

词典

散列：循对象访问

We are shaped by our thoughts; we become what we think.

- Buddha

Man's thought is shaped by his tongue.

- Anonymous

于是在熟人中，我们话也少了，我们“眉目传情”，我们“指石相证”，
我们抛开了比较间接的象征原料，而求更直接的会意了。

E9 - A7

邓俊辉

deng@tsinghua.edu.cn

联合数组：更直接、更有效的访问

❖ 数组？再常见不过，比如：

```
fib[0] = 0  
fib[1] = 1  
fib[2] = 1  
fib[3] = 2  
fib[4] = 3  
fib[5] = 5  
fib[6] = 8  
  
...
```

❖ Associative Array
——与常规的数组有何区别？

❖ 根据数据元素的取值，直接访问！

```
style["关羽"] = "云长"  
style["张飞"] = "翼德"  
style["赵云"] = "子龙"  
style["马超"] = "孟起"
```

❖ 下标不再是整数，甚至没有大小次序
——更为直观、便捷

❖ 支持的语言：

Snobol4、MUMPS、SETL、Rexx、AWK、
Java、Python、Perl、Ruby、PHP、...

词条 ~ 映射/词典

❖ entry = (key, value)

❖ Map/Dictionary: 词条的集合

- 关键码禁止/允许雷同

- `get(key)`

- `put(key, value)`

- `remove(key)`

❖ 关键码未必可定义大小，元素类型较BST更多样

查找对象不限于最大/最小词条，接口功能较PQ更强大



get("翼德")

SanguoHero

云长

关羽

翼德

张飞

子龙

赵云

孟起

马超

汉升

黄忠

"张飞"

Dictionary

❖ template <typename K, typename V> //key、value

```
struct Dictionary {  
  
    virtual Rank size() = 0;  
  
    virtual bool put( K, V ) = 0;  
  
    virtual V* get( K ) = 0;  
  
    virtual bool remove( K ) = 0;  
};
```

❖ 词典中的词条只需支持**判等/比对操作**, 尽管

诸如Java:::TreeMap等实现仍支持**大小/比较器**

get("翼德")

SanguoHero

云长 关羽

翼德 张飞

子龙 赵云

孟起 马超

汉升 黄忠

"张飞"

Java: HashMap + Hashtable

```
import java.util.*;  
  
public class Hash {  
  
    public static void main(String[] args) {  
  
        HashMap HM = new HashMap(); //Map  
  
        HM.put("东岳", "泰山"); HM.put("西岳", "华山"); HM.put("南岳", "衡山");  
        HM.put("北岳", "恒山"); HM.put("中岳", "嵩山"); System.out.println(HM);  
  
        Hashtable HT = new Hashtable(); //Dictionary  
  
        HT.put("东岳", "泰山"); HT.put("西岳", "华山"); HT.put("南岳", "衡山");  
        HT.put("北岳", "恒山"); HT.put("中岳", "嵩山"); System.out.println(HT);  
    }  
}
```

Perl: %Hash Type

❖ 由字符串 (string) 标识的一组无序标量 (scalar) //亦即MAP

❖ `my %hero = ("云长"=>"关羽", "翼德"=>"张飞", "子龙"=>"赵云", "孟起"=>"马超") ;`

`foreach $style (keys %hero) # Hash类型的变量由%引导`

`{ print "$style => $hero{$style}\n"; }`

❖ `$hero{"汉升"} = "黄忠";`

`foreach $style (keys %hero)`

`{ print "$style => $hero{$style}\n"; }`

`foreach $style (reverse sort keys %hero)`

`{ print "$style => $hero{$style}\n"; }`

Python: Dictionary Class

```
❖ beauty = dict( { "沉鱼": "西施", "落雁": "昭君", "闭月": "貂蝉", "羞花": "玉环" } )  
print beauty  
  
❖ beauty["红颜"] = "圆圆"  
print beauty  
  
❖ for alias, name in beauty.items():  
    print alias, ":", name  
  
❖ for alias, name in sorted(beauty.items()):  
    print alias, ":", name  
  
❖ for alias in sorted(beauty.keys(), reverse = True):  
    print alias, ":", beauty[alias]
```

Ruby: Hash Table

```
scarborough = # declare and initialize a hashtable  
{ "P"=>"parsley", "S"=>"sage", "R"=>"rosemary", "T"=>"thyme" }  
  
puts scarborough # output the hash table  
  
for k in scarborough.keys # output hash table items  
  puts k + "=>" + scarborough[k] # 1-by-1  
  
end  
  
for k in scarborough.keys.sort # output hash table items  
  puts k + "=>" + scarborough[k] # 1-by-1 in order  
  
end
```

词典

散列：原理

09-A2

书者，散也。欲书先散怀抱，任情恣性，然后书之。

“如果是活着的人，想谁，找谁一趟不就完了？”

“找不得，找不得，当年就是因为个找，我差点丢了命。”

邓俊辉

deng@tsinghua.edu.cn

电话：号码 ~ 人

General inquiries

Tel: Toll Free: 1-800-IBM-4YOU
E-mail: askibm@vnet.ibm.com
www.ibm.com/us/en/

Shopping

Tel: Toll Free: 1-888-SHOP-IBM

Sales Center

1-855-2-LENOVO (1-855-253-6686)
Mon - Fri: 9am-9pm (EST)
Sat - Sun: 9am-6pm (EST)

Customer Service

1-855-2-LENOVO (1-855-253-6686)
Mon - Fri: 9am-9pm (EST)
Sat - Sun: 9am-6pm (EST)



电话簿

❖ 需求：为一所学校制作电话簿

号码 \rightarrow 教员、学生、员工、办公室

❖ 蛮力：用户记录 ~ 数组Rank ~ 电话号码

$O(1)$ 效率！

❖ 以清华为例（2003）

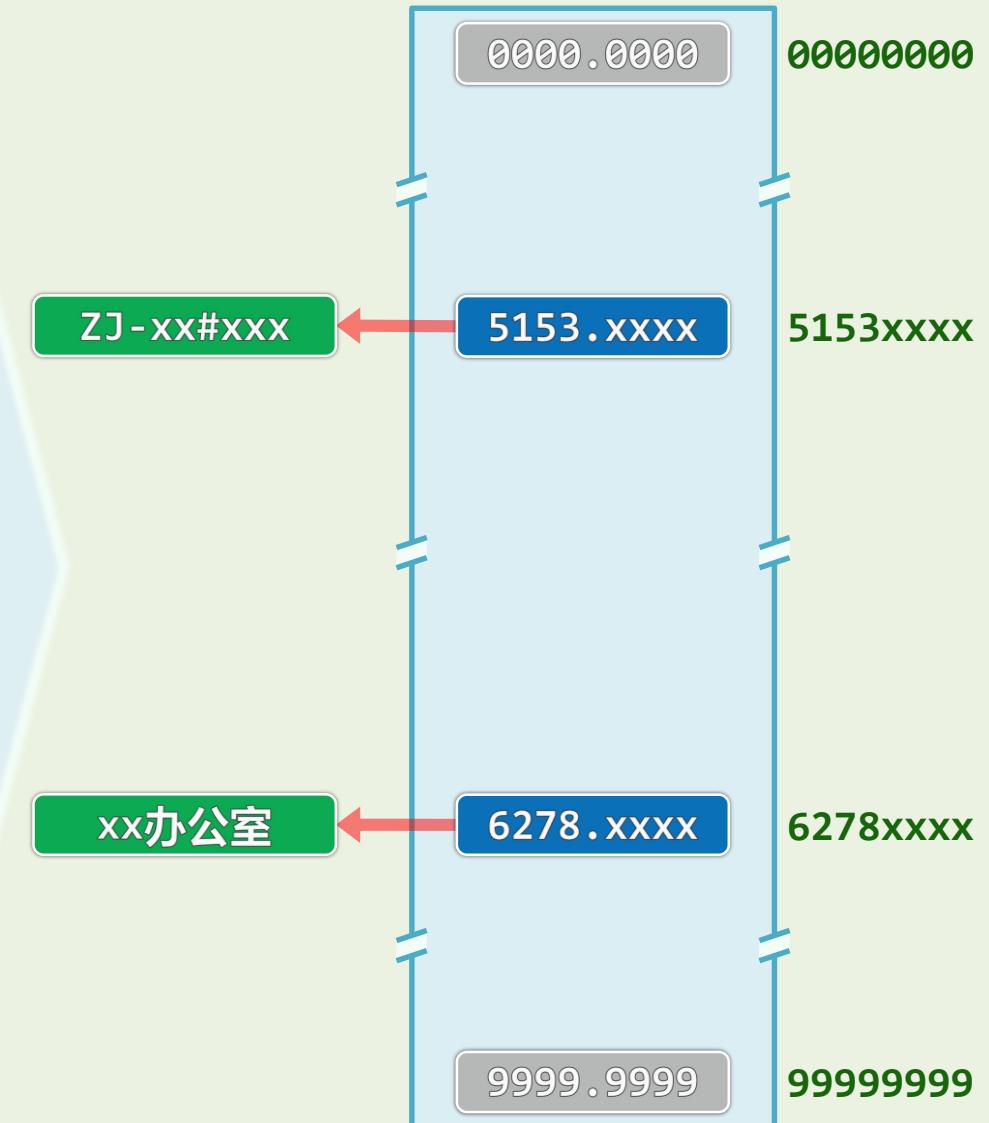
#可能的电话 = R = 10^8 = 100M

#实有的电话 = N = 25,000 = 25K

❖ 问题：空间 = $O(R + N) = O(100M + 25K)$

效率 = 25K / 100M = 0.025%

❖ 如何在保持查找速度的同时，降低存储消耗？



散列表 / 散列函数

❖ 桶 (bucket) : 直接存放或间接指向一个词条

❖ Bucket array ~ Hashtable

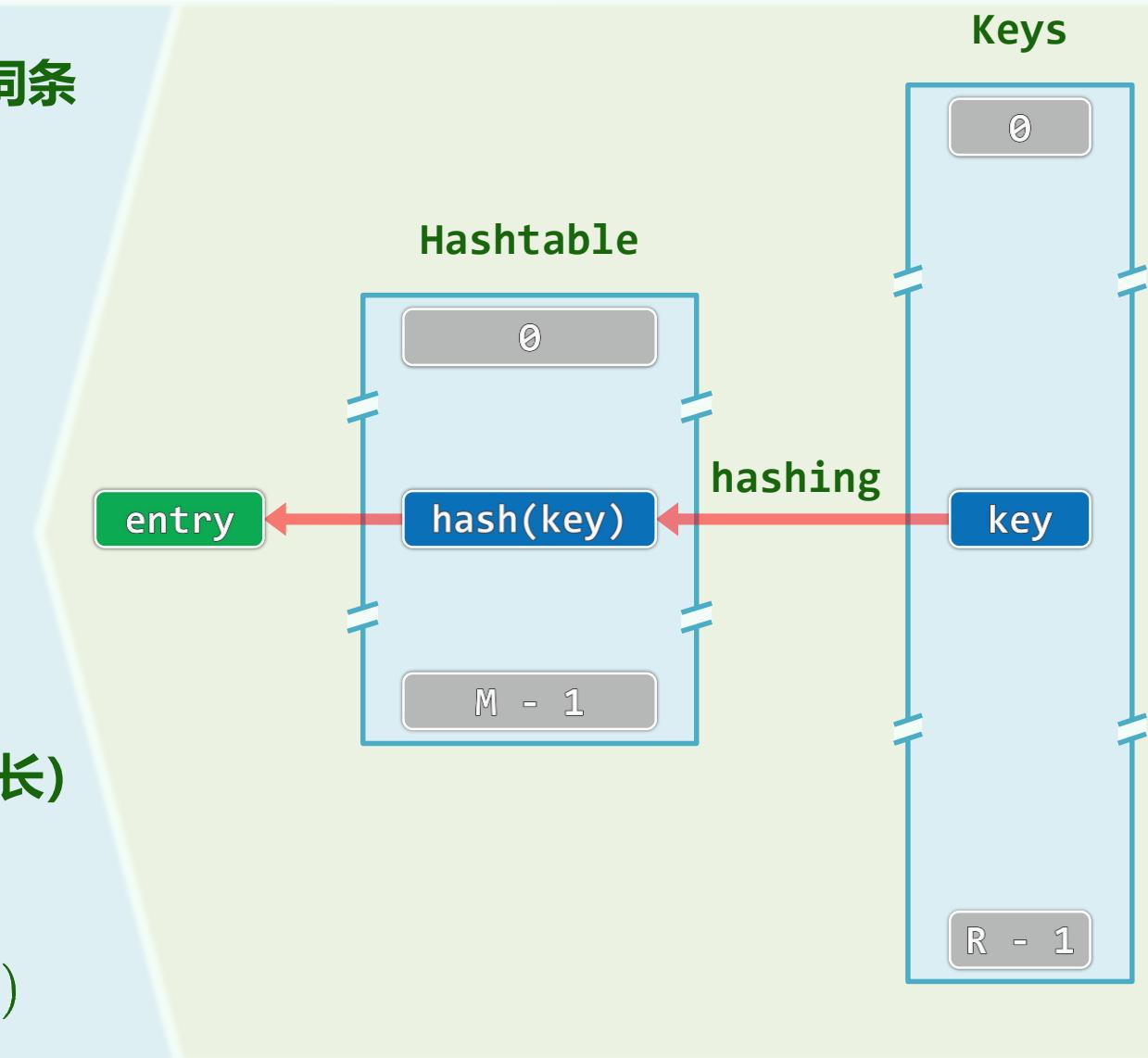
- 容量: M
- 满足: $N < M \ll R$
- 空间: $\mathcal{O}(N + M) = \mathcal{O}(N)$

❖ 定址/杂凑/散列

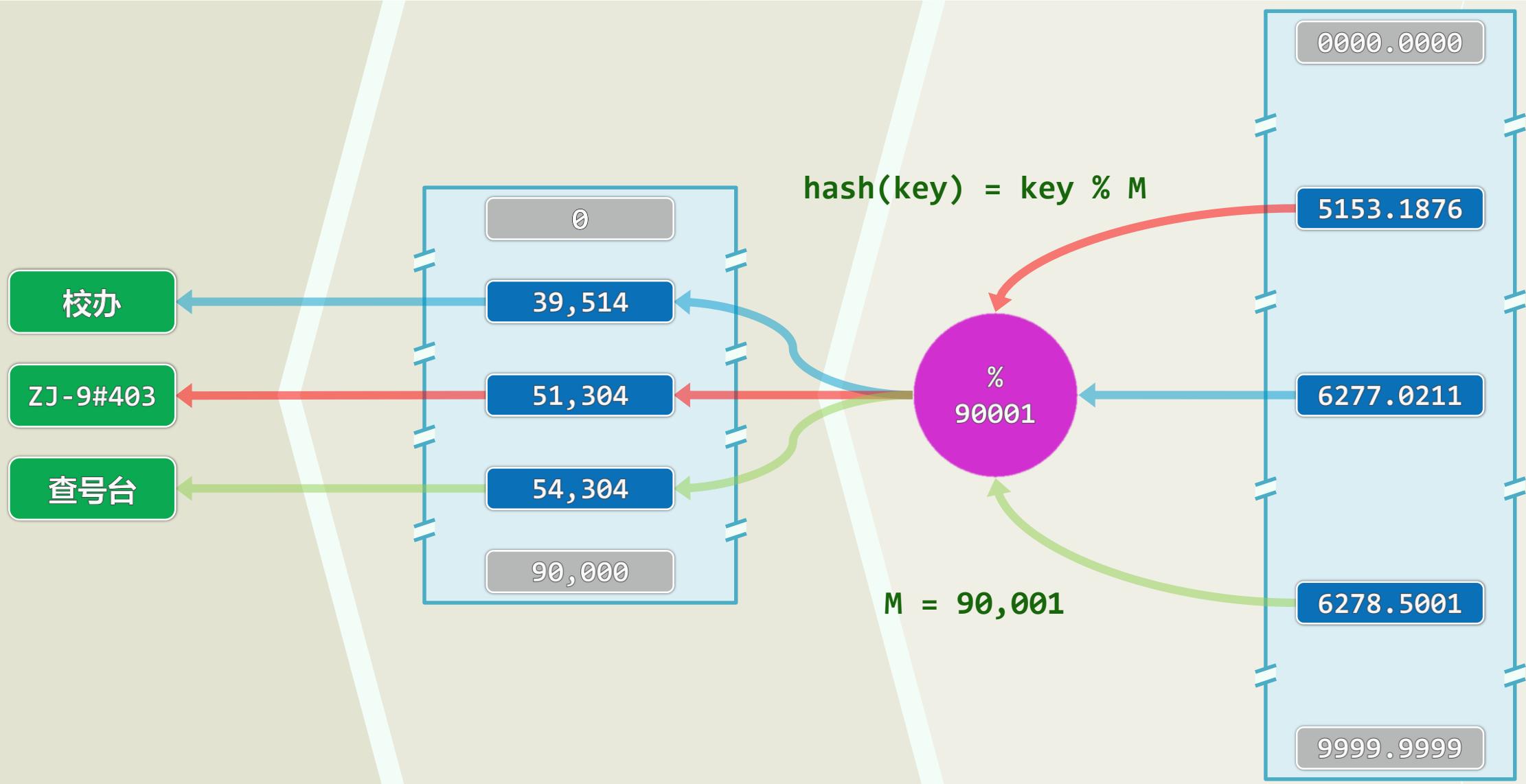
- 根据词条的key (未必可比较)
- “直接” 确定散列表入口 (无论表有多长)

❖ 散列函数: $hash() : key \mapsto \&entry$

❖ “直接” : $expected - \mathcal{O}(1) \neq \mathcal{O}(1)$



实例



词典

散列：冲突

e9-A3

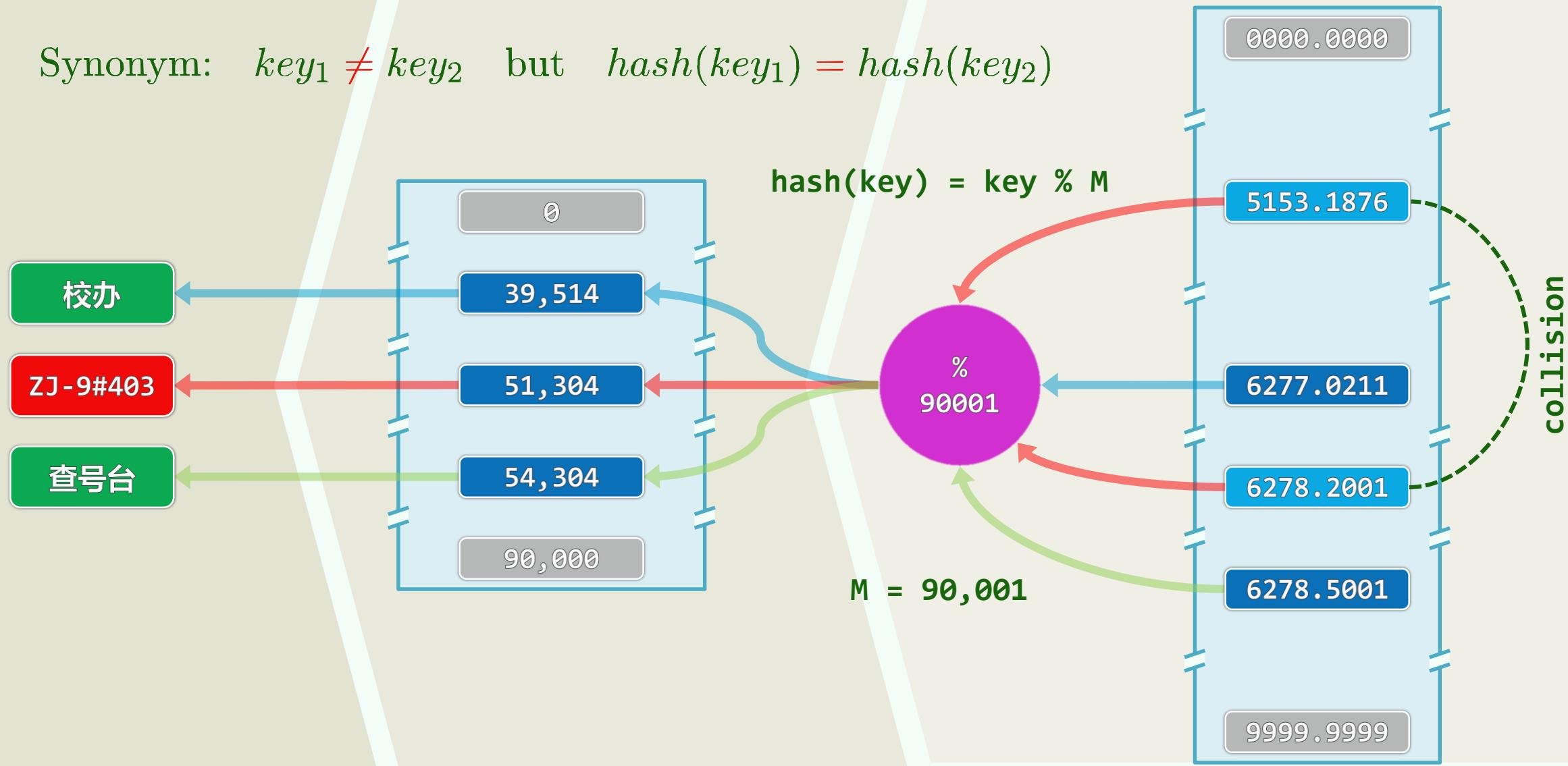
邓俊辉

deng@tsinghua.edu.cn

宝玉道：“已经完了，怎么又作揖？”袭人笑道：“这是他来
给你拜寿。今儿也是他的生日，你也该给他拜寿。”宝玉听了，
喜的忙作下揖去，说：“原来今儿也是姐姐的芳诞。”

同义词

Synonym: $key_1 \neq key_2$ but $hash(key_1) = hash(key_2)$



装填因子 vs. 冲突

❖ load factor: $\lambda = N/M$

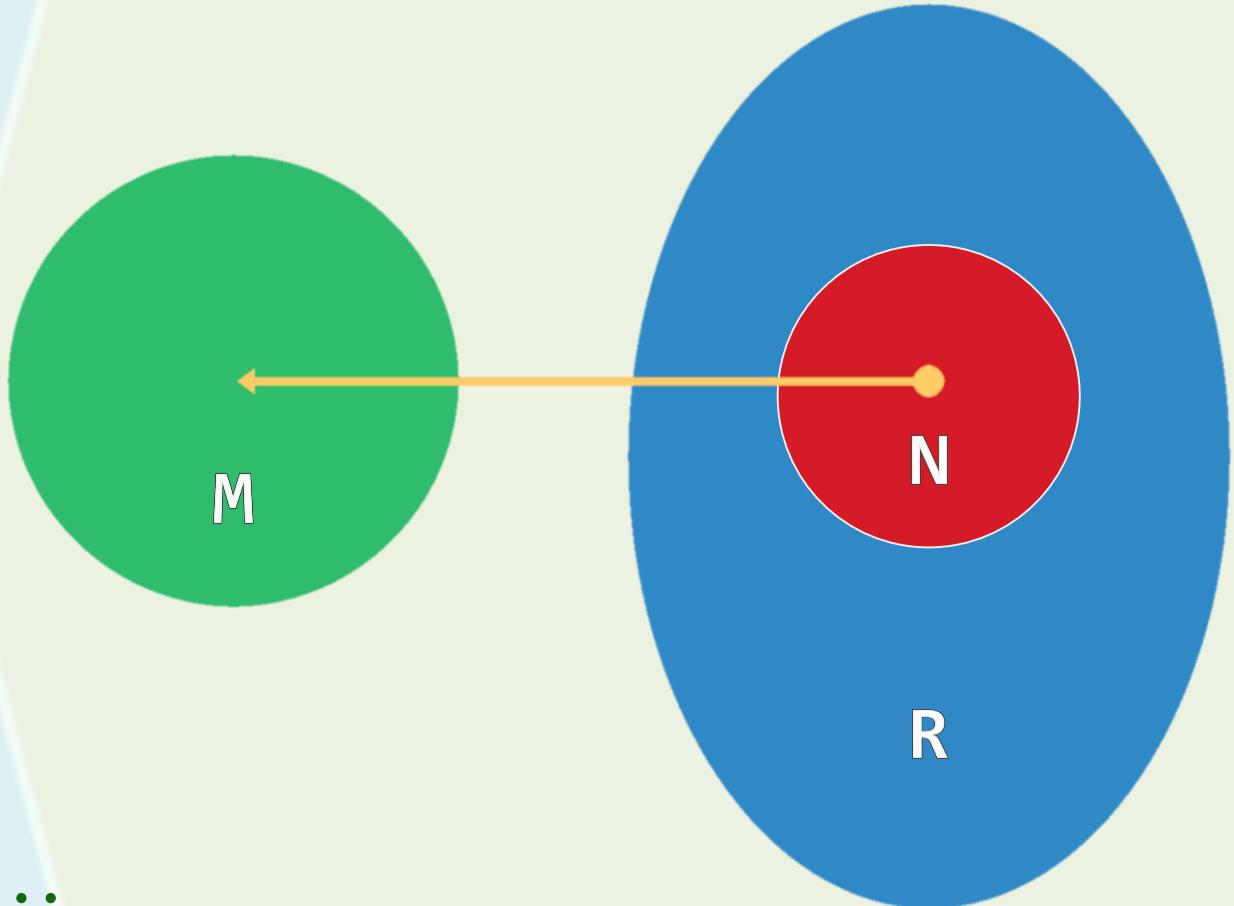
❖ λ 选多大才合适?

❖ λ 越大/小

- 空间利用率越高/低
- 冲突的情况越严重/轻微

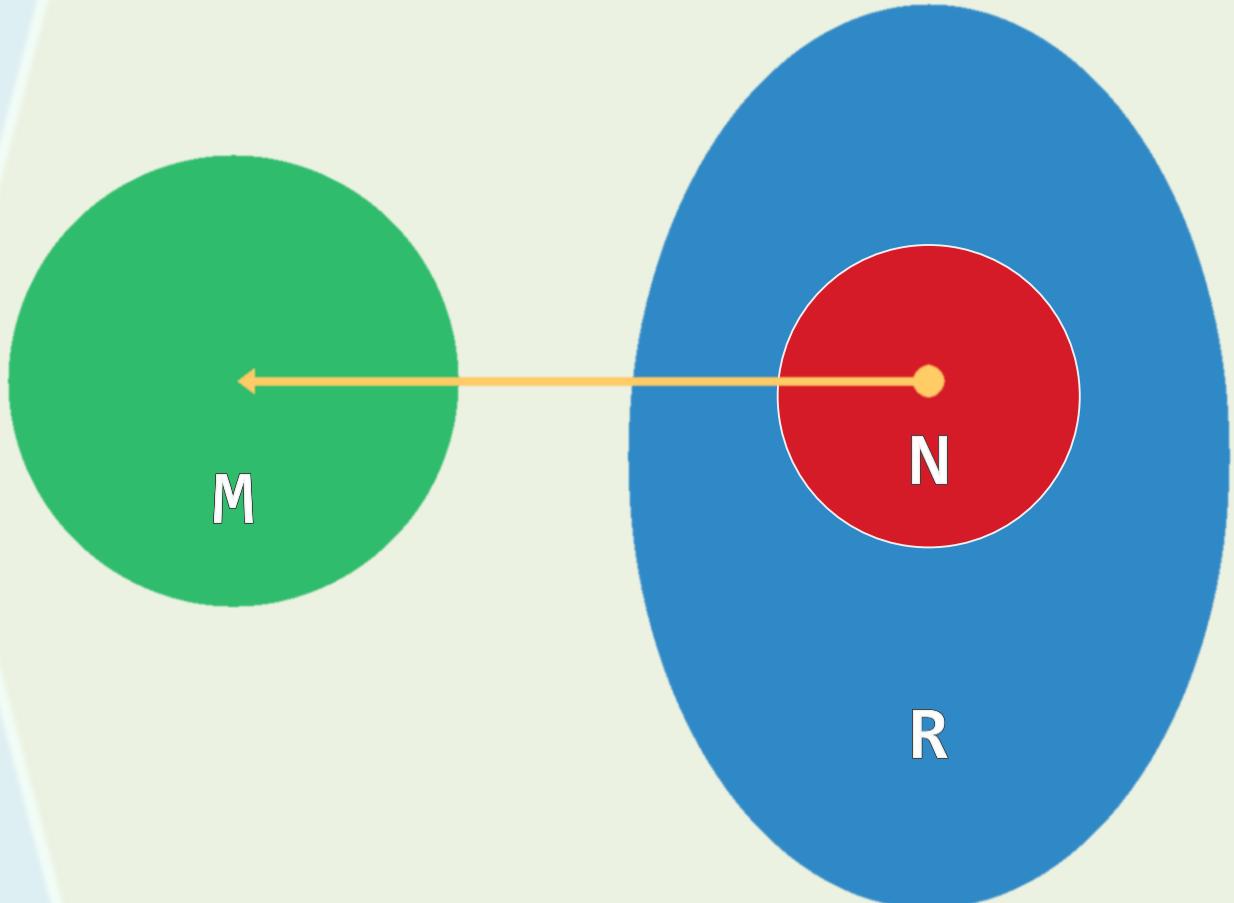
❖ 通过降低 λ ，冲突程度将会有所改善

但只要数据集在动态变化，就无法彻底杜绝...



完美散列

- ❖ 在某些条件下，的确可以实现
单射 (injection) 式散列，比如...
- ❖ 数据集已知且固定时，可实现
完美散列 (perfect hashing)
 - 采用两级散列模式
 - 仅需 $\mathcal{O}(n)$ 空间
 - 关键码之间互不冲突
 - 最坏情况下的查找时间也不过 $\mathcal{O}(1)$
- ❖ 不过在一般情况下，完美散列可期不可求...



生日悖论

❖ 将在座同学（对应的词条）按生日（月/日）做散列存储

散列表长 $M = 365$, 装填因子 = 在场人数 $N / 365$

❖ 冲突（至少有两位同学生日相同）的可能性 $P_{365}(n) = ?$

// [概率论与数理统计] 讲义第一章, 清华大学数学系王晓峰

$P_{365}(1) = 0, P_{365}(2) = 1/365, \dots, P_{365}(22) = 47.6\%, P_{365}(23) = 50.7\%, \dots$

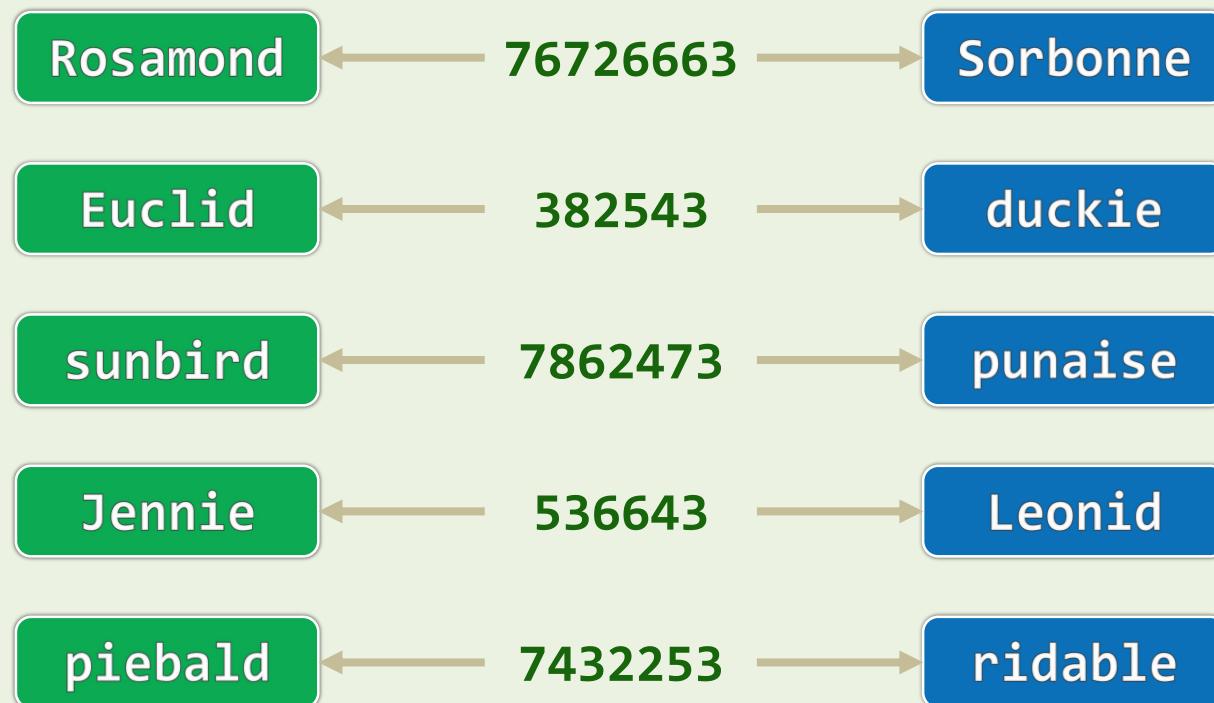
❖ 100人的集会: $1 - p_{365}(100) = 0.000,031\%$

- 自7岁起, 不吃不喝、无休无息, 每小时参加四次
- 到100岁, 才有可能期望遇到一次无冲突的集会

❖ 因此, 在装填因子确定之后, 散列策略的选取将至关重要, 散列函数的设计也很有讲究...

两项基本任务

- ❖ 首先（下一节）：精心设计散列表及散列函数，尽可能降低冲突的概率
- ❖ 同时（再下节）：制定可行的预案，以便在发生冲突时，能够尽快予以排解



词典

散列函数：基本

09-B1

此刻他就在占卜，方法是要从办公室到法庭扶手椅座位的步数可以被三除尽，那么新的疗法肯定能治好他的胃炎；要是除不尽，那就治不好。走下来是二十六步，但他把最后一步缩小，这样就正好走了二十七步。

邓俊辉

deng@tsinghua.edu.cn

评价标准 + 设计原则

❖ 确定 (determinism)

同一关键码总是被映射至同一地址

❖ 快速 (efficiency)

expected- $\mathcal{O}(1)$

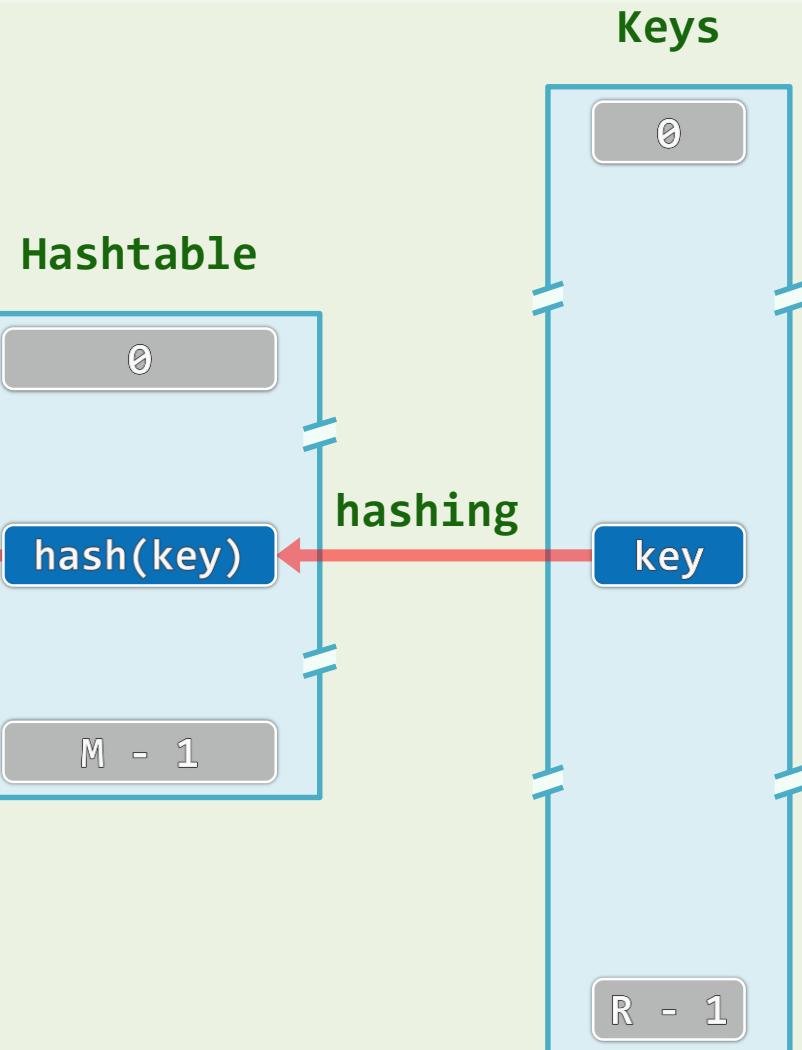
❖ 满射 (surjection)

尽可能充分地利用整个散列空间

❖ 均匀 (uniformity)

关键码映射到散列表各位置的概率尽量接近

有效避免聚集 (clustering) 现象



除余法

❖ $hash(key) = key \% M$ //前例中，为何选 $M = 90001$?

❖ 据说： M 为素数时，数据对散列表的覆盖最充分，分布最均匀

其实：对于理想随机的序列，表长是否素数，无关紧要！

❖ 序列的Kolmogorov复杂度：生成该序列的算法，最短可用多少行代码实现？

- 算术级数： 7 12 17 22 27 32 37 42 47 ... //单调性 / 线性：从7开始，步长为5

- 循环级数： 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 ... //周期性：12345不断循环

- 英文：data structures and algorithms ... //局部性：频率、关联、词根、...

❖ 实际应用中的数据序列远非理想随机，上述规律性普遍存在

❖ 蝉的哲学：经长期自然选择，生命周期“取”作素数

❖ 除余法的缺陷

- 不动点：无论表长 M 取值如何，总有： $\text{hash}(0) \equiv 0$
- 相关性： $[0, R)$ 的关键码尽管系平均分配至 M 个桶；但相邻关键码的散列地址也必相邻



❖ Multiply - Add - Divide

$$\text{hash}(\text{key}) = (a \times \text{key} + b) \% \mathcal{M}, \mathcal{M} \text{ prime, } a > 1, b > 0, \text{ and } \mathcal{M} \nmid a$$

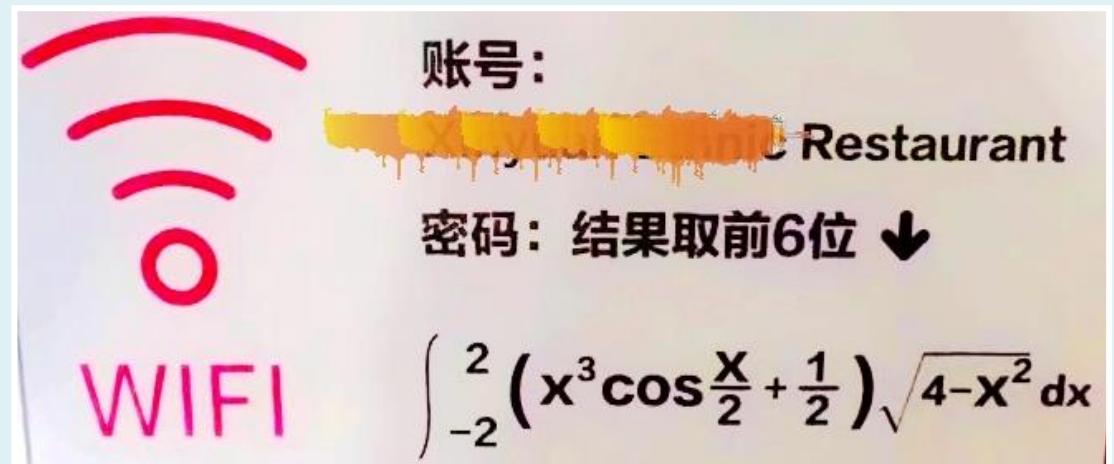
更多散列函数

❖ 数字分析/selecting digits

抽取key中的某几位，构成地址

- 比如，取十进制表示的奇数位

- $hash(3\ 1\ 4\ 1\ 5\ 9\ 2\ 6\ 5\ 4) = 34525$



❖ 平方取中/mid-square

取key²的中间若干位，构成地址

- $hash(123) = middle(123 \times 123) = 1\boxed{512}9 = 512$

- $hash(1234567) = 15241\boxed{556}77489 = 556$

更多散列函数

❖ 折叠法[folding]: 将key分割成等宽的若干段, 取其总和作为地址

- $\text{hash}(123456789) = 123 + 456 + 789 = 1368$ //自左向右
- $\text{hash}(123456789) = 123 + 654 + 789 = 1566$ //往复折返

❖ 位异或法[XOR]: 将key分割成等宽的二进制段, 经异或运算得到地址

- $\text{hash}(110011011_b) = 110 \wedge 011 \wedge 011 = 110_b$ //自左向右
- $\text{hash}(110011011_b) = 110 \wedge 110 \wedge 011 = 011_b$ //往复折返

❖

❖ 总之, 越是随机, 越是没有规律, 越好

散列

散列函数：随机数



As I have said so many times, God doesn't play dice with the world.
- A. Einstein

那妇人道：“不好，不好！我这里有一方手帕，你顶在头上，遮了脸，撞个天婚，教我女儿从你跟前走过，你伸开手扯倒那个就把那个配了你罢。”

邓俊辉
deng@tsinghua.edu.cn

(伪) 随机数法

❖ 循环: $\text{rand}(x + 1) = [\text{a} \times \text{rand}(x)] \% M$ //M素数, $a \% M \neq 0$

$$a = 7^5 = 16,807 = \boxed{100000110100111}_b$$

$$M = 2^{31} - 1 = 2,147,483,647 = 01111111 \boxed{11111111} 11111111 \boxed{11111111}_b$$

❖ 径取: $\text{hash(key)} = \text{rand(key)} = [\text{rand}(0) \times a^{\text{key}}] \% M$

种子: $\text{rand}(0) = ?$

❖ 把难题推给伪随机数发生器, 但是...

❖ (伪) 随机数发生器的实现, 因具体平台、不同历史版本而异

创建的散列表可移植性差——故需慎用此法!

(伪) 随机数法: The C Programming Language (2nd edn), p46

```
❖ unsigned long int next = 1; //sizeof(long int) = 8  
  
void srand(unsigned int seed) { next = seed; } //sizeof(int) = 4 or 8  
  
int rand(void) { //1103515245 = 3^5 * 5 * 7 * 129749  
  
    next = next * 1103515245 + 12345;  
  
    return (unsigned int)(next/65536) % 32768;  
  
}
```

rand  2^{15}

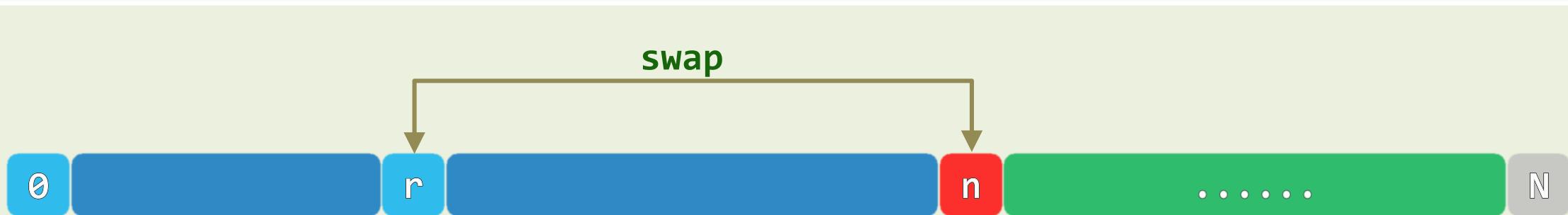
next  2^{15} 2^{32}

```
❖ int rand() { int uninitialized; return uninitialized; }
```

```
char* rand( t_size n ) { return ( char* ) malloc( n ); }
```

就地随机置乱：任给一个数组A[0, n)，理想地将其中元素的次序**随机打乱**

```
// [R. Fisher & F. Yates, 1938], [R. Durstenfeld, 1964], [D. E. Knuth, 1969]  
void shuffle( int A[], int n ) {  
    for ( ; 1 < n; --n ) //自后向前，依次将各元素  
        swap( A[ rand() % n ], A[ n - 1 ] ); //与随机选取的某一前驱（含自身）交换  
} //  $20! < 2^{64} < 21!$ 
```



❖ 的确可以**等概率地生成所有** $n!$ **种排列?** $20! < 2^{64} < 21!$

词典

散列函数：hashCode与多项式法

09-B3

有意整齐与有意变化，皆是一方死法。

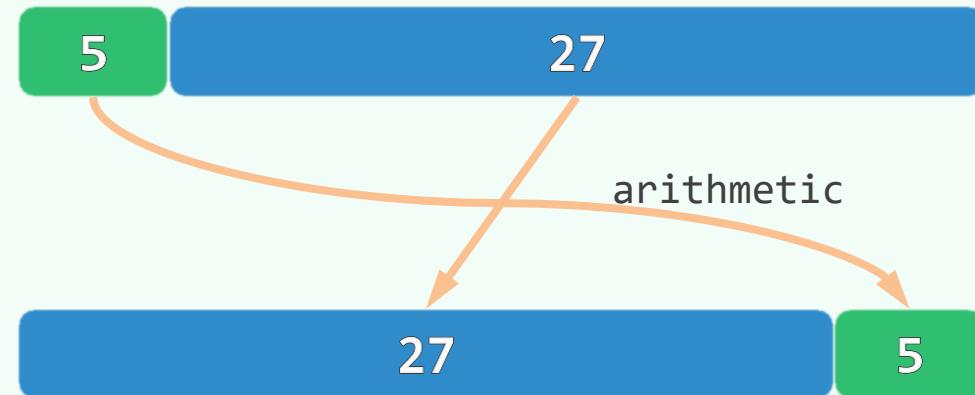
于是父亲只得求助于拈阄的办法，把两个姑娘的姓氏写在两方小红纸片上，把它们揉成两团，拿在手里，走到祖宗的神主面前诚心祷告了一番，然后随意拈起一个来。李家的亲事就这样地决定了。

邓俊辉

deng@tsinghua.edu.cn

String/Object To Integer

```
static Rank hashCode( char s[] ) {  
  
    Rank n = strlen(s); Rank h = 0;  
  
    for ( Rank i = 0; i < n; i++ ) {  
  
        h = (h << 5) | (h >> 27);  
  
        h += s[i];  
  
    } //乘以32，加上扰动，累计贡献  
  
    return h;  
  
} //有必要如此复杂吗？能否使用更简单的散列，比如...
```



$$\begin{aligned} \text{hashCode}(x_{n-1} \dots x_3 x_2 x_1 x_0) &= x_{n-1} \cdot a^{n-1} + \dots + x_2 \cdot a^2 + x_1 \cdot a^1 + x_0 \\ &= (\dots ((x_{n-1} \cdot a + x_{n-2}) \cdot a) + \dots + x_1) \cdot a + x_0 \end{aligned}$$

冲突 ~ 巧合

❖ 比如: $hashCode(S) = \sum_{c \in S} code(upper(c))$
 $hashCode("hash") = 8 + 1 + 19 + 8 = 36$

❖ 字符相对次序信息丢失, 将引发大量冲突

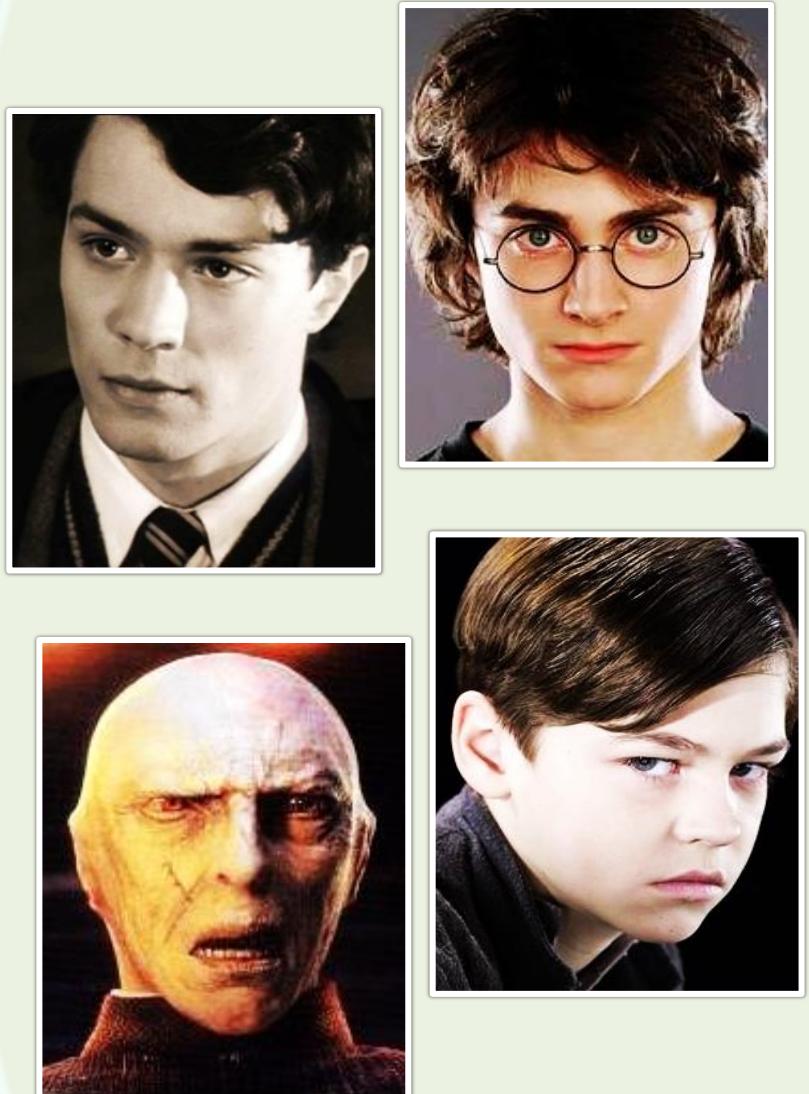
- I am Lord Voldemort
- Tom Marvolo Riddle

❖ 即便字符不同、数目不等...

- He's Harry Potter

❖ Key to improving your programming skills

Learning Tsinghua Data Structures & Algorithms



词典

排解冲突：开放散列

e9 - C1

尽人事，听天命

Every mistake I've ever made
Has been rehashed and then replayed
As I got lost along the way.

邓俊辉
deng@tsinghua.edu.cn

多槽位

❖ Multiple Slots

- 桶单元细分成若干槽位
- 存放（与同一单元）冲突的词条

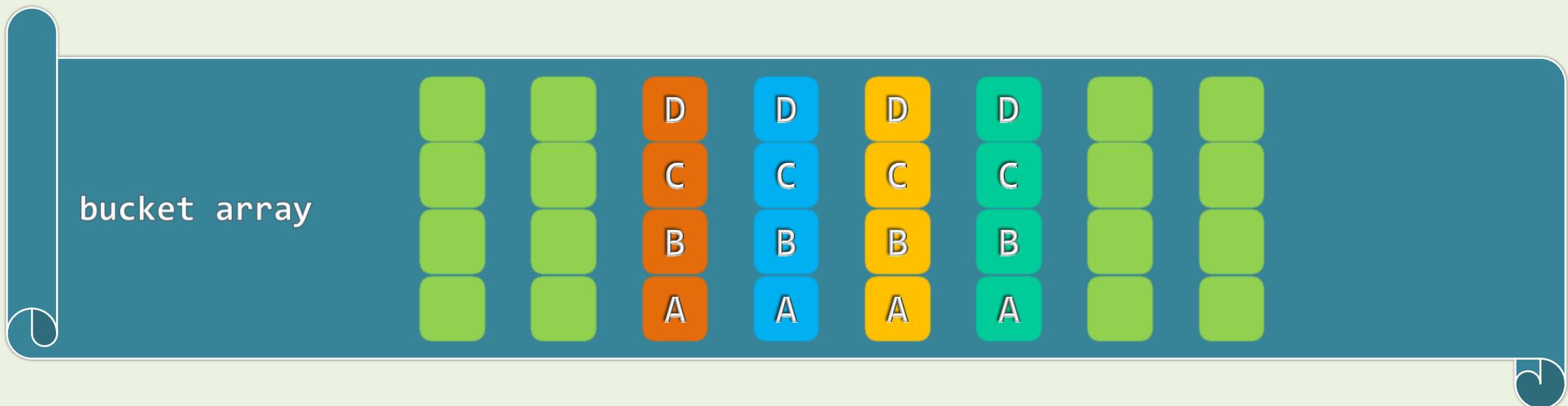
❖ 只要槽位数目不太多

依然可以保证 $\mathcal{O}(1)$ 的时间效率

❖ 但是，究竟需要细分到什么程度？

难以预测！

- 过细，空间浪费；反过来
- 无论多细，极端情况下仍可能不够



公共溢出区 / Overflow Area

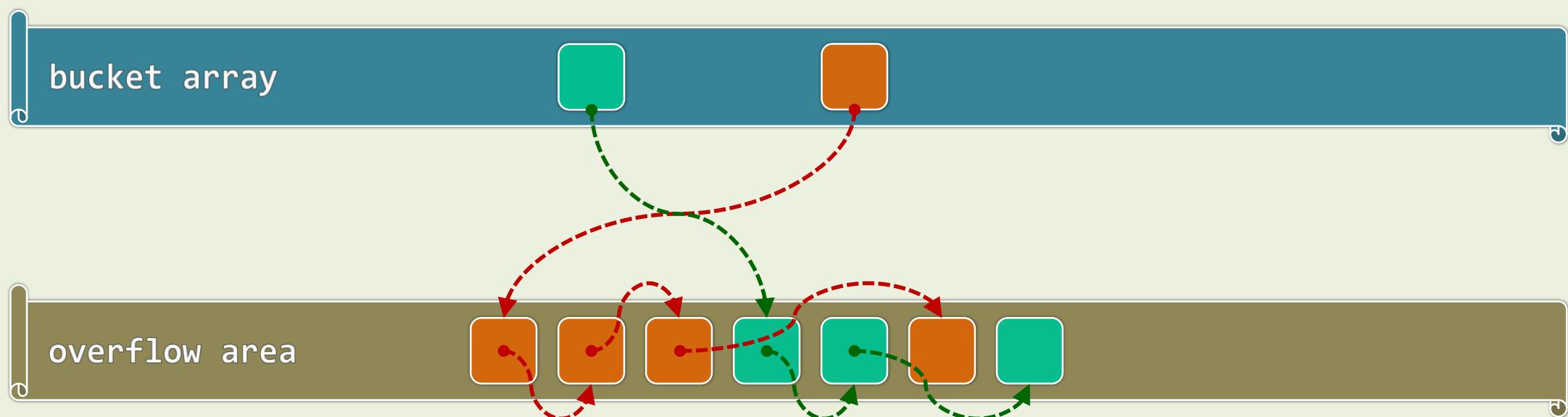
❖ 单独开辟一块连续空间

发生冲突的词条，顺序存入此区域

❖ 结构简单，算法易于实现

❖ 但是，不冲突则已，一旦发生冲突

最坏情况下，处理冲突词条所需的时间将
正比于溢出区的规模



独立链 / Linked-List Chaining / Separate Chaining

❖ 每个桶拥有一个**列表**, 存放对应的一组同义词

❖ 优点 无需为每个桶预备多个槽位

任意多次的冲突都可解决

删除操作实现简单、统一

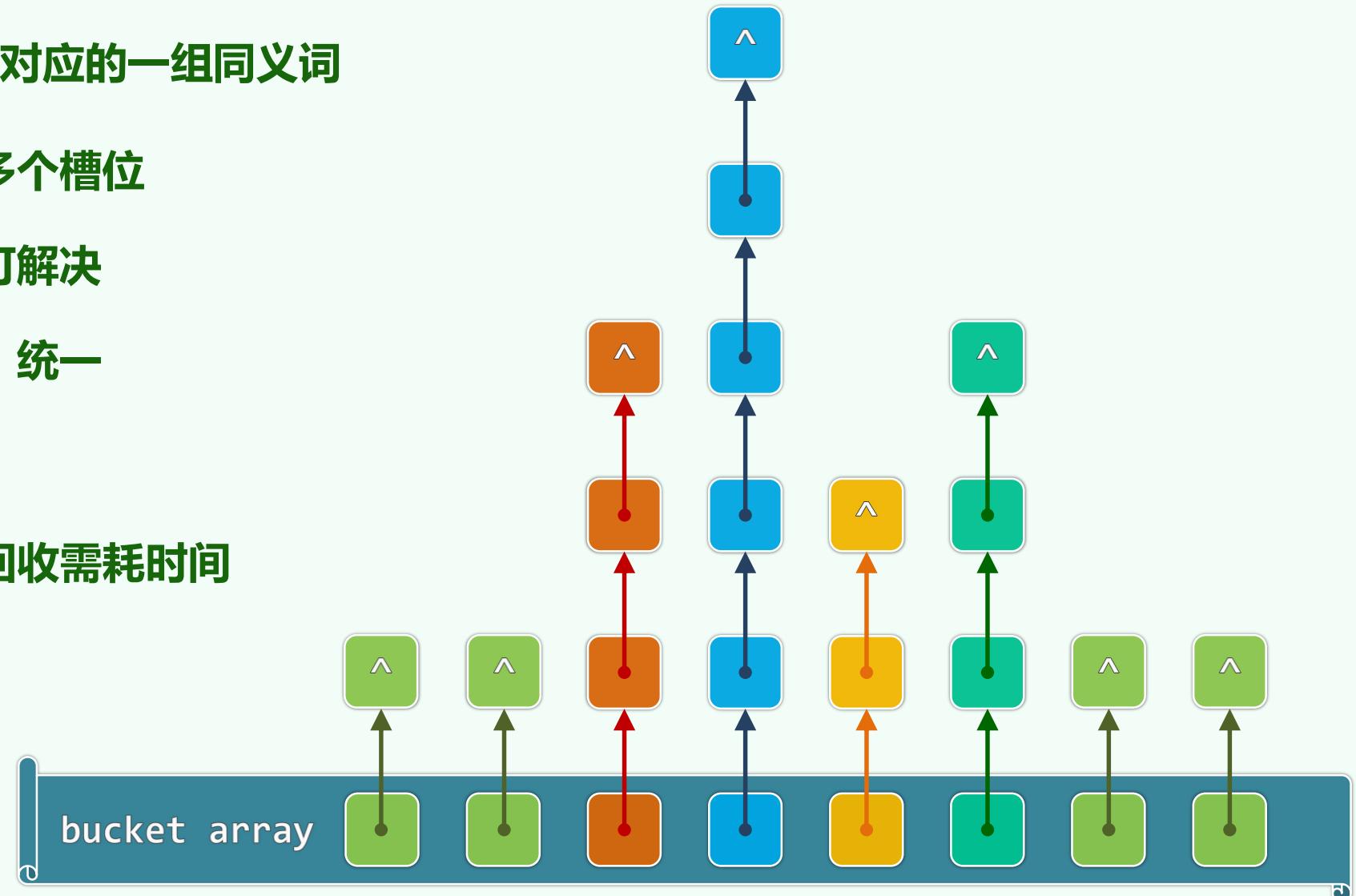
❖ 但是 指针本身占用空间

节点的动态分配和回收需耗时间

更重要的是...

❖ 空间未必**连续分布**

系统**缓存**很难生效



词典

排解冲突：封闭散列

邓俊辉

deng@tsinghua.edu.cn

旅客要在每个生人口敲叩
才能敲到自己的家门
人要在外面到处漂流
最后才能走到最深的内殿



在我们出生之前，一切都在没有我们的宇宙里开着
在我们活着的时候，一切都在我们身体里闭着
当我们死去，一切重又打开
打开、关闭、打开，我们就是这样

开放定址

❖ Closed Hashing, 必然对应于Open Addressing

- 只要有必要，任何散列桶都可以接纳任何词条

❖ Probe Sequence/Chain

- 为每个词条，都需事先约定若干备用桶，优先级逐次下降

❖ 查找算法：沿试探链，逐个转向下一桶单元，直到

- 命中成功，或者
- 抵达一个空桶（存在则必能找到？）而失败

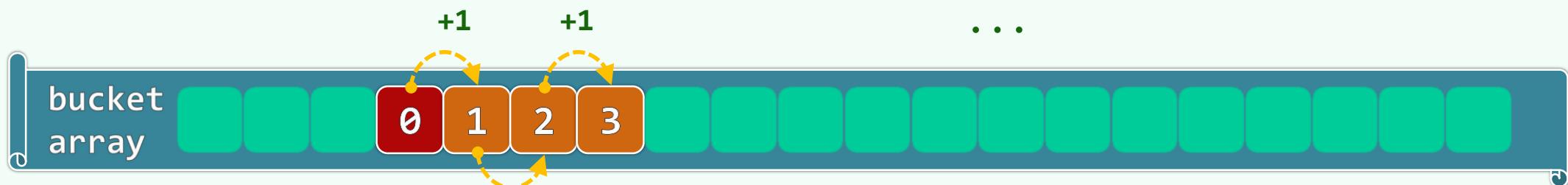
❖ 相应地，试探链又应如何约定？

线性试探

❖ Linear Probing

- 一旦冲突，则试探后一紧邻的桶
- 直到命中（成功），或抵达空桶（失败）

❖ $[\text{hash}(\text{key}) + 1] \% M$
 $[\text{hash}(\text{key}) + 2] \% M$
 $[\text{hash}(\text{key}) + 3] \% M$
 $[\text{hash}(\text{key}) + 4] \% M$
...



❖ 在散列表内部解决冲突

无需附加的指针、链表或溢出区等
整体结构保持简洁

❖ 只要还有空桶，迟早会找到

❖ 新增非同义词之间的冲突

❖ 数据堆积（clustering）现象严重

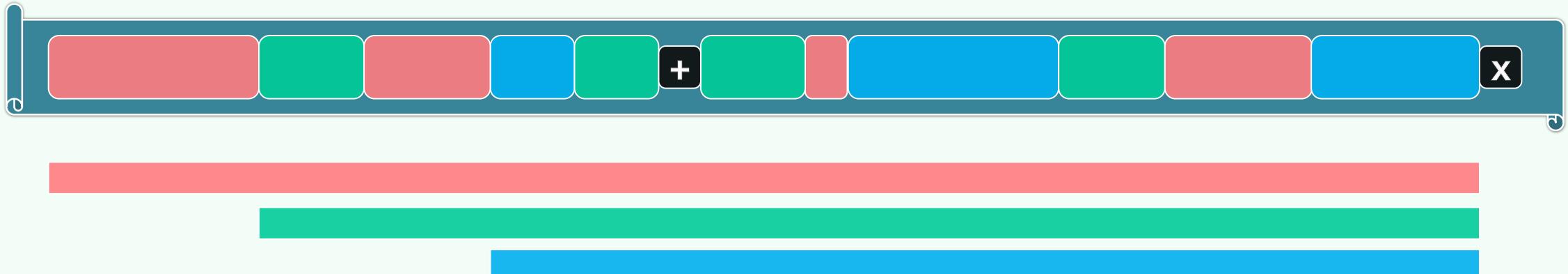
❖ 好在，试探链连续，数据局部性良好

❖ 通过装填因子，冲突与堆积都可有效控制

插入 + 删除

❖ 插入：新词条若尚不存在，则存入试探**终止处**的空桶

❖ 试探链：可能因而彼此**串接、重叠**！



❖ 删除：简单地清除命中的桶？

经过它的试探链都将因此**断裂**，导致后续词条**丢失**——明明存在，却访问不到

❖ 那么，如何才能**简明、高效**地完成删除呢？

词典

排解冲突：懒惰删除

e9-c3

邓俊辉

deng@tsinghua.edu.cn

江山故宅空文藻，云雨荒台岂梦思
最是楚宫俱泯灭，舟人指点到今疑

Lazy Removal: 故居 ~ 空宅

❖ `Bitmap* removed; //用Bitmap懒惰地标记被删除的桶`

`int L; //被标记桶的数目`



❖ 仅做标记，不对试探链做更多调整——此后，带标记的桶，**角色因具体的操作而异**

- **查找词条时，被视作“必不匹配的非空桶”，试探链在此得以延续**
- **插入词条时，被视作“必然匹配的空闲桶”，可以用来存放新词条**

两种试探算法

```
template <typename K, typename V> int Hashtable<K, V>::probe4Hit(const K& k) {  
    int r = hashCode(k) % M; //按除余法确定试探链起点  
  
    while ( ( ht[r] && (k != ht[r]->key) ) || removed->test(r) )  
        r = ( r + 1 ) % M; //线性试探（跳过带懒惰删除标记的桶）  
  
    return r; //调用者根据ht[r]是否为空及其内容，即可判断查找是否成功  
}
```

```
template <typename K, typename V> int Hashtable<K, V>::probe4Free(const K& k) {  
    int r = hashCode(k) % M; //按除余法确定试探链起点  
  
    while ( ht[r] ) r = (r + 1) % M; //线性试探，直到空桶（无论是否带有懒惰删除标记）  
  
    return r; //只要有空桶，线性试探迟早能找到  
}
```

词典

排解冲突：重散列

e9-c4

“天地會壞否？”

“不會壞。只是相將人無道極了，便一齊打合，混沌一番，人物都盡，又重新起。”

邓俊辉

deng@tsinghua.edu.cn

Rehashing

```
template <typename K, typename V> //随着装填因子增大，冲突概率、排解难度都将激增
void Hashtable<K, V>::rehash() { //此时，不如“集体搬迁”至一个更大的散列表
    int oldM = M; Entry<K, V>** oldHt = ht;
    ht = new Entry<K, V>*[ M = primeNLT( 4 * N ) ]; N = 0; //新表“扩”容
    memset( ht, 0, sizeof( Entry<K, V>* ) * M ); //初始化各桶
    release( removed ); removed = new Bitmap(M); L = 0; //懒惰删除标记
    for ( int i = 0; i < oldM; i++ ) //扫描原表
        if ( oldHt[i] ) //将每个非空桶中的词条
            put( oldHt[i]->key, oldHt[i]->value ); //转入新表
    release( oldHt ); //释放——因所有词条均已转移，故只需释放桶数组本身
}
```

插入

```
template <typename K, typename V> bool Hashtable<K, V>::put( K k, V v ) {  
    if ( ht[ probe4Hit( k ) ] ) return false; //雷同元素不必重复插入  
    int r = probe4Free( k ); //为新词条找个空桶 (只要装填因子控制得当, 必然成功)  
    ht[ r ] = new Entry<K, V>( k, v ); ++N; //插入  
    if ( removed->test( r ) ) { removed->clear( r ); --L; } //懒惰删除标记  
    if ( (N + L)*2 > M ) rehash(); //若装填因子高于50%, 重散列  
    return true;  
}
```

删除

```
template <typename K, typename V> bool Hashtable<K, V>::remove( K k ) {  
  
    int r = probe4Hit( k ); if ( !ht[r] ) return false; //确认目标词条确实存在  
  
    release( ht[r] ); ht[r] = NULL; --N; //清除目标词条  
  
    removed->set(r); ++L; //更新标记、计数器  
  
    if ( 3*N < L ) rehash(); //若懒惰删除标记过多，重散列  
  
    return true;  
}
```

词典

排解冲突：平方试探

09 - C5

邓俊辉

deng@tsinghua.edu.cn

三十六计，走为上计

我真的以为，这样何尝不是一种所谓的解脱
要背负的辛苦又有谁能够清楚，那内心的冲突

平方试探

❖ Quadratic Probing

以平方数为距离，确定下一试探桶单元

$[\text{hash(key)} + 1^2] \% M$

$[\text{hash(key)} + 2^2] \% M$

$[\text{hash(key)} + 3^2] \% M$

$[\text{hash(key)} + 4^2] \% M$

...

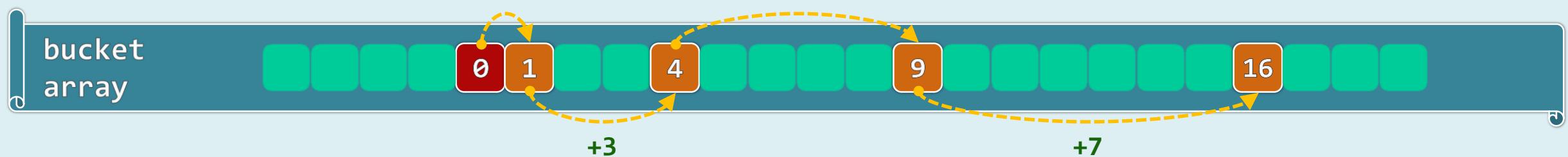
❖ 数据聚集现象有所缓解

- 试探链上，各桶间距**线性递增**
- 一旦冲突，可“**聪明**”地跳离**是非之地**

❖ 对于大散列表，**I/O操作有所增加**

❖ 只要有空桶，就...一定能...找出来吗？

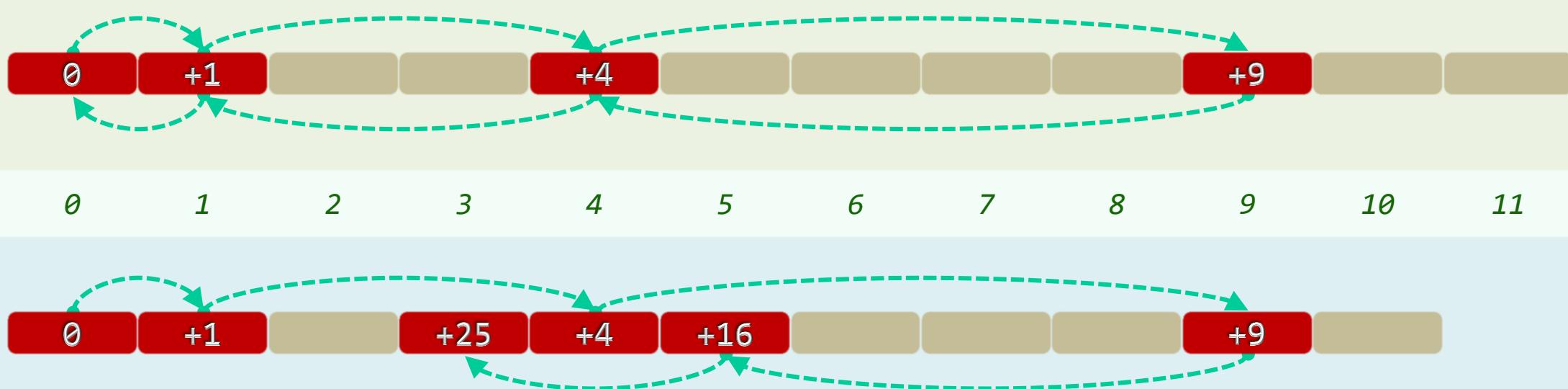
//毕竟不是逐个试探



素数表长时，只要 $\lambda < 0.5$ 就一定能够找出；否则，不见得

❖ $\{ 0, 1, 2, 3, 4, 5, \dots \}^2 \% 12 = \{ 0, 1, 4, 9 \}$

若为合数： $n^2 \% M$ 可能的取值可能少于 $\lceil M/2 \rceil$ 种——此时，只要对应的桶均非空...



❖ $\{ 0, 1, 2, 3, 4, 5, \dots \}^2 \% 11 = \{ 0, 1, 4, 9, 5, 3 \}$

❖ 若 M 为素数，则 $n^2 \% M$ 恰有 $\lceil M/2 \rceil$ 种取值，且由试探链的前 $\lceil M/2 \rceil$ 项取遍 //Quadratic Residue

每一条试探链，都有足够长的无重前缀

❖ 反证：假设存在 $0 \leq a < b < \lceil \mathcal{M}/2 \rceil$ ，使得

沿着试探链，第 a 项和第 b 项彼此冲突

❖ 于是： a^2 和 b^2 自然关于 \mathcal{M} 同余，亦即

$$a^2 \equiv b^2 \pmod{\mathcal{M}}$$

$$b^2 - a^2 = (b + a) \cdot (b - a) \equiv 0 \pmod{\mathcal{M}}$$

❖ 然而， $0 < b - a \leq b + a < \lceil \mathcal{M}/2 \rceil + (\lceil \mathcal{M}/2 \rceil - 1) \leq \lceil \mathcal{M}/2 \rceil + \lfloor \mathcal{M}/2 \rfloor = \mathcal{M}$

无论 $b - a$ 还是 $b + a$ 都不可能整除 \mathcal{M}

❖ 那么，另一半的桶，可否也利用起来呢...

词典

排解冲突：双向平方试探

e9 - C6

邓俊辉

deng@tsinghua.edu.cn

绕树三匝，何枝可依

伺候父亲躺下，我正准备离去，父亲拉住了我的手，轻轻地问我，女儿，你知道什么是无枝可栖吗？

策略：交替地沿两个方向试探，均按平方确定距离

[$\text{hash(key)} + 1^2$] % M

[$\text{hash(key)} + 2^2$] % M

[$\text{hash(key)} + 3^2$] % M

[$\text{hash(key)} + 4^2$] % M

[$\text{hash(key)} - 1^2$] % M

[$\text{hash(key)} - 2^2$] % M

[$\text{hash(key)} - 3^2$] % M

[$\text{hash(key)} - 4^2$] % M

...

...



子试探链，彼此独立？

❖ 正向和反向的子试探链，各自包含 $\lceil \mathcal{M}/2 \rceil$ 个互异的桶

$$\underline{-\lfloor \mathcal{M}/2 \rfloor, \dots, -3, -2, -1, 0, 1, 2, 3, \dots, \lfloor \mathcal{M}/2 \rfloor}$$

$\pm i^2$		-36	-25	-16	-9	-4	-1	0	1	4	9	16	25	36
M	5					1	4	0	1	4				
	7				5	3	6	0	1	4	2			
	11		8	6	2	7	10	0	1	4	9	5	3	
	13	3	1	10	4	9	12	0	1	4	9	3	12	10

❖ 除了起点0，这两个序列是否还有...其它公共的桶？

$4k + 3$

❖ 两类素数: 3 5 7 11 13 17 19 23 29 31

❖ 诀窍: 表长取作素数 $\mathcal{M} = 4 \cdot k + 3$, 即必然可以保证试探链的前 \mathcal{M} 项均互异

$\pm i^2$		-36	-25	-16	-9	-4	-1	0	1	4	9	16	25	36
M	5					1	4	0	1	4				
	7					5	3	6	0	1	4	2		
	11					8	6	2	7	10	0	1	4	9
	13					3	1	10	4	9	12	0	1	4

❖ 反之, $\mathcal{M} = 4 \cdot k + 1$ 就...必然不能使用?

Two-Square Theorem of Fermat

❖ 素数 p 不能表示为一对整数的平方和, 当且仅当

$$p \equiv 3 \pmod{4}$$

❖ 只要注意到:

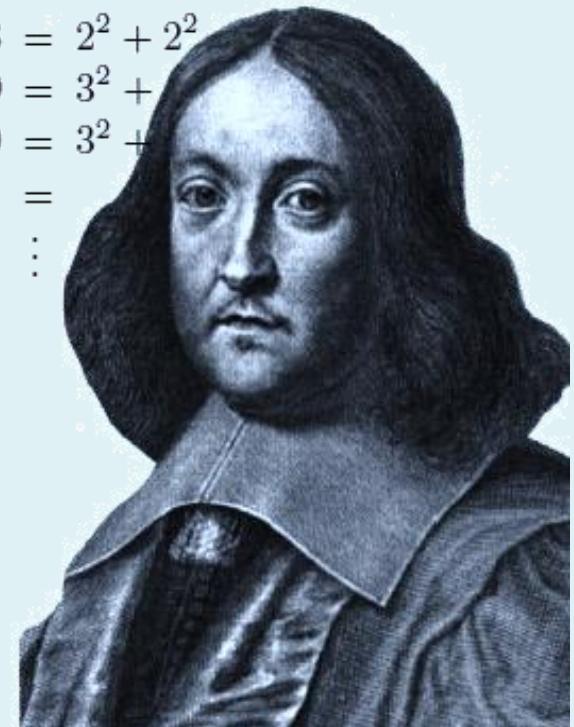
$$\begin{aligned}(u^2 + v^2) \cdot (s^2 + t^2) &= (us + vt)^2 + (ut - vs)^2 \\ &= (us - vt)^2 + (ut + vs)^2\end{aligned}$$

$$\begin{aligned}(2^2 + 3^2) \cdot (5^2 + 8^2) &= (10 + 24)^2 + (16 - 15)^2 \\ &= (10 - 24)^2 + (16 + 15)^2\end{aligned}$$

❖ 就可以推知:

- 自然数 n 可表示为一对整数的平方和, 当 (且仅当)
- 它的每一 $M = 4 \cdot k + 3$ 类的素因子均为偶数次方

$$\begin{aligned}1 &= 1^2 + 0^2 \\ 2 &= 1^2 + 1^2 \\ 3 &= \\ 4 &= 2^2 + 0^2 \\ 5 &= 2^2 + 1^2 \\ 6 &= \\ 7 &= \\ 8 &= 2^2 + 2^2 \\ 9 &= 3^2 + 0^2 \\ 10 &= 3^2 + 1^2 \\ 11 &= \\ \vdots &\end{aligned}$$



e9-c7

词典

排解冲突：双散列

邓俊辉

deng@tsinghua.edu.cn

Double Hashing

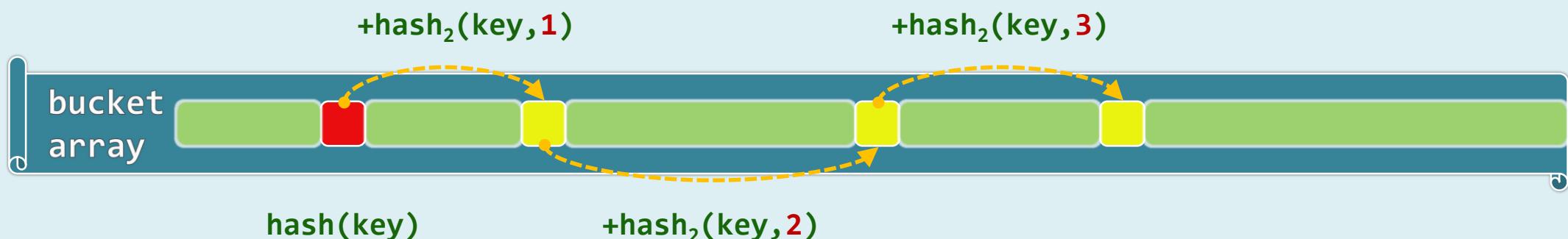
❖ 预先约定第二散列函数: $hash_2(key, i)$

冲突时, 由其确定偏移增量, 确定下一试探位置: $[hash(key) + \sum_{i=1}^k hash_2(key, i)] \% \mathcal{M}$

❖ 线性试探: $hash_2(key, i) \equiv 1$

平方试探: $hash_2(key, i) = 2i - 1$

更一般地, 偏移增量同时还与key相关



词典

桶排序：算法

e9-D1

邓俊辉

deng@tsinghua.edu.cn

Don't put all your eggs in one basket.

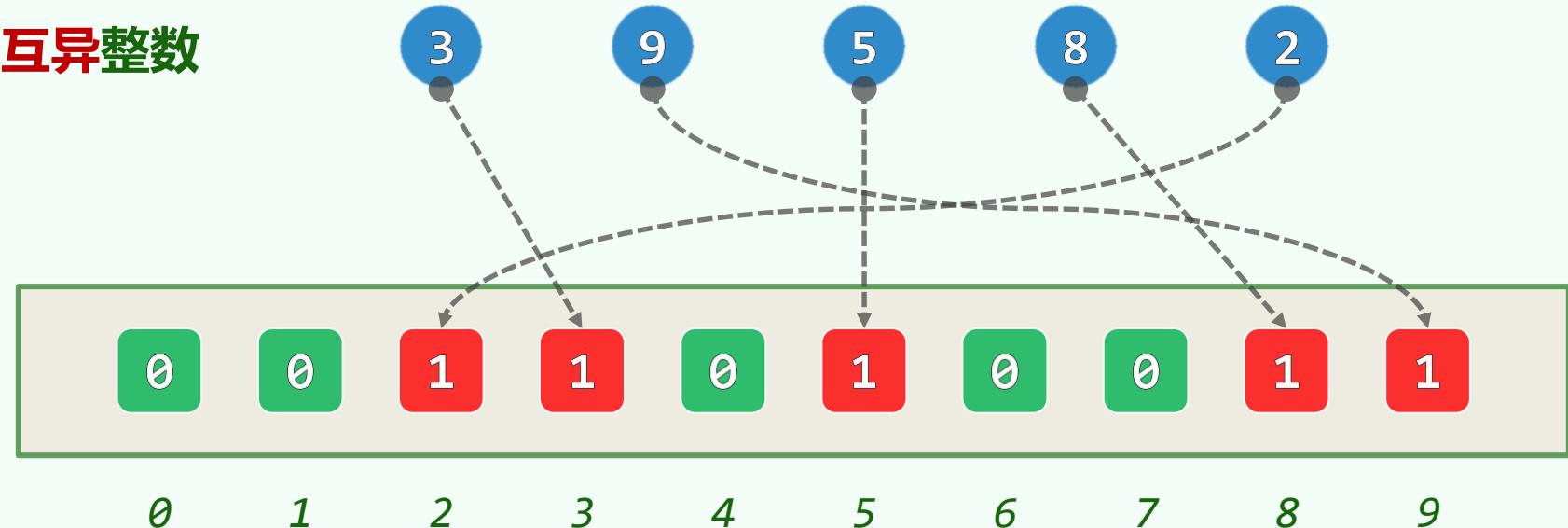
简单情况

❖ 对 $[0, m)$ 内的 n ($< m$) 个互异整数

借助散列表 $\pi[]$ 做排序

❖ 空间 = $\Theta(m)$

时间 = $\Theta(n)$!



❖ initialization: `for i = 0 to m - 1, let $\pi[i] = 0$ // $\Theta(m)$ --> $\Theta(1)$`

distribution: `for each key in the input, let $\pi[key] = 1$ // $\Theta(n)$`

enumeration: `for i = 0 to m - 1, output i if $\pi[i] = 1$ // $\Theta(m)$ --> $\Theta(n)$`

一般情况

❖ 若允许**重复**

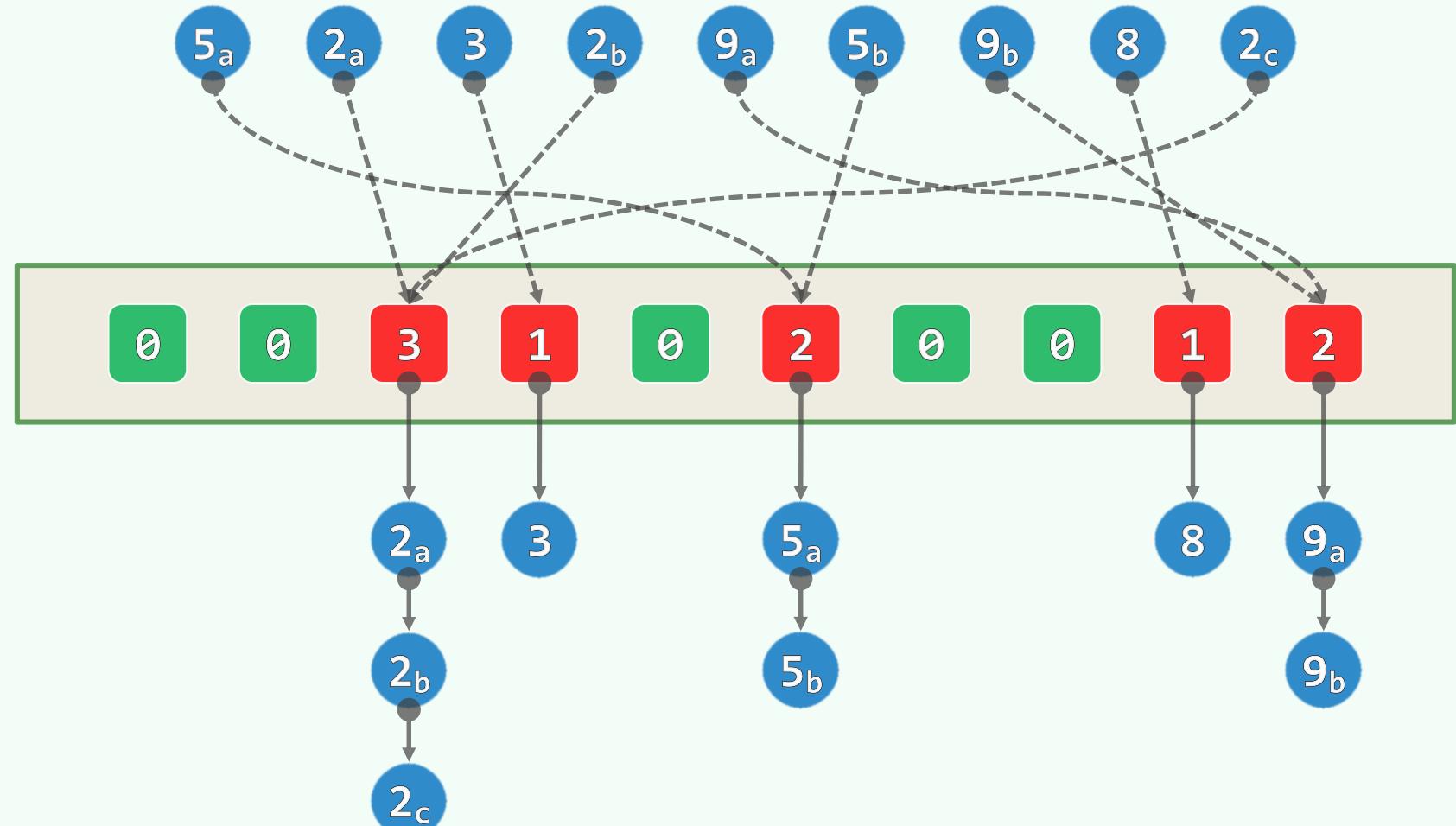
(可能 $m \ll n$)

❖ 依然使用散列表

- 每一组同义词
- 各成一个链表

❖ 空间 = $\theta(m + n)$

时间 = $\theta(n)$!



词典

桶排序：最大缝隙

e9 - D2

“恒纪元能持续多长时间？”

“一天或一个世纪，每次多长谁都说不准。”

“那乱纪元会持续多长时间呢？”

“不是说过嘛，除了恒纪元都是乱纪元，两者互为对方的间隙。”

邓俊辉

deng@tsinghua.edu.cn

MaxGap

❖ 任意n个互异点均将实轴分为n-1段有界区间，其中的哪一段最长？

❖ 如果不追求效率，显而易见的方法莫过于…

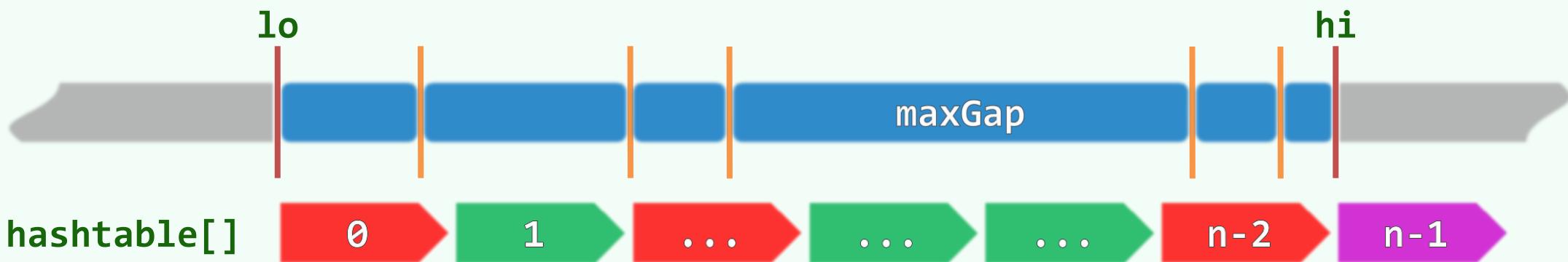


- 对所有点排序 //可惜，这需要 $\Omega(n \log n)$ 时间
- 依次计算各相邻点对的间距，保留最大者 // $\Theta(n)$

❖ 可否更快？

❖ 采用分桶策略，可改进至 $\mathcal{O}(n)$ 时间…

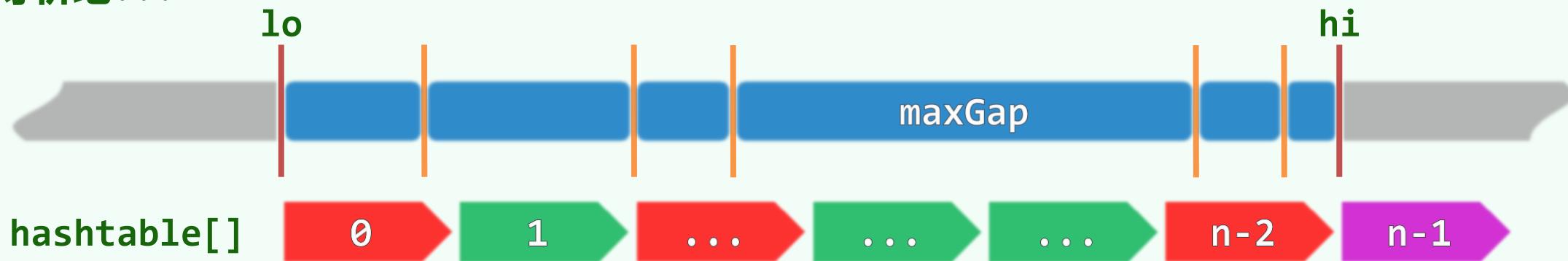
线性算法

找到最左点、最右点	$\theta(n)$	//一趟线性扫描
将有效范围均匀地划分为n-1段 (n个桶)	$\theta(n)$	//相当于散列表
		
hashtable[]	0 → 1 → ... → ... → n-2 → n-1	
通过散列，将各点归入对应的桶	$\theta(n)$	//模余法
在各桶中，动态记录最左点、最右点	$\theta(n)$	//可能相同甚至没有
算出相邻 (非空) 桶之间的“距离”	$\theta(n)$	//一趟遍历足矣
最大的距离即MaxGap	$\theta(n)$	//画家算法

正确性

- ❖ 正确性：MaxGap至少跨越两个桶

等价地...



MaxGap不可能局限于某一个桶内

- ❖ 对称的MinGap问题：n-1段有界区间中，何者最短？

可否沿用上法，以突破 $\Omega(n \log n)$ 下界？

词典

基数排序：算法与实现

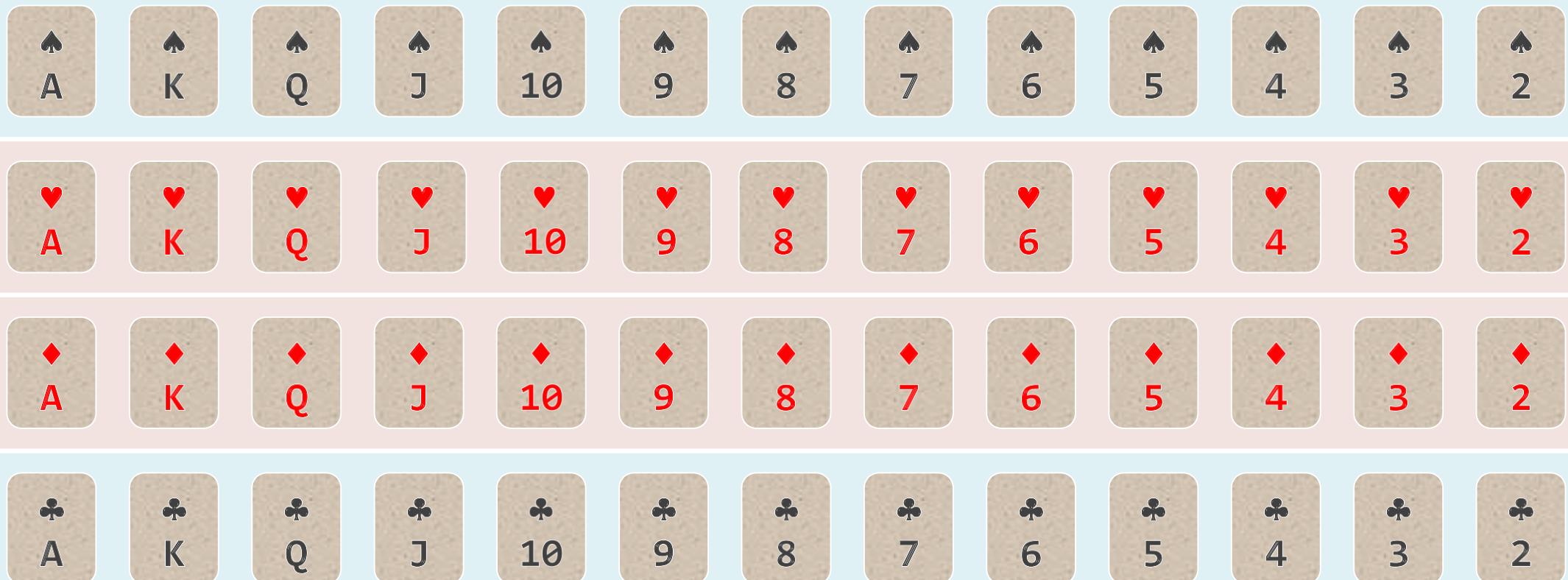
09-E1

邓俊辉

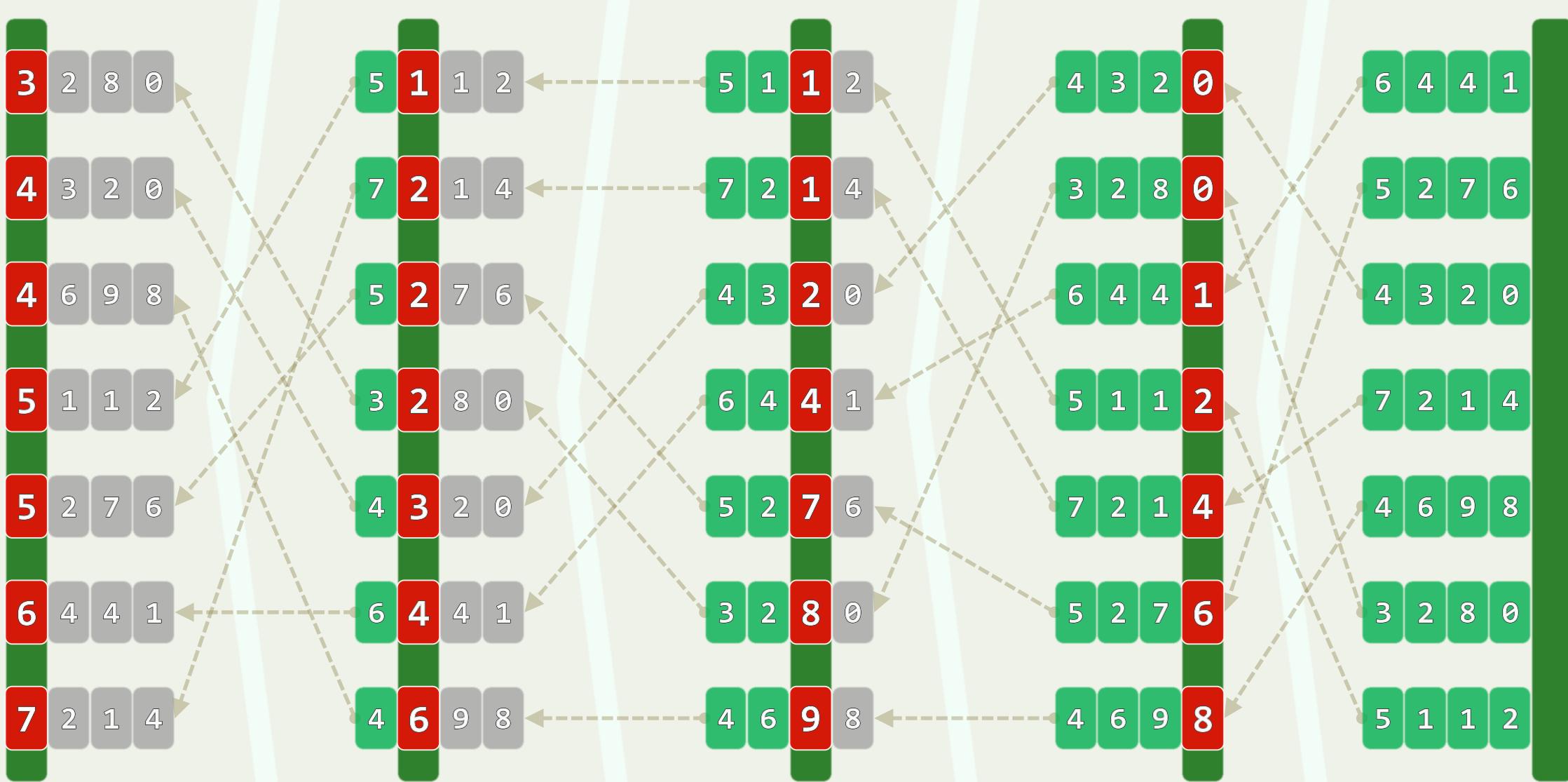
deng@tsinghua.edu.cn

词典序

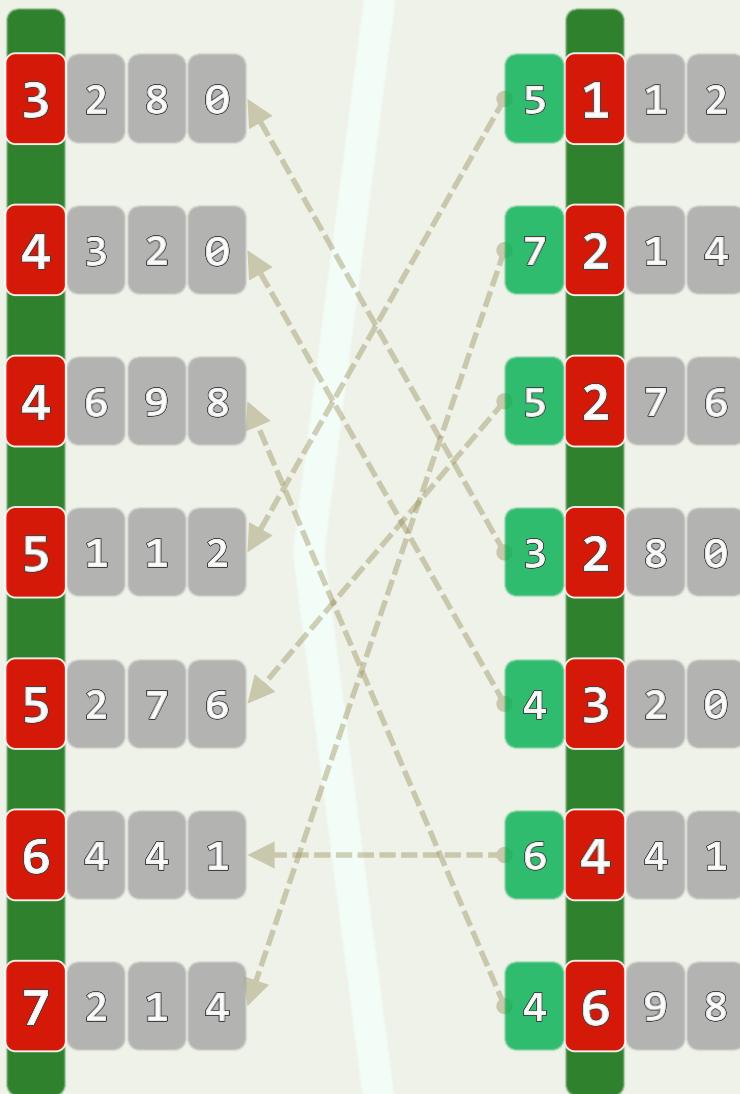
- ❖ 有时，关键码由多个域组成： k_d, k_{d-1}, \dots, k_1 // (suit, point) in bridge
- ❖ 若将各域视作字母，则关键码即单词——按词典的方式排序 (lexicographic order)



算法：自 k_1 到 k_t (低位优先) , 依次以各域为序做一趟桶排序

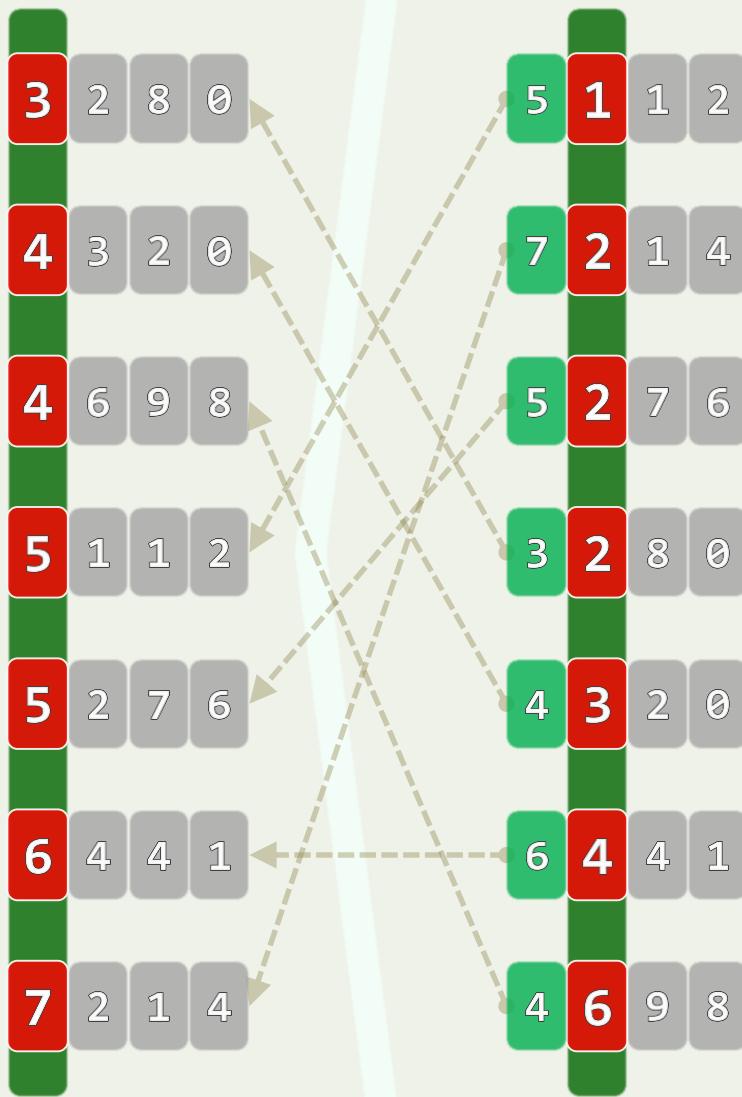


正确性



- ❖ 归纳假设：前*i*趟排序后，所有词条关于低*i*位有序
(第1趟显然)
- ❖ 假设前*i*-1趟均成立，现考查第*i*趟排序之后的时刻
- ❖ 无非两种情况
 - 凡第*i*位不同的词条
即便此前曾是逆序，现在亦必已转为有序
 - 凡第*i*位相同的词条
得益于桶排序的稳定性，必保持原有次序

时间成本



= 各趟桶排序所需时间之和

$$= n + 2m_1$$

$$+ n + 2m_2$$

+ ...

+ n + 2m_d // m_k 为各域的取值范围

$$= O(d \times (n + m))$$

// $m = \max\{m_1, \dots, m_d\}$

❖ 当 $m = O(n)$ 且 d 可视作常数时, $O(n)!$

❖ 在一些特定场合, Radixsort 非常高效 // 稍后举例

实现 (以二进制无符号整数为例)

```
❖ typedef unsigned int U; //约定: 类型T或就是U; 或可转换为U, 并依此定序

❖ template <typename T> void List<T>::radixSort( ListNodePosi<T> p, int n ) {

    ListNodePosi<T> head = p->pred; ListNodePosi<T> tail = p;

    for ( int i = 0; i < n; i++ ) tail = tail->succ; //待排序区间为(head, tail)

    for ( U radixBit = 0x1; radixBit && (p = head); radixBit <<= 1 ) //以下反复地

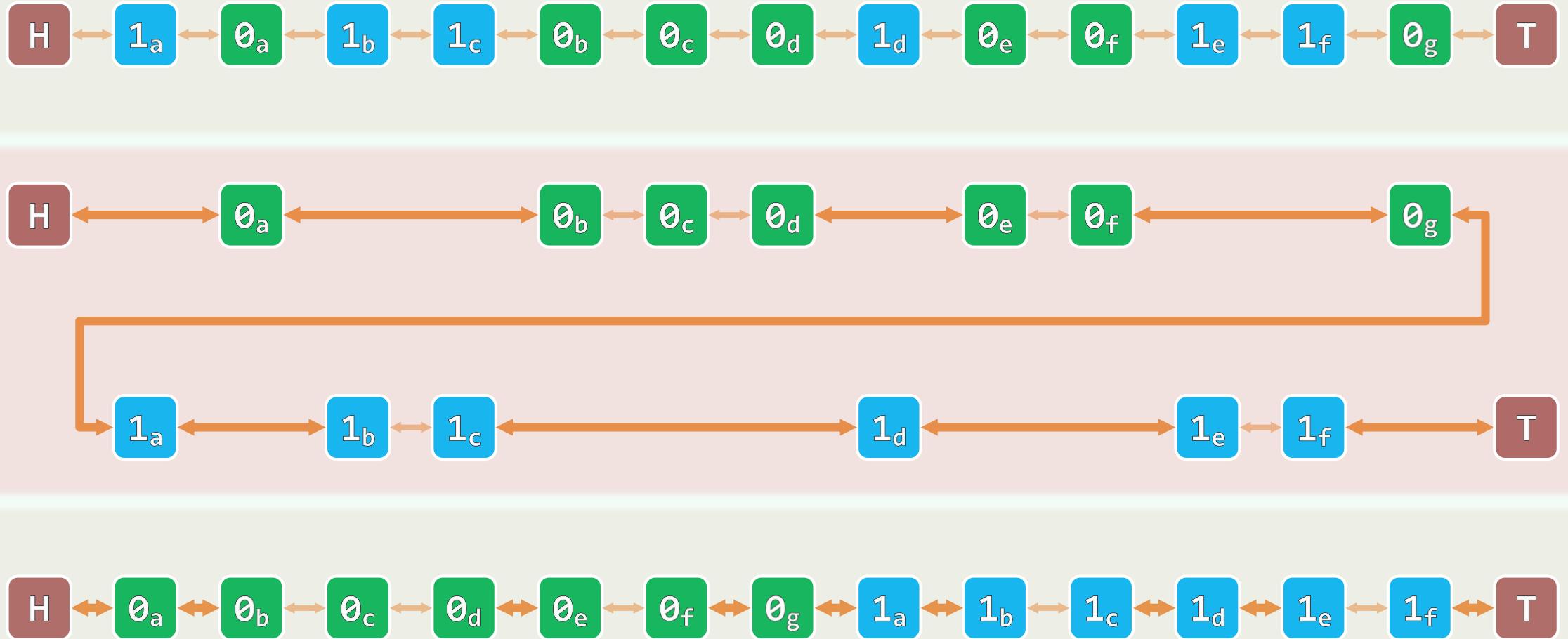
        for ( int i = 0; i < n; i++ ) //根据当前基数位, 将所有节点

            radixBit & U (p->succ->data) ? //分拣为前缀 (0) 与后缀 (1)

                insert( remove( p->succ ), tail ) : p = p->succ;

} //为避免remove()、insert()的低效率, 可拓展List::move(p,tail)接口, 将节点p直接移至tail之前
```

实例



词典

基数排序：整数排序

09-E2

邓俊辉

deng@tsinghua.edu.cn

常对数密度的整数集

❖ 设 $d > 1$ 为常数

❖ 考查取自 $[0, n^d)$ 内的 n 个整数

$$\text{- 常规密度} = \frac{n}{n^d} = \frac{1}{n^{d-1}} \mapsto 0$$

$$\text{- 对数密度} = \frac{\ln n}{\ln n^d} = \frac{1}{d} = \mathcal{O}(1)$$

❖ 亦即，这类整数集的对数密度不超过常数

❖ 这一附加条件，在实际应用中不难满足…

❖ 若取 $d = 4$ ，则即便是64位整数

也只需 $n > (2^{64})^{1/4} = 2^{16} = 65,536$

❖ 对于这类整数集

有无效率为 $\mathcal{O}(n \log n)$ 的排序算法？

线性排序算法

❖ 预处理：将所有元素转换为n进制形式：

$$x = (x_d, \dots, x_3, x_2, x_1)$$



❖ 于是，每个元素均转化为d个域，故可直接套用Radixsort算法

❖ 排序时间 $= d \cdot (n + n) = \mathcal{O}(n)$ //“突破”了此前确定的下界！

❖ 原因在于：
- 整数取值范围有限制
- 不再是基于比较的计算模式

❖ 进制转换需要多少时间？回忆一下此前的相关内容...

e9.-F

词典

计数排序

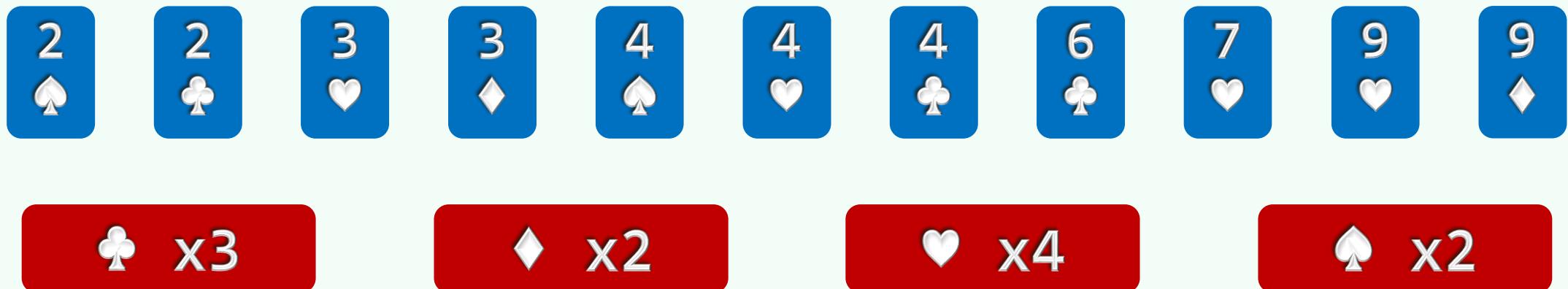
邓俊辉

deng@tsinghua.edu.cn

The purpose of computing is insight, not numbers.

算法 (1)

- ❖ 回忆：基数排序中反复做的桶排序...
- ❖ 亦属“小集合 + 大数据”类型，是否可以更快？



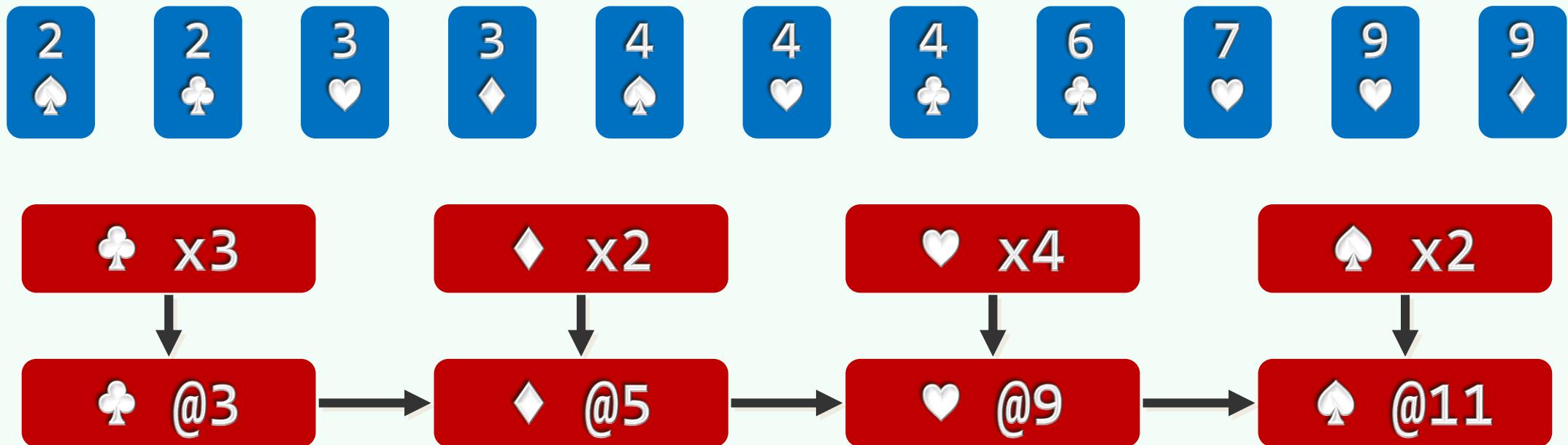
❖ 仍以纸牌排序为例 ($n \gg m = 4$)，假设已按点数排序，以下对花色排序

1) 经过分桶，统计出各种花色的数量 // $\mathcal{O}(n)$

算法 (2+3)

2) 自前向后扫描各桶，依次累加 //cumulative sum, $O(m)$

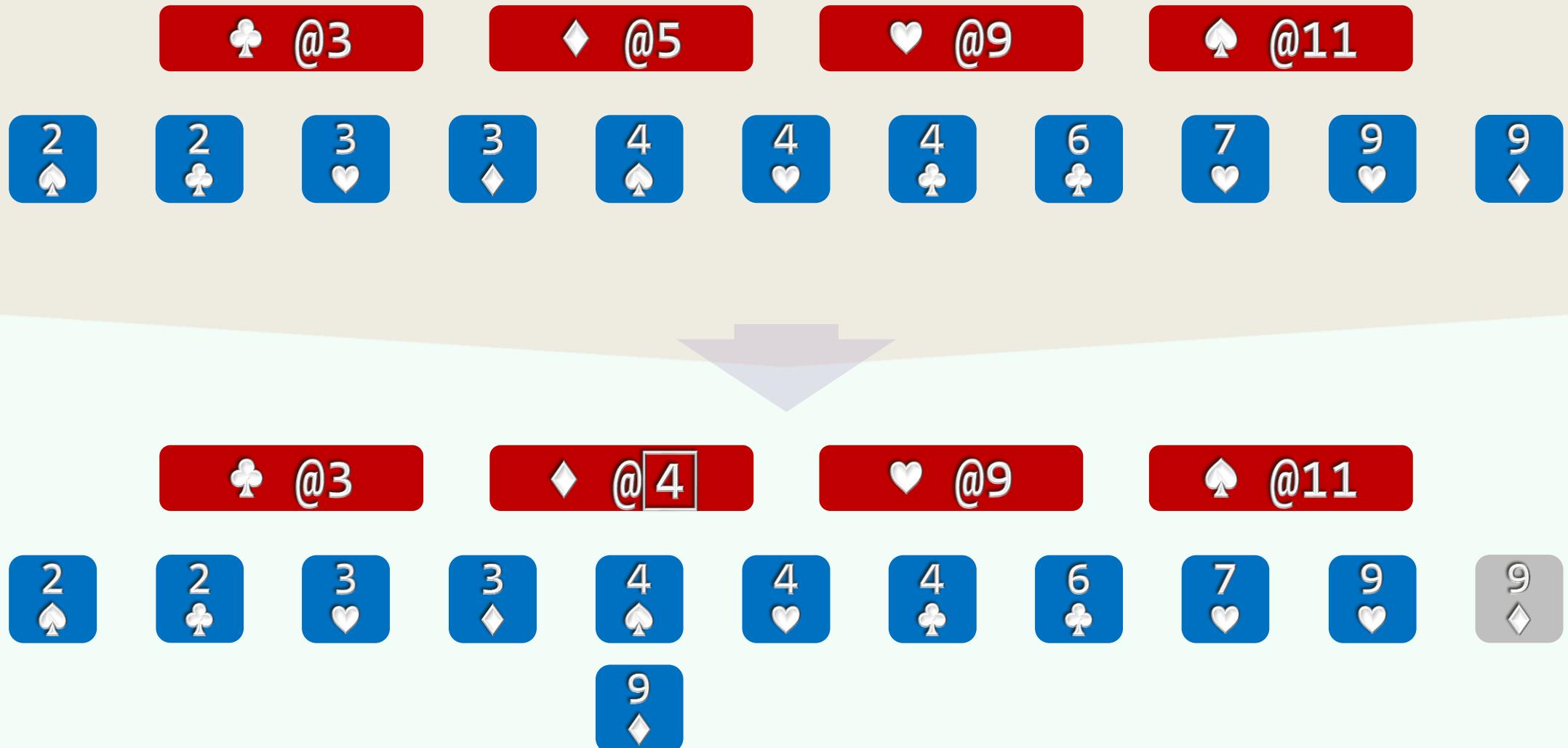
即可确定各套花色所处的秩区间： $[0, 3) + [3, 5) + [5, 9) + [9, 11)$



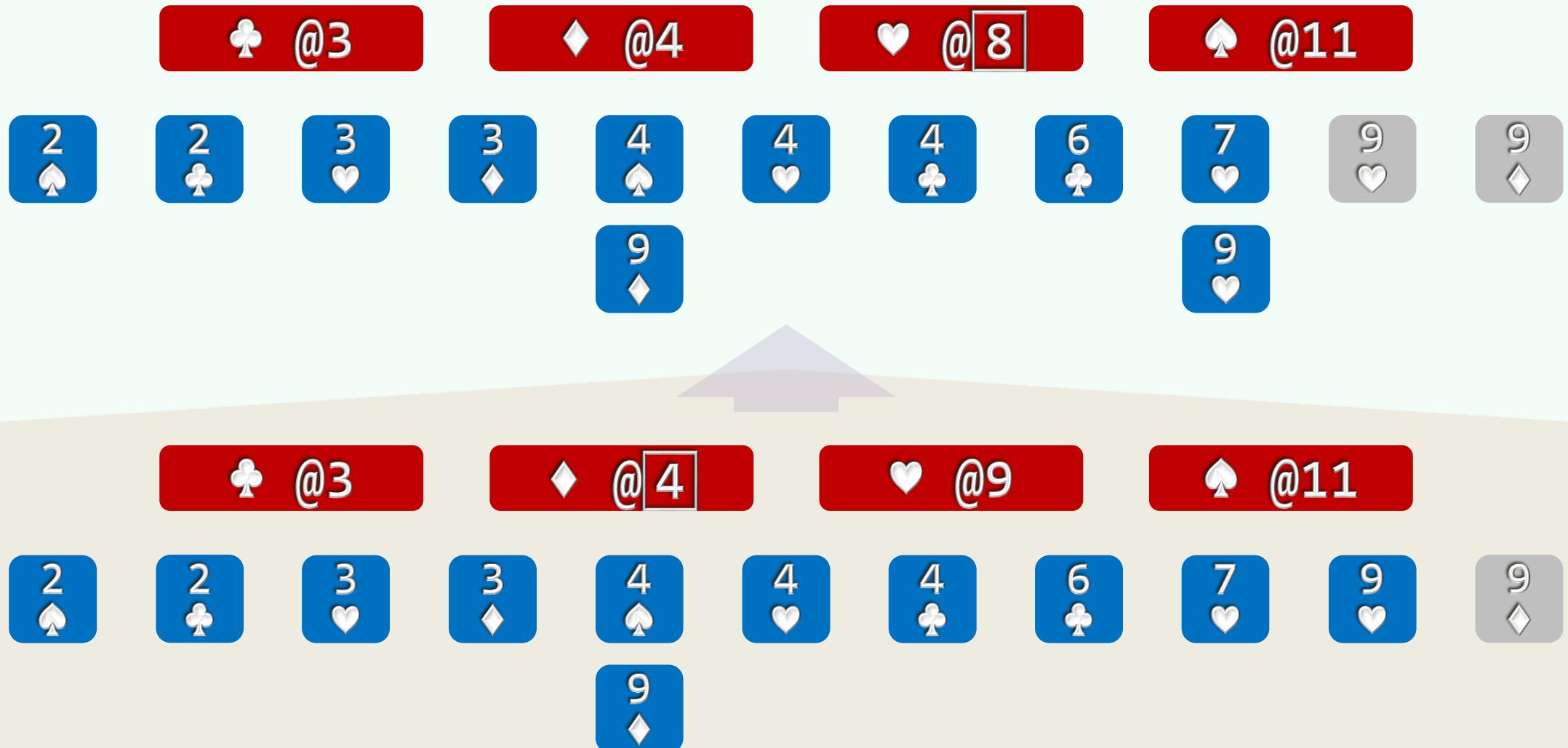
3) 自后向前扫描每一张牌 // $O(n)$

对应桶的计数减一，即是其在最终有序序列中对应的秩

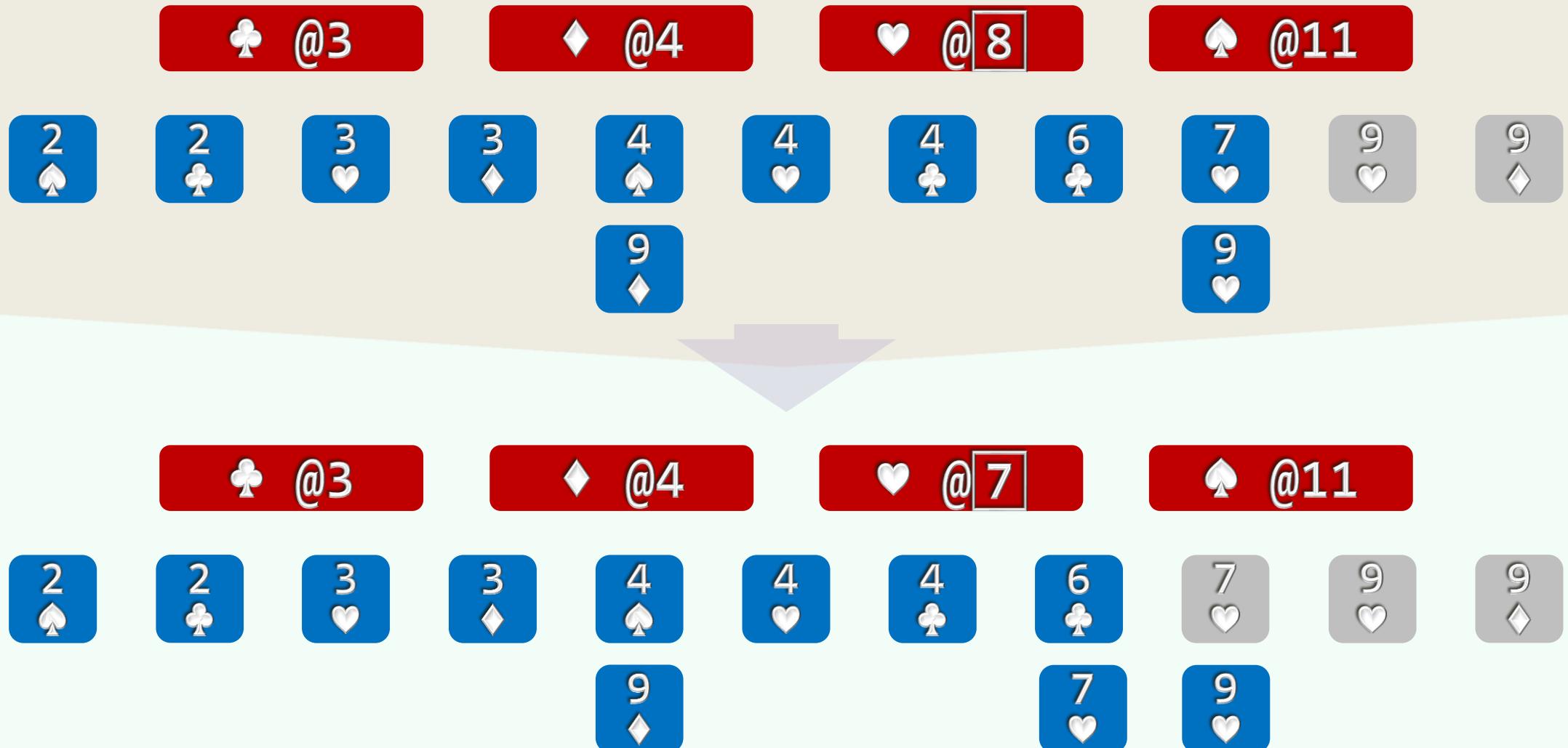
实例 (1/11)



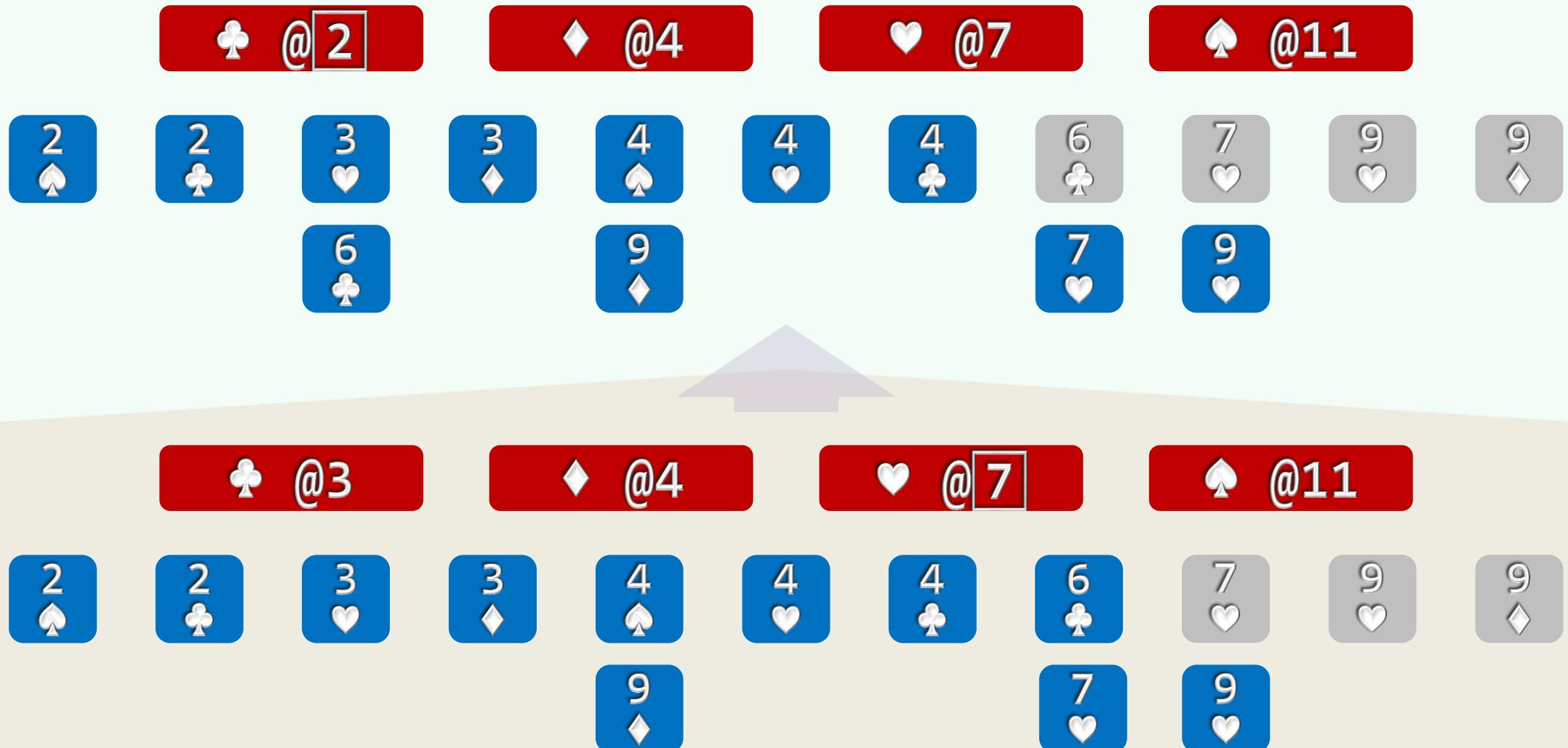
实例 (2/11)



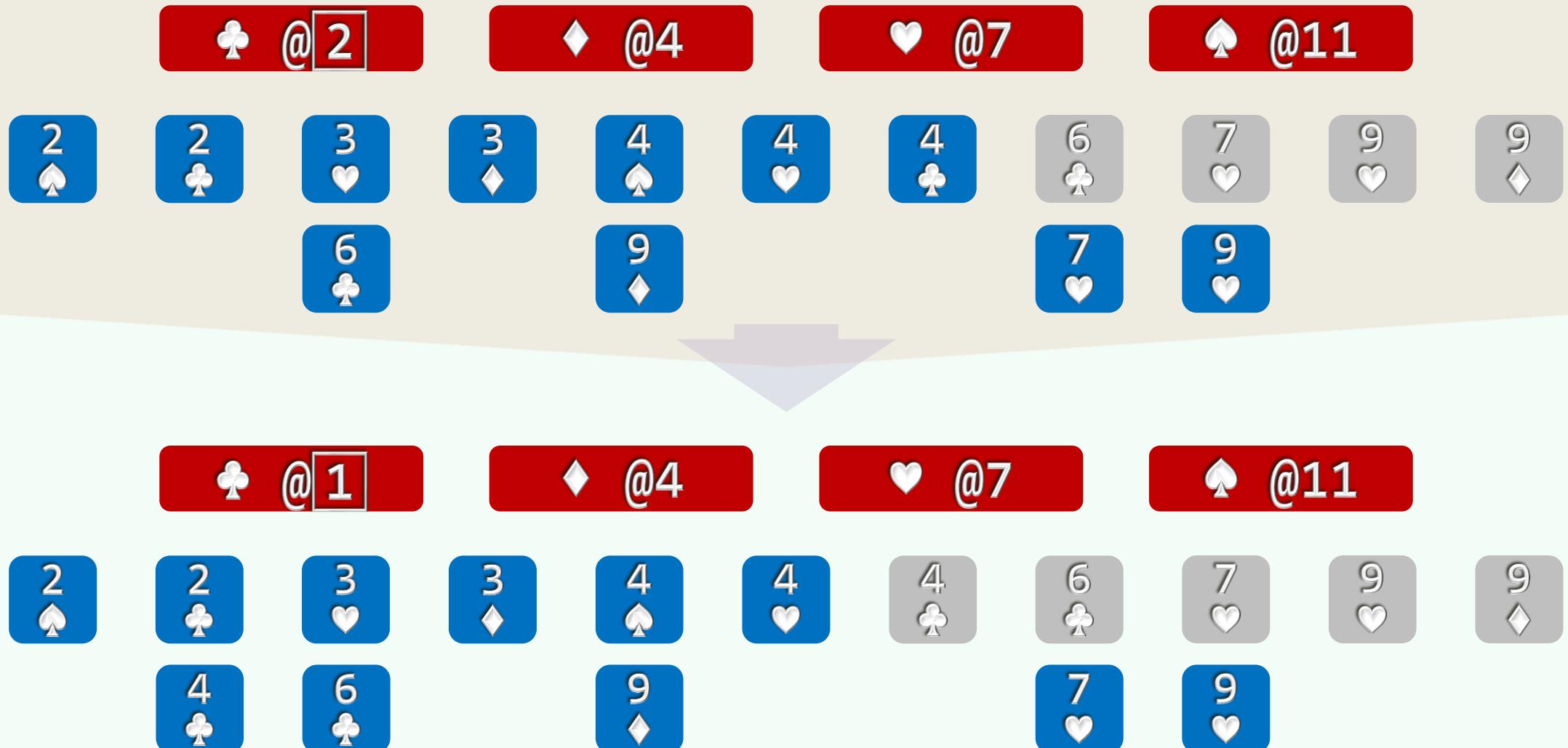
实例 (3/11)



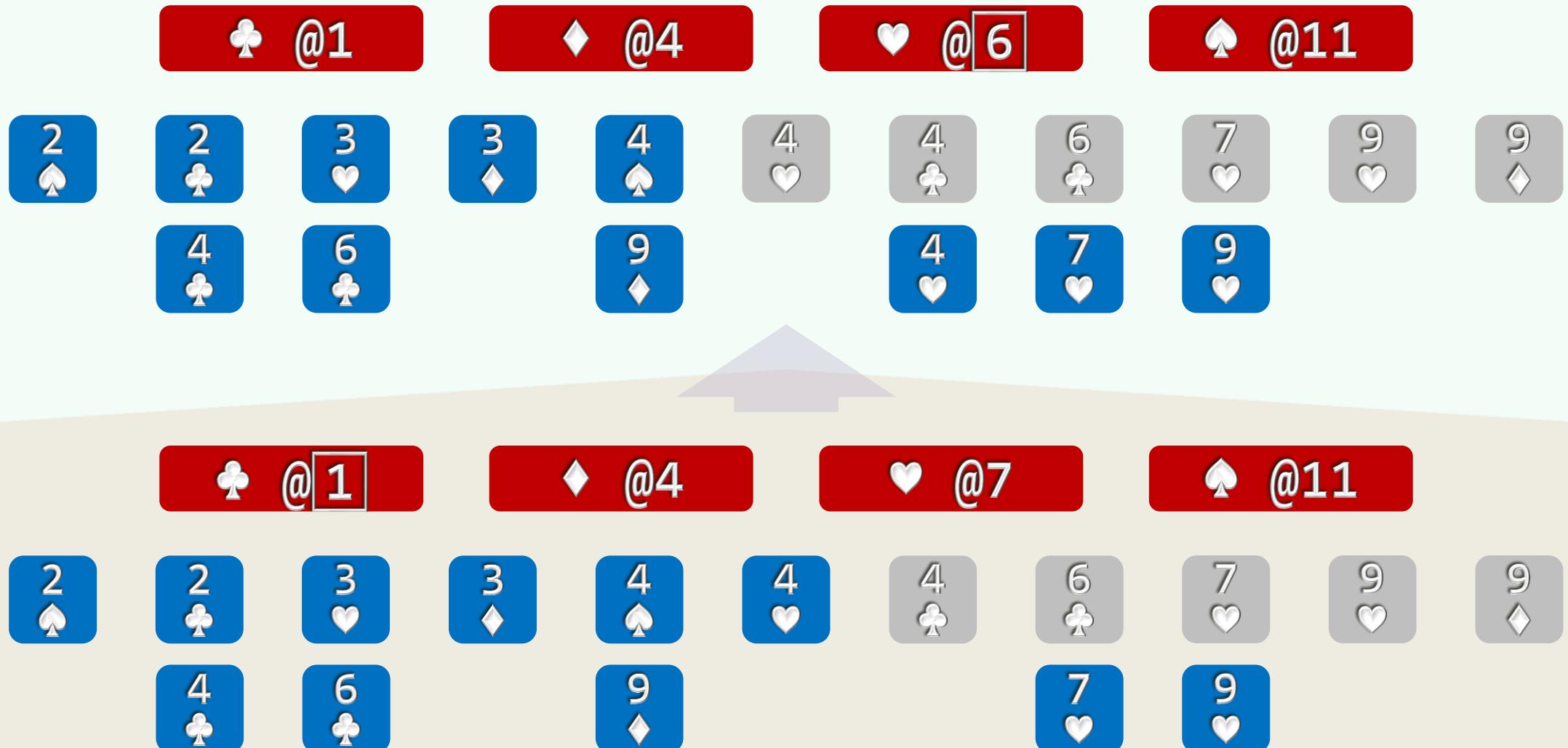
实例 (4/11)



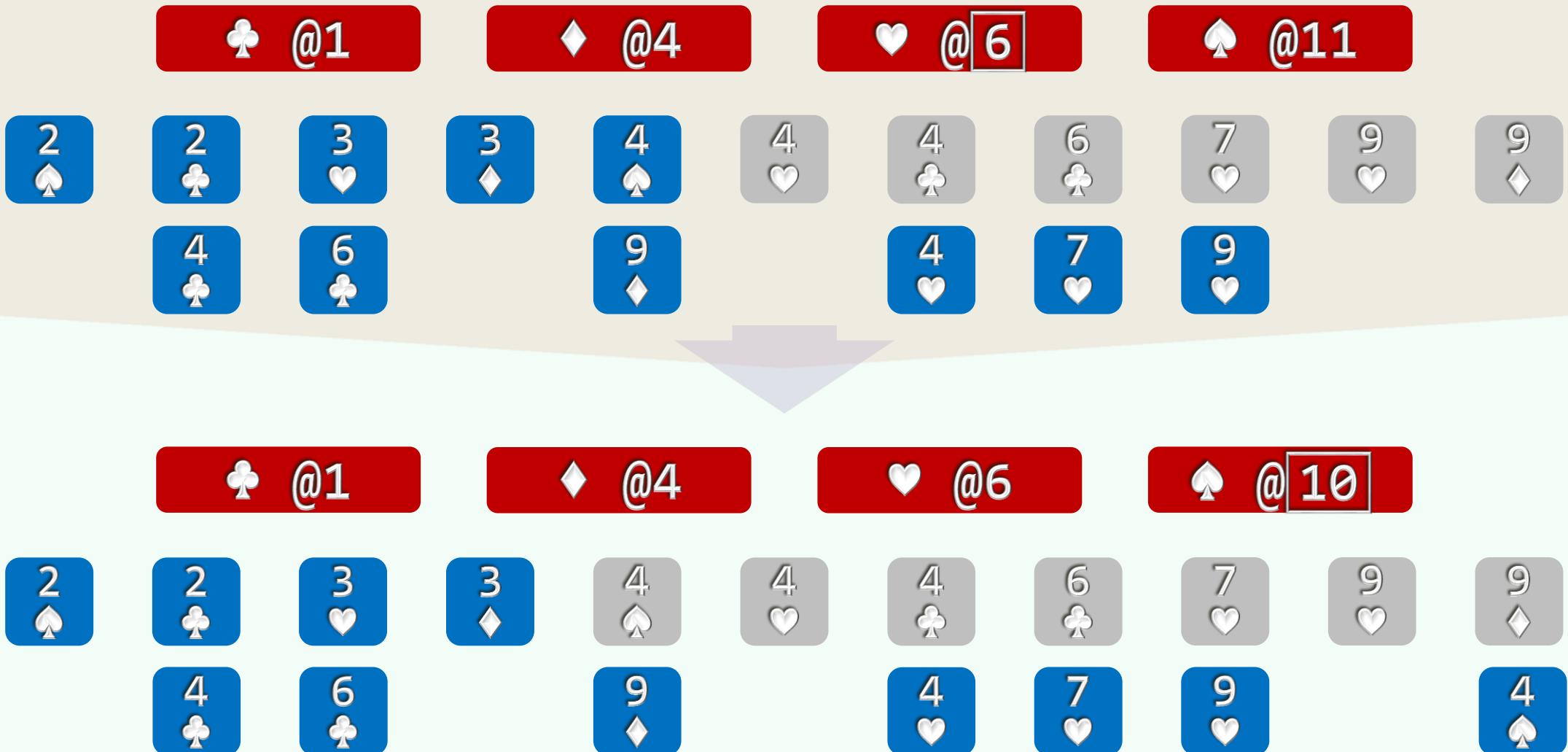
实例 (5/11)



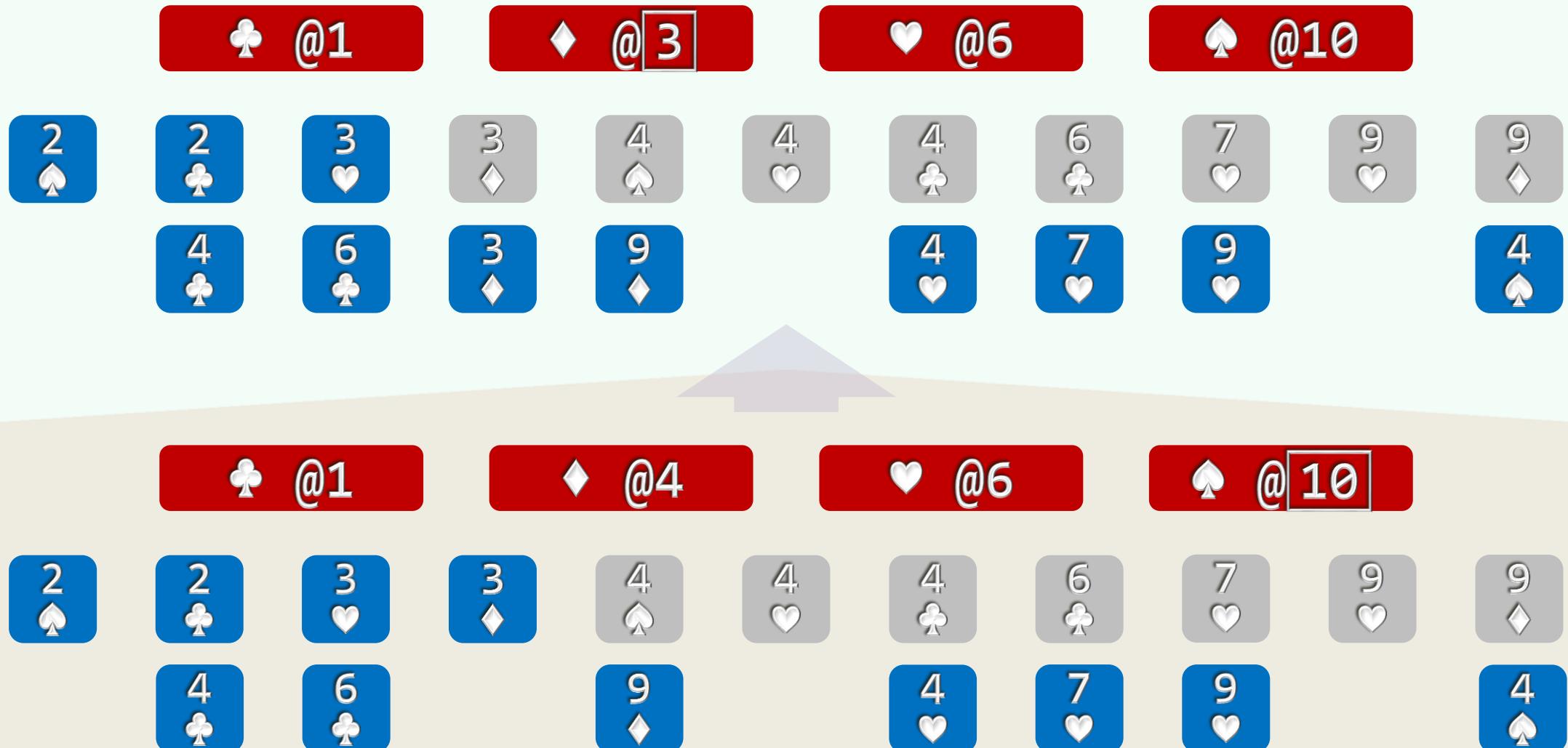
实例 (6/11)



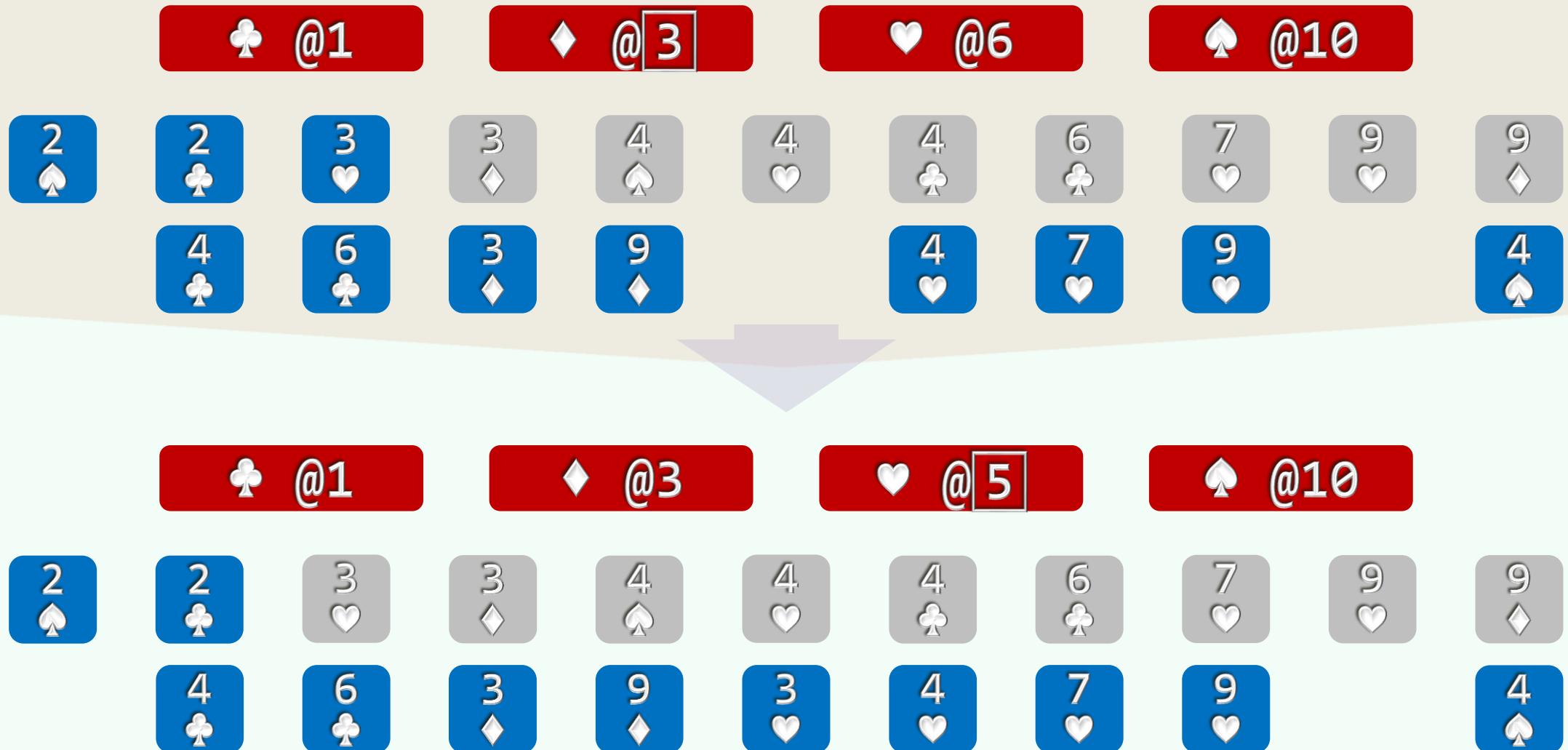
实例 (7/11)



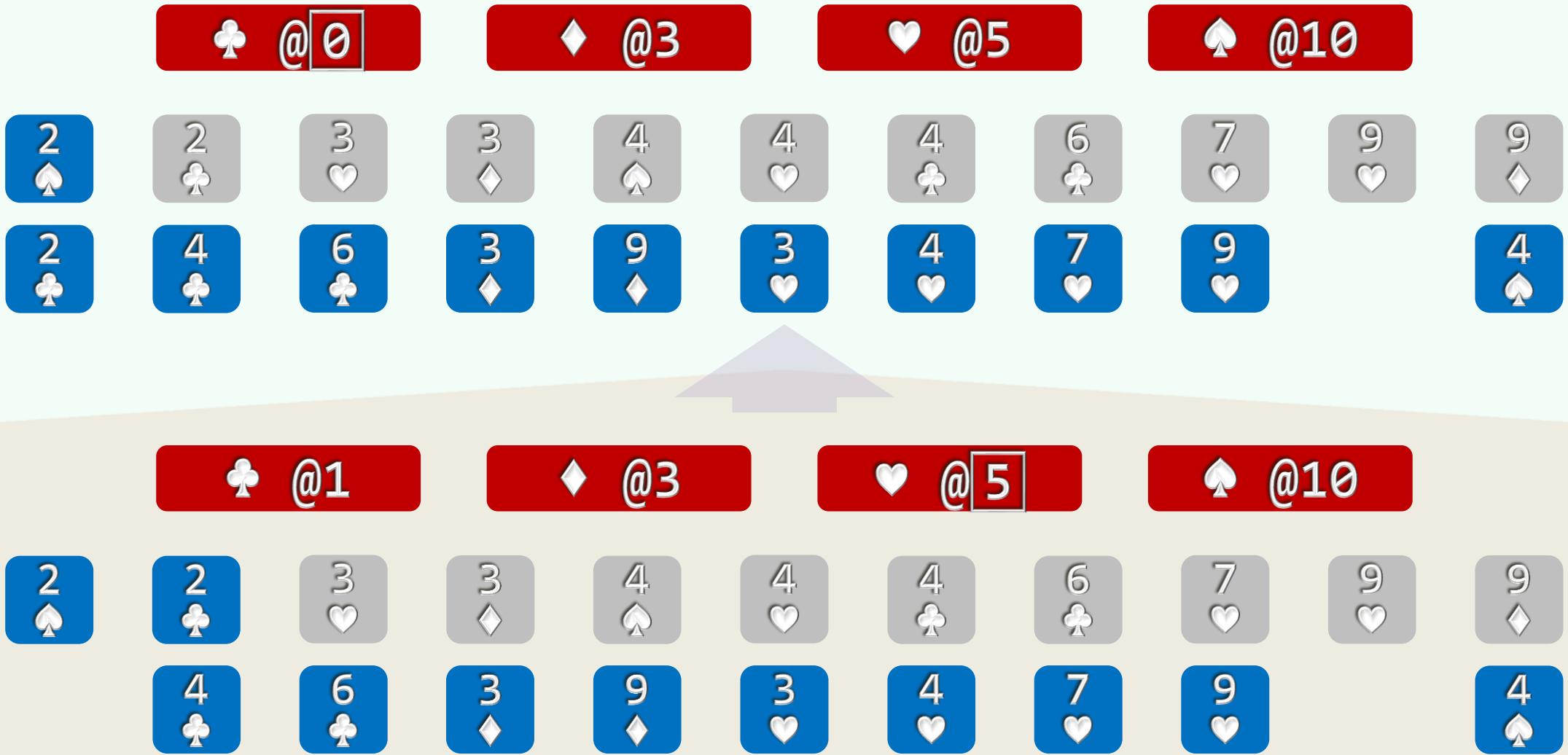
实例 (8/11)



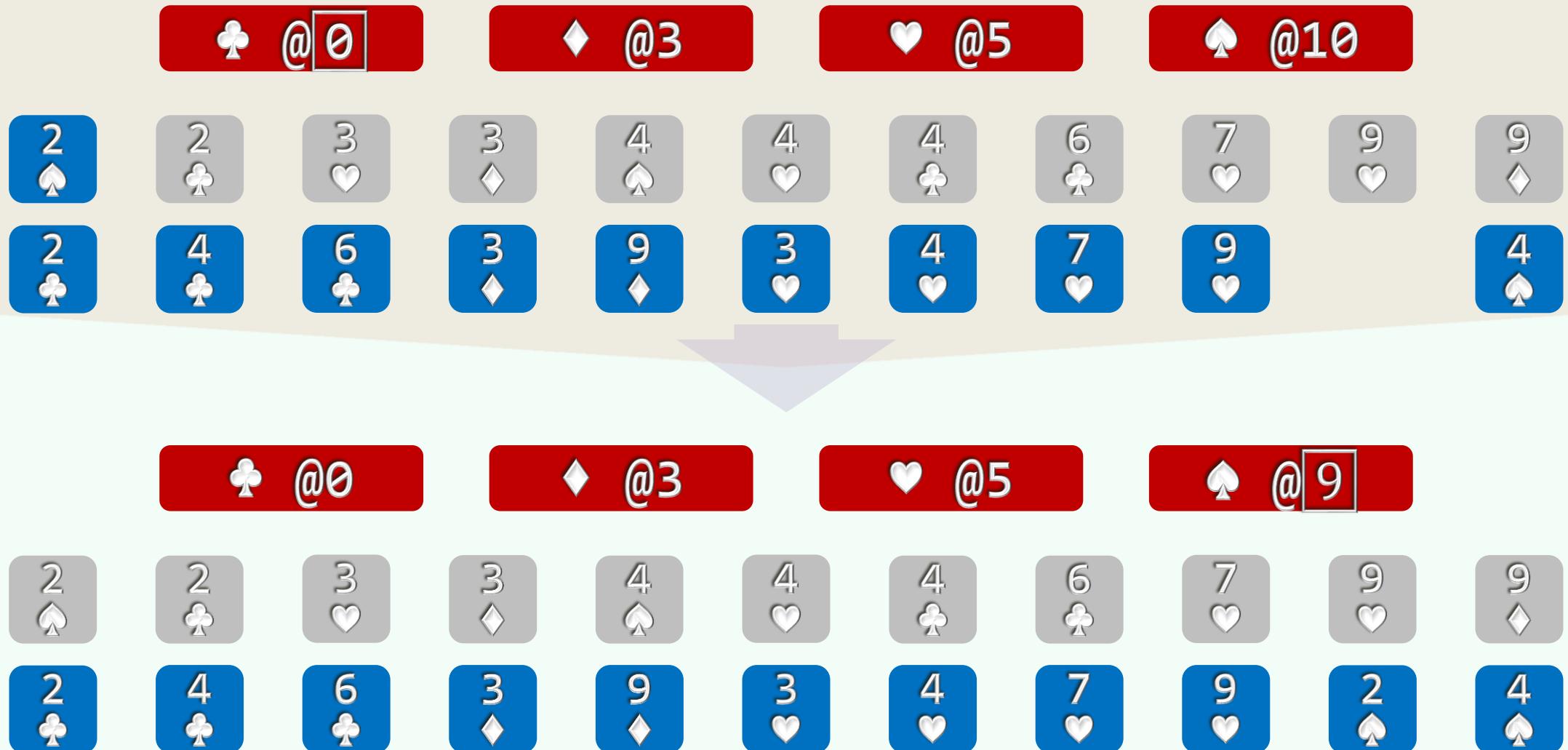
实例 (9/11)



实例 (10/11)



实例 (11/11)



分析

- ❖ 时间复杂度 = $\mathcal{O}(n + m + n) = \mathcal{O}(n)$ ——高效处理大规模数据
- ❖ 空间复杂度 = $\mathcal{O}(n)$ ——不能就地完成，需借助外存
- ❖ 最后一步的扫描次序，可否改为自前向后？



词典

跳转表：结构

e9-G1

邓俊辉

deng@tsinghua.edu.cn

去沿江上下，或二十里，或三十里，选高阜处置一烽火台，每台用五十军守之。

动机与思路

- ❖ [William Pugh, 1989]

Skip Lists: A Probabilistic Alternative to Balanced Trees

Skip lists are data structures that use

probabilistic balancing rather than

strictly enforced balancing

Algorithms for insertion and deletion in skip lists

are much simpler and significantly faster than

equivalent algorithms for balanced trees



结构

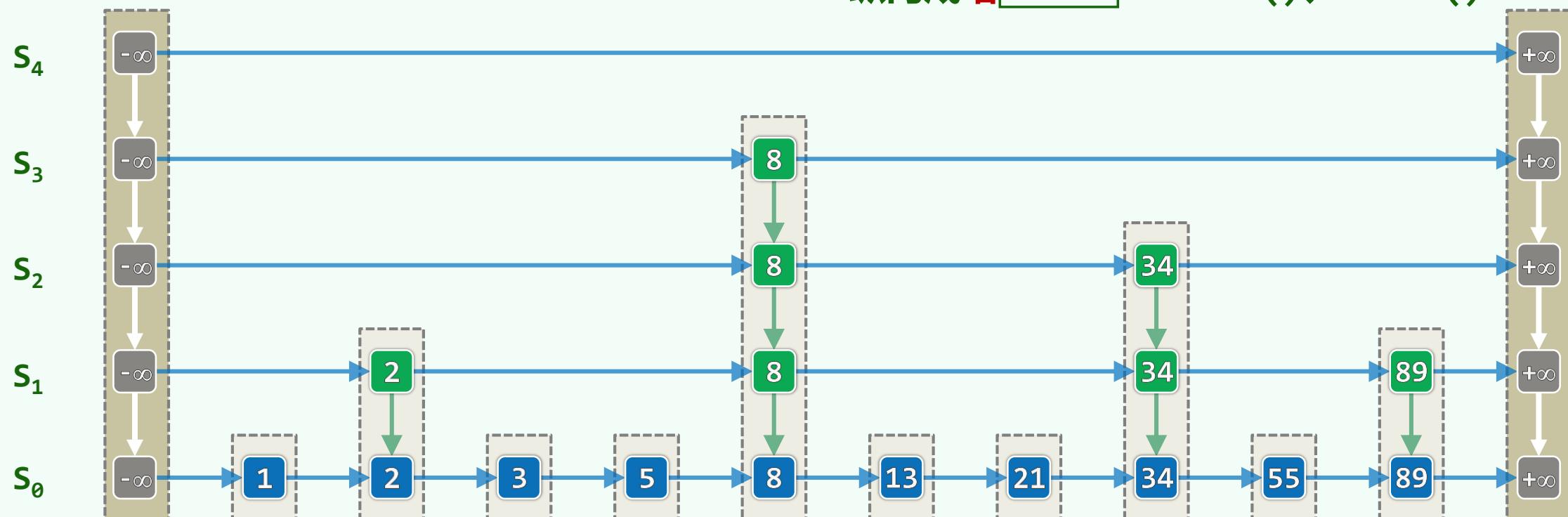
❖ 分层耦合的多个列表: s_0, s_1, \dots, s_h

s_0/s_h 称作底/顶层

❖ 横向为层[**level**]: `prev()`、`next()`

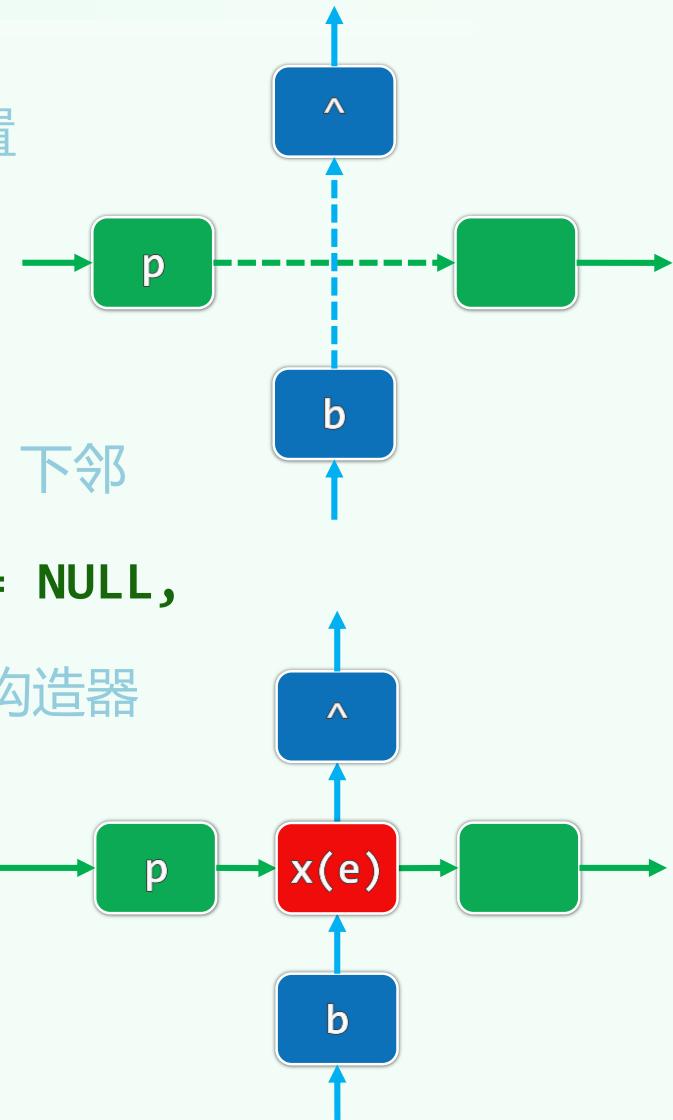
设有头、尾哨兵

❖ 纵向成塔[**tower**]: `above()`、`below()`



QuadNode

```
template <typename T> using QNodePosi = QNode<T>*; //节点位置  
template <typename T> struct QNode { //四联节点  
    T entry; //所存词条  
    QNodePosi<T> pred, succ, above, below; //前驱、后继、上邻、下邻  
    QNode( T e = T(), QNodePosi<T> p = NULL, QNodePosi<T> s = NULL,  
           QNodePosi<T> a = NULL, QNodePosi<T> b = NULL ) //构造器  
        : entry(e), pred(p), succ(s), above(a), below(b) {}  
    QNodePosi<T> insert( T const& e, QNodePosi<T> b = NULL );  
    //将e作为当前节点的后继、b的上邻插入  
};
```



QuadList

```
template <typename T> struct Quadlist { //四联表
    Rank _size; //节点总数
    QNodePosi<T> header, trailer; //头、尾哨兵
    void init(); int clear(); //初始化、清除
    Quadlist() { init(); } //构造
    ~Quadlist() { clear(); delete header; delete trailer; } //析构
    T remove( QNodePosi<T> p ); //删除p
    QNodePosi<T> insert( T const & e, QNodePosi<T> p, QNodePosi<T> b = NULL );
        //将e作为p的后继、b的上邻插入
};
```

Skiplist

```
template < typename K, typename V > struct Skiplist :  
public Dictionary<K, V>, public List< Quadlist< Entry<K, V> >*> {  
Skiplist() { insertAsFirst( new Quadlist< Entry<K, V> > ); } //至少有一层空列表  
QNodePosi< Entry<K, V> > search( K ); //由关键码查询词条  
Rank size() { return empty() ? 0 : last()->data->size(); } //词条总数  
Rank height() { return List::size(); } //层高, 即Quadlist总数  
bool put( K, V ); //插入 (Skiplist允许词条重复, 故必然成功)  
V * get( K ); //读取  
bool remove( K ); //删除  
};
```

空间性能

❖ 较之常规的单层列表

- 每次操作过程中，需访问的节点是否会实质地增多？
- 每个节点都至多可能重复 h 份，空间复杂度是否因此有实质增加？ //先来回答后者...

❖ 逐层随机减半： S_k 中的每个关键码，在 S_{k+1} 中依然出现的概率，均为 $p = 1/2$

——稍后将会看到，这一性质可以通过投掷硬币来模拟

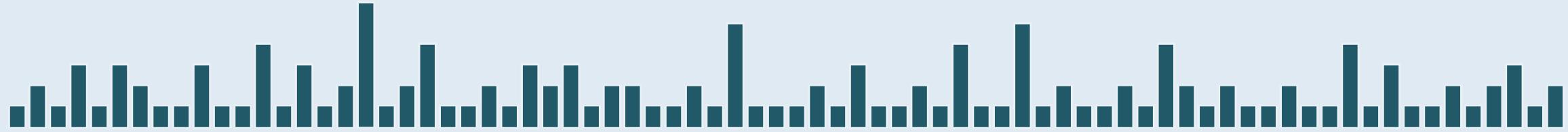
❖ 可见，各塔的高度符合几何分布： $Pr(h = k) = p^{k-1} \cdot (1 - p)$

❖ 于是，期望的塔高： $\mathbb{E}(h) = 1/(1 - p) = 2$

❖ 什么，没有学过概率？不要紧，有直观的解释...

空间性能

- 既然逐层随机减半，故 S_0 中任一关键码在 S_k 中依然出现的概率为 2^{-k}



第 k 层节点数的期望值 $\mathbb{E}(|S_k|) = n \cdot 2^{-k} = n/2^k$

- 于是，所有节点期望的总数（即各层列表所需空间总和）为

$$\mathbb{E}\left(\sum_k |S_k|\right) = \sum_k \mathbb{E}(|S_k|) = n \times \sum_k 2^{-k} < 2n = \mathcal{O}(n)$$

- 结论：跳转表所需空间为 expected- $\mathcal{O}(n)$
- 类比：半衰期为1年的放射性物质中，各粒子的平均寿命不过2年
- 更为细致地，塔高的方差是否足够小？ // 比照稍后对层高的分析

词典

跳转表：查找

e9 - G2

只见参仙老怪梁子翁笑嘻嘻的站起身来，向众人拱了拱手，缓步走到庭中，忽地跃起，左足探出，已落在欧阳克插在雪地的筷子之上，拉开架子，……，把一路巧打连绵的“燕青拳”使了出来，脚下纵跳如飞，每一步都落在竖直的筷子之上。

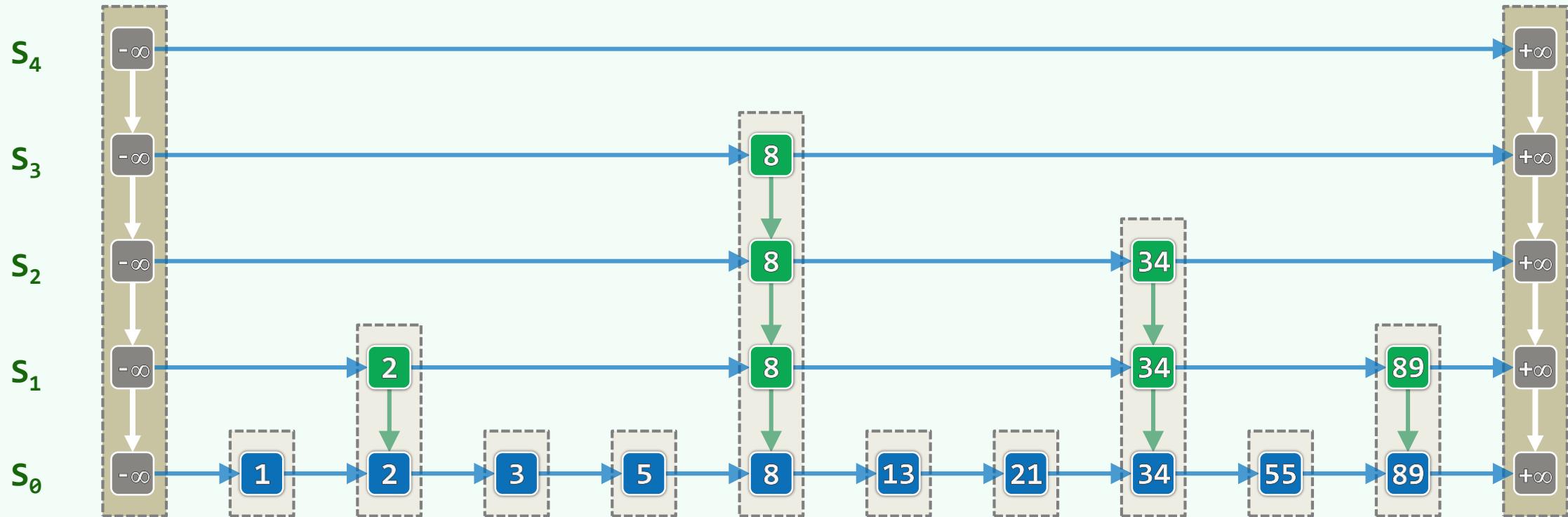
邓俊辉

deng@tsinghua.edu.cn

减而治之：由粗到细 = 由高到低

❖ 成功: `search(21 | 34 | 1 | 89)`

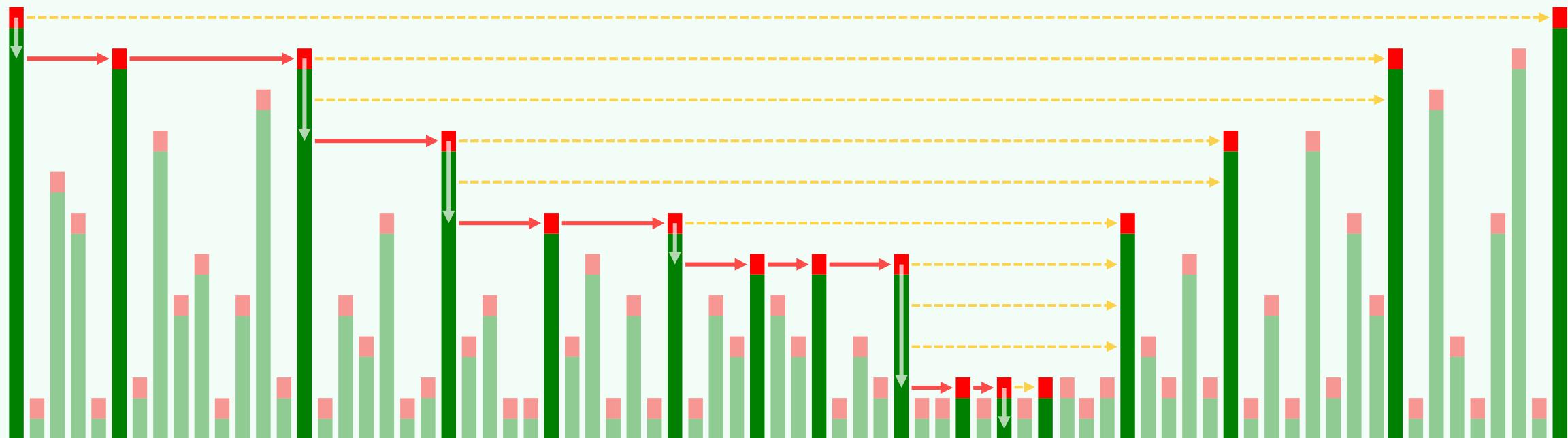
❖ 失败: `search(80 | 0 | 99)`



减而治之：总览

❖ 查找时间取决于横向、纵向的累计跳转次数

❖ 会否因层次过多，首先导致纵向跳转过多？



实现

```
template <typename K, typename V> //关键码不大于k的最后一个词条（所对应塔的基座）

QNodePosi< Entry<K, V> > Skiplist<K, V>::search( K k ) {

    for ( QNodePosi< Entry<K, V> > p = first()->data->header; ;  ) //从顶层的首节点出发

        if ( (p->succ->succ) && (p->succ->entry.key <= k) ) p = p->succ; //尽可能右移

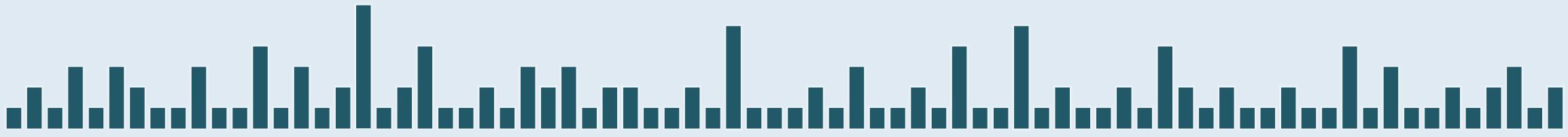
    else if ( p->below ) p = p->below; //水平越界时，下移

    else return p; //验证：此时的p符合输出约定（可能是最底层列表的header）

} //体会：得益于哨兵的设置，哪些环节被简化了？
```

纵向跳转 ~ 层高

❖ 观察：一座塔高度不低于 | 低于 k 的概率 = p^k | $1 - p^k$



❖ 引理：随着 k 的增加， S_k 为空/非空的概率急剧上升/下降

$$Pr(|S_k| = 0) = (1 - p^k)^n \geq 1 - n \cdot p^k \quad Pr(|S_k| > 0) \leq n \cdot p^k$$

❖ 推论：跳转表高度 $h = \mathcal{O}(\log n)$ 的概率极大

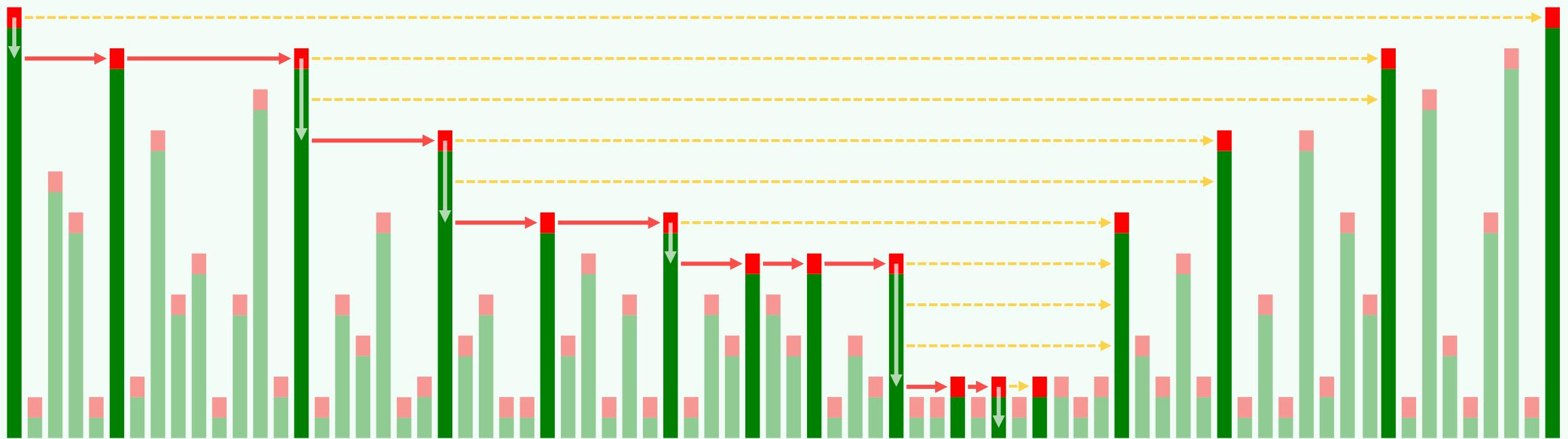
❖ 比如：若 $p = 1/2$ ，则第 $k = 3 \cdot \log_{1/p} n$ 层非空（当且仅当 $h \geq k$ ）的概率为

$$Pr(|S_k| > 0) \leq n \cdot p^k = n \cdot n^{-3} = 1/n^2 \rightarrow 0$$

❖ 结论：查找过程中，纵向跳转的次数，累计不过 expected- $\mathcal{O}(\log n)$

横向跳转 ~ 紧邻塔顶

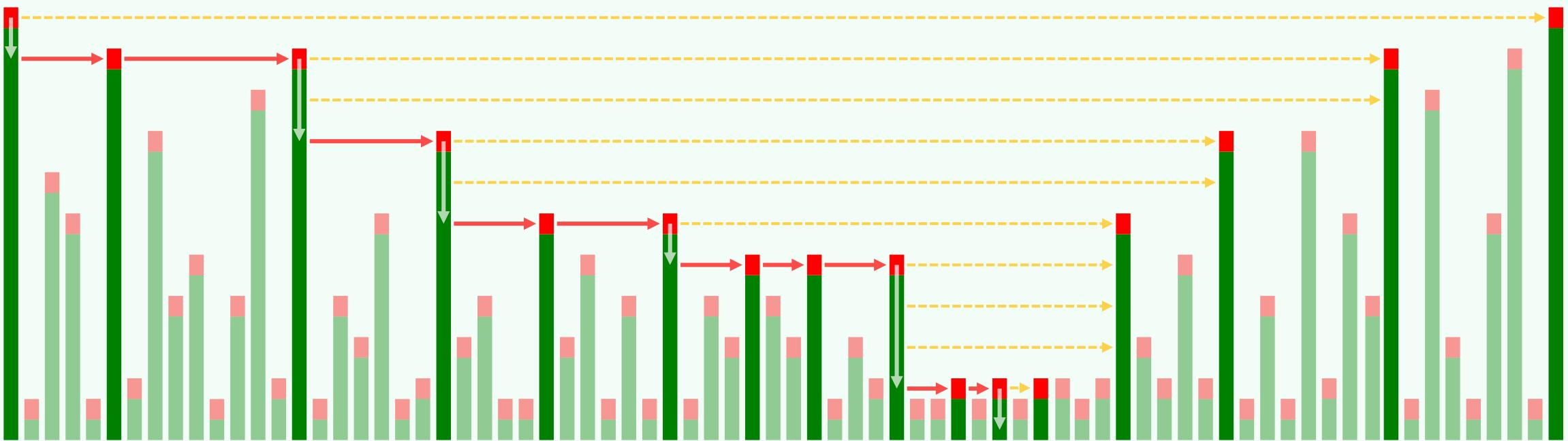
- ❖ 那么：横向跳转是否可能很多次？比如 $\omega(\log n)$ ，甚至 $\Omega(n)$ ？
- ❖ 观察：在同一水平列表中，横向跳转所经节点必然依次紧邻，而且每次抵达都是塔顶



- ❖ 于是：若将沿同一层列表跳转的次数记作 Y ，则它符合几何分布 $Pr(Y = k) = (1 - p)^k \cdot p$

横向跳转 ~ 期望时间成本

- ❖ 定理: $\mathbb{E}(Y) = (1 - p)/p = (1 - 0.5)/0.5 = 1$ 次
- ❖ 因此: 在同一层列表中连续跳转的**期望时间成本** = 1次跳转 + 1次驻足 = 2



- ❖ 结论: 跳转表的每次查找, 都可在 $\leq \text{expected-}(2h) = \text{expected-}\mathcal{O}(\log n)$ 时间内完成

词典

跳转表：插入与删除

e9-G3

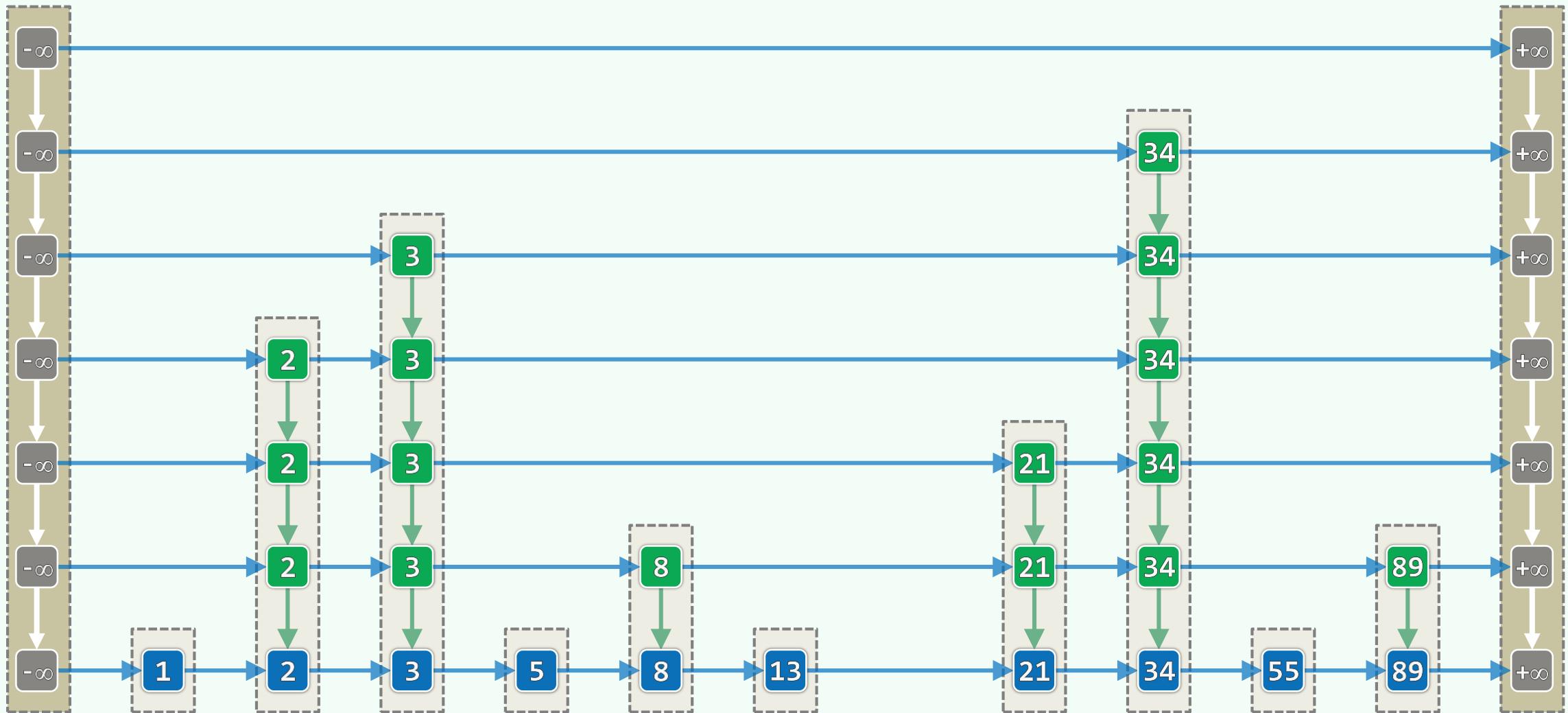
邓俊辉

deng@tsinghua.edu.cn

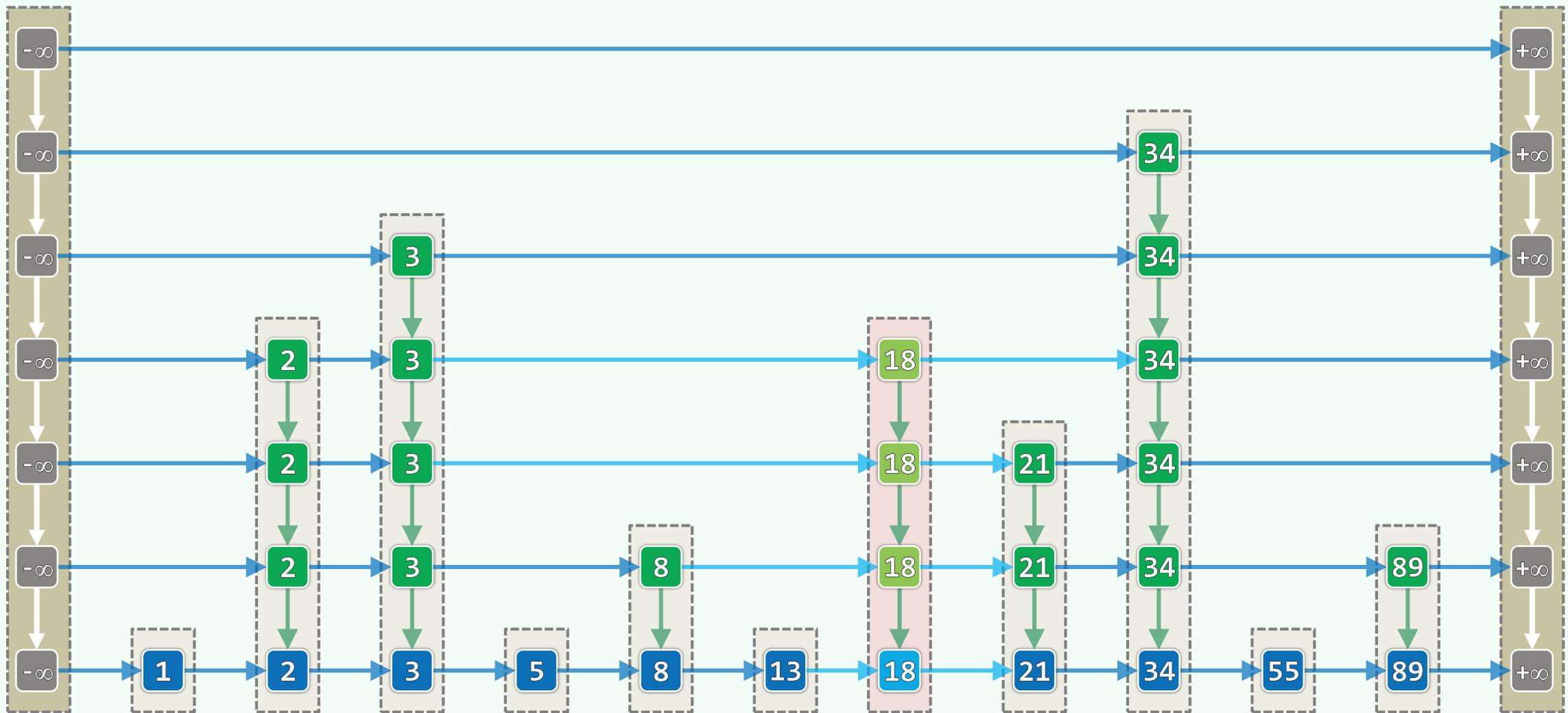
如果一个人遇到不可解之事，把脑子想穿了，也找不到其中的原因，怎么办呢？

他或许会去庙里烧香，把自己的难题交给算命先生，听任他们的摆布。

实例：put(18)之前 = remove(18)之后



实例：put(18)之后 = remove(18)之前



插入算法：整体

```
template <typename K, typename V> bool Skiplist<K, V>::put( K k, V v ) {  
    Entry< K, V > e = Entry< K, V >( k, v ); //待插入的词条 (将被同一塔中所有节点共用)  
    QNodePosi< Entry<K, V> > p = search( k ); //查找插入位置：新塔将紧邻其右，逐层生长  
    ListNodePosi< Quadlist< Entry<K, V > *> qlist = last(); //首先在最底层  
    QNodePosi< Entry<K, V > > b = qlist->data->insert( e, p ); //创建新塔的基座  
    while ( rand() & 1 ) {  
        /* ... 建塔 ... */  
    }  
    return true; //Dictionary允许重复元素，故插入必成功  
} //体会：得益于哨兵的设置，哪些环节被简化了？
```

插入算法：建塔

```
while ( rand() & 1 ) { //经投掷硬币，若新塔需再长高，则  
    while ( p->pred && !p->above ) p = p->pred; //找出不低于此高度的最近前驱  
    if ( !p->pred && !p->above ) { //若该前驱是header，且已是最顶层，则  
        insertAsFirst( new Quadlist< Entry<K, V> > ); //需要创建新的一层  
        first()->data->header->below = qlist->data->header;  
        qlist->data->header->above = first()->data->header;  
    }  
    p = p->above; qlist = qlist->pred; //上升一层，并在该层  
    b = qlist->data->insert( e, p, b ); //将新节点插入p之后、b之上  
}
```

删除算法 (1/3): 预备

```
template <typename K, typename V> bool SkipList<K, V>::remove( K k ) {  
  
    QNodePosi< Entry<K, V> > p = search( k ); //查找目标词条  
  
    if ( !p->pred || (k != p->entry.key) ) return false; //若不存在, 直接返回  
  
    ListNodePosi< Quadlist< Entry<K, V> *>* qlist = last(); //从底层Quadlist开始  
  
    while ( p->above ) { qlist = qlist->pred; p = p->above; } //升至塔顶  
  
    /* ... 2. 拆塔 ... */  
  
    /* ... 3. 删除空表 ... */  
  
    return true; //删除成功  
}  
//体会: 得益于哨兵的设置, 哪些环节被简化了?
```

删除算法 (2/3): 拆塔

```
template <typename K, typename V> bool SkipList<K, V>::remove( K k ) {  
    /* ... 1. 预备 ... */  
  
    do { QNodePosi< Entry<K, V> > lower = p->below; //记住下一层节点，并  
        qlist->data->remove( p ); //删除当前层节点，再  
        p = lower; qlist = qlist->succ; //转入下一层  
    } while ( qlist->succ ); //直到塔基  
  
    /* ... 3. 删除空表 ... */  
  
    return true; //删除成功  
} //体会：得益于哨兵的设置，哪些环节被简化了？
```

删除算法 (3/3): 删除空表

```
template <typename K, typename V> bool SkipList<K, V>::remove( K k ) {  
    /* ... 1. 预备 ... */  
  
    /* ... 2. 拆塔 ... */  
  
    while ( (1 < height()) && (first()->data->_size < 1) ) { //逐层清除  
        List::remove( first() );  
  
        first()->data->header->above = NULL;  
    } //已不含词条的Quadlist (至少保留最底层空表)  
  
    return true; //删除成功  
}  
//体会: 得益于哨兵的设置, 哪些环节被简化了?
```