# 《L4-CSP-J 复赛进阶算法与数据结构》

## Day11 集合

## (并查集-Hash表)

主讲人：小武老师

可达信奥_小武老师_keda.ac

# 并查集

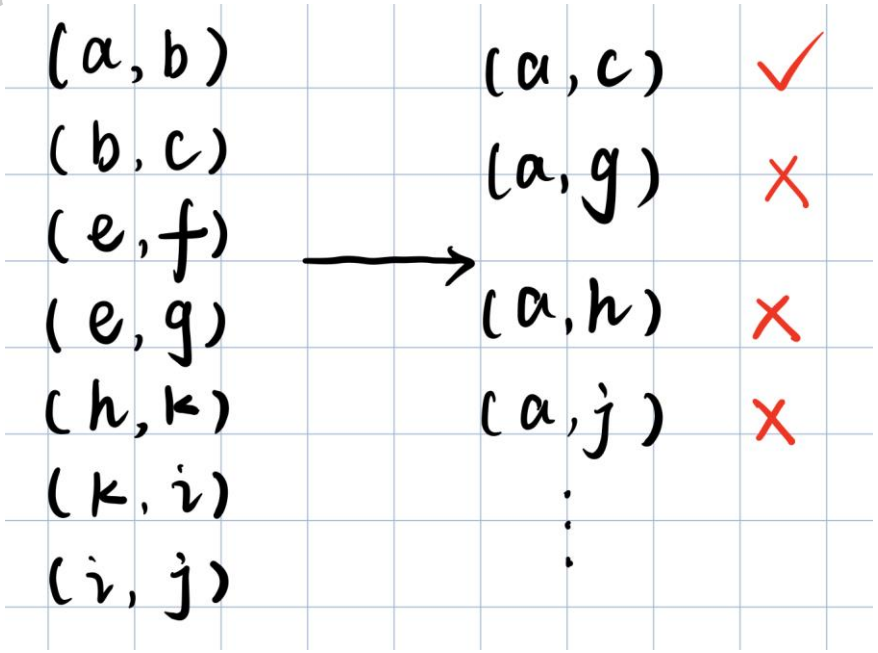
有时候，并不关心数据之间的前后关系，也不关心数据的层次关系。一些确定元素只是单纯的聚集在一起，这样的元素聚集体被称为集合。

当希望知道某个数据是否存在一个集合中，或者两个元素是否在同一个集合中，就需要使用一些集合数据结构来维护集合元素之间的关系。

可达信奥_小武老师_keda.ac

# 并查集

假设有n个村庄，有些村庄之间有连接的路，有些村庄之间并没有连接的路，给出有道路直接连通的村庄的组合列表，如(a,b), (c,d), (a,c)分别表示两个村庄之间有道路直接连通。基于此，我们希望能快速查询任意两个村庄之间是否连通？有什么办法？




图的深度优先搜索

图的广度优先搜索

并查集?

# 并查集

并查集(union-find set)就是根据对象两两之间的直接关系来快速查询任意两个对象之间是否存在关系。本质上，并查集是一组集合，存在直接或间接关系的对象放到一个集合中，不存在任何关系的对象被分割到不同的集合中。

查：查询两个对象之间是否存在关系。

并：实际上就是将两个集合合并在一起，如果隶属于两个集合的某两个对象之间存在关系，我们就将这两个集合合并为一个集合。

# 并查集

并查集是一种数组存储的树型数据结构，用于处理一些不相交集合的合并及查询问题。

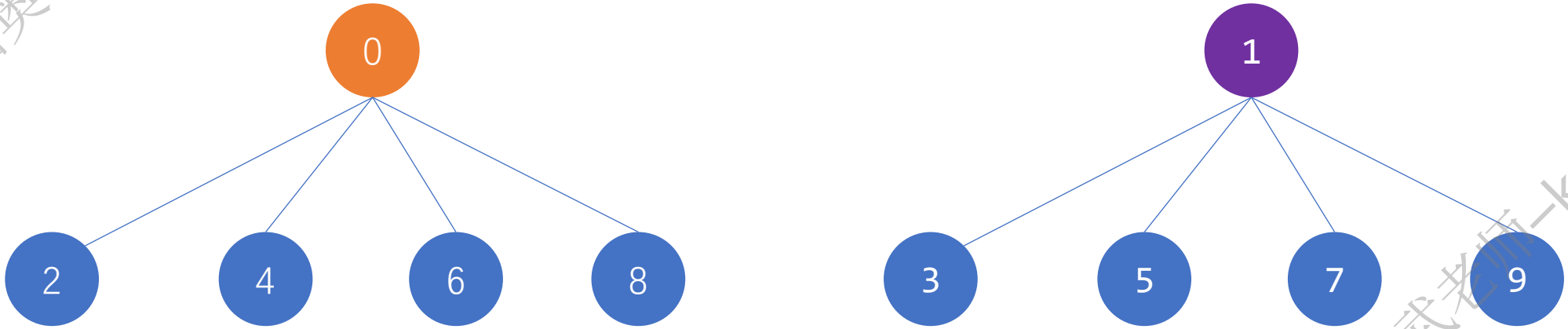
并查集的思想是用一个数组表示了整片森林（parent），树的根节点唯一标识了一个集合，我们只要找到了某个元素的的树根，就能确定它在哪个集合里。

| 元素编号 | id | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 集合编号 | n | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |

| 元素编号 | id | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 集合编号 | n | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |

# 并查集

| 元素编号 id | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 集合编号 n | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |



# 集合

集合，数学中默认指无序集，用于表达元素的聚合关系。两个元素只有属于同一个集合与不属于同一集合两种关系。

常见应用场景：

- 两个元素是否属于同一个集合？
- 一个元素是否在集合中存在？

常见实现方式：

- std::unordered_set、std::unordered_map
- 并查集、哈希表
- 启发式可并堆

# 集合：偏序集

在实际应用中，可能需要关心集合元素顺序。集合上定义偏序关系（即≤号），可构成一个偏序集。有序性作为重要规律，可引入算法（如二分法）提升运行效率。

常见应用场景：

- 某个元素在的前驱/后继是什么？
- 插入/删除若干元素，使集合仍然有序？
- 某个元素是第几名？第几名是哪个元素？

常见实现方式：

- std::set，std::map
- 二叉排序树（平衡二叉树）

男
张三 100分
李四 98分
靓仔59分

# 第一个问题：“两个元素是否属于同一个集合？”

P0574. 亲戚

并查集

# P0574. 亲戚

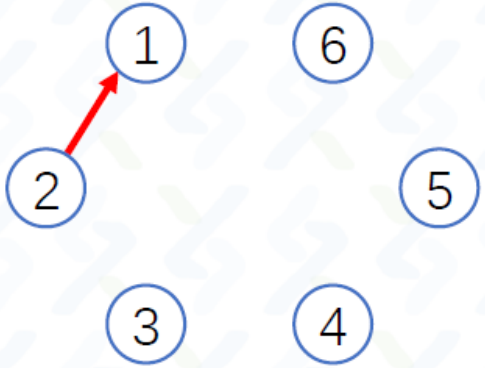
观察样例的亲戚关系，1 的亲戚都可以通过某种方式追溯到1。是否可以利用这种追溯关系，让1来代表1 的整个家族呢？将家族视为集合，若两人是亲戚，等价两个元素属于同一集合。

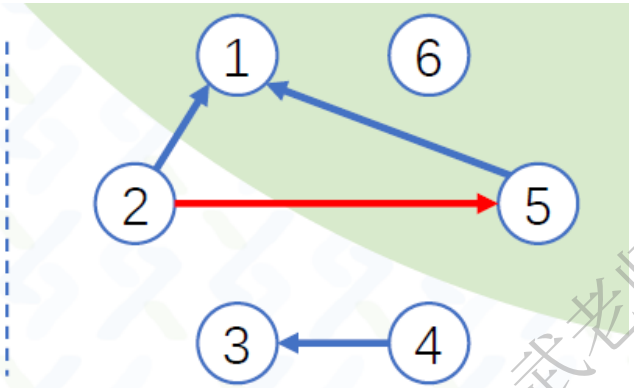

初始时，我们不知道任何人的亲属关系。可以认为每一个人单独属于一个集合，其代表为自己，如上图右。每添加一组亲属关系，则等价于合并两者所属的集合。

# P0574. 亲戚

考察第一组亲属关系<1,2>。
将1 所属的集合{1} 与2 所属的集合{2} 合并为{1,2}。
此处，我们假定以左侧元素1 作为这个集合的代表。那么1 的代表仍是1，2 的代表变为1。

```
1 2
1 5
3 4
5 2
1 3
```



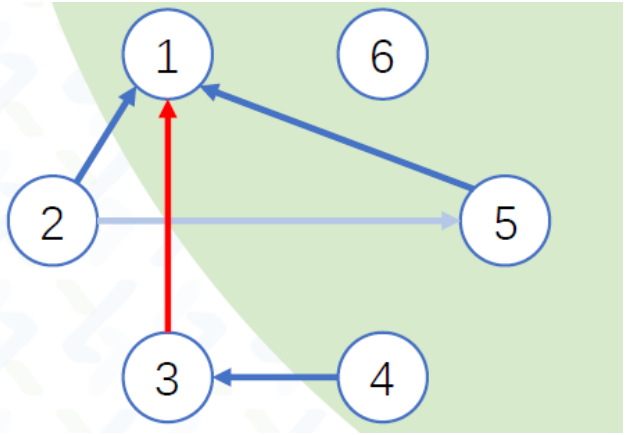




考察第四组亲属关系<5,2>。查询2、5 的代表，发现代表均为1，意味他们已属于同一个集合。
因此，这一条关系并没有增加任何的信息，可以忽略。

并查集

## P0574. 亲戚

```
1 2
1 5
3 4
5 2
1 3
```




考察第五组关系<1,3>。此时1、3 属于不同的集合{1,2,5} 和{3,4}，代表分别为1、3。

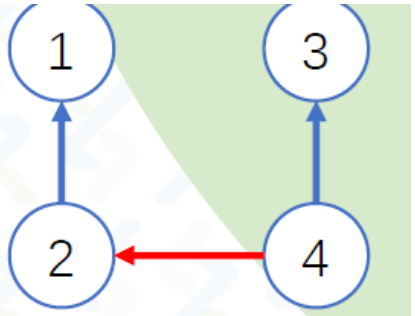
我们设置3 的代表为1。图中可以看到，代表关系具有传递性；4的代表也随着3 变为了1。

这样一来，{3,4} 也并入了集合{1,2,5}。

## P0574. 亲戚

但是考虑下面一种情况：

```
1 2
3 4
2 4
```




我们期望的结果是1、2、3、4 均有亲属关系。

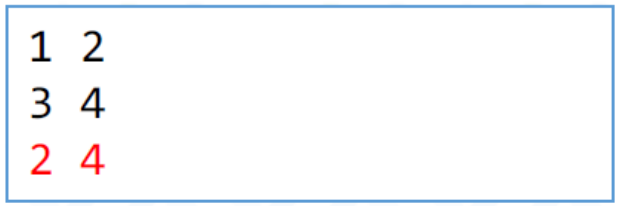
但若直接修改4 的代表为2，则丢失3、4 之间的关联，怎么办？

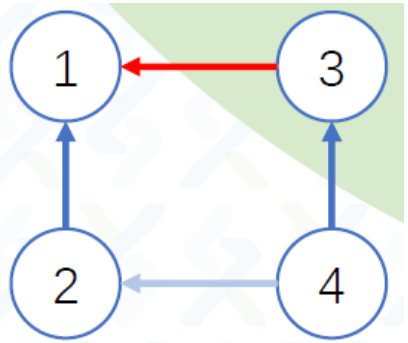
并查集

# P0574. 亲戚

代表关系具有传递性。我们不需要直接修改2、4的代表，而是修改2和4的代表的代表，那么2、4的关系也会改变，同时所属集合中其他所有元素也随代表被更改！

因此，先查找2 的代表为1，4 的代表为3。




然后合并，将3的代表设置为1。此时，4的代表也随着3变成了1。

这样，集合的合并与查询就都实现了，“并查集”因此得名。

并查集

# P0574. 亲戚

实现时，用数组fa来存储“代表”。代表具有传递性。当查找代表时，需要递归地向上，直到代表为自身。

```
int find(int x) { // 查询集合的“代表”
        if (x == fa[x])return x;
        return find(fa[x]);
}
```

当合并两个元素时，需先判断两者是否属于同一集合。若否，则将其中一个集合的代表设置为另一方的代表。

```
void join(int c1, int c2) { // 合并两集合
// f1为c1的代表，f2为c2的代表
     int f1 = find(c1), f2 = find(c2);
     if (f1 != f2) fa[f1] = f2;
}
```

# 并查集

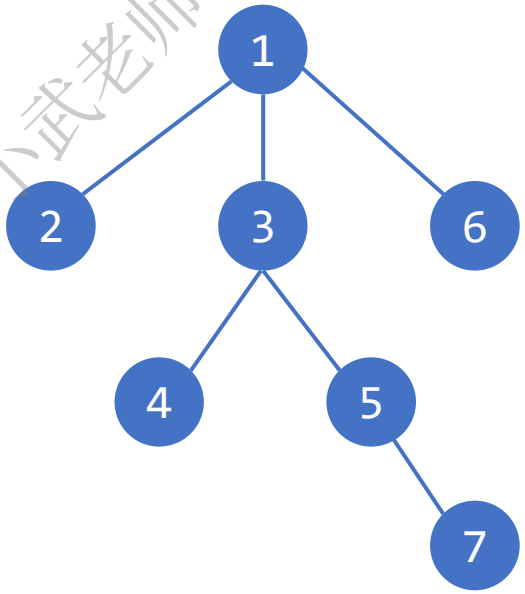
## P0574. 亲戚

```
int find(int x) { // 查询集合的“代表”
        if (x == fa[x])return x;
        return find(fa[x]);
}
```

```
void join(int c1, int c2) { // 合并两集合
// f1为c1的代表，f2为c2的代表
     int f1 = find(c1), f2 = find(c2);
     if (f1 != f2) fa[f1] = f2;
}
```

```
// 初始化
for (int i = 1; i <= n; ++i)
        fa[i] = i;
// 合并亲属关系
for (int i = 0; i < m; ++i) {
        cin >> x >> y;
        join(x, y);
}
// 根据代表是否相同，查询亲属关系
for (int i = 0; i < p; ++i) {
        cin >> x >> y;
    if (find(x) == find(y))
        cout << "Yes" << endl;
    else
        cout << "No" << endl;
}
```

# 并查集优化—路径压缩







```
int find(int x){
    if (x == f[x]) return x;
    return find(f[x]);
    //return f[x] = find(f[x]);
}
```

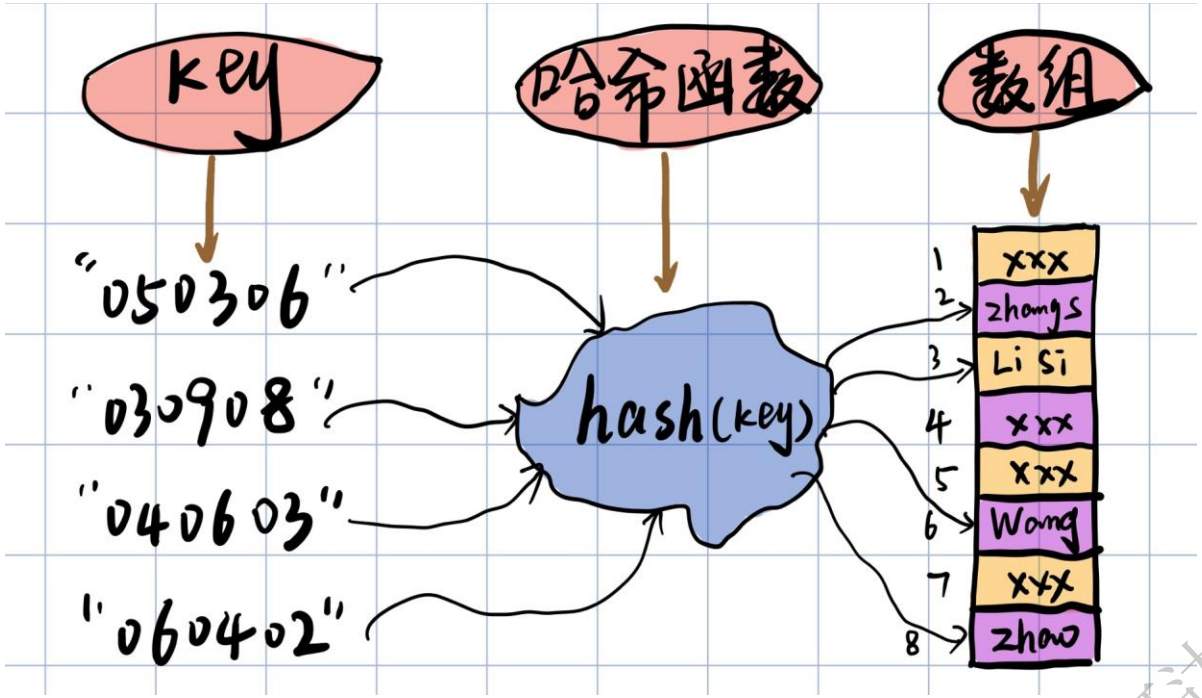
# Hash表

哈希表(hash table)是数组的一种扩展，由数组演化而来，底层依赖数组支持按下标快速访问元素的特性。

可达信奥_小武老师_keda.ac

# 哈希思想

假如有89名同学参加学校运动会，为了方便记录成绩，每个选手胸前都会贴上自己的参赛号码。每个参赛号码有6位数字组成，分别表示年级、班级及运动员编号。如051167表示，5年级11班的67号选手。现在我们希望编程实现这样一个功能：通过编号快速找到对应的选手信息，如何实现这个功能？



# 哈希函数

发现规律：哈希表利用的是数组按下标访问元素的时间复杂度是 O(1)这一特性。我们通过哈希函数把元素的键值映射为下标，然后，将对应的数据存储在数组中对应下标的位置。当按照键值查询元素时，我们使用同样的哈希函数，将键值转化为数组下标，从数组中这个下标对应的位置取数据。

在哈希函数设计时的3个基本要求：

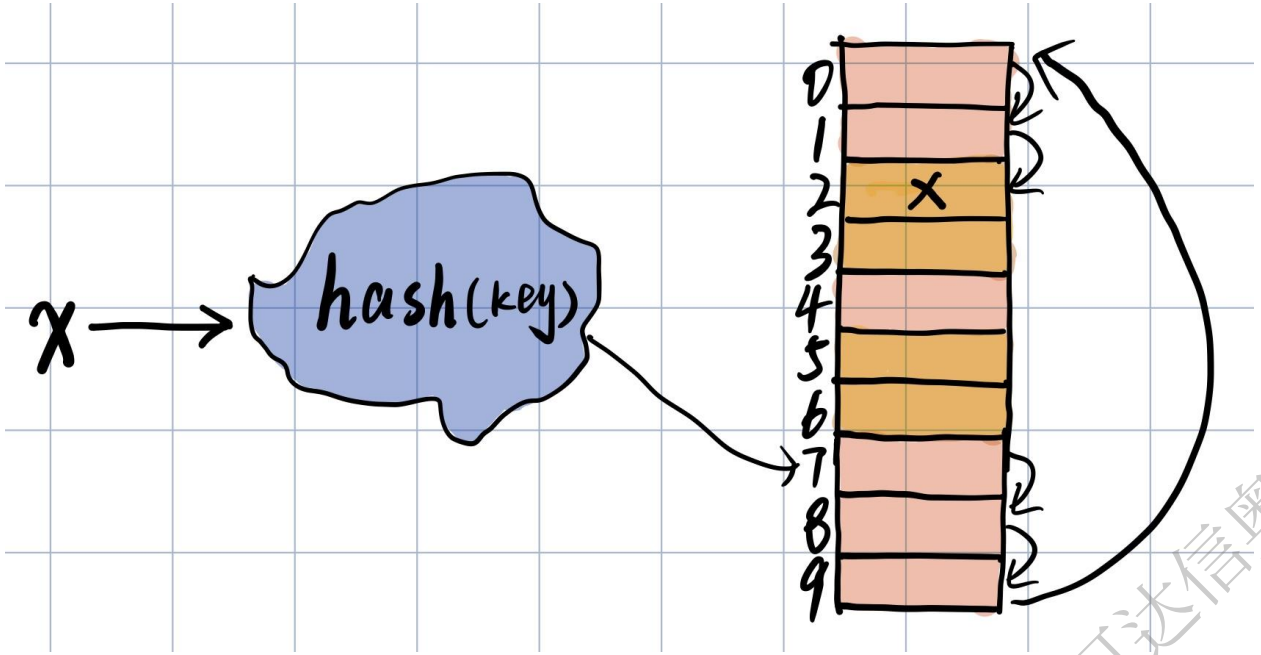
1. 哈希函数计算得到的哈希值是一个非负整数;
2. 如果key1 == key2, 那么hash(key1) == hash(key2);
3. 如果key1 ≠ key2, 那么 hash(key1) ≠ hash(key2) ;

# 哈希冲突

正常情况下，我们希望key不同，对应的哈希值也应该不同（否则就叫哈希冲突）。但这几乎是不可能的，因为数组的存储空间是有限的,所以也会加大哈希冲突的概率如何解决哈希冲突呢？常用的放法有两种：开放寻址法（open addressing）和链表法（chaining）

## 1. 开放寻址法

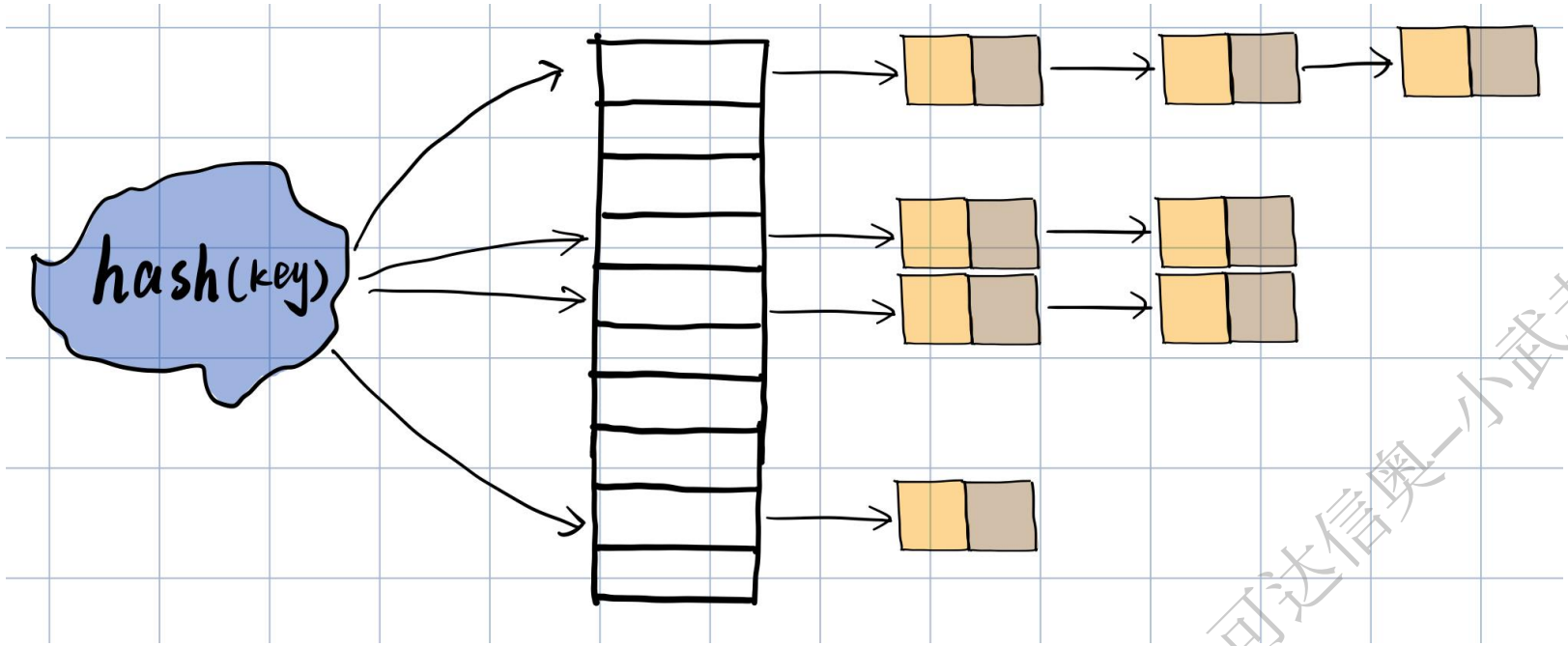
核心思想：有冲突往后放



# 哈希冲突

正常情况下，我们希望key不同，对应的哈希值也应该不同（否则就叫哈希冲突）。但这几乎是不可能的，因为数组的存储空间是有限的,所以也会加大哈希冲突的概率如何解决哈希冲突呢？常用的放法有两种：开放寻址法（open addressing）和链表法（chaining）

2．链表法　　核心思想：有冲突放在链表后



# 第二个问题：“一个元素是否存在于这个集合中？”

P0598 字符串哈希

# 字符串哈希

任意两个字符串两两比较时间上必然是不可取的，时间复杂度。时间只允许处理每一个字符串仅一次。联想计数排序的思想，我们可以将每一个字符串装入一个投票箱，使得相同的投票箱里只有同一类字符串。

| 票箱#1 | 票箱#2 | 票箱#3 | 票箱#4 | 票箱#5 |
|---|---|---|---|---|
| 1 | 7 | 0 | 0 | 2 |

至于如何将字符串变为数字，就取决于Hash函数如何构造了。

先考虑一个简单的问题：给定N个自然数，值域是[0, $10^{9}$]，求出这N 个自然数中共有多少个不同的自然数。

# 字符串哈希

将整数映射到较小的整数，不难想到一个经典的运算：取模

| 原数 | 1 | 14 | 514 | 1919810 | 1145141919810 |
|---|---|---|---|---|---|
| 模 11 | 1 | 3 | 8 | 2 | 9 |

该如何将字符串变成为整数编号呢？

回顾：字符在计算机中是以ASCII码（8位整数）的形式存储的。所以可进行整数运算，把字符串视作超长的256进制高精度整数。

| 字符 | L | u | o | g | u | 4 | ! |
|---|---|---|---|---|---|---|---|
| ASCII | 76 | 117 | 111 | 103 | 117 | 52 | 33 |

```
string s; // ......
int hash = 0;
for (int i = 0; s[i]; i++)
// 计算base进制下模mod的值作为hash值
hash = ((long long)hash * base + s[i]) % mod;
```

# 字符串哈希

计算哈希函数的base 和mod 根据经验选取。

- base 应当选择不小于字符集数的质数。例如，a-z 字符串为26，任意ASCII 字符串为256。
- 而mod 应该选取允许范围内尽可能大的质数。

```
int n, ans;
char s[MAXN];
vector <string> linker[mod + 2];
void insert();
int main() {
cin >> n;
for (int i = 1; i <= n; i++)
    cin >> s, insert();
cout << ans << endl;
return 0;
}
```

```
void insert() {
int hash = 1;
for (int i = 0; s[i]; i++)
hash = (hash * 1ll * base + s[i]) % mod;
//计算出字符串的hash值
string t = s;
for (int i = 0; i < linker[hash].size(); i++)
//遍历hash值为该字符串hash值的链表，检查字符串是否存在
if (linker[hash][i] == t)
return; //如果找到同样的字符串，这个字符串不计答案
linker[hash].push_back(t); //否则计入答案
ans++;
}
```

这里取base=261，mod=23333

# STL set/map

STL中也有集合的实现，分为无序集和偏序集。
其中分为集合(set) 和映射(map)。

可达信奥_小武老师_keda.ac

# STL set

集合在STL 中有两种，分别是有序集合和无序集合

```
unordered_set<Type> s; //创建Type类型的集合
s.insert(x); // 插入元素x
s.erase(x); // 删除值为x的元素
s.erase(it); // 删除it所指的元素
s.end(); // 返回末位哨兵的迭代器
s.find(x); // 查询x；不存在则返回s.end()
s.empty(); // 判断是否为空
s.size(); // 返回集合的大小
```

```
set<Type> s; // 创建一个Type类型的集合
s.insert(x); // 插入元素x
s.erase(x); // 删除值为x的元素
s.erase(it); // 删除it所指的元素
s.end(); // 返回末位哨兵的迭代器
s.find(x); // 查询x；不存在则返回s.end()
s.empty(); // 判断是否为空
s.size(); // 返回集合的大小
s.upper_bound(x); // 查询大于x的最小元素
s.lower_bound(y); // 查询不小于x的最小元素
// 使用方法与二分查找一章所介绍的一致
```

# STL map

映射在STL 中也有两种，分别是有序映射和无序映射

```
unordered_map <T1, T2> m; // 创建T1到T2的映射
// 其中T1称为键key，T2称为值value
m.insert({a,b});// 创建映射a->b
m.erase(a); // 删除key为a的映射
m.erase(it); // 删除it所指的映射
m.end(); // 返回末位哨兵的迭代器
m.find(x); // 寻找键x；若不存在则返回
m.end()
m.empty(); // 判断是否为空
m.size(); // 返回映射数目
m[a] = b; // 修改a映射为b；若不存在则创建
```

```
map<T1, T2> m; // 创建一个T1到T2的映射
// 其中T1称为键key，T2称为值value
m.insert({a,b});// 创建映射a->b
m.erase(a); // 删除key为a的映射
m.erase(it); // 删除it所指的映射
m.end(); // 返回末位哨兵的迭代器
m.find(x); // 寻找键x；若不存在则返回
m.end()
m.empty(); // 判断是否为空
m.size(); // 返回映射数目
m[a] = b; // 修改a映射为b；若不存在则创建
m.upper_bound(x); // 查询大于x的最小键
m.lower_bound(x); // 查询不小于x的最小键
// 使用方法与二分查找一章所介绍的一致
```

# 课后习题与实验

Talk is cheap, show me the code !

# 下节课见啦！