

极客时间算法训练营

第三课

哈希表、集合、映射

李煜东

《算法竞赛进阶指南》作者



目录

1. 哈希表的原理与实现
2. 无序集合、映射的实现与应用
3. 实战：实现一个 LRU

哈希表的原理与实现

哈希表

哈希表（Hash table）又称散列表，是一种可以通过“关键码”（key）直接进行访问的数据结构。

哈希表由两部分组成

- 一个数据结构，通常是链表、数组
- Hash 函数，输入“关键码”（key），返回数据结构的索引

对外表现为可以通过关键码直接访问： $\text{hash_table}[\text{key}] = \text{value}$

实际上是在数据结构的 $\text{hash}(\text{key})$ 位置处存储了 value： $\text{data_structure}[\text{hash}(\text{key})] = \text{value}$

哈希表

最简单的例子，关键码是整数，定义 $\text{hash}(\text{key}) = \text{key}$

那这个哈希表其实就是一个数组了，key 自己就是下标

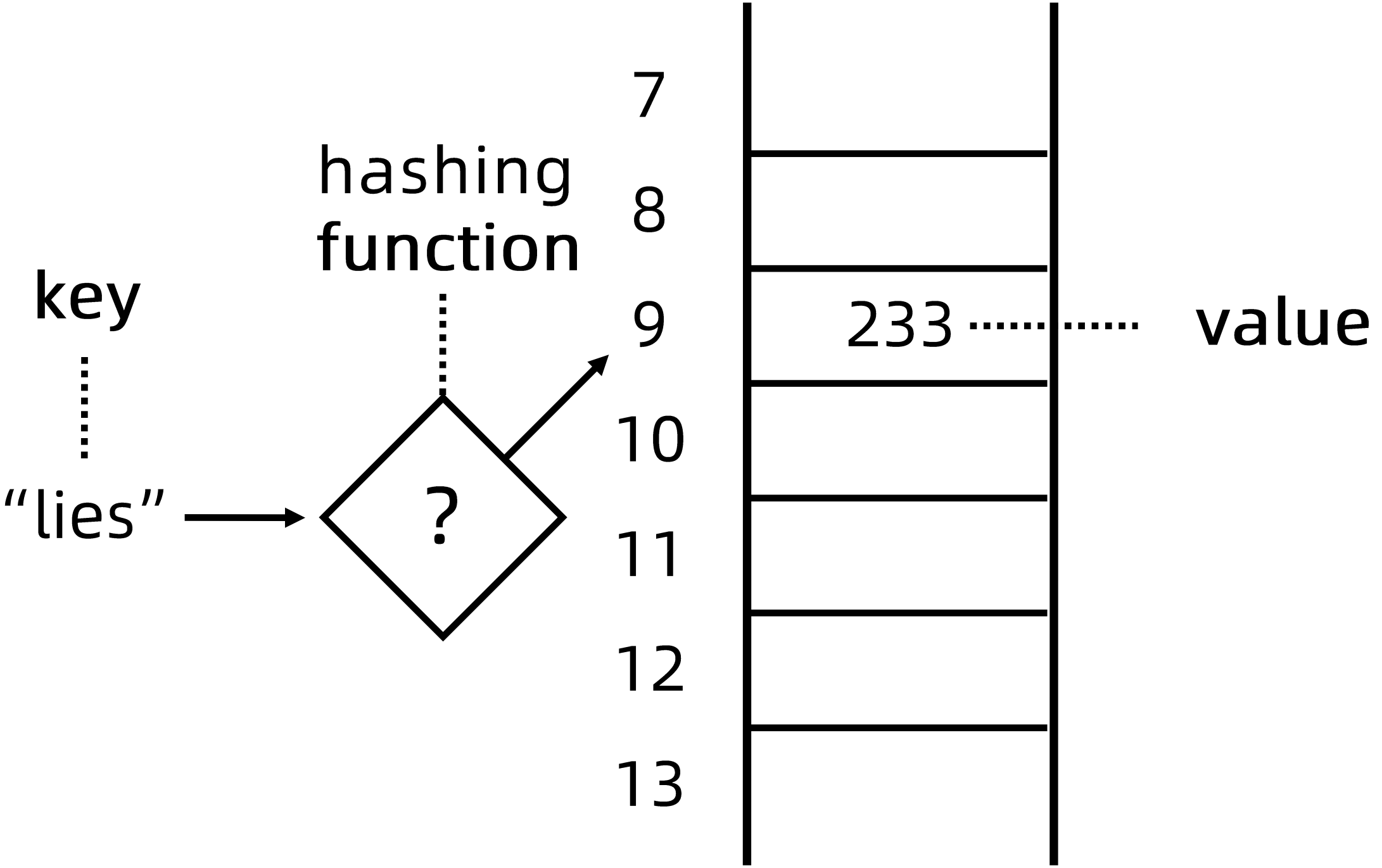
当然，一般情况下，关键码 key 是一个比较复杂的信息，比如很大的数、字符串

这时候 key 就不能直接作为数据结构的下标了

此时就需要设计一个 Hash 函数，把复杂信息映射到一个较小的值域内，作为索引

哈希表

一个简单的 `hash_table["lies"] = 233` 的例子，以各字符 ASCII 码相加 mod 20 为 Hash 函数



“ l i e s ”

↓ ↓ ↓ ↓

$$108 + 105 + 101 + 115 = 429$$
$$429 \bmod 20 = 9$$

哈希碰撞

哈希碰撞（Collisions）指的是两个不同的 key 被计算出同样的 Hash 结果

把复杂信息映射到小的值域，发生碰撞是不可避免的

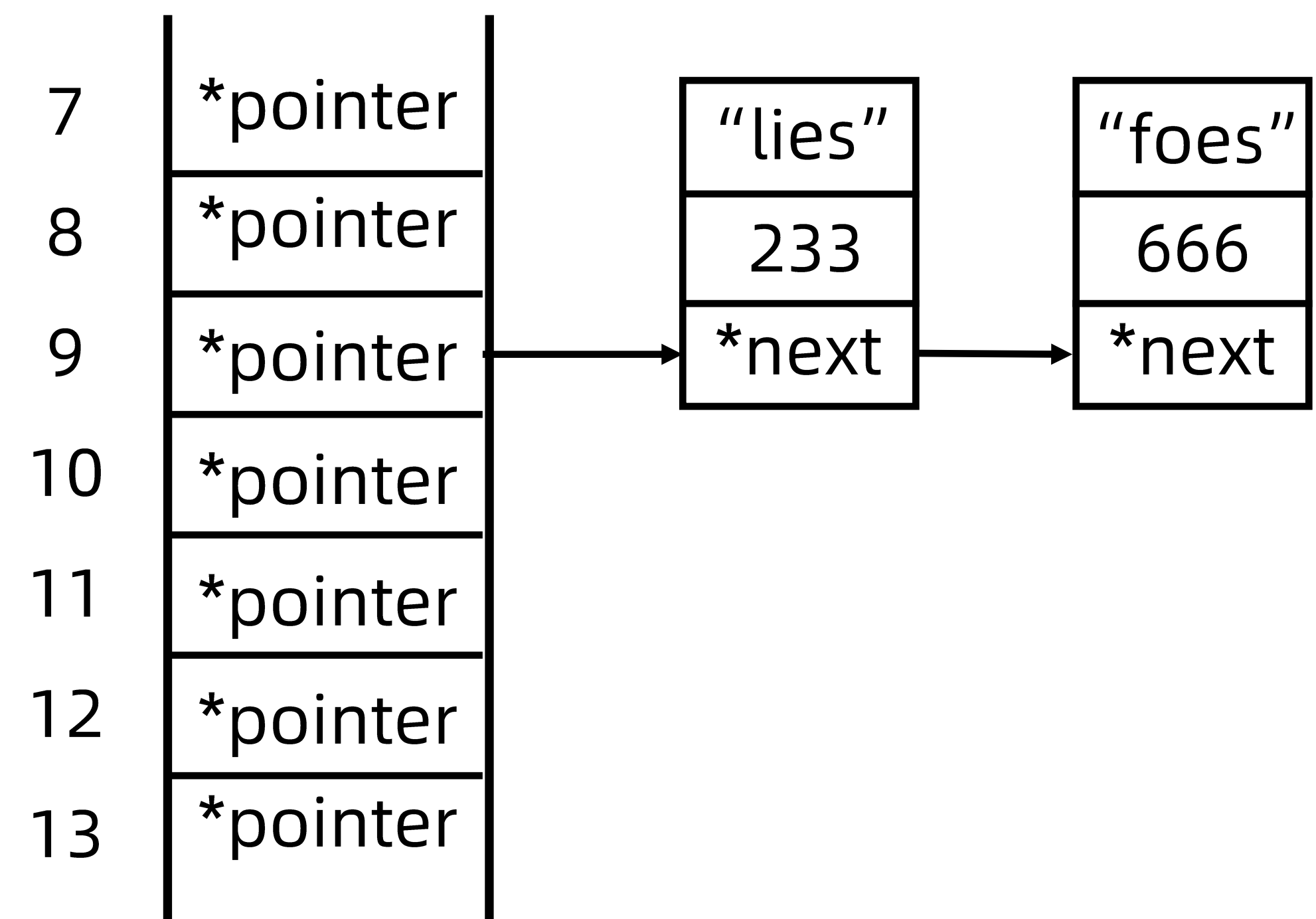
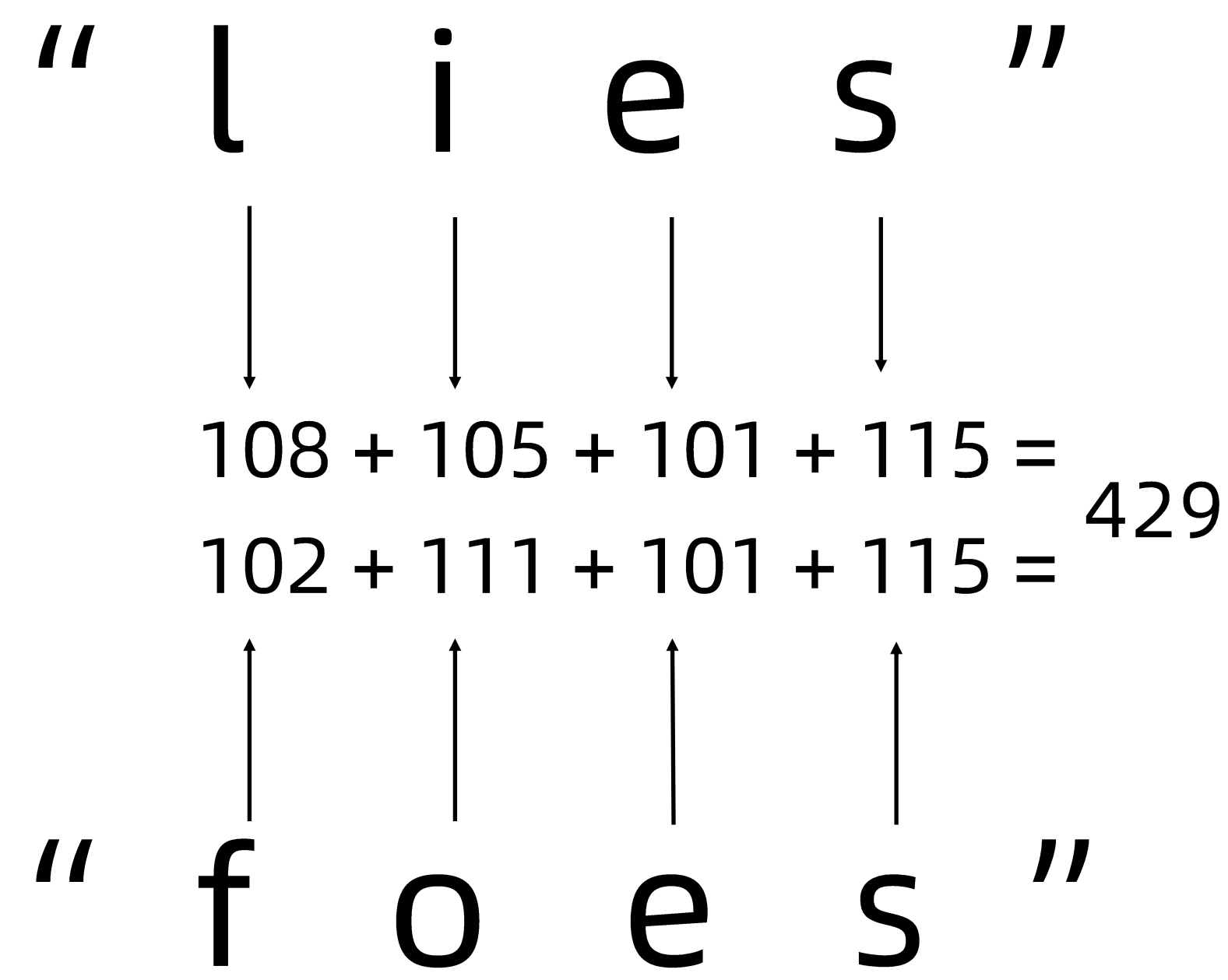
好的 Hash 函数可以减少碰撞发生的几率，让数据尽可能地均衡分布

开散列是最常见的碰撞解决方案

- Hash 函数依然用于计算数组下标
- 数组的每个位置存储一个链表的表头指针（我们称它为表头数组）
- 每个链表保存具有同样 Hash 值的数据

形象描述：“挂链”——表头数组每个位置“挂”着一个链表

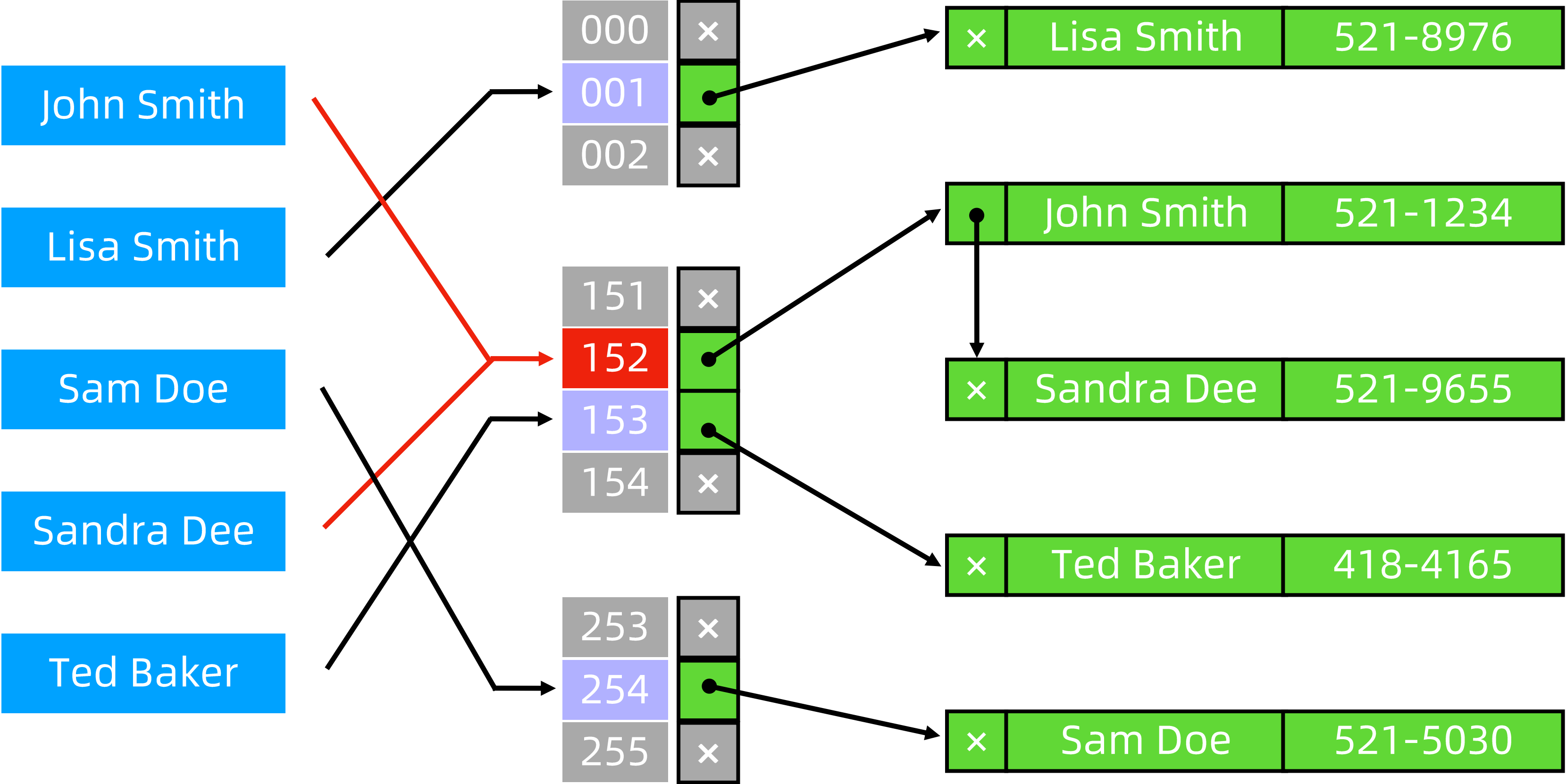
哈希碰撞 + 开散列



工程应用

- 电话号码簿
- 用户信息表
- 缓存 (LRU Cache)
- 键值对存储 (Redis)

完整结构图



时间复杂度

- 期望：插入、查询、删除 $O(1)$
 - 数据分布比较均衡时
- 最坏：插入、查询、删除 $O(n)$
 - 数据全部被映射为相同的 Hash 值时

无序集合、映射的实现与应用

集合与映射

集合 (set) 存储不重复的元素

- 有序集合，遍历时按元素大小排列，**一般用平衡二叉搜索树实现**， $O(\log N)$
- **无序集合，一般用 hash 实现**， $O(1)$

映射 (map) 存储关键码 (key) 不重复的键值对 (key-value pair)

- 有序集合，遍历时按照 key 大小排列，一般用平衡二叉搜索树实现， $O(\log N)$
- 无序集合，一般用哈希表实现， $O(1)$

对于语言内置的类型 (int, string)，已经有默认的优秀 hash 函数，可以直接放进 set/map 里使用

C++ code

set 与 unordered_set

- [文档](#) Unordered sets are containers that store **unique elements** in **no particular order**
- `unordered_set<string> s;` In an unordered_set, the value of an element is at the same time its key, Keys are immutable, therefore, the elements in an unordered_set **cannot be modified**
- insert, find, erase, clear 等方法
- *multiset*

map 与 unordered_map

- [文档](#)
- `unordered_map<string, int> h;`
- `h[key] = value`
- find(key), erase(key), clear 等方法
- *multimap*

Java code

Set: 不重复元素的集合, [文档](#), [示例](#)

- `HashSet<...> set = new HashSet<>()`
- `set.add(value)`
- `set.contains(value)`
- `set.remove(value)`

Map: key-value对, key不重复, [文档](#), [示例](#)

- `HashMap<..., ...> map = new HashMap<>()`
- `map.put(key, value)`
- `map.get(key)`
- `map.remove(key)`
- `map.clear()`

Python code

```
list_a = list([1, 2, 3, 4])
```

集合

```
set_a = {'jack', 'selina', 'Andy'}  
set_b = set(list_a)
```

字典

```
map_a = {  
    'Jack': 100,  
    '张三': 80,  
    'Candela': 90,  
}
```

实战

两数之和

<https://leetcode-cn.com/problems/two-sum/description/>

基本思路：枚举一个数 x ，找它前面有没有 $target - x$

所以建立一个数值到下标的hash map就可以了

对于每个数 x ，先查询 $target - x$ ，再插入 x

时间复杂度 $O(n)$

实战

模拟行走机器人

<https://leetcode-cn.com/problems/walking-robot-simulation/>

可以用 set 或者 map 存储障碍物，从而快速判断一个格子里有没有障碍
可以利用方向数组简化实现（代替 if）

实战

字母异位词分组

<https://leetcode-cn.com/problems/group-anagrams/>

对字符串分组，其实就是进行 Hash

让同一组的字符串具有相同的 Hash 函数值，不同组的字符串具有不同的 Hash 函数值

然后就可以用 hash map 分组了

方案一：把每个字符串中的字母排序，排序后的串作为 hash map 的 key

map<string, group>

方案二：统计每个字符串中各字母出现次数，把长度为 26 的计数数组作为 key

map<array<26, int>, group> (C++ std::array, Python tuple)

C++ 中内置支持对于数组array类型的hash

实战

串联所有单词的子串

<https://leetcode-cn.com/problems/substring-with-concatenation-of-all-words/>

遇到难题，先分解

不会求解，可以先想想判定：

给出一个 s 的子串、words，判定这个子串是不是 words 的串联？

把子串划分以后，其实就是比较两个 Hash map 是否相等

"barfoothe foo bar man"	mapA = {"bar": 1, "foo": 1}
["foo","bar"]	mapB = {"bar": 1, "foo": 1}
	mapA == mapB

实战

串联所有单词的子串

<https://leetcode-cn.com/problems/substring-with-concatenation-of-all-words/>

回到原问题：

枚举子串的所有起始位置， $O(\text{length of } s * \text{total length of words})$

barfoothefoobarman → barfoothefoobarman → ...

枚举部分起始位置 + 滑动窗口， $O(\text{length of } s * \text{length of one word})$

barfoothefoobarman → barfoothefoobarman → ...

barfoothefoobarman → barfoothefoobarman → ...

...

实战：实现一个 map

选做：自己动手，用哈希表（开散列）实现一个无序映射（map）

固定以 string 为关键码

支持 find, put, remove 方法即可

实战：实现一个 LRU

Cache

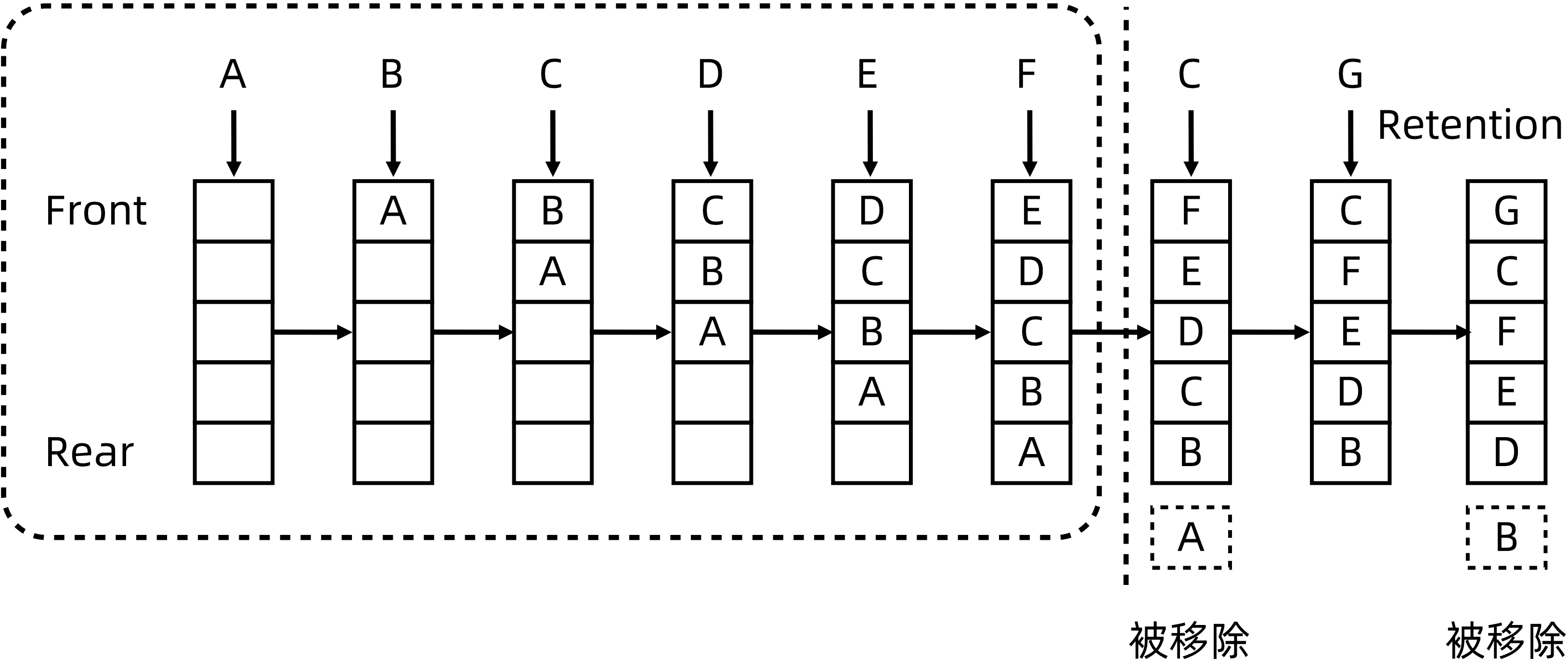
缓存的两个要素：大小、替换策略

常见替换算法：

- LRU - least recently used, 最近最少使用（淘汰最旧数据）
- LFU - least frequently used, 最不经常使用（淘汰频次最少数据）

LRU cache

根据LRU算法的定义与功能分析，需要：
线性表
支持操作 头部插入、尾部删除、中间删除、查询访问某个给定元素
==>链表+无序映射map



实战：实现一个 LRU

<https://leetcode-cn.com/problems/lru-cache/>

哈希表 + 双向链表

- 双向链表用于按时间顺序保存数据
- 哈希表用于把 key 映射到链表结点（指针 / 引用）

$O(1)$ 访问：直接检查哈希表

$O(1)$ 更新：通过哈希表定位到链表结点，删除该结点（若存在），在表头重新插入

$O(1)$ 删除：总是淘汰链表末尾结点，同时在哈希表中删除

Homework

子域名访问计数

<https://leetcode-cn.com/problems/subdomain-visit-count/>

数组的度

<https://leetcode-cn.com/problems/degree-of-an-array/>

元素和为目标值的子矩阵数量

<https://leetcode-cn.com/problems/number-of-submatrices-that-sum-to-target/>

THANKS

 极客时间 | 训练营