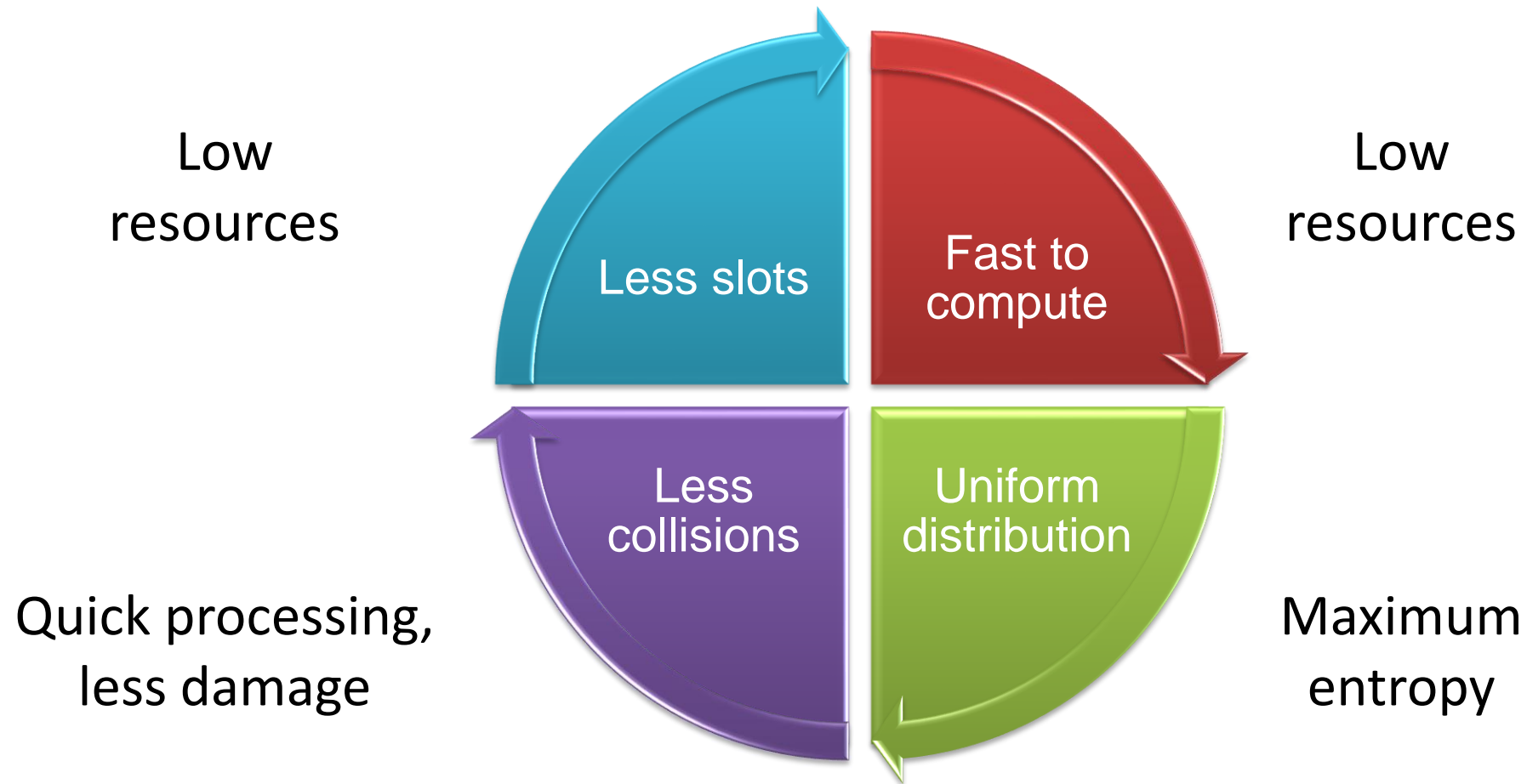
**Definition 1 ( $\ell$  – bit random hash).** Consider a random function  $h: \mathcal{X} \rightarrow \{0,1\}^\ell$ , chosen uniformly over that set of all function from  $\mathcal{X}$  to  $\{0,1\}^\ell$ . The output  $h(x)$  is called the  $\ell$ -random hash of the input  $x$ .

**Lemma 1 (Collision-resistance property).** For a random function  $h$  in Definition 1 and every distinct  $x \neq x' \in \mathcal{X}$ , we have

$$P(h(x) = h(x')) = 2^{-\ell}.$$



# Criteria of Good Hash Functions



## Example of **Bad** Hash Function

- Select Digits – e.g. choose the 4<sup>th</sup> and 7<sup>th</sup> digits of an ID
  - hash(081**2**48**1**) = **21**
  - hash(242**1**21**0**) = **10**
- So many students share the same hash values.

# Perfect Hash Functions

- A **perfect hash function** provides a **one-to-one mapping** between keys and their hash values, meaning **no collisions** occur.
- This is achievable when **all keys are known in advance**.
- A **minimal perfect hash function** ensures that the **hash table size exactly matches the number of keys**.
- Applications:
  - Used in **compilers** and **interpreters** to efficiently locate reserved words.
  - Utilized by **shell interpreters** for identifying built-in commands.
- **GNU gperf** is a freely available tool written in C++ that generates perfect hash functions from a **user-defined list of keywords**.

# Uniform hash functions

**Definition 1 ( $\ell$  – bit uniform hash function).** Consider a random function  $h: \mathcal{X} \rightarrow \{0,1\}^\ell$  that maps all events from  $\mathcal{X}$  to  $\{0,1\}^\ell$ . The hash function  $h$  is uniform if every distinct  $x \neq x' \in \mathcal{X}$ , we have

$$P(h(x) = h(x')) = 2^{-\ell}.$$

- Distributes keys **evenly** in the hash table.
- Example: If ID integers are **uniformly** distributed among **0** and  **$X - 1$** , we can map the values to a hash table of size  **$p$**  ( $p < X$ ) using the hash function beside.

$$\text{ID} \in [0, X)$$

$$h(\text{ID}) = \left\lfloor \text{ID} \times \frac{p}{X} \right\rfloor$$

ID is the key value and  $0 \leq \text{ID} < X$

$\lfloor \quad \rfloor$ : is the **floor** function

## Division method (mod operator)

- Map into a hash table of  $p$  slots.
- Use the **modulo** operator (**%** in C++) to map an integer to a value between 0 and  $p - 1$ .
- $k \bmod p$  = remainder of  $k$  divided by  $p$ , where  $k$  and  $p$  are positive integers.

$$h_p(k) := k \% p$$

The most popular method.

# How to pick $m$ ?

- The choice of  $p$  (or **hash table size**) is important.
- If  $p$  is power of two, say  $2^n$ , then key modulo of  $p$  is the same as extracting the last  $n$  bits of the key.
- If  $p$  is  $10^n$ , then our hash values is the last  $n$  digit of keys.
- Both are not good.

- Pick  $p$  as a good prime number.
- A good prime is a prime number whose square is greater than the product of any two primes at the same number of positions before and after it in the sequence of primes.

$$p_n^2 > p_{n-i}p_{n+i} \text{ for } 1 \leq i \leq n - 1$$

5, 11, 17, 29, 37, 41, 53, 59,  
67, 71, 97, 101, 127, 149, 179,  
191, 223, 227, 251, 257, 269,  
307, 311, 331, 347, 419, 431,  
541, 557, 563, 569, 587, 593,  
599, 641, 727, 733, 739, 809,  
821, 853, 929, 937, 967

## Multiplication method

1. Multiply by a constant real number  $\epsilon$  between 0 and 1.
2. Extract the fractional part.
3. Multiply by  $p$ , the hash table size.

$$h(k) := \lfloor p(k\epsilon - \lfloor k\epsilon \rfloor) \rfloor$$

1. Choose  $r$  as an irrational number and set  $\epsilon = r - \lfloor r \rfloor$ .
2. Can choose  $\epsilon$  as the **golden ratio**  $= \frac{\sqrt{5} - 1}{2} = 0.618033$  (recommended by Knuth).

# Hash of strings

An example hash function for strings:

```
int hash(const string& s, int p) {
    int sum = 0;
    for (char c : s) {
        sum += static_cast<int>(c); // Add ASCII value of each
        character
    }
    return sum % p; // Modulo with hash table size
}
```

hash("Bui Van Thach")

= (66 + 117 + 105 + 32 + 86 + 97 + 110 + 32  
+ 84 + 104 + 97 + 99 + 104)% 821 = 312

B	: 66
u	: 117
i	: 105
(space)	: 32

V	: 86
a	: 97
n	: 110
(space)	: 32

T	: 84
h	: 104
a	: 97
c	: 99
h	: 104



- All 3 strings below have the **same hash value**! Why?
  - Bui Van Thach (Bùi Văn Thạch)
  - Thua Bich Van (Thừa Bích Vân)
  - Van Thi Bachu (Vạn Thi Bá Chủ)
- **Problem:** This hash function value does not depend on positions of characters! – Bad

- A more effective hash function for strings adjusts the accumulated sum after processing each character, ensuring that character positions influence the final hash value.

```
int hash(const std::string& s, int p) {  
    unsigned long long sum = 0;  
    for (char c : s) {  
        sum = sum * 821 + static_cast<unsigned char>(c); // Use  
        unsigned char to handle extended ASCII  
    }  
    return sum % p;  
}
```

## Analysis: Performance of Hash Table

- $n$ : number of keys in the hash table
- $p$ : size of the hash tables – number of slots
- $\alpha$ : load factor

$$\alpha := \frac{n}{p}.$$

a measure of **how full** the hash table is. If the table size is the number of linked lists, then  $\alpha$  is the average length of the linked lists.

# Outline

1. Direct Addressing Table
2. Hash Table
3. Hash Functions
  - Good/bad/perfect/uniform hash function
4. Collision Resolution
  - Separate Chaining
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

# Probability of Collision (1/2)

- **The Birthday Paradox:**

“How many people must be in a room before the probability that some **share a birthday**, ignoring the year and leap days, becomes **at least 50 percent**?”

- $q(n)$  = Probability of **unique** birthday for  $n$  people

$$= \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \dots \times \frac{365 - n + 1}{365}$$

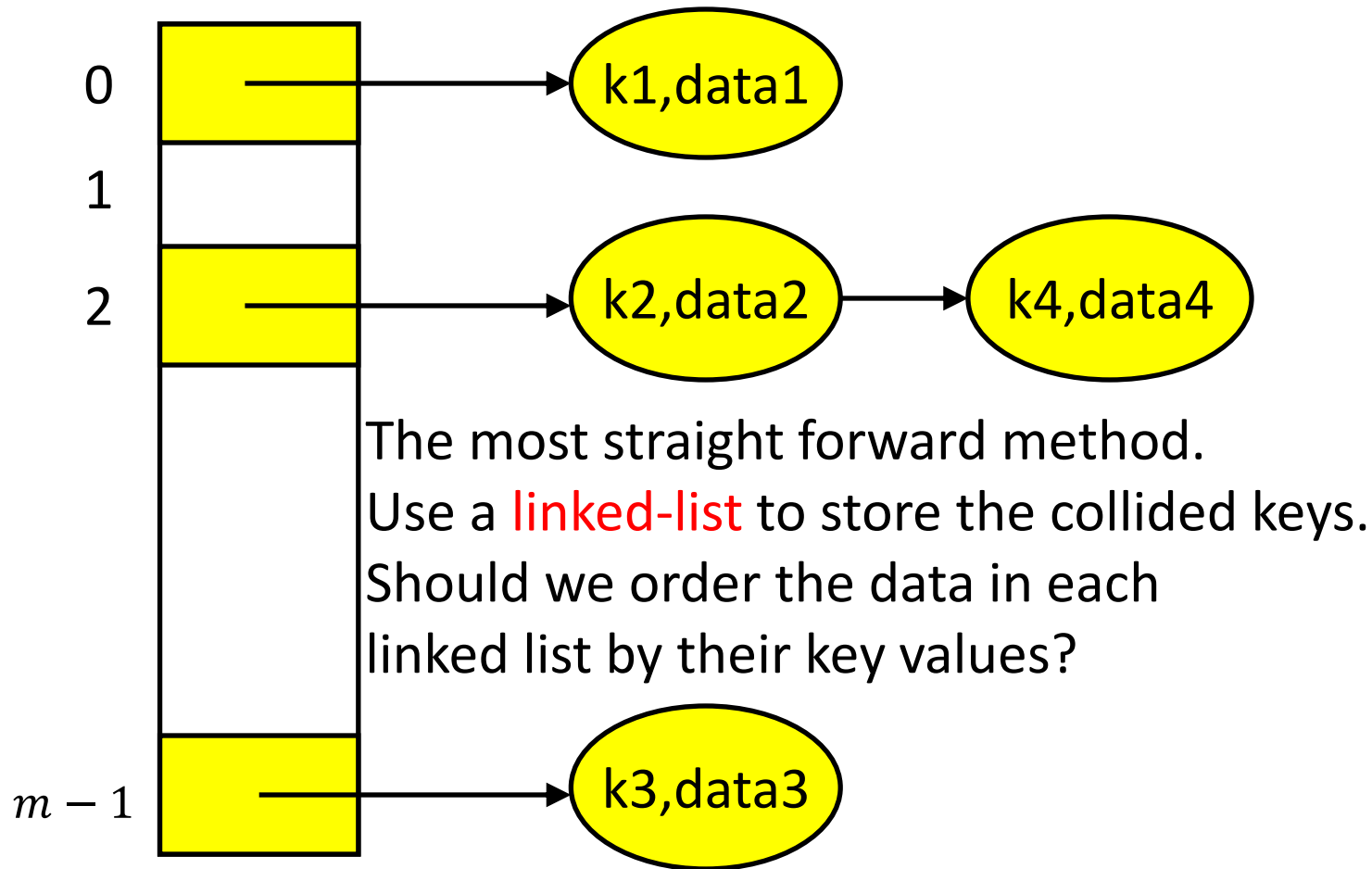
- $p(n)$  = Probability of **collisions** (same birthday) for  $n$  people =  $1 - q(n)$
- $p(\mathbf{23}) = 0.507$
- Hence, you need **only 23 people** in the room!

## Probability of Collision (2/2)

- This means that if there are **23** people in a room, the probability that some people share a birthday is **50.7%**!
- In the hashing context, if we insert **23** keys UNIFORMLY into a table with **365** slots, more than half of the time we will get collisions!
- So, **collision** is very likely!

# Outline

1. Direct Addressing Table
2. Hash Table
3. Hash Functions
  - Good/bad/perfect/uniform hash function
4. Collision Resolution
  - Separate Chaining
  - Linear Probing
  - Quadratic Probing
  - Double Hashing



### insert (key, data)

Insert data into the **list**  $a[h(\text{key})]$

Takes  $O(1)$  time

### find (key)

Find key from the **list**  $a[h(\text{key})]$

Takes  $O(c)$  time, where  $c$  is length of the chain

### delete (key)

Delete data from the **list**  $a[h(\text{key})]$

Takes  $O(c)$  time, where  $c$  is length of the chain

# Reconstructing Hash Table

- $n$ : number of keys in the hash table
- $p$ : size of the hash tables – number of slots
- $\alpha$ : load factor

$$\alpha := \frac{n}{p}.$$

- To keep  $\alpha$  bounded, we may need to **reconstruct** the whole table when the load factor exceeds the bound.
- Whenever the load factor exceeds the bound, we need to **rehash** all keys into a **bigger** table (increase  $p$  to reduce  $\alpha$ ), say double the table size  $p$ .



# Outline

1. Direct Addressing Table
2. Hash Table
3. Hash Functions
  - Good/bad/perfect/uniform hash function
4. Collision Resolution
  - Separate Chaining
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

# Linear probing: insertion

$$h(k) = k \bmod 7$$

Here, the table size  $p = 7$

Note: 7 is a prime number.

0	
1	
2	
3	
4	
5	
6	

In **linear probing**, when we get a **collision**, we scan through the table looking for the **next empty slot** (wrapping around when we reach the last slot).

```
linear_probing_insert(K)
    if (table is full)
        error
    probe = h(K)
    while (table[probe] occupied)
        probe = (probe + 1) mod p
    table[probe] = K
```

# Linear probing

The **probe sequence** of this linear probing is:

insertion

$$h(k) = k \bmod p$$

collision

$$h(\text{key}, \text{step}) = (h(\text{key}) + \text{step}) \% p$$

$$h(\text{key}, 0) = h(\text{key}) \% p$$

$$h(\text{key}, 1) = (h(\text{key}) + 1) \% p$$

$$h(\text{key}, 2) = (h(\text{key}) + 2) \% p$$

1. Given  $k_1$  and  $k_2$ , if  $h(k_1) = h(k_2)$  (**collision**), then  $h(k_2) := h(k_2, i)$  for  $i = 1, 2, \dots$  until there is an empty slot at  $h(k_2, i)$ .
2. Put  $k_2$  at  $h(k_2, i)$ .

# Insert 7, 14, and 25

$$h(k) = k \bmod 7$$

$$h(7) = 7 \bmod 7 = 0$$

$$h(14) = 14 \bmod 7 = 0$$

$$h(25) = 25 \bmod 7 = 4$$

0	7
1	14
2	
3	
4	25
5	
6	

Insert 7

Insert 14

Collision occurs!  
Look for next empty slot.

Insert 25

# Insert 1 and 35

$$h(k) = k \bmod 7$$

$$h(7) = 7 \bmod 7 = 0$$

$$h(14) = 14 \bmod 7 = 0$$

$$h(1) = 1 \bmod 7 = 1$$

$$h(35) = 35 \bmod 7 = 0$$

$$h(25) = 25 \bmod 7 = 4$$

0	7
1	14
2	1
3	35
4	25
5	
6	

Collides with 21 (hash value 0). Look for **next empty slot**.

Collision, need to check **next 3 slots**.

Insert 1

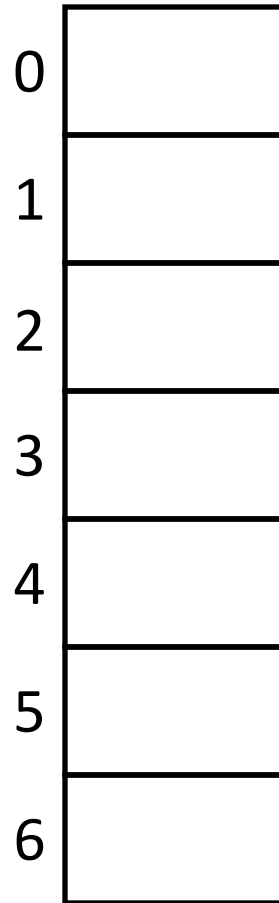
Insert 35

# Linear probing: finding

1. Check the slot at the initial index  $h(k)$ .
2. If it contains the key, return the index.
3. If the slot is empty or contains a different key (collision), proceed to the next slot.
4. Use the linear probing formula to compute the next index:  

$$h'(k,i)=(h(k)+i)\bmod p$$

where:  $i$  is the probe number (starting from 0),  $p$  is the table size.



In **linear probing**, when we get a **collision**, we scan through the table looking for the **next empty slot** (wrapping around when we reach the last slot).

```
bool linear_probing_find(K)
    if (table is full)
        error
    do {
        probe = h(K)
        if table[probe]==K
            return 1
    } while (table[probe] occupied)
    return 0
```

# Find 35

$$h(k) = k \bmod 7$$

$$h(35) = 0$$

0	7
1	14
2	1
3	35
4	25
5	
6	

Found 35, after 4 probes.

# Find 8

$$h(k) = k \bmod 7$$

$$h(8) = 1$$

0	7
1	14
2	1
3	35
4	25
5	
6	

8 NOT found.  
Need 5 probes!



# Delete 14 (1/2)

$$h(k) = k \bmod 7$$

$$h(14) = 0$$

0	7
1	14
2	1
3	35
4	25
5	
6	

We **cannot** simply **remove** a value, because it can affect **find()**!

# Delete 14 (2/2)

$$h(k) = k \bmod 7$$

$$\text{hash}(35) = 0$$

Hence for deletion, **cannot** simply remove the key value!

0	7
1	
2	1
3	35
4	25
5	
6	

We **cannot** simply **remove** a value, because it can affect **find()**!

35 NOT found!  
**Incorrect!**

# How to delete?

$$h(k) = k \bmod 7$$

$$\text{hash}(14) = 0$$

- **Lazy** Deletion
- Use **three** different **states** of a slot
  - Occupied
  - Occupied but mark as deleted
  - Empty

0	7
1	<del>14</del>
2	1
3	35
4	25
5	
6	

Slot 1 is occupied but now **marked as deleted**.

- When a value is removed from linear probed hash table, we just **mark** the status of the slot as “**deleted**”, instead of emptying the slot.

# Find 35

$$h(k) = k \bmod 7$$

$$h(35) = 0$$

0	7
1	<del>14</del>
2	1
3	35
4	25
5	
6	

Found 35, after 4 probes.

Now we can find 35!

# Insert 15 (1/2)

$$h(k) = k \bmod 7$$

$$h(15) = 1$$

0	7
1	<del>14</del>
2	1
3	35
4	25
5	
6	

Slot 1 is marked as deleted.

We **continue to search** for 15, and found that 15 is not in the hash table (total 5 probes).

So, we insert this new value 15 into the slot that has been marked as deleted (i.e. slot 1).

# Insert 15 (2/2)

$$h(k) = k \bmod 7$$

$$h(15) = 1$$

0	7
1	15
2	1
3	35
4	25
5	
6	

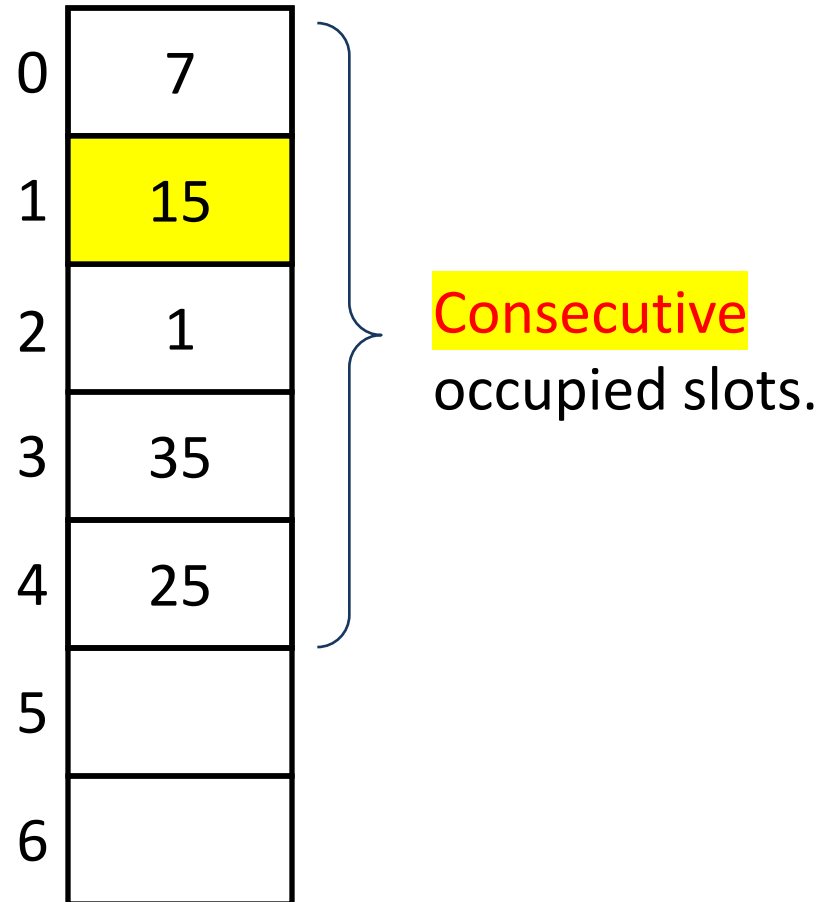
So, 15 is inserted into slot **1**, which was marked as deleted.

**Note:** We should insert a new value in **first** available slot so that the find operation for this value will be the fastest.

# Problem of Linear Probing

It can create many **consecutive occupied slots**, increasing the running time of find/insert/delete.

This is called **Primary Clustering**



# Modified Linear Probing

Q: How do we **modify** linear probing to avoid **primary clustering**?

We can modify the probe sequence as follows:

insertion

$$h(k) = k \bmod p$$

collision

$$\left\{ \begin{array}{l} h(\text{key}, \text{step}) = (h(\text{key}) + \text{step} \times d) \% p \\ h(\text{key}, 0) = h(\text{key}) \% p \\ h(\text{key}, 1) = (h(\text{key}) + 1 \times d) \% p \\ h(\text{key}, 2) = (h(\text{key}) + 2 \times d) \% p \end{array} \right.$$

where  $d$  is some constant integer larger than 1 and is co-prime to  $p$ , *i.e.*,  $\text{gcd}(d, p) = 1$ .

Note: Since  $d$  and  $p$  are co-primes, the probe sequence **covers all** the slots in the hash table.



# Outline

1. Direct Addressing Table
2. Hash Table
3. Hash Functions
  - Good/bad/perfect/uniform hash function
4. Collision Resolution
  - Separate Chaining
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

# Quadratic probing

The **quadratic sequence** of this quadratic probing is:

insertion

$$h(k) = k \bmod p$$

collision

$$\left\{ \begin{array}{l} h(\text{key}, \text{step}) = (h(\text{key}) + \text{step}^2) \% p \\ h(\text{key}, 0) = h(\text{key}) \% p \\ h(\text{key}, 1) = (h(\text{key}) + 1^2) \% p \\ h(\text{key}, 2) = (h(\text{key}) + 2^2) \% p \end{array} \right.$$

1. Given  $k_1$  and  $k_2$ , if  $h(k_1) = h(k_2)$  (**collision**), then find an index  $h(k_2, i)$  for  $i = 1, 2, \dots$  until there is an empty slot at  $h(k_2, i)$ .
2. Put  $k_2$  at  $h(k_2, i)$ .

# Insert 3 and 18

$$h(k) = k \bmod 7$$

$$h(3) = 3 \% 7 = 3$$

$$h(18) = 18 \% 7 = 4$$

0	
1	
2	
3	3
4	18
5	
6	

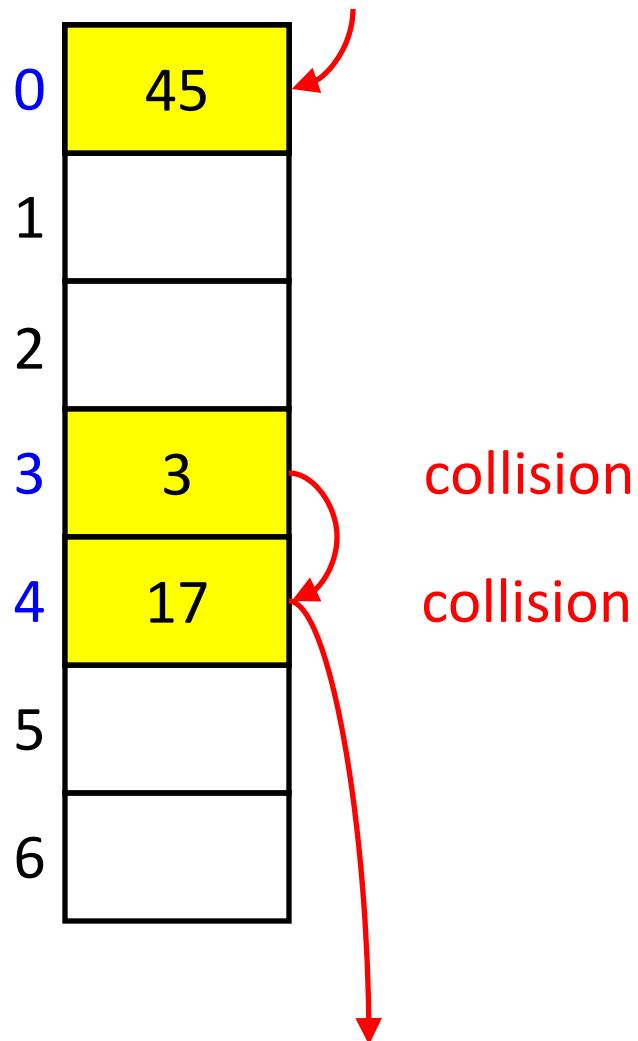
# Insert 45

$$h(k) = k \bmod 7$$

$$h(45) = 45 \% 7 = 3$$

➔  $h(45, 1) = (45 + 1^2) \% 7 = 4$

$$h(45, 2) = (45 + 2^2) \% 7 = 0$$

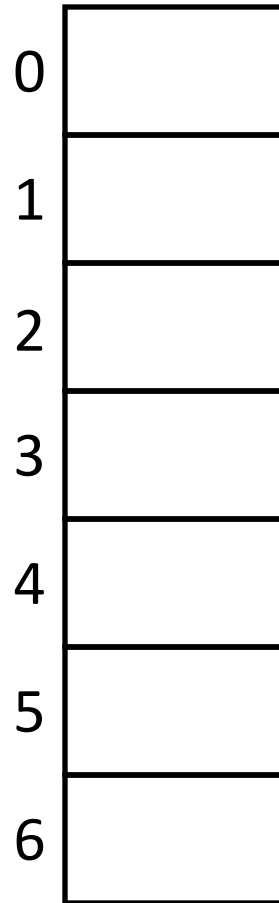


# Quadratic probing: finding

1. Check the slot at the initial index  $h(k)$ .
2. If it contains the key, return the index.
3. If the slot is empty or contains a different key (collision), proceed to the next **index**.
4. Use the quadratic probing formula to compute the next index:

$$h(k, i) = (h(k) + i^2) \% p$$

where:  $i$  is the probe number (starting from 0),  $p$  is the table size.



```
bool quadratic_probing_find(K)
    if (table is full)
        error
    count = 0
    do {
        probe = h(K, count)
        if table[probe]==K
            return 1
        count++;
    } while (table[probe] occupied)
    return 0
```

# Find 8

$$h(k) = k \bmod 7$$

$$h(8, 0) = (8 + 0^2) \% 7 = 1$$

$$h(8, 1) = (8 + 1^2) \% 7 = 2$$

0	
1	48
2	
3	3
4	17
5	
6	

8 NOT found.  
Need 2 probes!

# Theorem of Quadratic Probing

**Theorem 1.** If  $\alpha < 0.5$ , and  $p$  is prime, then we can always find an empty slot.  
( $p$  is the table size and  $\alpha$  is the load factor)

- Note:  $\alpha < 0.5$  means the hash table is less than half full.
- Q: How can we be sure that quadratic probing **always terminates**?
- Insert 14 into the previous example, followed by 11. See what happen?

## Problem of Quadratic Probing

- If two keys have the **same** initial position, their probe sequences are the **same**.
- This is called **secondary clustering**.
- But it is not as bad as linear probing.



# Outline

1. Direct Addressing Table
2. Hash Table
3. Hash Functions
  - Good/bad/perfect/uniform hash function
4. Collision Resolution
  - Separate Chaining
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

# Double hashing: Toy example

The **double hashing function** uses **two** hash functions. The secondary hash function indicates the number of slots to jump each time a collision occurs :

insertion

$$h(k) = k \bmod p$$

collision

$$\left\{ \begin{array}{l} h(\text{key}, \text{step}) = (h(\text{key}) + \text{step} \times h_2(\text{key})) \% p \\ h(\text{key}, 0) = h(\text{key}) \% p \\ h(\text{key}, 1) = (h(\text{key}) + h_2(\text{key})) \% p \\ h(\text{key}, 2) = (h(\text{key}) + 2 \times h_2(\text{key})) \% p \end{array} \right.$$

1. Given  $k_1$  and  $k_2$ , if  $h(k_1) = h(k_2)$  (**collision**), then  $h(k_2) := h(k_2, i)$  for  $i = 1, 2, \dots$  until there is an empty slot at  $h(k_2, i)$ .
2. Put  $k_2$  at  $h(k_2, i)$ .

# Insert 3 and 9

$$h(k, i) = (h(k) + i \times h_2(k)) \% 7$$

$$h(k) = k \bmod 7$$

$$h_2(k) = k \bmod 5$$

$$h(3) = 3 \% 7 = 3$$

$$h(9) = 9 \% 7 = 2$$

0	
1	
2	9
3	3
4	
5	
6	

# Insert 16

$$h(k, i) = (h(k) + i \times h_2(k)) \% 7$$

$$h(k) = k \bmod 7$$

$$h_2(k) = k \bmod 5$$

$$h(16) = 16 \% 7 = 2$$

$$h(16, 1) = (16 \% 7 + 1 \times 16 \% 5) \% 7 = 5$$

0	
1	
2	9
3	3
4	
5	16
6	

Collision occurs

# Insert 17

$$h(k, i) = (h(k) + i \times h_2(k)) \% 7$$

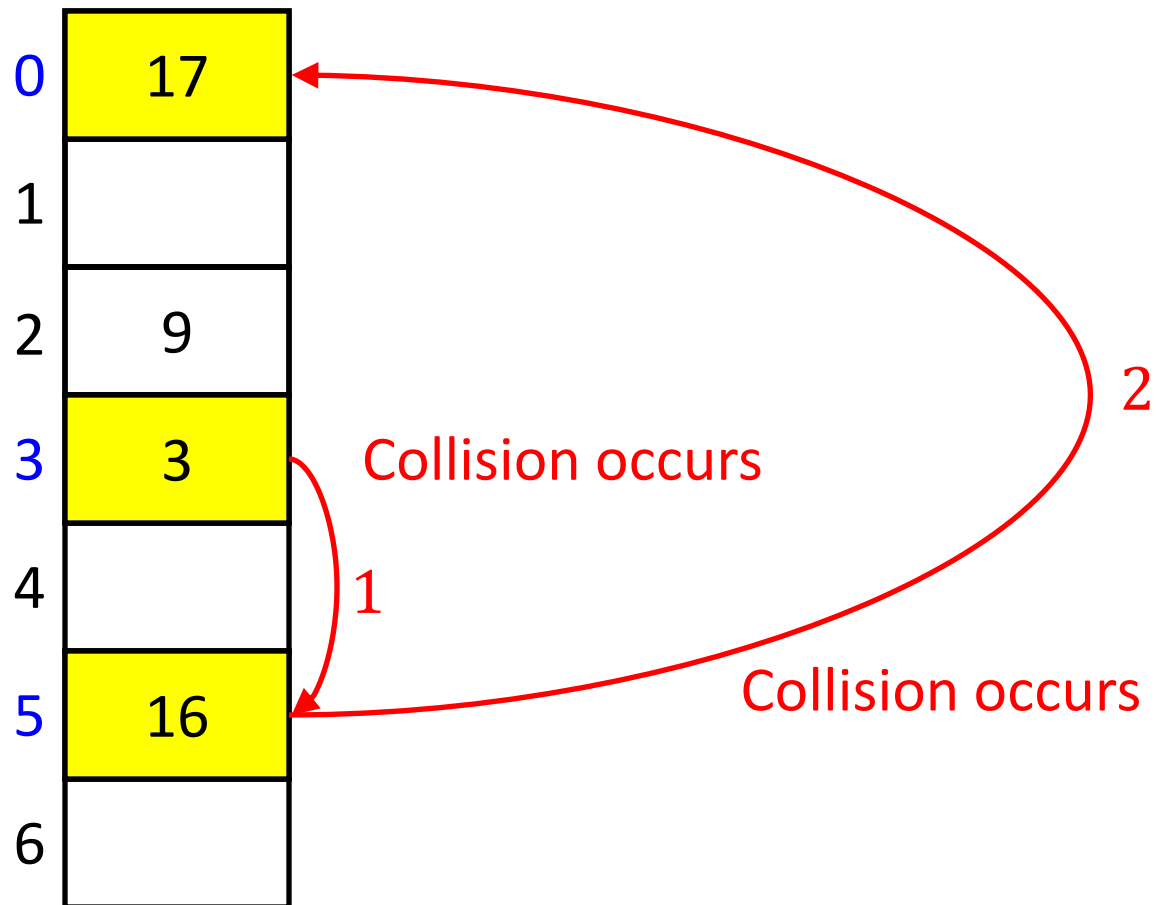
$$h(k) = k \bmod 7$$

$$h_2(k) = k \bmod 5$$

$$h(17) = 17 \% 7 = 3$$

$$h(17, 1) = (17 \% 7 + 1 \times 17 \% 5) \% 7 = 5$$

$$h(17, 2) = (17 \% 7 + 2 \times 17 \% 5) \% 7 = 0$$



# Insert 35

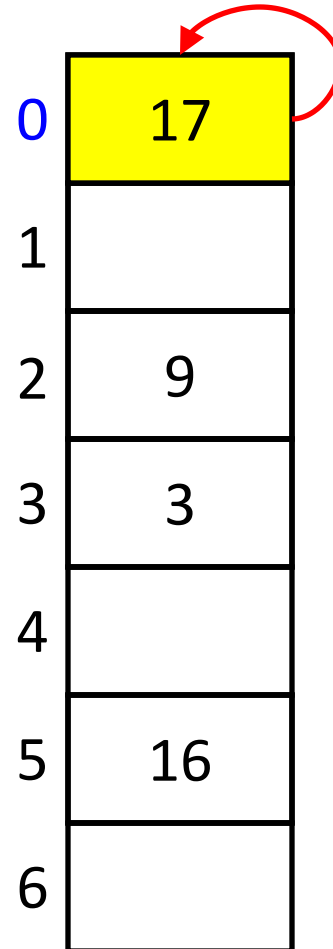
$$h(k, i) = (h(k) + i \times h_2(k)) \% 7$$

$$h(k) = k \bmod 7$$

$$h_2(k) = k \bmod 5$$

$$h(35) = 35 \% 7 = 0$$

$$h(35, i) = (35 \% 7 + i \times 35 \% 5) \% 7 = 0$$



0	17
1	
2	9
3	3
4	
5	16
6	

Collision occurs

But if we insert 35, the probe sequence is 0, 0, 0, ...

No **NEW** probe is found!

What is wrong?

Since  $h_2(35)=0$ .

**Not acceptable!**

# Caution

- Secondary hash function must **not** evaluate to **0**!
- To solve this problem, simply change  $h_2(\text{key})$  in the above example to:

$$h_2(\text{key}) = 5 - (2 * \text{key} \% 5)$$

## Note:

- ❑ If  $h_2(k) = 1$ , then it is the same as linear probing.
- ❑ If  $h_2(k) = d$ , where  $d$  is a constant integer  $> 1$ , then it is the same as modified linear probing.

# Criteria of Good Collision Resolution Method

- Minimize clustering.
- Always find an empty slot if it exists.
- Give different probe sequences when 2 initial probes are the same (i.e. no secondary clustering).
- Fast.



# The Efficiency of Hashing

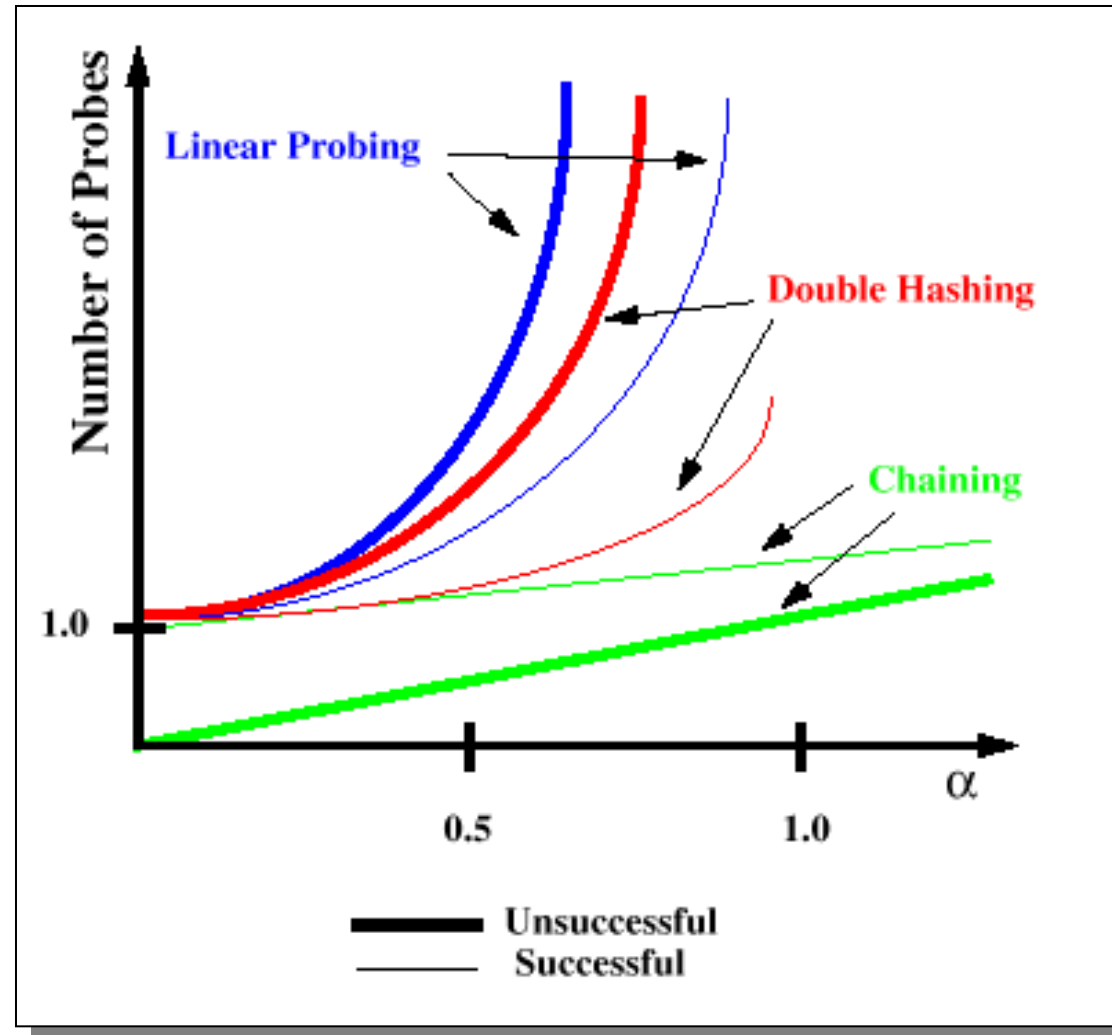
- $n$ : number of keys in the hash table;  $p$ : size of the hash tables – number of slots
- $\alpha$ : load factor

$$\alpha := \frac{n}{p}.$$

a measure of **how full** the hash table is. If the table size is the number of linked lists, then  $\alpha$  is the average length of the linked lists.

	Average number of probes	
	Successful search	Unsuccessful search
Linear probing	$1 + \alpha$	$1 + \alpha/2$
Quadratic probing	$\frac{1}{2} + \frac{1}{2(1 - \alpha)^2}$	$\frac{1}{2} + \frac{1}{2(1 - \alpha)}$
Double hashing	$\frac{1}{1 - \alpha}$	$\frac{1}{\alpha} \ln \frac{1}{1 - \alpha}$

# Experimental results



Q & A