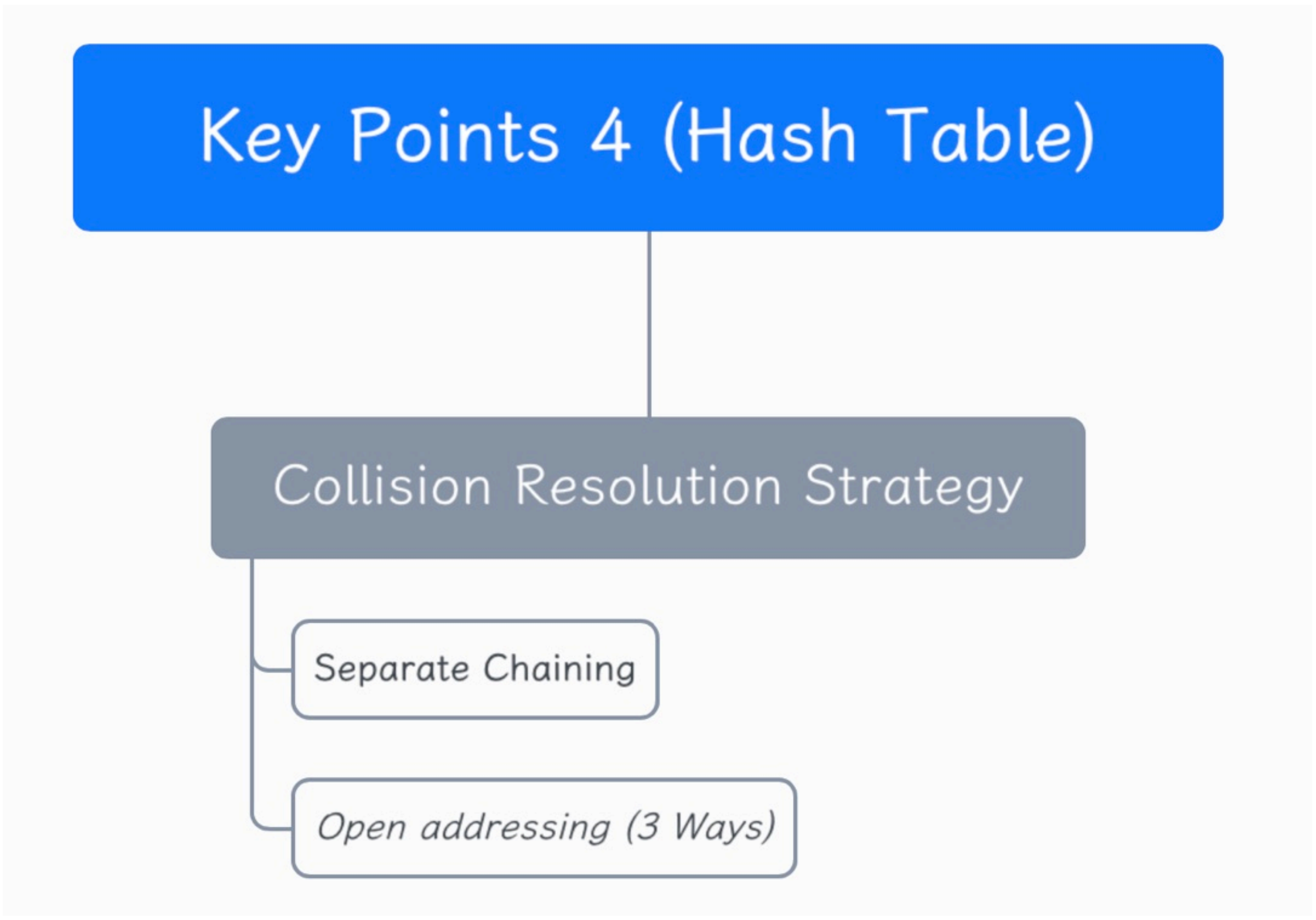
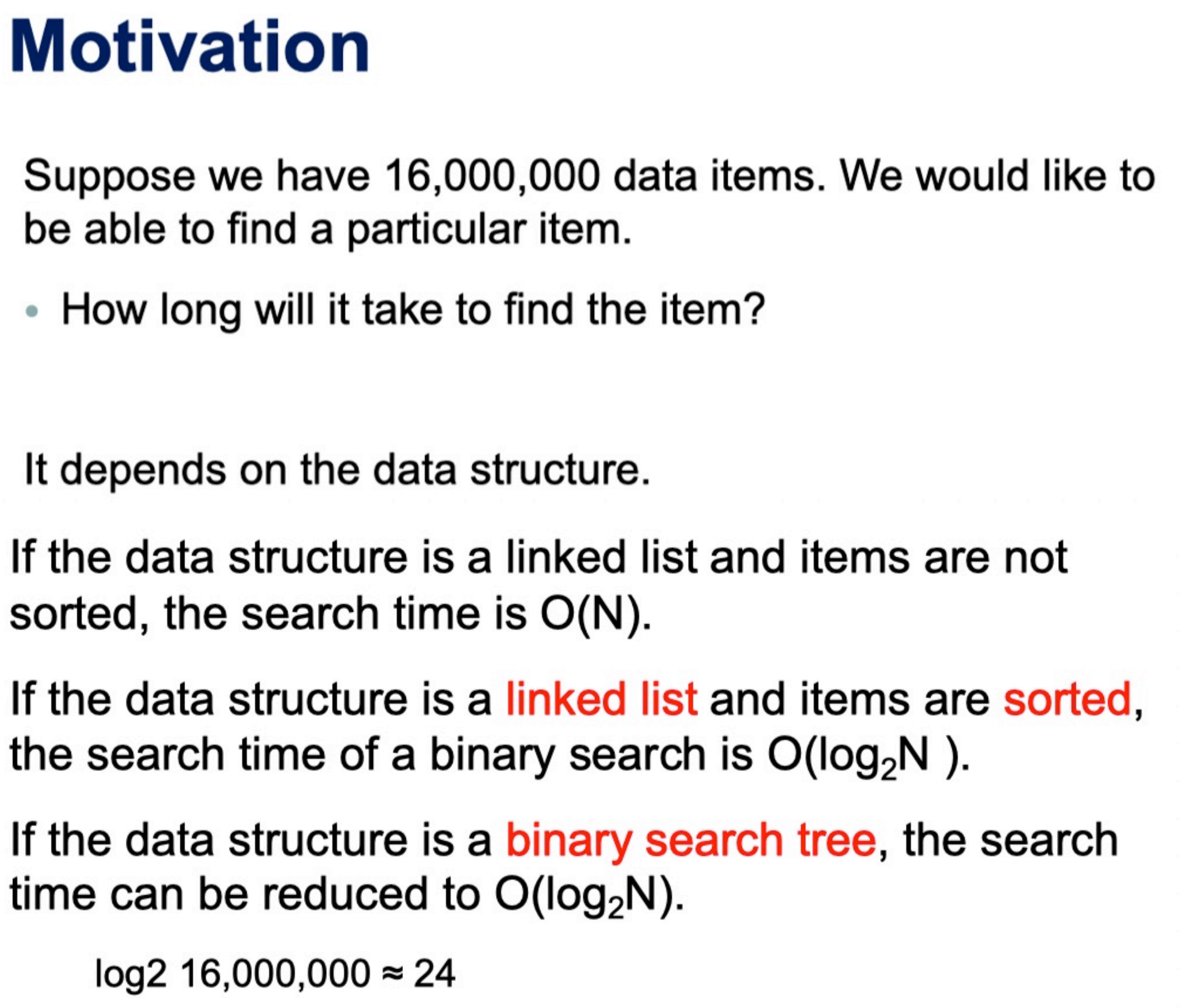
CS211FZ Note 4 | Algorithms & Data Structures | DSA2

Key Points 4 (Hash Table)

- Collision Resolution Strategy
    - **Separate Chaining**
    - *Open addressing (3 Ways)*



5.0 When we begin to learn Hash Table



Motivation

Suppose we have 16,000,000 data items. We would like to be able to find a particular item.

- How long will it take to find the item?

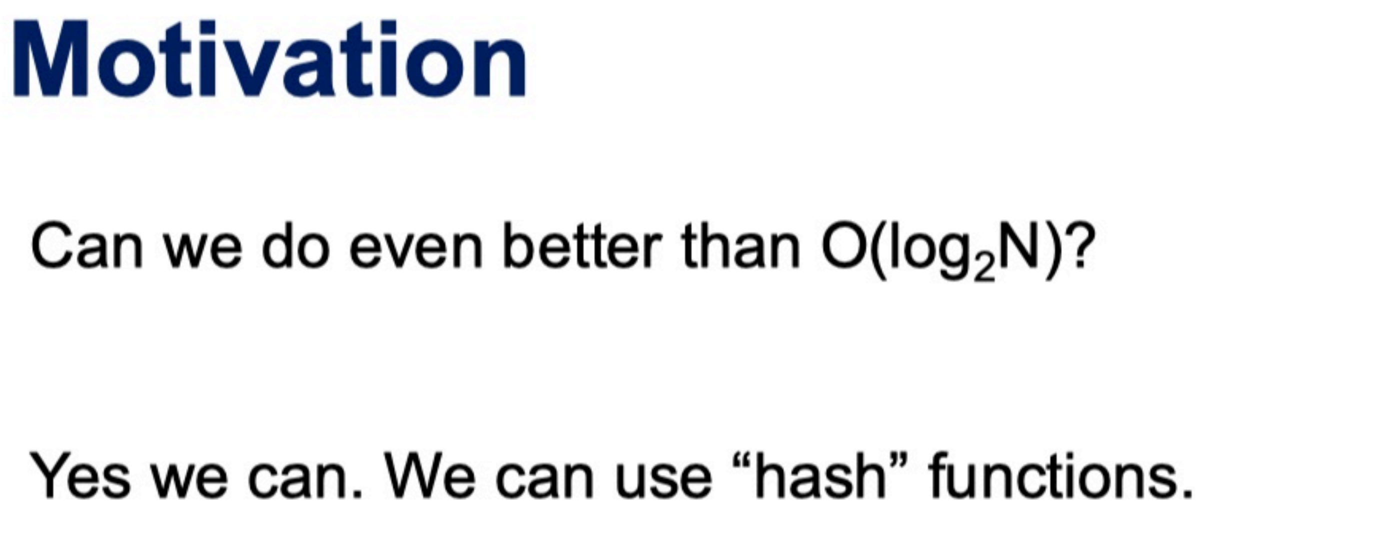
It depends on the data structure.

If the data structure is a linked list and items are not sorted, the search time is O(N).

If the data structure is a linked list and items are sorted, the search time of a binary search is $O(\log_2 N)$.

If the data structure is a binary search tree, the search time can be reduced to $O(\log_2 N)$.

$\log 2\ 16,000,000 \approx 24$



Motivation

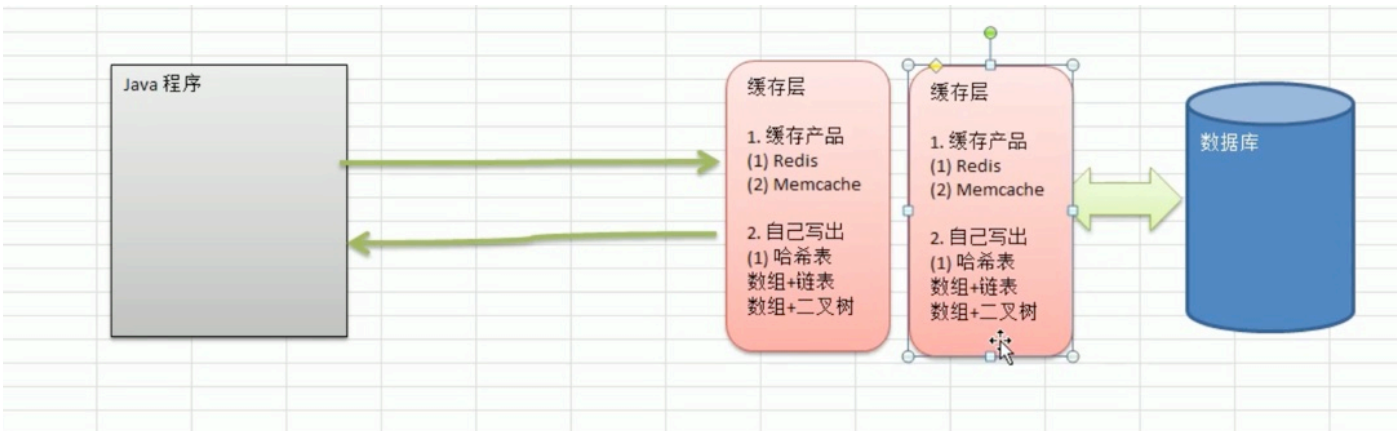
Can we do even better than $O(\log_2 N)$?

Yes we can. We can use "hash" functions.

Lecture 5 Hash Table

5.1 Basic

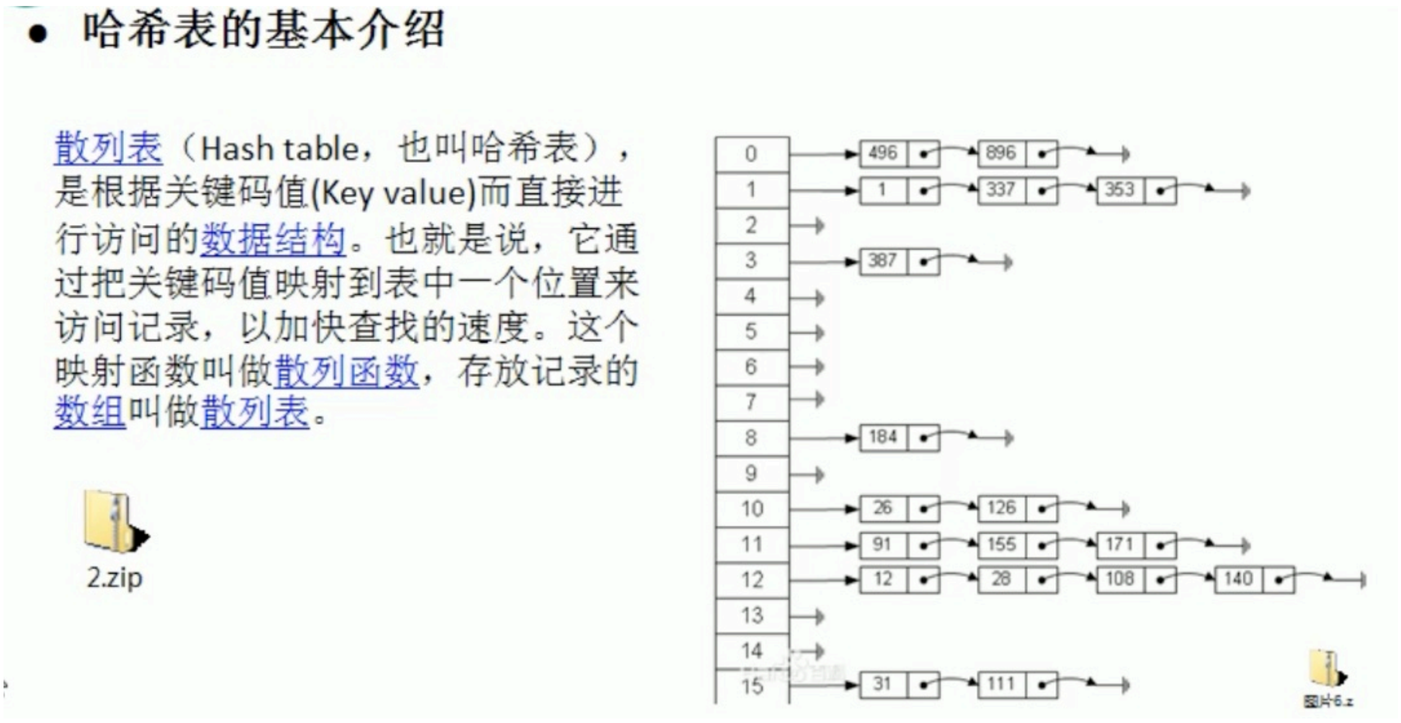
哈希表，根据关键码值*(key value)*直接进行访问的数据结构，通过把关键码值映射到表中的一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。



哈希表的两种实现方式：

1\哈希表 ＝ 数组 ＋ 链表；

2\哈希表 ＝ 数组 ＋ 二叉树；



● 哈希表的基本介绍

散列表（Hash table，也叫哈希表），是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

Hash Tables

- A hash table is a data structure that offers very fast insertion and searching.
- No matter how many data items there are, insertion and searching take close to constant time, nearly instantaneous or O(1)
- The disadvantage is that because they are based on arrays, they are difficult to expand
- Performance may degrade when a table becomes too full
- Items cannot be visited in any order
- However, if you know the size of your database and don’t need to access items in order hash tables are excellent.

哈希表是一种能够快速插入和搜索的数据结构；

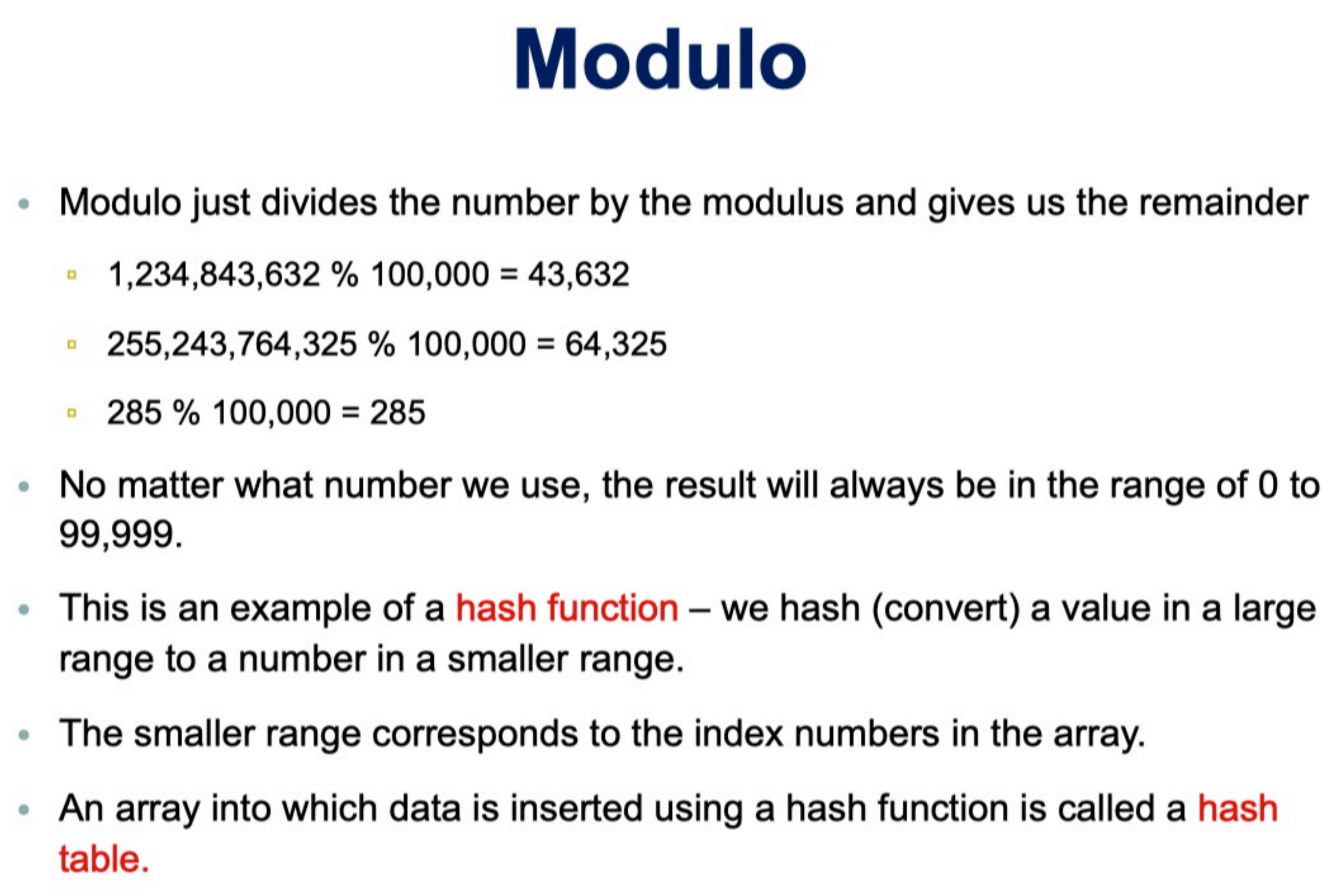
无论有多少数据项，插入和搜索的时间接近常数，几乎是瞬间或O(1)；

缺点是由于它们是基于数组的，所以很难扩展；

当表太满时，性能可能会下降。

0704：晚上好好休息，明天早起继续投入复习！加油，最后几天了，坚持住！💪

5.2 Modulo



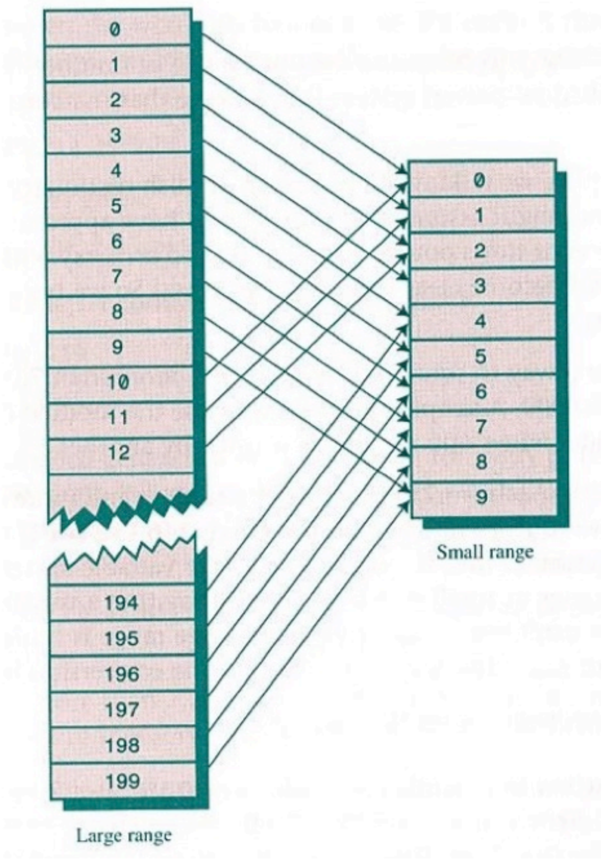
Modulo

- Modulo just divides the number by the modulus and gives us the remainder
  - 1,234,843,632 % 100,000 = 43,632
  - 255,243,764,325 % 100,000 = 64,325
  - 285 % 100,000 = 285
- No matter what number we use, the result will always be in the range of 0 to 99,999.
- This is an example of a hash function – we hash (convert) a value in a large range to a number in a smaller range.
- The smaller range corresponds to the index numbers in the array.
- An array into which data is inserted using a hash function is called a hash table.

Hashing

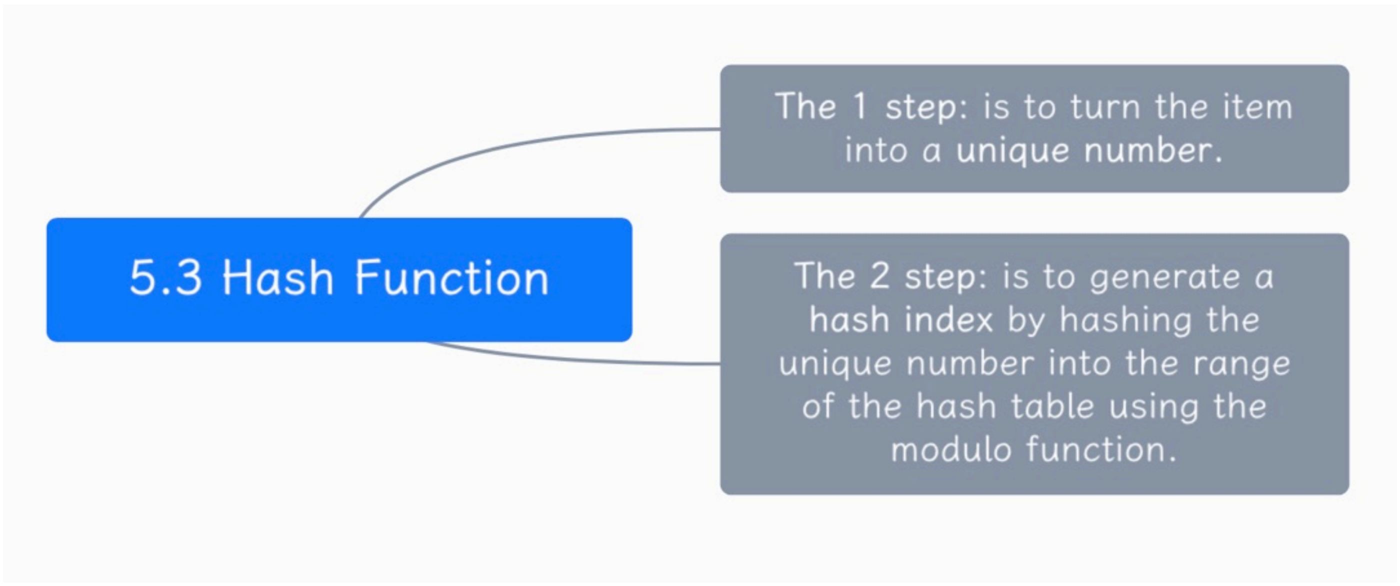
Use the modulo operator to squeeze a huge range of numbers into a range about twice as big as the number we want to store

- arraySize = numberWords * 2
- arrayIndex = hugeNumber % arraySize



5.3 Hash Function

- **The 1 step:** is to turn the item into a **unique number.**
- **The 2 step:** is to generate a **hash index** by hashing the unique number into the range of the hash table using the modulo function.


5.3 Hash Function
The 1 step: is to turn the item into a unique number.
The 2 step: is to generate a hash index by hashing the unique number into the range of the hash table using the modulo function.

The item is then inserted into the slot in the hash table, which has the same number as the hash index.

Prime Sized

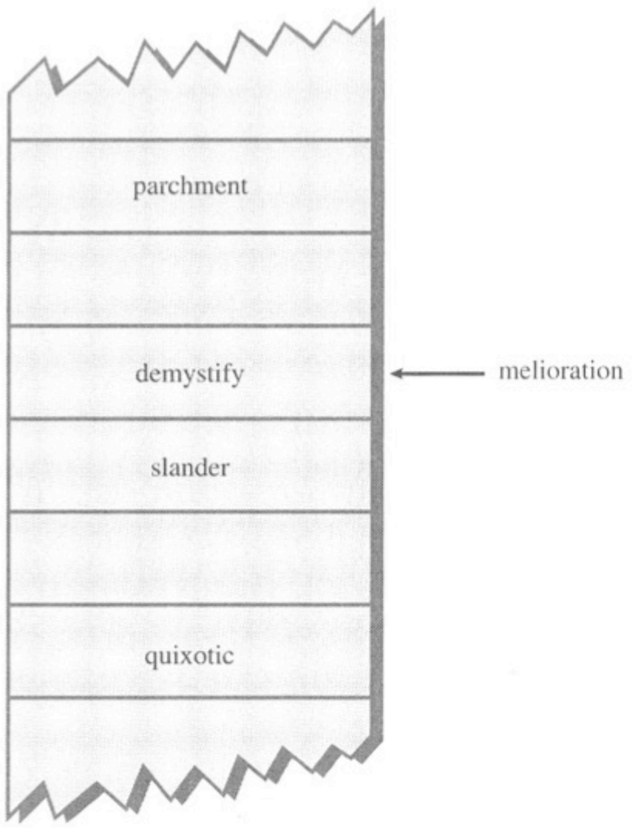
1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

- Prime-sized hash tables are better because a prime modulus will eliminate regular patterns in the data.
- Imagine we're hashing ten exam marks which have all been rounded up to the nearest 10%
- If we pick an array size of 20 then every mark will either be hashed into slot 10 (10%, 30%, 50%, 90%) or slot 0 (20%, 40%, 60%, 80%)
- If we use a prime number size, say 19, then all of these marks will hash into different slots.
- Any regular patterns in the data are eliminated by using a prime modulus.

5.4 Hash Collision

Collisions

- On average, there should be one word in every second slot.
- However, there's no guarantee that two words won't hash to the same array index.
- Imagine you go to insert the word melioration, and you find that the appropriate slot already contains the word demystify.
- This is called a collision.



访问 Access : $\boxed{X}$

搜索 Search : O(1)　　碰撞: O(K) ↗ 碰撞元素的个数

插入 Insert : O(1)

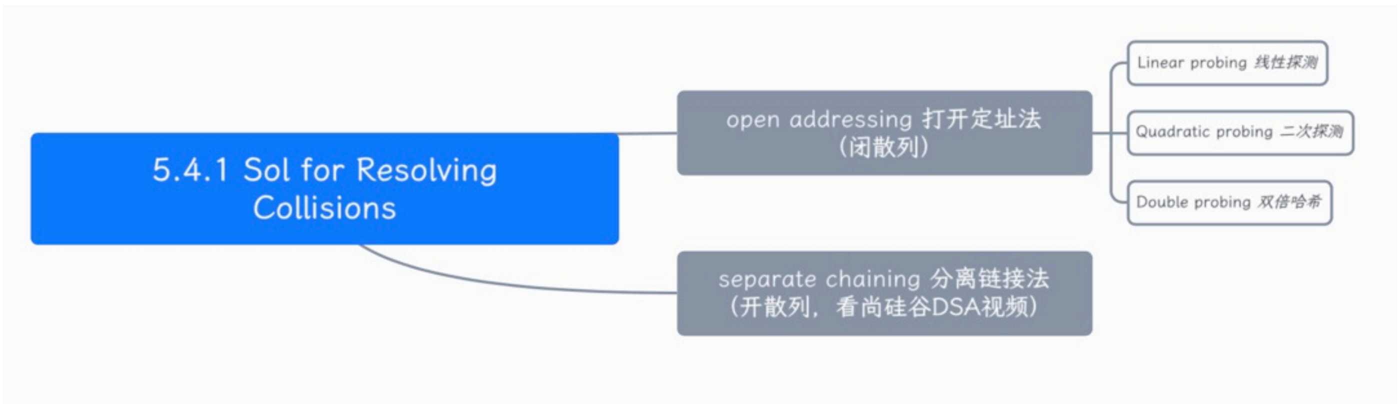
删除 Delete : O(1)

5.4.1 Sol for Resolving Collisions

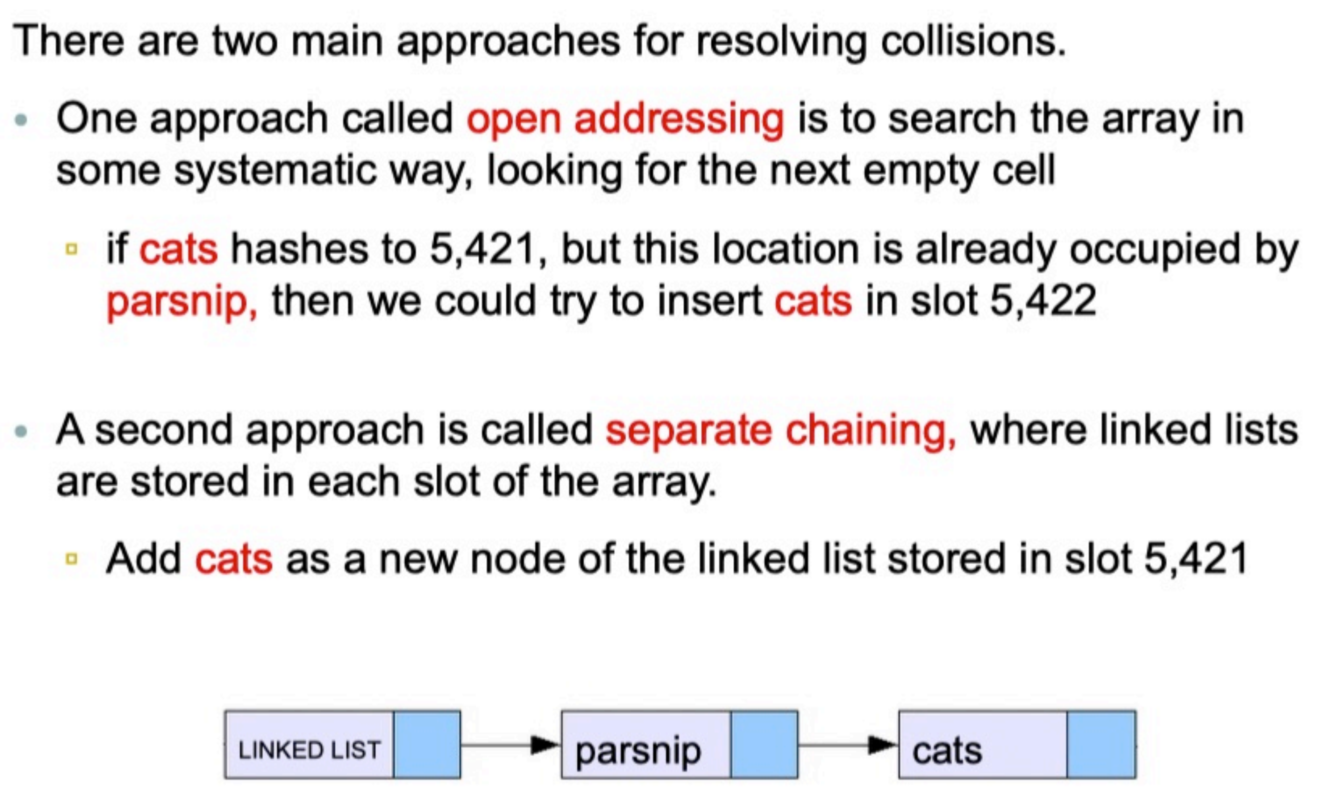
- open addressing **打开定址法（闭散列）**
    - Linear probing *线性探测*

- Quadratic probing *二次探测*
- Double probing *双倍哈希*

- Separate Chaining **分离链接法（开散列，看尚硅谷DSA视频）**


5.4.1 Sol for Resolving Collisions
open addressing 打开定址法（闭散列）
Linear probing 线性探测
Quadratic probing 二次探测
Double probing 双倍哈希
separate chaining 分离链接法（开散列，看尚硅谷DSA视频）

Lance说：不要受他人的影响，生活是自己的，未来也是自己的。不在一个轨道上的人，何必为之浪费时间。



There are two main approaches for resolving collisions.

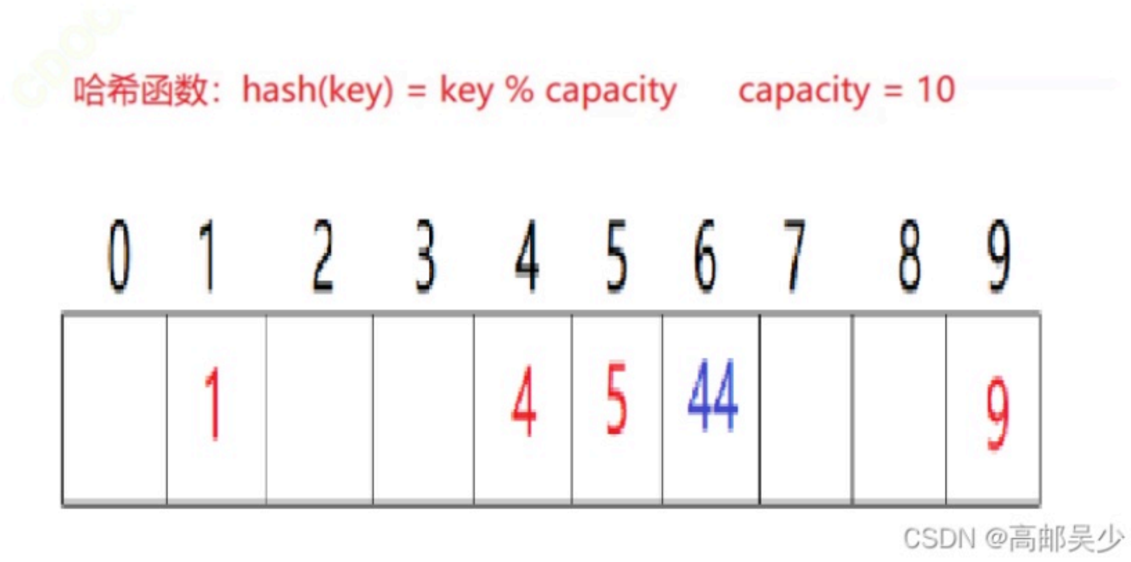
- One approach called open addressing is to search the array in some systematic way, looking for the next empty cell
  - if cats hashes to 5,421, but this location is already occupied by parsnip, then we could try to insert cats in slot 5,422
- A second approach is called separate chaining, where linked lists are stored in each slot of the array.
  - Add cats as a new node of the linked list stored in slot 5,421

1–1 Linear probing

线性探测：从发生冲突的位置开始，依次向后探测，直到寻找到下一个空位置为止

缺点：线性探测会把冲突的元素挨在一起，这就会造成，你如果要进行删除，比如我们现在要删除4，一旦我们不采取其他措施直接删除，我们后面想找到44就困难了

我们现在要往表里面放44,hash（44）=44%10=4，放在4位置，但是4位置已经被4占了，再往后一格；下标5的位置也有数；再往后一格，下标6位置没有数据，44放下标6的位置。



1–2 Quadratic probing

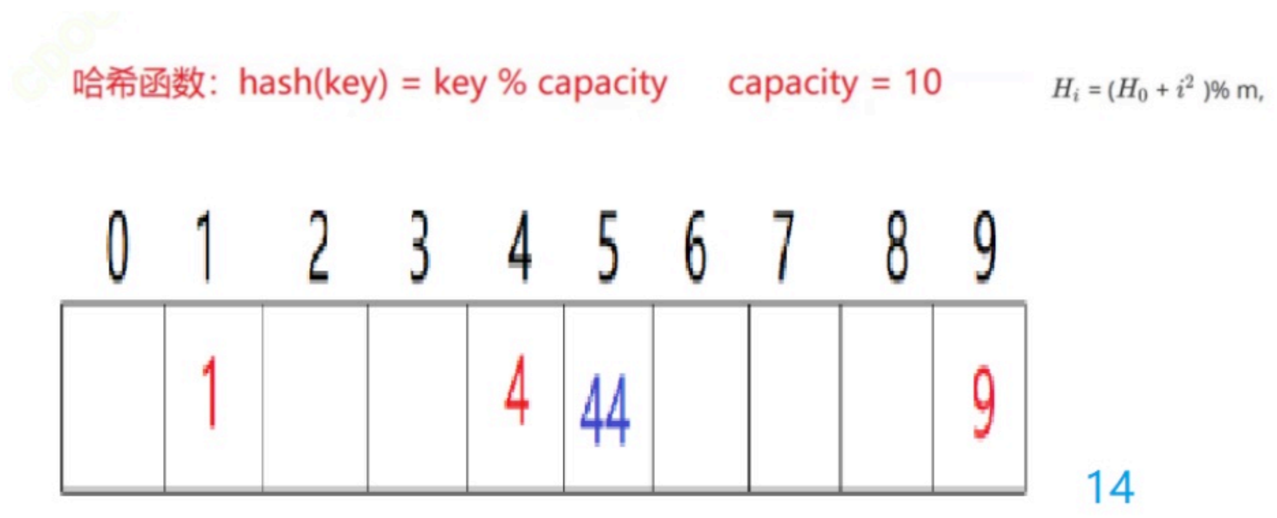
法二.二次探测：

线性探测的缺陷是产生冲突的数据堆积在一块，这与其找下一个空位置有关系，因为找空位置的方式就是挨着往后逐个去找，因此二次探测为了避免该问题，找下一个空位置的方法为：H_i = (H_0 + i^2)% m, 或者：H_i = (H_0 - i^2)% m。其中：i = 1,2,3…，H_0是通过散列函数Hash(x)对元素的关键码 key 进行计算得到的位置，m是表的大小。

简单来说就是冲突后放法改变了，放在Hi的位置，H0是你通过哈希函数得到的位置，比如hash(14)=14%10=4,这个4就是H0，i是第几次冲突，m是表大小，示例如下：

```
Hi = (H0 + i^2)% m
```

hash（44）=44%4=4,4与下标4里面的数据冲突，那就放在Hi=（4+1^2）%10=5，也就是下标5的位置；下标5内没有数据，不发生冲突，44就放进去了。



hash（44）= 44 % 4 = 4

4与下标4里面的数据冲突，那就放在Hi=（4+1^2）%10=5，也就是下标5的位置；

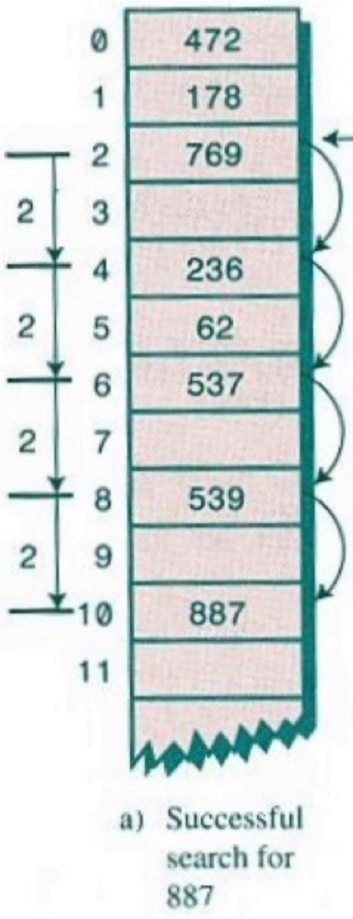
下标5内没有数据，不发生冲突，44就放进去了。

> 因此：比散列最大的缺陷就是空间利用率比较低，这也是哈希的缺陷。
>
> *二次探测是防止集群（cluster）形成的一种尝试；二次探测可以消除主要的聚类，但它们会导致更微妙的聚类问题。当许多项散列到同一个槽位时，会导致二次集群。*

1–3 Double probing

Double Hashing

- The secondary hash function takes in the unique number for the item and outputs a jump size in a given range.
  - The jump size is obviously smaller than the size of the hash table.
- The secondary hash function must have certain characteristics.
  - It must not be the same as the primary hash function.
  - It must never output a 0 (or else there would be no step, the algorithm would go into an infinite loop, and the program would seize up).
- The following formula outputs a value between 1 and MAX
  - jumpSize = MAX – (unique number % MAX)
  - Let MAX be 7 and unique number be 258
  - jumpSize = 7 – (258 % 7) = 1



Array size of 59

Unique number = 887

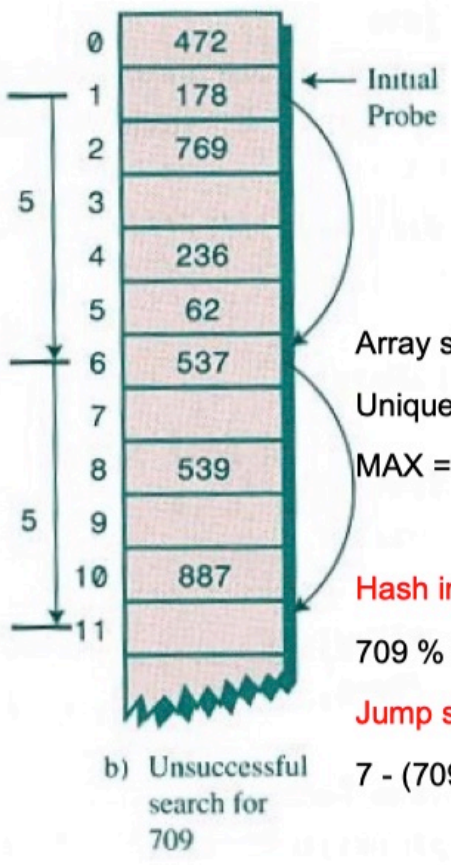
MAX = 7

Hash index =

887 % 59 = 2

Jump size =

7 - (887 % 7)= 2



Array size of 59

Unique number = 709

MAX = 7

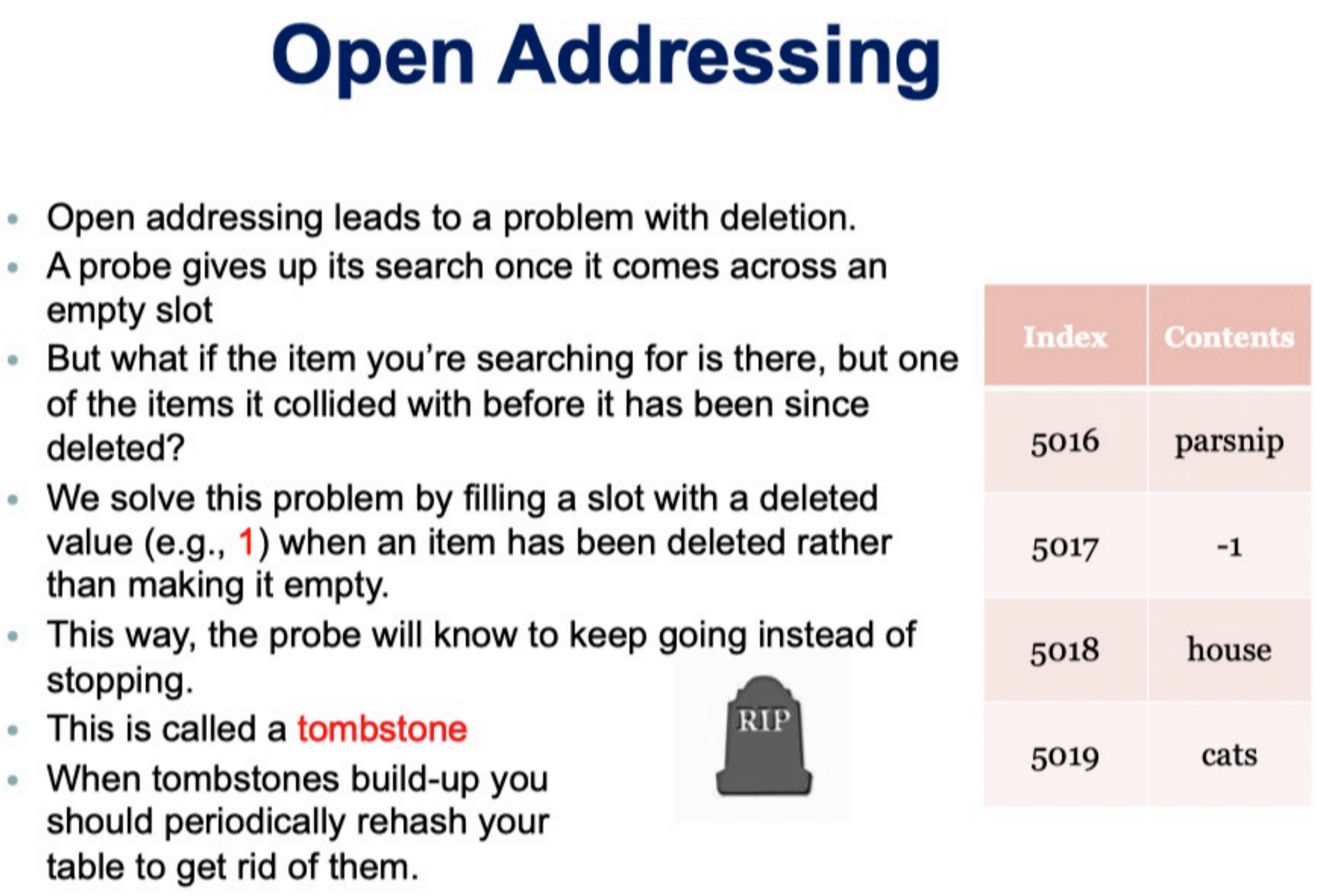
Hash index =

709 % 59 = 1

Jump size =

7 - (709 % 7)= 5

Open Addressing (overall)



Open Addressing

- Open addressing leads to a problem with deletion.
- A probe gives up its search once it comes across an empty slot
- But what if the item you're searching for is there, but one of the items it collided with before it has been since deleted?
- We solve this problem by filling a slot with a deleted value (e.g., 1) when an item has been deleted rather than making it empty.
- This way, the probe will know to keep going instead of stopping.
- This is called a tombstone
- When tombstones build-up you should periodically rehash your table to get rid of them.

Index	Contents
5016	parsnip
5017	-1
5018	house
5019	cats

Tomb：删除之后，为避免探针放弃搜索，使用（1&–1）代替被删除的数值。

此时（1&–1）被成为Tomb

2 Separate Chaining

开散列法又叫链地址法(开链法)，首先对关键码集合用散列函数计算散列地址，**具有相同地址的关键码归于同一子集合**，每一个子集合称为一个桶，各个桶中的元素通过一个单链表链接起来，**各链表的头结点存储在哈希表中。**

开散列，可以认为是把一个在大集合中的搜索问题转化为在小集合中做搜索了。

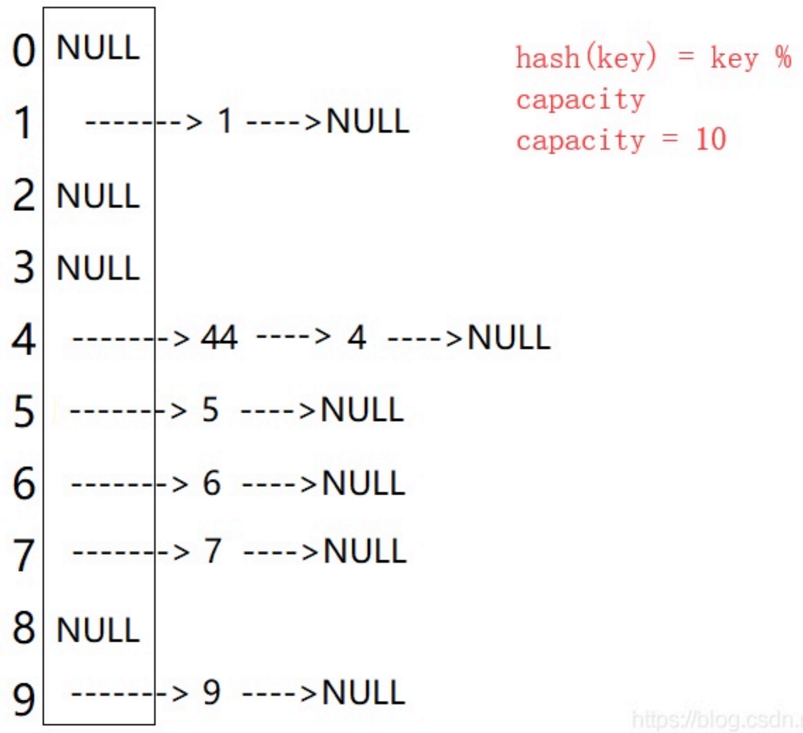
Separate Chaining

- Rather than trying to put each item in a separate slot, another approach is to install a linked list in each slot.
- If collisions occur and several items end up in the same slot, then they are simply added to the linked list.
- When looking for an item at a particular slot, simply search through the linked list containing all items hashing to that slot.
- So long as the linked lists don't become too long, then the performance shouldn't be too badly affected.

开散列（链地址法）

概念

对关键码集合用散列函数计算散列地址，将具有相同地址的关键码归于同一子集合，每一个子集合称为一个桶 (哈希桶)，各个桶中的元素通过一个单链表链接起来，各链表的头结点存储在哈希表中。这样就解决了哈希冲突。



```
0 NULL
1    -------> 1 ---->NULL
2 NULL
3 NULL
4   -------> 44 ----> 4  ---->NULL
5   -------> 5  ---->NULL
6   -------> 6  ---->NULL
7   -------> 7  ---->NULL
8 NULL
9   -------> 9  ---->NULL
```

```
hash(key) = key % capacity
capacity = 10
```

Load Factor 负载因子

负载因子 = 存储散列表的元素个数 / 散列表的长度

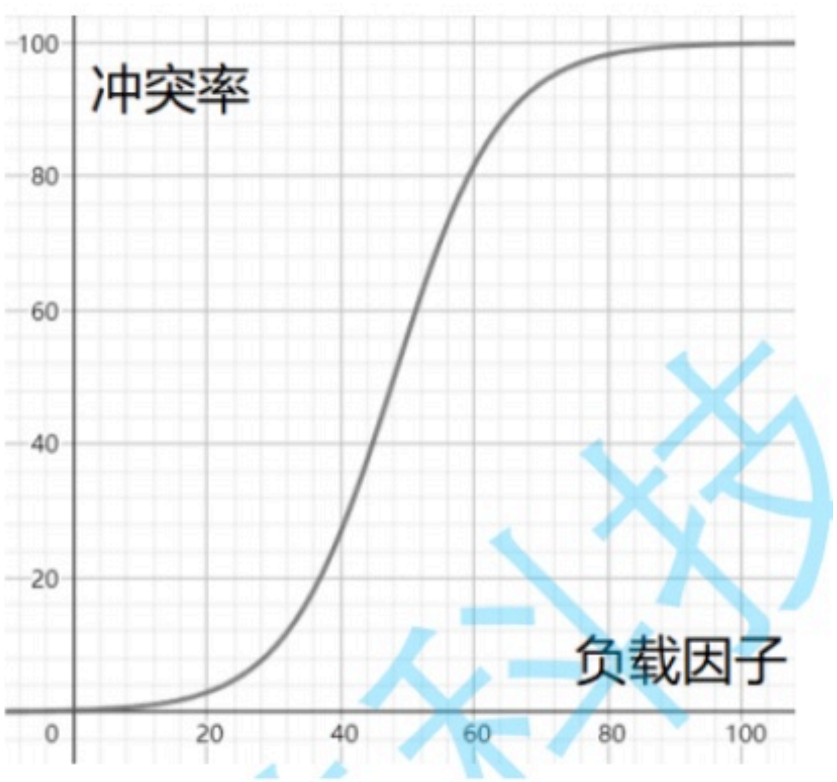
% 我们需要增加散列表的长度，来间接减小负载因子

3.2负载因子调节（重点）

负载因子=存储散列表的元素个数/散列表的长度

一般我们的哈希Map负载因子是0.75，如果你的负载因子超过0.75，就需要做出修改了

负载因子和冲突率的关系粗略演示

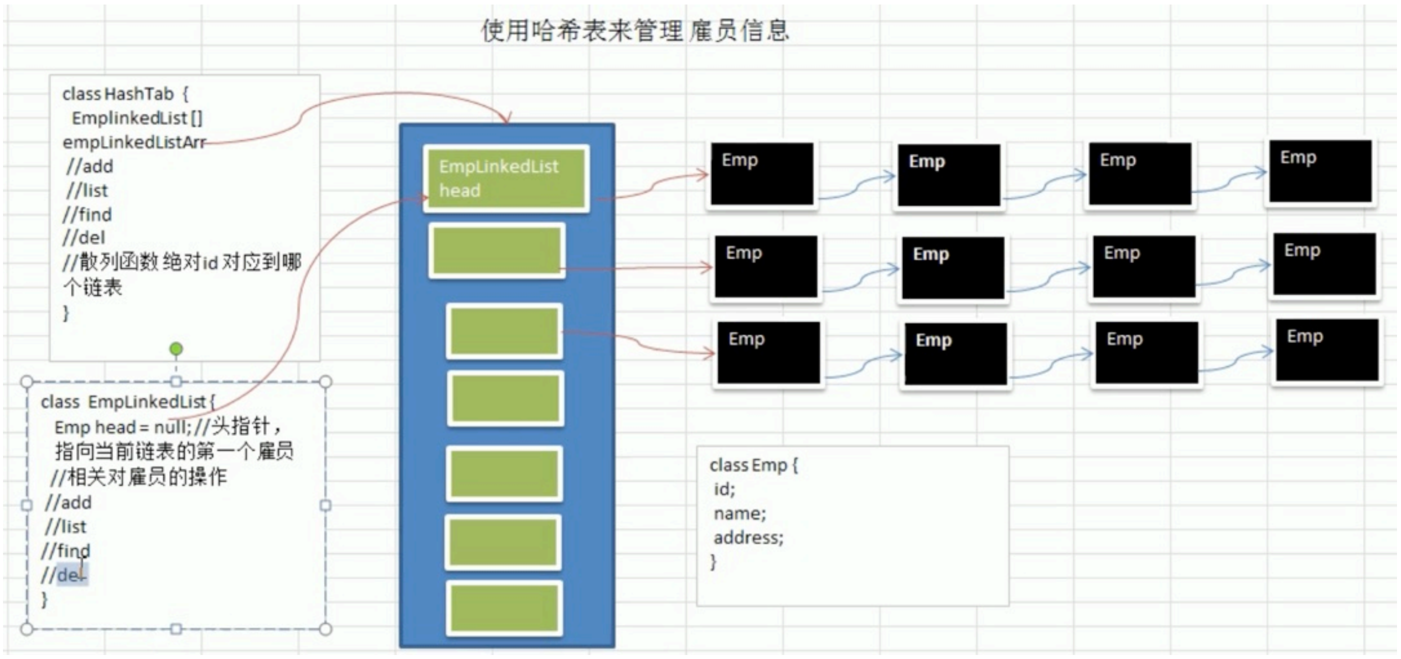


Ex 使用哈希表来管理雇员信息（尚硅谷）：

```
class HashTable{};

class EmpLinkedList{};

class Emp{};
```



```
class HashTab {
 EmplinkedList []
empLinkedListArr
 //add
 //list
//find
//del
//散列函数 绝对id 对应到哪
个链表
}
```

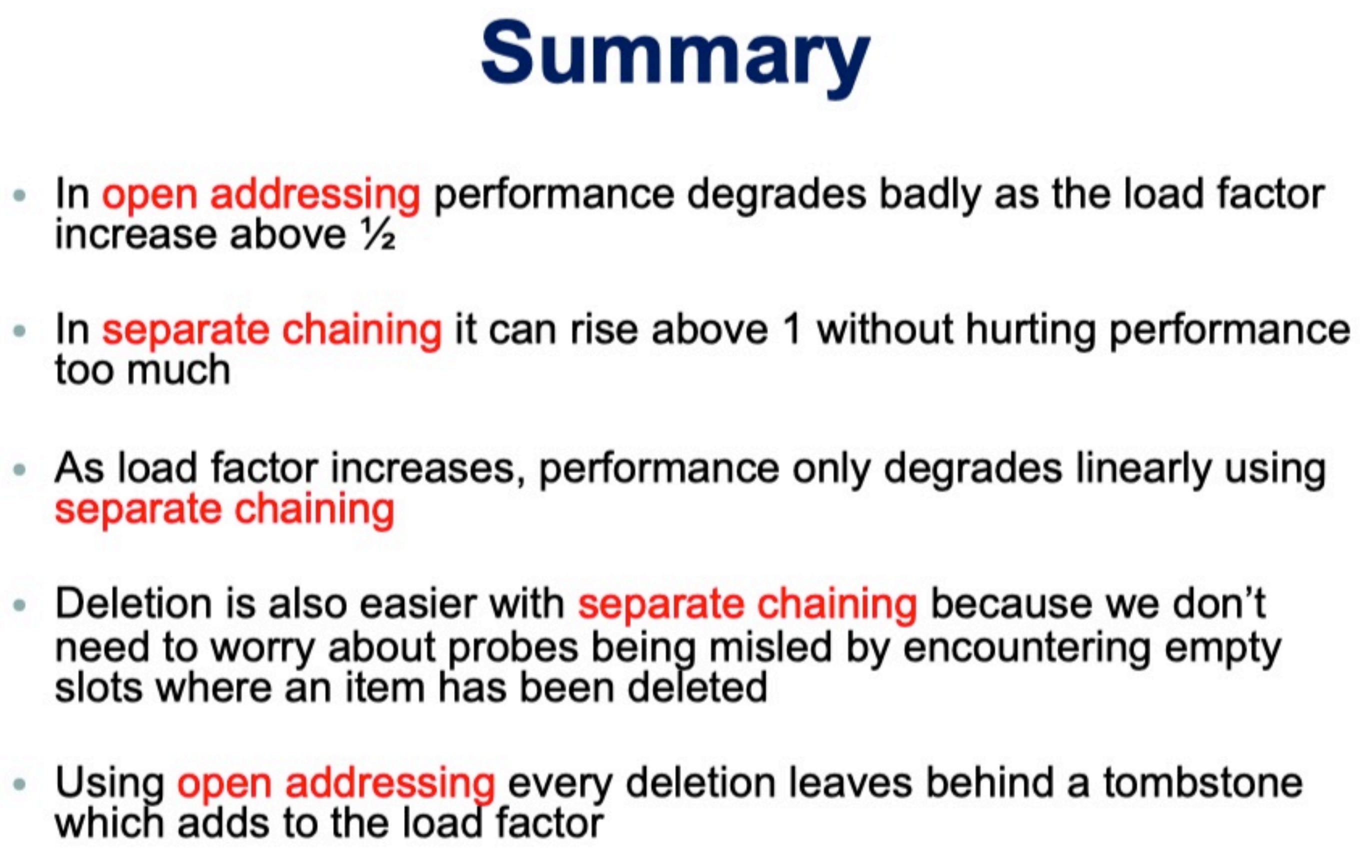
```
class EmpLinkedList{
  Emp head = null; //头指针，
  指向当前链表的第一个雇员
 //相关对雇员的操作
//add
 //list
//find
//del
}
```

```
class Emp {
 id;
 name;
 address;
}
```

具体代码请见：

`http://t.csdn.cn/qpTy2`

5.5 Overall



Summary

- In open addressing performance degrades badly as the load factor increase above ½
- In separate chaining it can rise above 1 without hurting performance too much
- As load factor increases, performance only degrades linearly using separate chaining
- Deletion is also easier with separate chaining because we don't need to worry about probes being misled by encountering empty slots where an item has been deleted
- Using open addressing every deletion leaves behind a tombstone which adds to the load factor

Summary

- If using open addressing, double hashing is slightly better than quadratic probing
- However, if there's plenty of memory and the data won't expand after the table is created then linear probing can be handier to implement and will cause little performance penalty with a load factor under ½
- If the number of items to be inserted into the hash table is unknown then separate chaining is a safer bet
- If there are going to be many deletions then separate chaining is the way to go because an open addressing hash table will fill up with tombstones

1、在*开放寻址*中，当负载因子增加到½以上时，性能会严重下降；

2、在*单独链接*中，它可以在不太影响性能的情况下超过1；

3、当负载因子增加时，使用*单独链接*只会线性降低性能；

4、使用*单独链接*删除也更容易，因为我们不需要担心在条目被删除的地方会遇到空槽，从而误导探测；

5、使用*开放寻址*，每次删除都会留下一个墓碑，增加负载因子；

6、如果使用*开放寻址*，双哈希略好于二次探测；

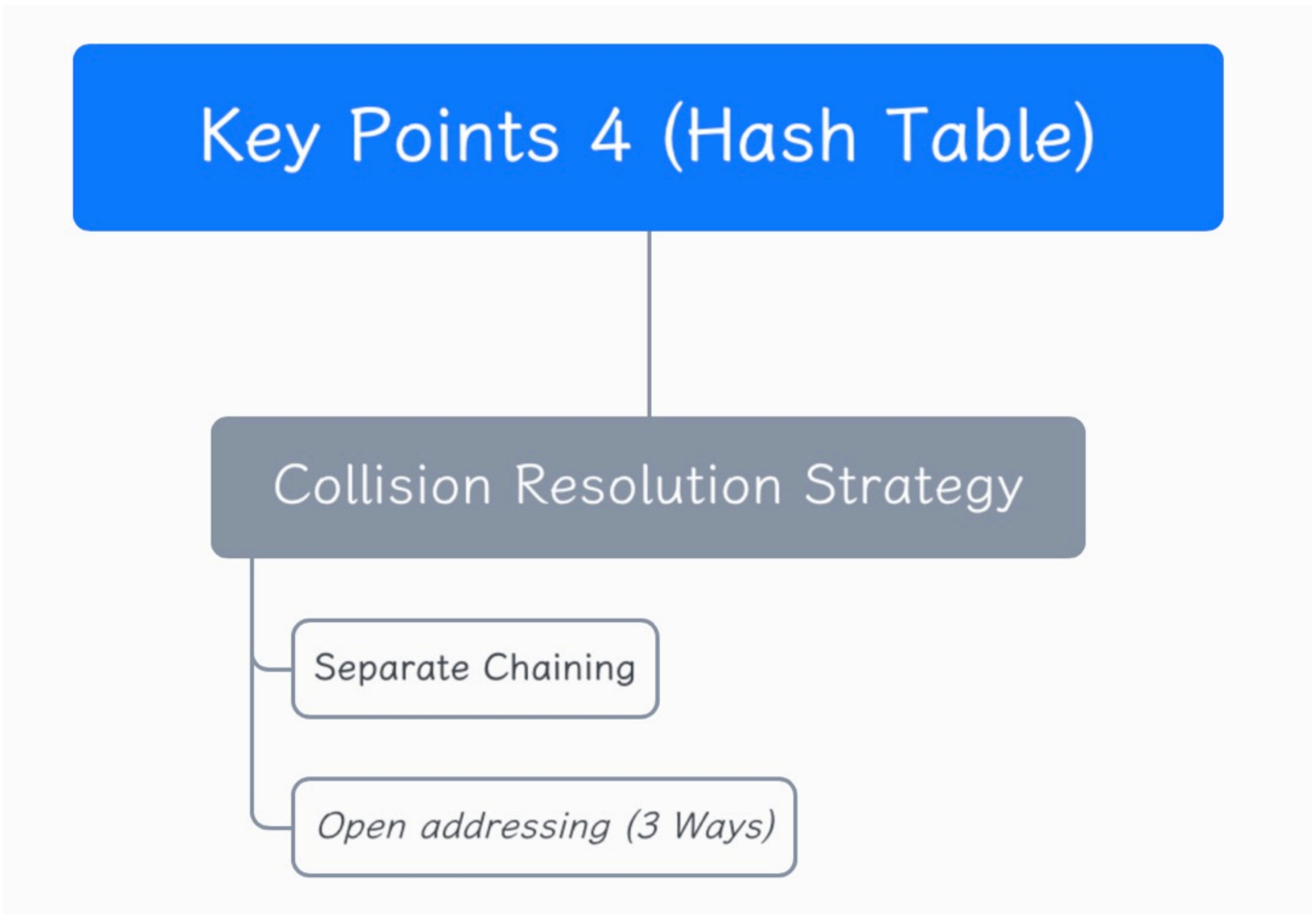
7、然而，如果有足够的内存，并且在表创建后数据不会扩展，那么*线性探测*可以更方便地实现，并且在负载系数小于½的情况下不会对性能造成什么损失；

8、如果插入哈希表的条目数量未知，那么*单独链接*是一个更安全的选择；

9、如果要进行多次删除，那么*单独链接*是最好的方法，因为一个*开放寻址*哈希表会被“墓碑”填满。

CS211FZ Note 4 Review (Hash Table)

- Collision Resolution Strategy
  - **Separate Chaining**
  - *Open addressing (3 Ways)*



`CS211 Note-4`

`by Lance Cai`

`2022/07/05`