



湖南大學
HUNAN UNIVERSITY

第二章 高级数据结构

—— 湖南大学信息科学与工程学院 ——



湖南大学
HUNAN UNIVERSITY

目录

- 第一节 堆**

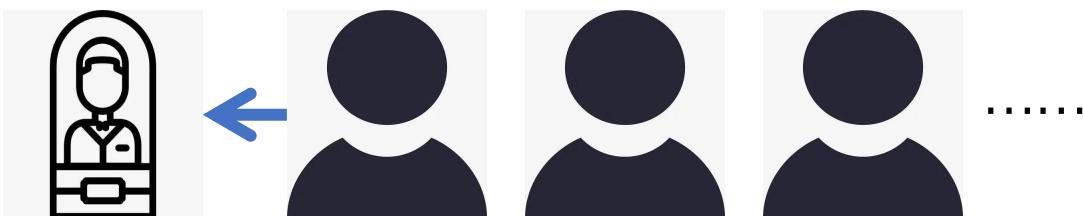
- 第二节 散列表**



队列



队列：满足先进先出（FIFO）的数据结构，数据从队列头部取出，新的数据从队列尾部插入，数据之间是平等的。类似于普通人到火车站排队买票。

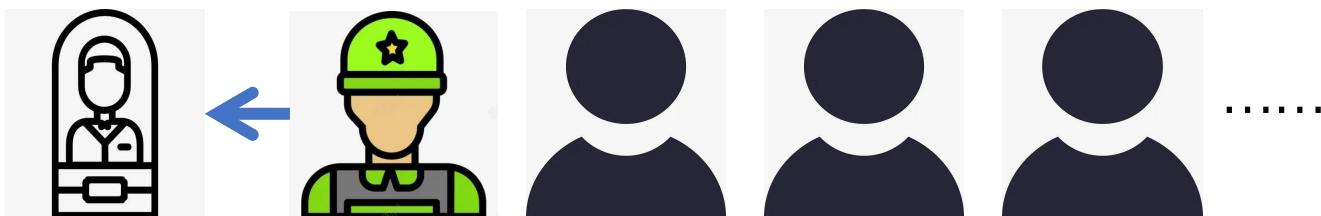




优先队列



优先队列：从头部取出数据，从尾部插入数据，但是这个过程根据数据的优先级而变化的，总是优先级高的先出来。类似于军人依法优先。





堆的作用

堆(Heap)是一类特殊的数据结构，是高效的优先队列实现方式

堆的应用场景：

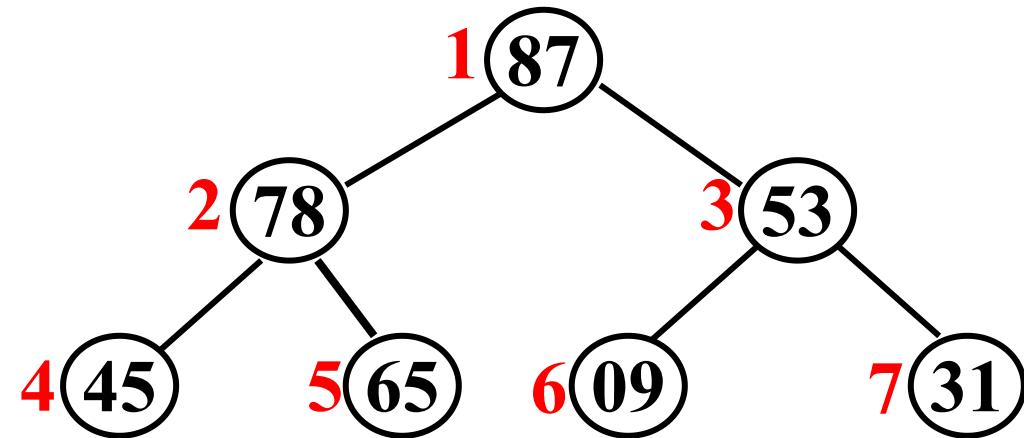
- 优先级队列的使用场景
- 求Top K值问题
- 求中位数、百分位数
- 大数据量日志统计搜索排行榜

课程目标：堆的定义



堆的定义

满二叉树：一棵深度为k且有 2^k-1 个结点的二叉树



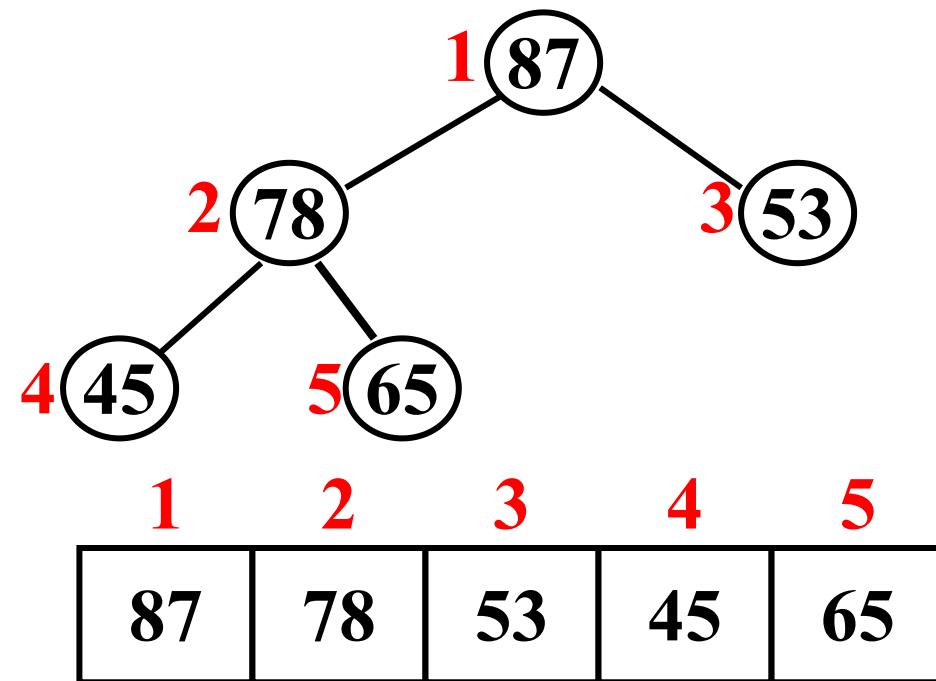
如果对满二叉树的结点进行编号，约定编号从根结点起，自上而下，自左而右，则可以形成一个长度为 2^k-1 的序列

1	2	3	4	5	6	7
87	78	53	45	65	09	31



堆的定义

完全二叉树：一棵深度为k的有n个结点的二叉树，对树中的结点按从上至下、从左到右的顺序进行编号，如果编号为i ($1 \leq i \leq n$) 的结点与满二叉树中编号为i的结点在二叉树中的位置相同，则这棵二叉树称为完全二叉树。





堆的定义

堆实质上是满足如下性质的完全二叉树：树中任一非叶结点的关键字均不大于(或不小于)其左右孩子(若存在)结点的关键字

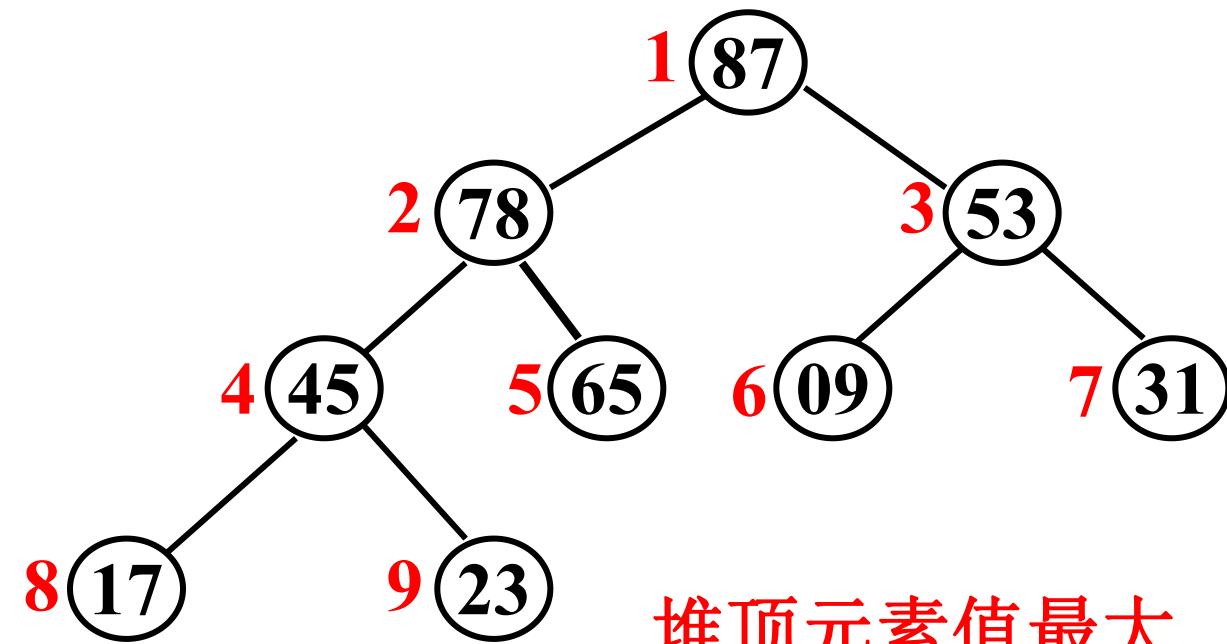
对应在序列上， n 个元素的序列 $\{k_1, k_2, \dots, k_n\}$ 当且仅当满足如下关系时，称之为**堆 (heap)**。若满足条件 $k_i \leq k_{i/2}$ 且则称**最大堆**；若满足条件 $k_i \geq k_{i/2}$ 则称**最小堆**



堆的实例

最大堆

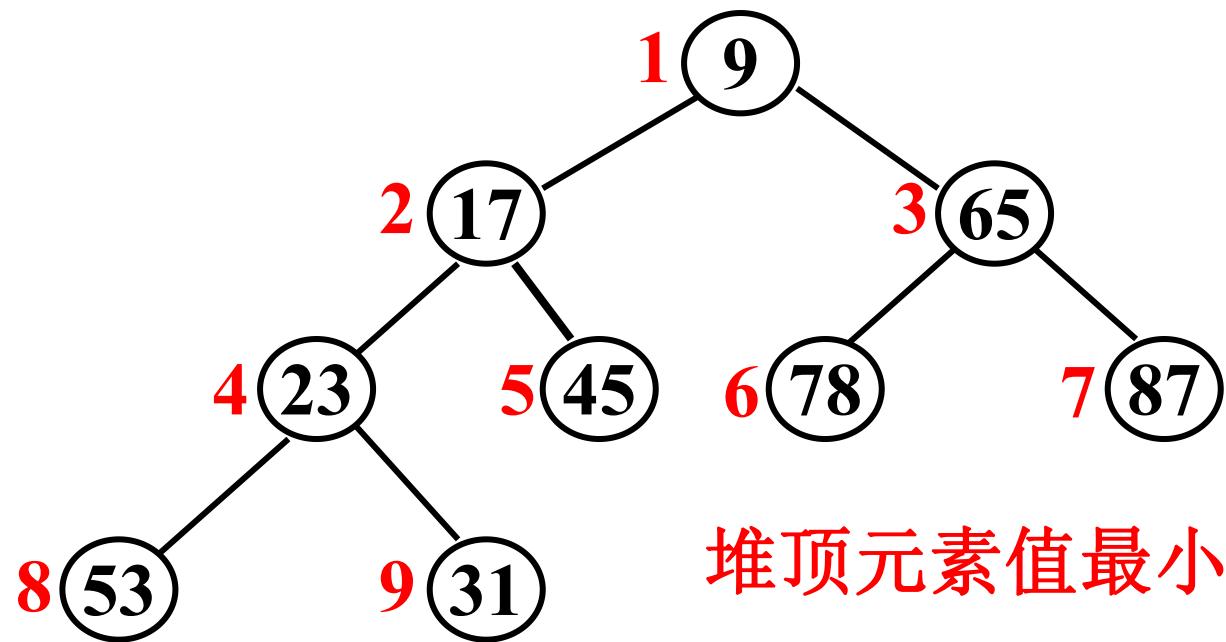
1	2	3	4	5	6	7	8	9
87	78	53	45	65	09	31	17	23





堆的实例

	1	2	3	4	5	6	7	8	9
最小堆	9	17	65	23	45	78	87	53	31



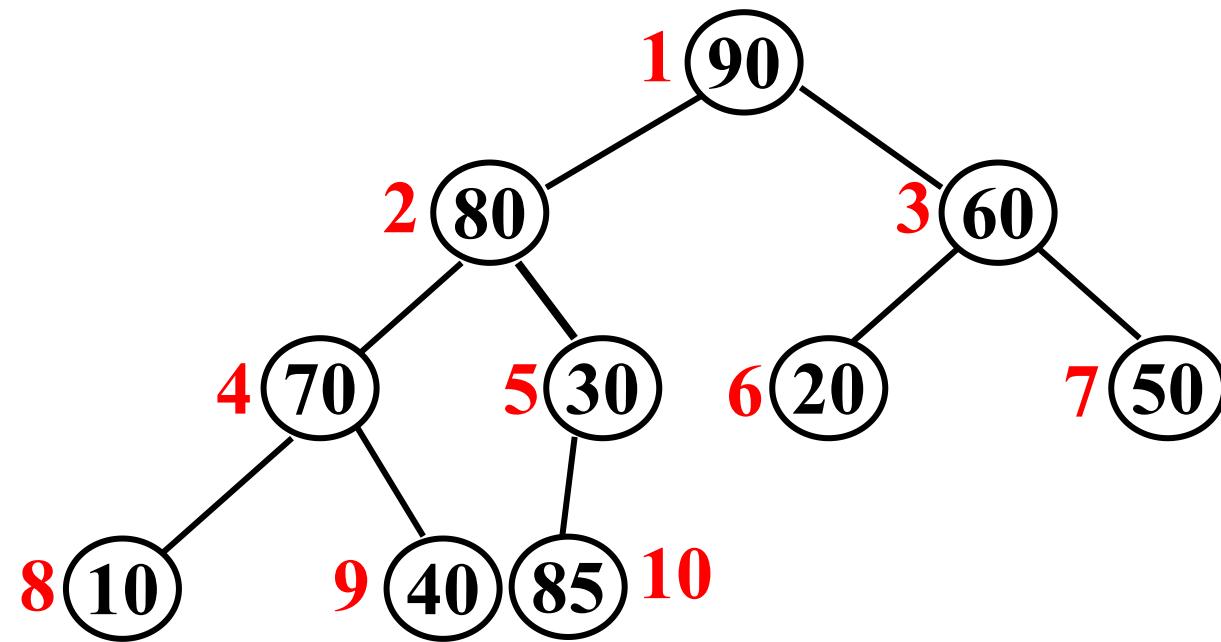
将堆用顺序存储结构来
存储，则堆对应一组序
列



堆的实例

插入新一个元素85之后呢？

90	80	60	70	30	20	50	10	40	85			
----	----	----	----	----	----	----	----	----	----	--	--	--





堆的实例

如何调整成一个堆？

90	80	60	70	30	20	50	10	40	85			
----	----	----	----	----	----	----	----	----	----	--	--	--

90	80	60	70	85	20	50	10	40	30			
----	----	----	----	----	----	----	----	----	----	--	--	--

90	85	60	70	80	20	50	10	40	30			
----	----	----	----	----	----	----	----	----	----	--	--	--



堆的概述

1、优先级队列的使用场景

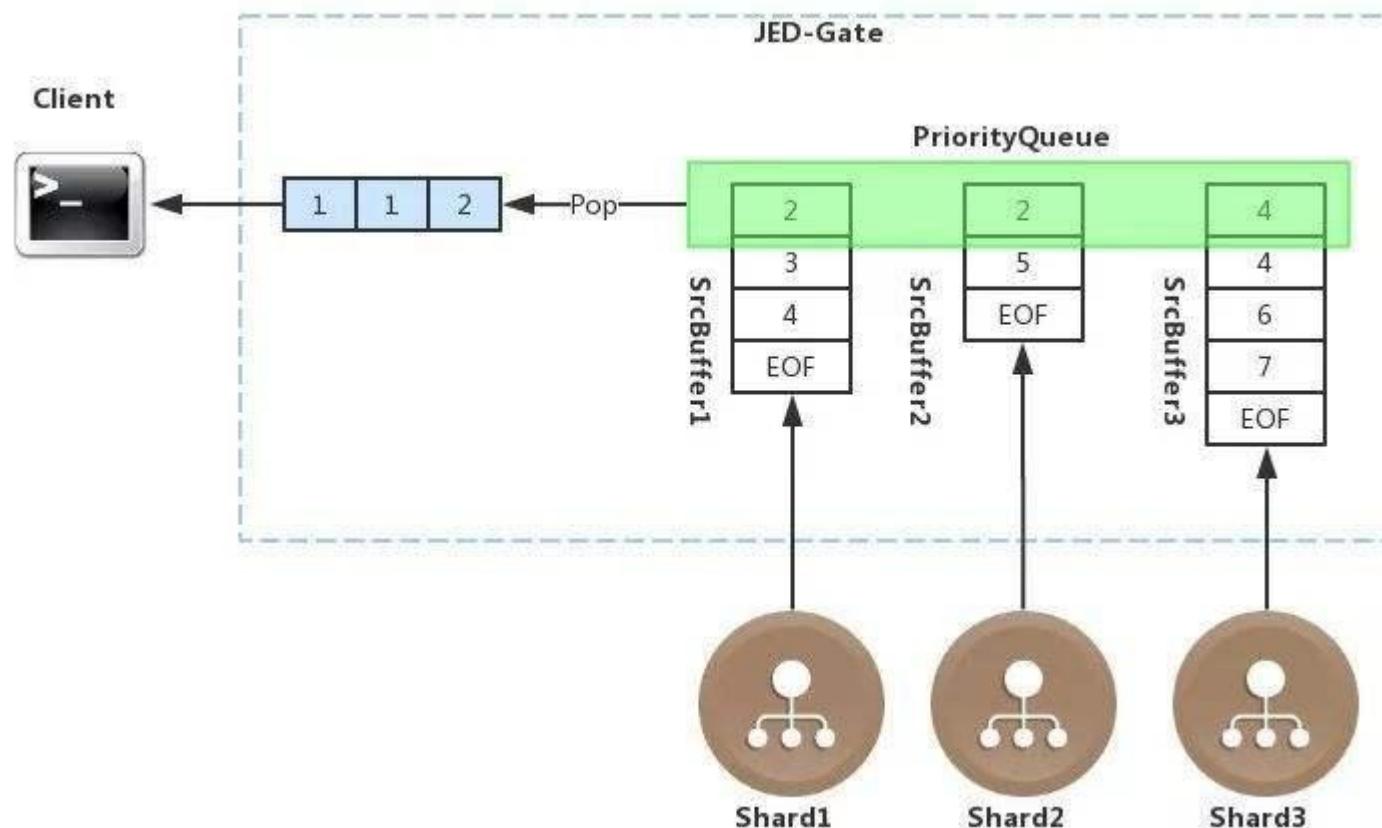
- 1)、定时任务轮训问题
- 2)、合并有序小文件

2、求Top K值问题

- 3、求中位数、百分位数
- 4、大数据量日志统计搜索排行榜



优先队列应用



优先级队列有两种：最大优先级队列和最小优先级队列



优先队列应用-最大优先级队列操作实现

优先队列是一种用来维护由一组元素构成的集合S的数据结构，其中每一个元素都有一个关键字(key)，关键字赋予了一个元素的优先级，故名为优先队列。

优先队列应用-最大优先级队列操作实现



优先队列的操作：

最大：

- 1) Insert(S, x): 插入 x 到集合 S 中;
- 2) Maximum(S): 返回 S 中具有最大关键字的元素;
- 3) Extract_Max(S): 去掉并返回集合 S 中具有最大关键字的元素;
- 4) Increase_Key(S, x, key): 将元素 x 的关键字增加到 key 。

优先队列应用-最大优先级队列操作实现



优先队列的操作：

最小

- 1) Insert(S, x): 插入 x 到集合 S 中;
- 2) Minimum(S): 返回 S 中具有最小关键字的元素;
- 3) Extract_Min(S): 去掉并返回集合 S 中具有最小关键字的元素;
- 4) Increase_Key(S, x, key): 将元素 x 的关键字增加到 key 。



优先队列应用-最大优先级队列操作实现

堆实现优先队列：

- (1) HEAP_MAXIMUM返回最大堆第一个元素的值（堆的调整）
- (2) HEAP_EXTRACT_MAX实现EXTRACT_MAX操作，删除最大堆中第一个元素，然后调整堆。（堆的调整）
- (3) MAX_HEAP_INSERT实现INSERT操作，向最大堆中插入新的关键字。（堆的调整）
- (4) HEAP_INCREASE_KEY实现INCREASE_KEY，通过下标来标识要增加的元素的优先级key，增加元素后需要调整堆，从该节点的父节点开始自顶向上调整。（维护堆的性质）



堆的概述

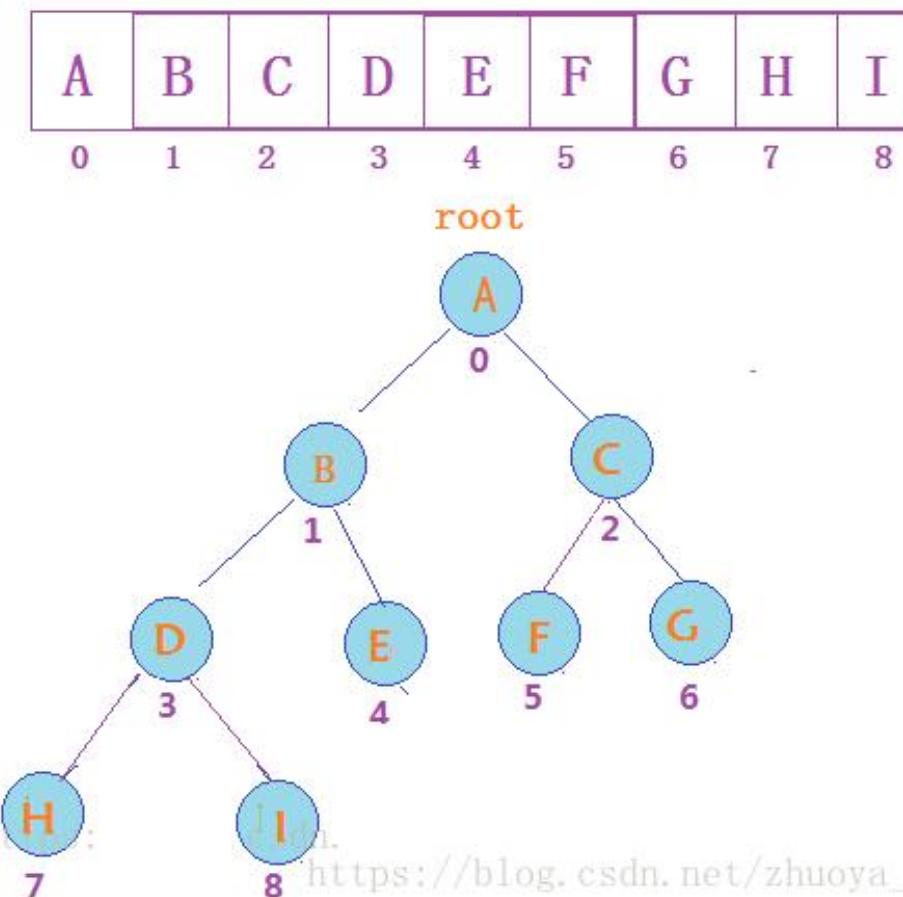
完全二叉树的存储方式（逻辑表达）

- ✓ 顺序存储（数组）（物理实现）
- ✓ 链式存储（链表）（物理实现）



堆的概述

完全二叉树的存储方式：
顺序存储

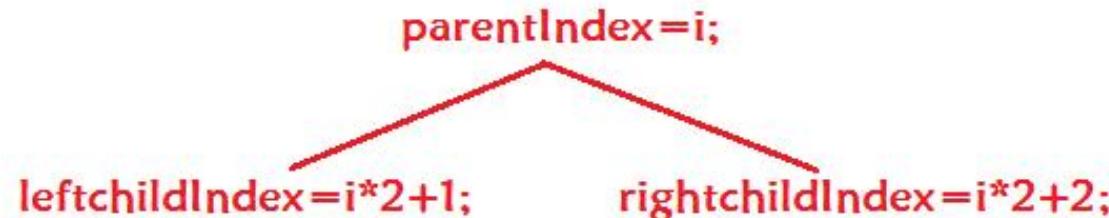




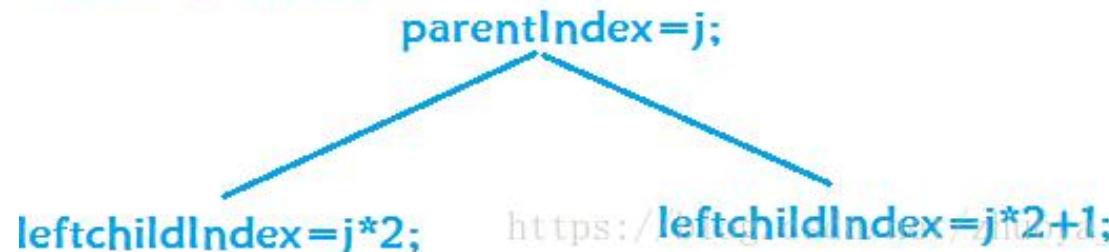
堆的概述

完全二叉树的存储 方式：顺序存储

从数组的0号下标开始存储时
逻辑意义上根结点和左、
右孩子的索引关系：



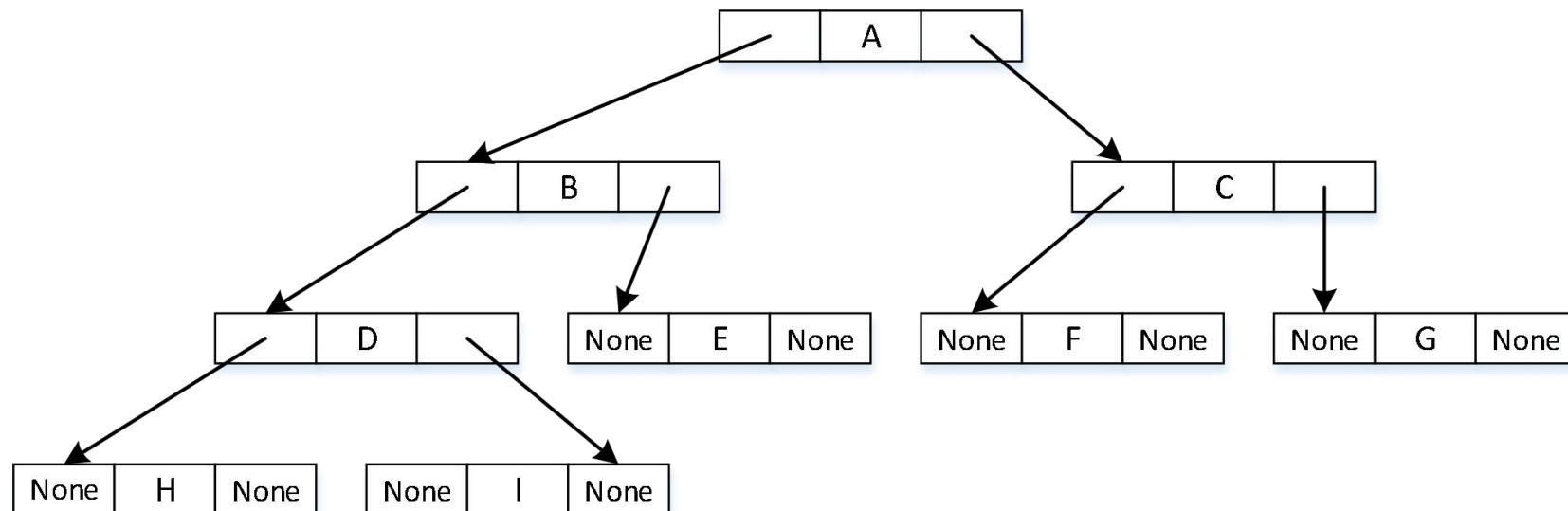
从数组的1号下标开始存储时
逻辑意义上根结点和左、
右孩子的索引关系：





堆的概述

完全二叉树的存储方式：链式存储





堆的调整

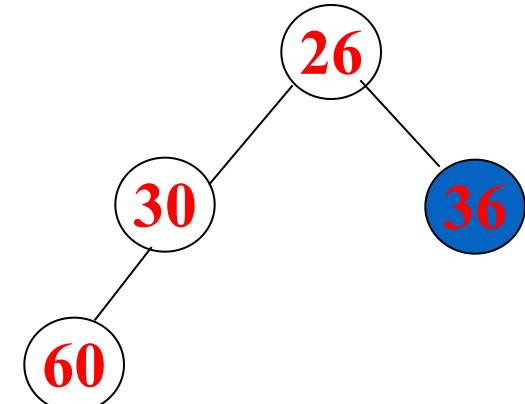
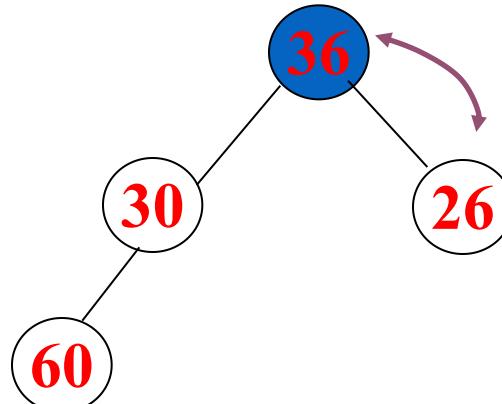
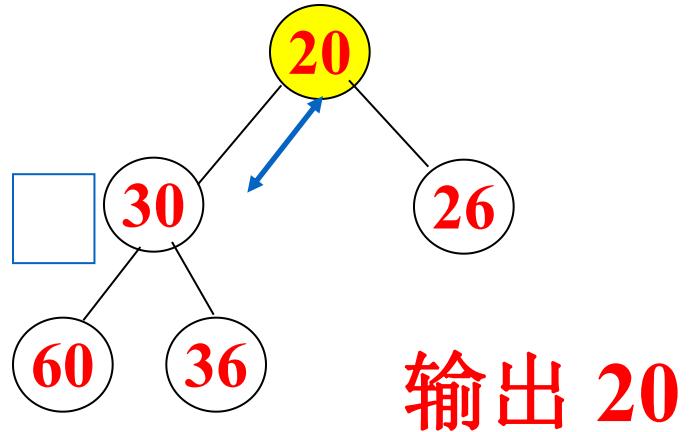
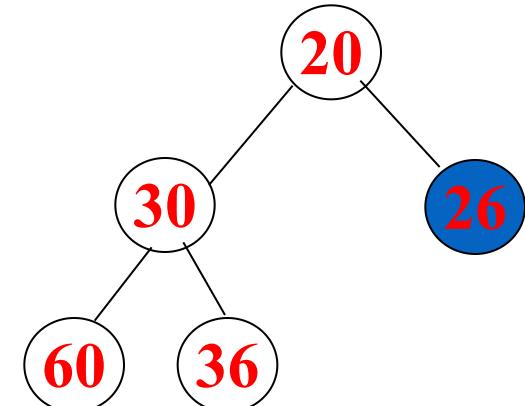
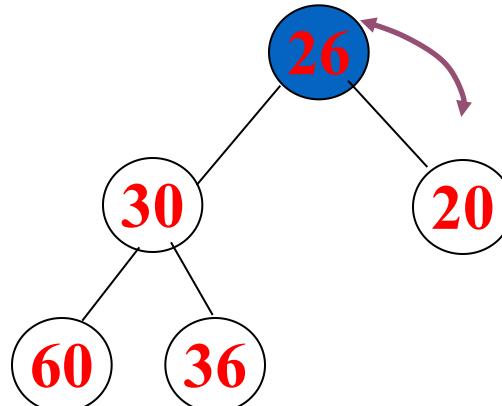
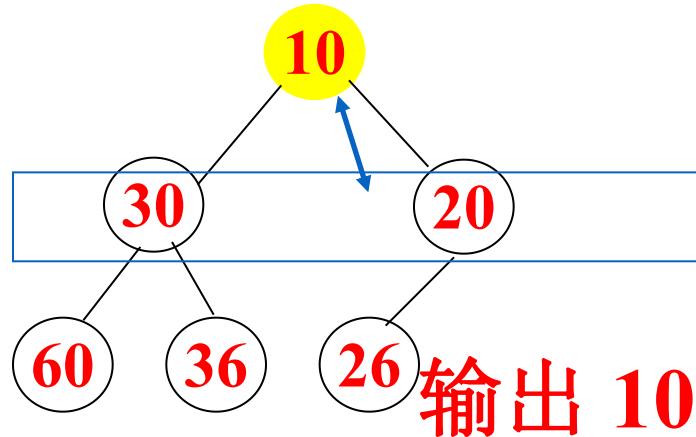
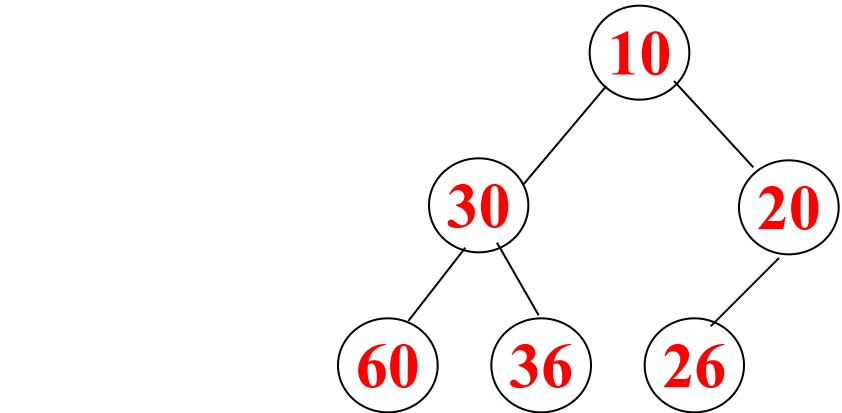
如何在输出堆顶元素后，调整剩余元素为一个新的堆？

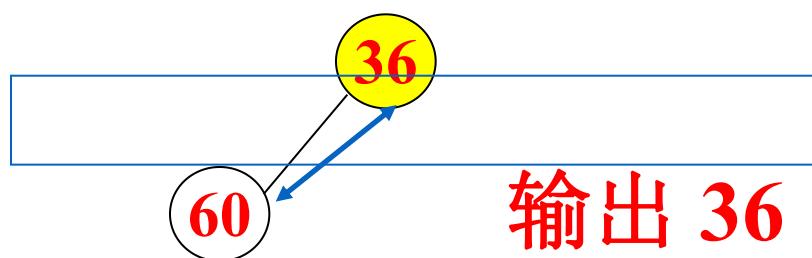
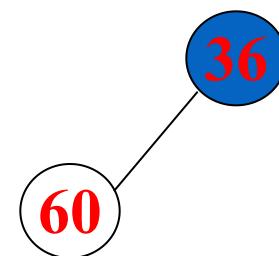
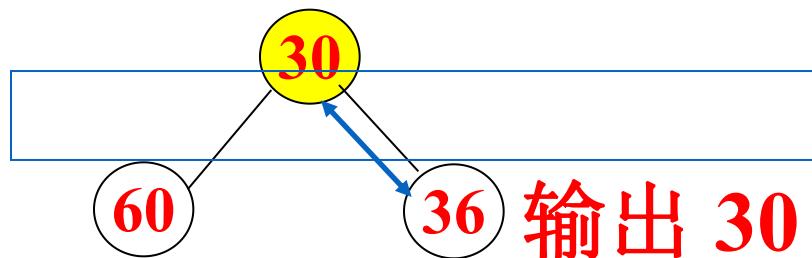
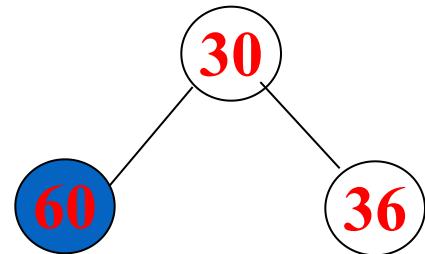
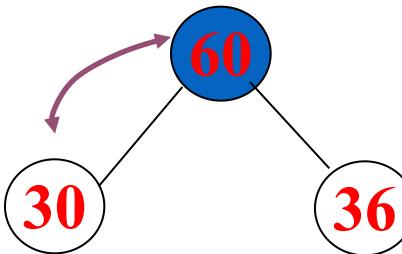
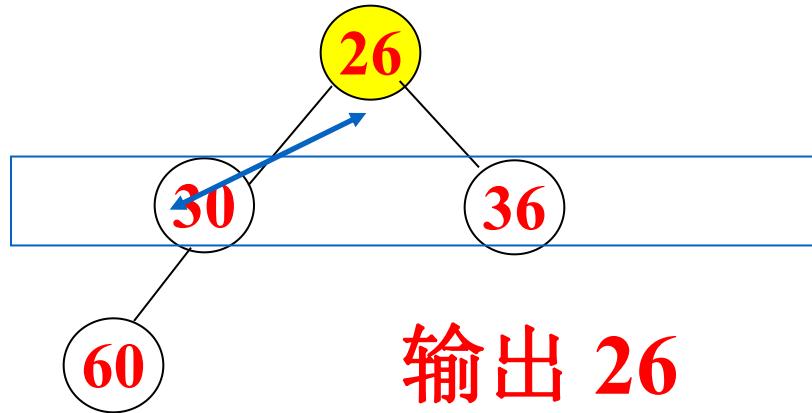
解决方法：

输出堆顶元素之后，以堆中最后一个元素替代之；然后将根结点值与左、右子树的根结点值进行比较，并与其中小者进行交换；重复上述操作，直至叶子结点，将得到新的堆，称这个从堆顶至叶子的调整过程为“筛选”。

堆的调整--动画演示

最小堆输出序列





递增输出序列为: 10 20 26 30 36 60



堆的调整

如何在堆中插入一个元素？

解决方法：

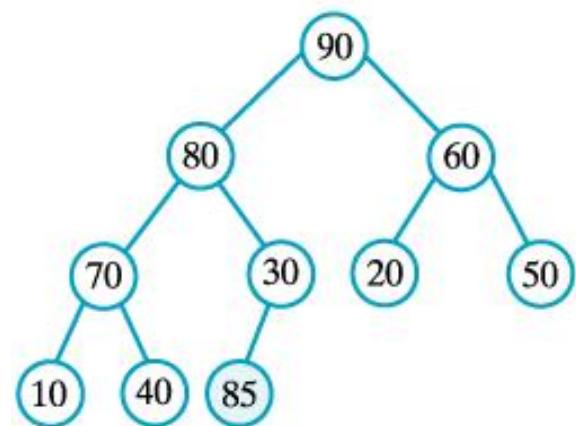
在堆的最后增加一个元素，然后新添加的元素不能比它的父元素大，如果比它的父元素大，就涉及到上浮的动作



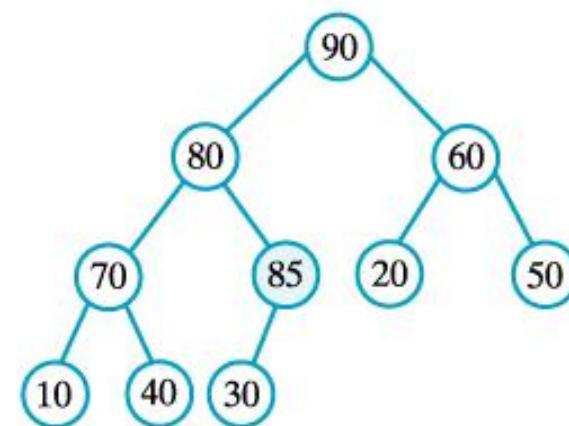
堆的调整

最大堆中加入元素85

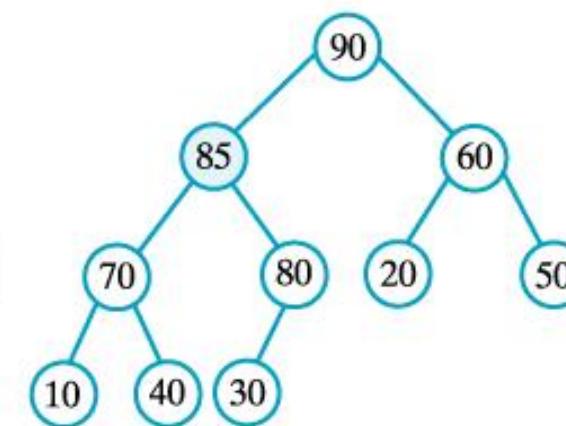
(a)



(b)



(c)





堆的调整

最大堆中加入元素85

(a)

	90	80	60	70	30	20	50	10	40			
0	1	2	3	4	5	6	7	8	9	10	11	12

$(10/2)$

$85 > 30$

(b)

	90	80	60	70		20	50	10	40	30		
0	1	2	3	4	5	6	7	8	9	10	11	12

Move 30



(c)

	90	80	60	70		20	50	10	40	30		
0	1	2	3	4	5	6	7	8	9	10	11	12

$(5/2)$

$85 > 80$



堆的调整

最大堆中加入元素85

(d)

	90		60	70	80	20	50	10	40	30		
0	1	2	3	4	5	6	7	8	9	10	11	12

Move 80

(e)

	90		60	70	80	20	50	10	40	30		
0	1 (2/2)	2	3	4	5	6	7	8	9	10	11	12

$85 < 90$

(f)

	90	85	60	70	80	20	50	10	40	30		
0	1	2	3	4	5	6	7	8	9	10	11	12

Insert 85



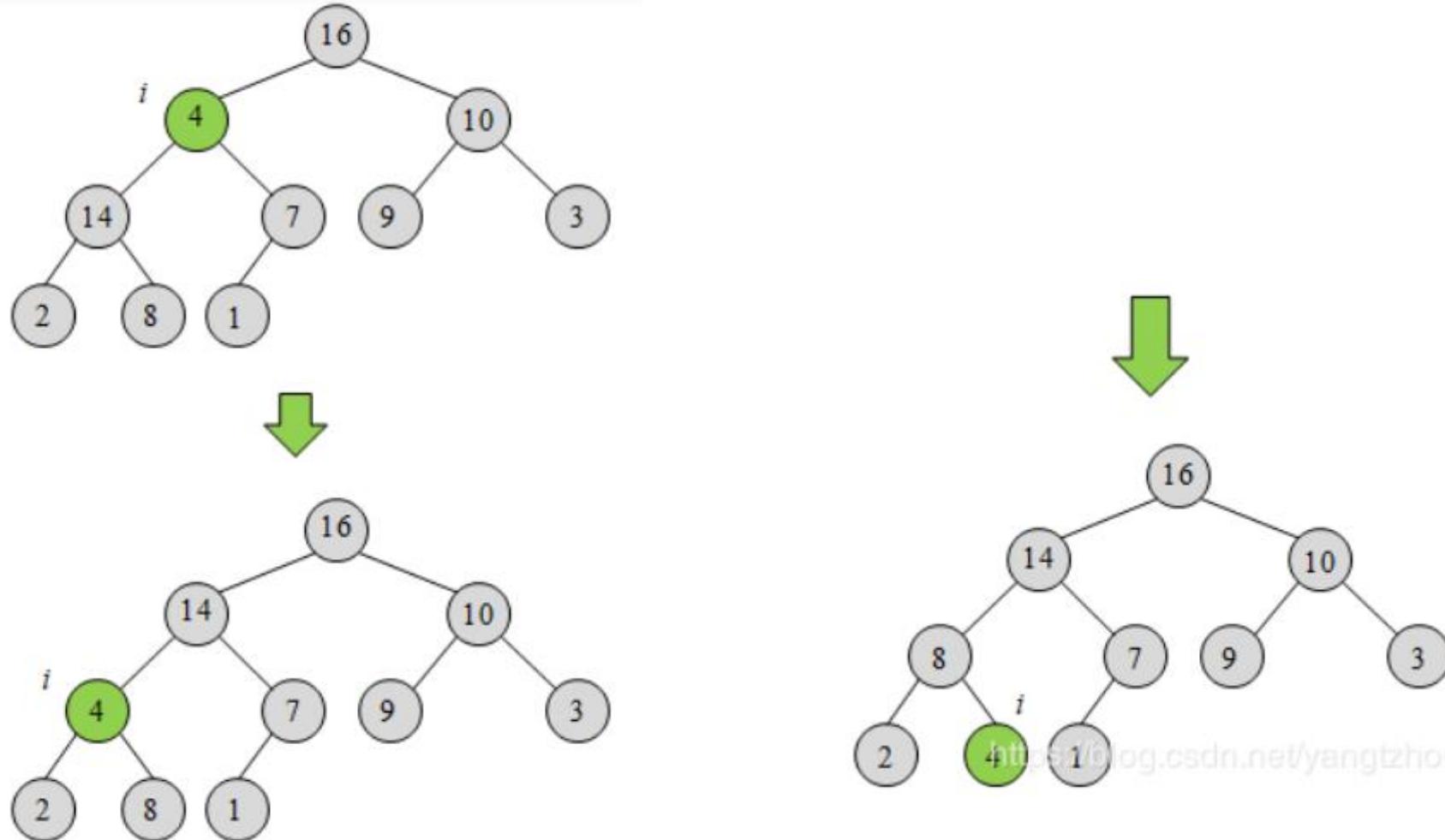
堆的调整

加入元素时向上调整

删除元素时向下调整



维护堆的性质





维护堆的性质

代码 6.2-1：维护最大堆性质

//参数A：一个最大堆的数组

//参数i：堆中的一个元素

```
1 MAX-HEAPFY(A,i)
2   l = LEFT (i)
3   r = RIGHT (i)
4   if l ≤ A.heap_size and A[l] > A[i]
5     largest = l
6   else
7     largest = i
8   if r ≤ A.heap_size and A[r] > A[largest]
9     largest = r
10  if largest ≠ i
11    exchange A[i] with A[largest]
12    MAX-HEAPFY(A,largest)
```



建堆

建堆方法：

- ✓ 自底向上
- ✓ 插入建堆



建堆

建堆方法1：自底向上（迭代实现）

代码 6.3-1：构建最大堆
//参数A：一个用来构建最大堆的数组

```
1 BUILD-MAX-HEAP(A,i)
2     A.heap_size = A.length
3     r = RIGHT (i)
4     for i = ⌊A.length/2⌋ down to 1
5         MAX-HEAPFY(A,i)
```

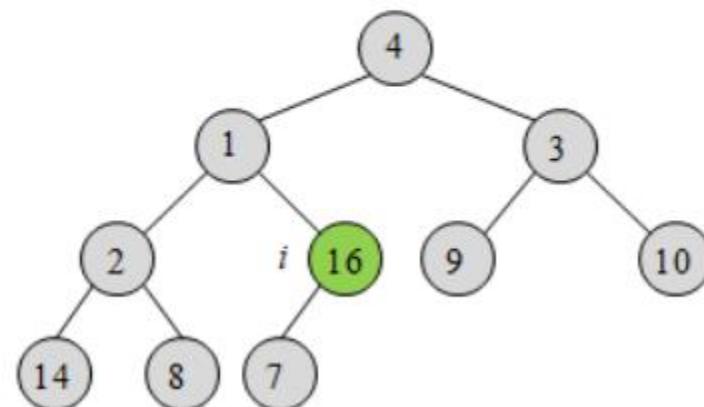


建堆

建堆方法1：自底向上（迭代实现）

原始数组 A

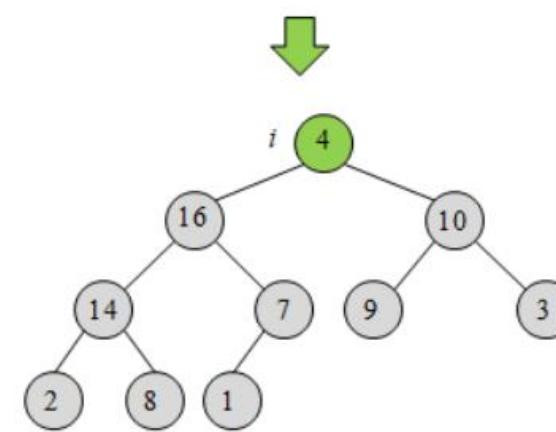
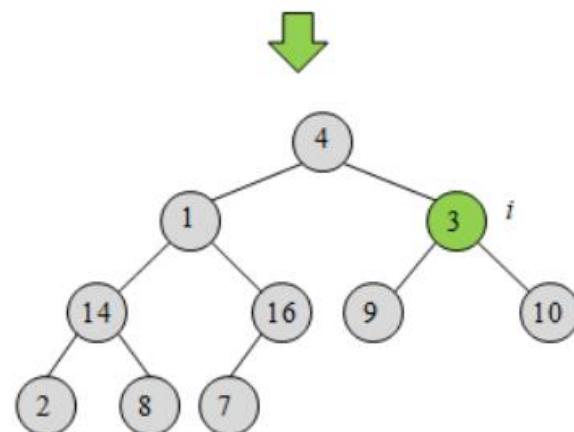
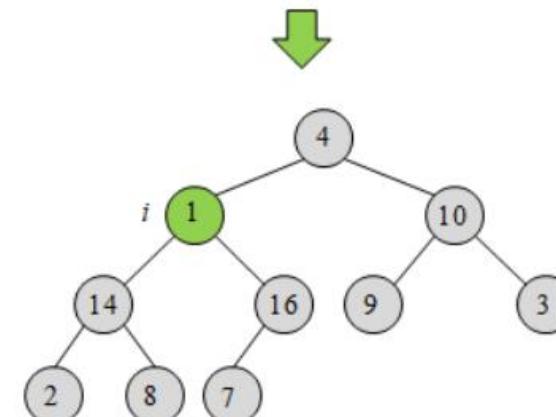
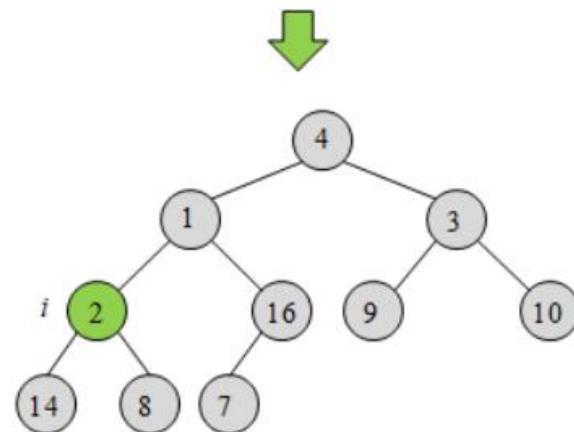
4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---





建堆

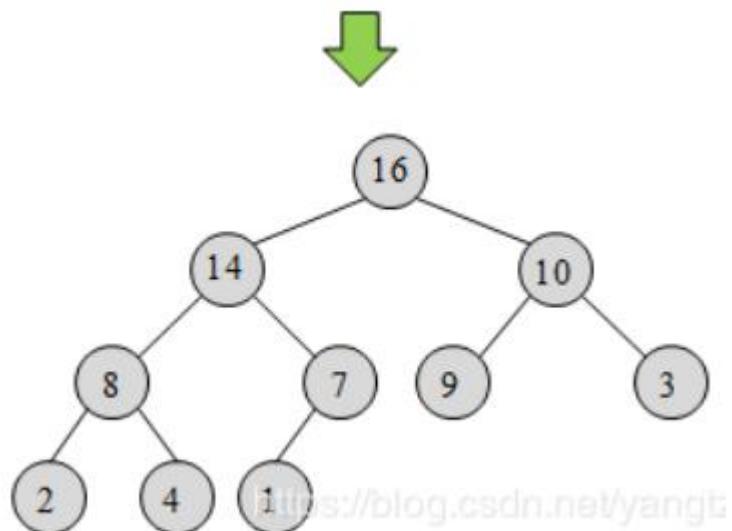
建堆方法1：自底向上（迭代实现）





建堆

建堆方法1：自底向上（迭代实现）



<https://blog.csdn.net/yangt>



建堆

建堆方法1：自底向上（迭代实现）

```
def max_heap(to_adjust_list, heap_size, index):
    "调整列表中的元素以保证以index为根的堆是一个最大堆"

    # 将当前结点与其左右子节点比较，将较大的结点与当前结点交换，然后递归地调整子树
    left_child = 2 * index + 1
    right_child = left_child + 1
    if left_child < heap_size and to_adjust_list[left_child] > to_adjust_list[index]:
        largest = left_child
    else:
        largest = index
    if right_child < heap_size and to_adjust_list[right_child] > to_adjust_list[largest]:
        largest = right_child
    if largest != index:
        to_adjust_list[index], to_adjust_list[largest] = \
            to_adjust_list[largest], to_adjust_list[index]
        max_heap(to_adjust_list, heap_size, largest)
```



建堆

建堆方法1：自底向上（迭代实现）

```
def build_max_heap(to_build_list):
    """建立一个堆"""

    # 自底向上建堆
    for i in range(len(to_build_list)//2 - 1, -1, -1):
        max_heap(to_build_list, len(to_build_list), i)

if __name__ == '__main__':
    to_sort_list = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]
    build_max_heap(to_sort_list)
    print(to_sort_list)
```



建堆

建堆方法2：插入建堆

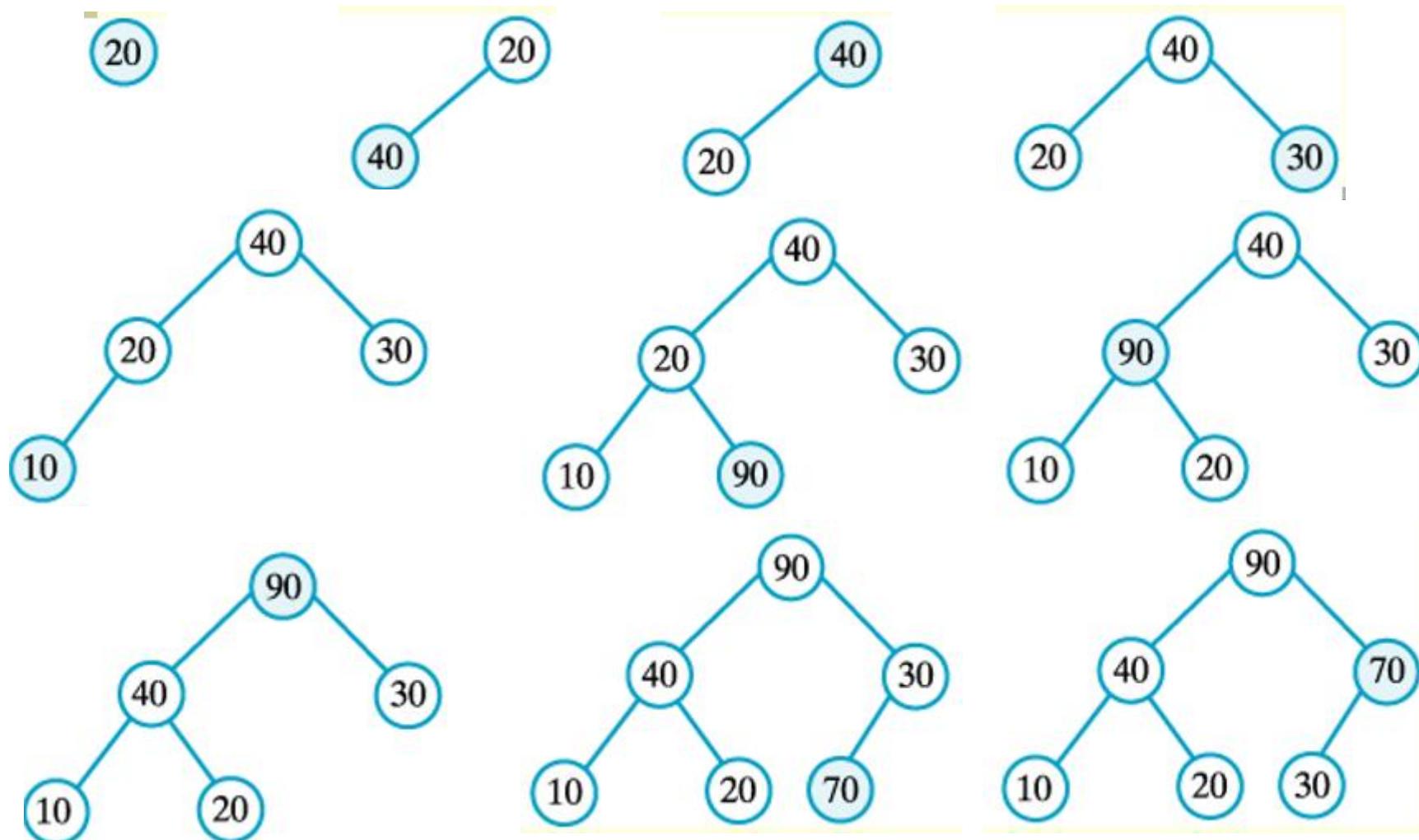
把一个数组看成两部分，左边是堆，右边是还没加入堆的元素，步骤如下：

- 1、数组里的第一个元素自然地是一个堆
- 2、然后从第二个元素开始，一个个地加入左边的堆，当然，每加入一个元素就破坏了左边元素堆的性质，得重新把它调整为堆。



建堆

建堆方法2：
插入建堆





堆排序算法

若在输出堆顶的最小值（最大值）后，使得剩余 $n-1$ 个元素的序列重又建成一个堆，则得到 n 个元素的次小值（次大值）……如此反复，便能得到一个有序序列，这个过程称之为**堆排序算法**。



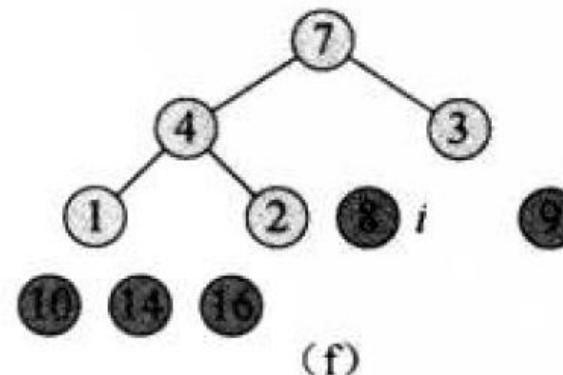
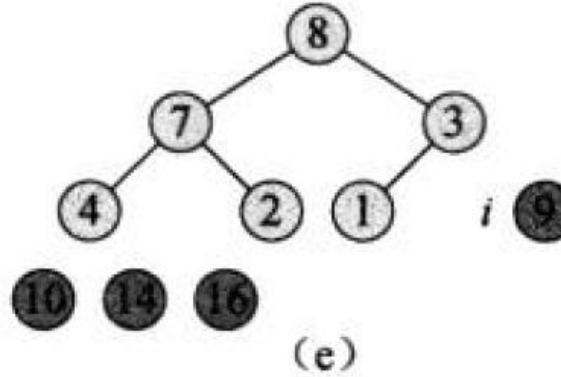
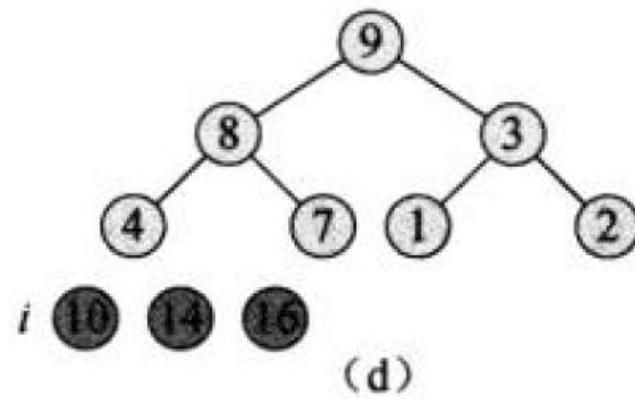
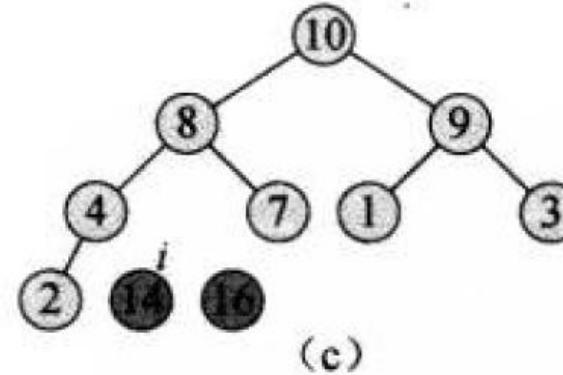
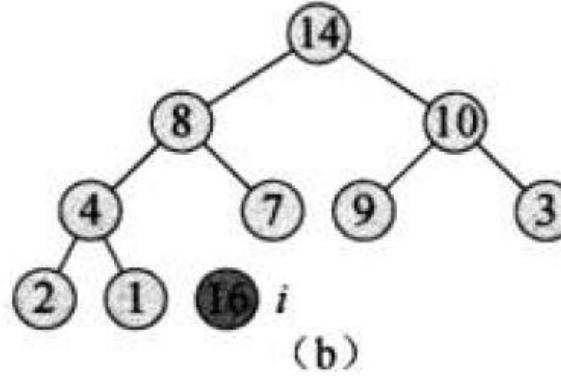
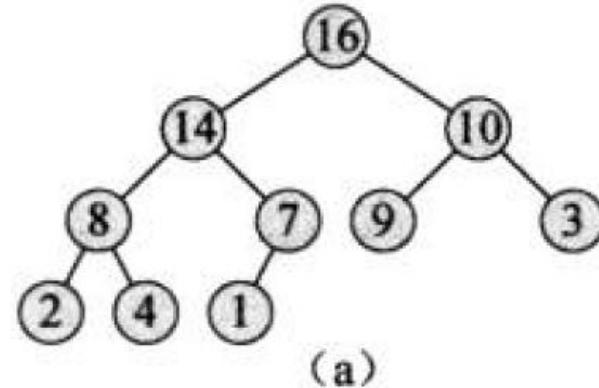
堆排序算法

HEAPSORT(A)

- 1 BUILD-MAX-HEAP(A)
- 2 for $i = \lfloor A.length/2 \rfloor$ downto 1
- 3 exchange $A[1]$ with $A[i]$
- 4 $A.heap_size = A.heap-size - 1$
- 5 MAX-HEAPIFY(A,1)



堆排序算法





堆排序算法

堆排序的时间复杂度

堆排序=堆调整 + 堆排序

堆排序时间复杂度=堆调整时间复杂度 + 堆排序时间复杂度

堆调整的时间复杂度为 $O(n)$

排序重建堆的时间复杂度为 $n \lg(n)$ （递归）

总的时间复杂度为 $O(n+n \lg n)=O(n \lg n)$



优先队列-最大优先级队列操作实现

优先队列的操作：

最大：

- 1) Insert(S, x):插入 x 到集合 S 中;
- 2) Maximum(S):返回 S 中具有最大关键字的元素;
- 3) Extract_Max(S):去掉并返回集合 S 中具有最大关键字的元素;
- 4) Increase_Key(S, x, key):将元素 x 的关键字增加到 key 。



优先队列-最大优先级队列操作实现

优先队列的操作：

最小

- 1) Insert(S, x):插入 x 到集合 S 中;
- 2) Minimum(S):返回 S 中具有最小关键字的元素;
- 3) Extract_Min(S):去掉并返回集合 S 中具有最小关键字的元素;
- 4) Increase_Key(S, x, key):将元素 x 的关键字增加到 key 。



优先队列-最大优先级队列操作实现

堆实现优先队列：

- (1) HEAP_MAXIMUM返回最大堆第一个元素的值（堆的调整）
- (2) HEAP_EXTRACT_MAX实现EXTRACT_MAX操作，删除最大堆中第一个元素，然后调整堆。（堆的调整）
- (3) MAX_HEAP_INSERT实现INSERT操作，向最大堆中插入新的关键字。（堆的调整）
- (4) HEAP_INCREASE_KEY实现INCREASE_KEY，通过下标来标识要增加的元素的优先级key，增加元素后需要调整堆，从该节点的父节点开始自顶向上调整。（维护堆的性质）

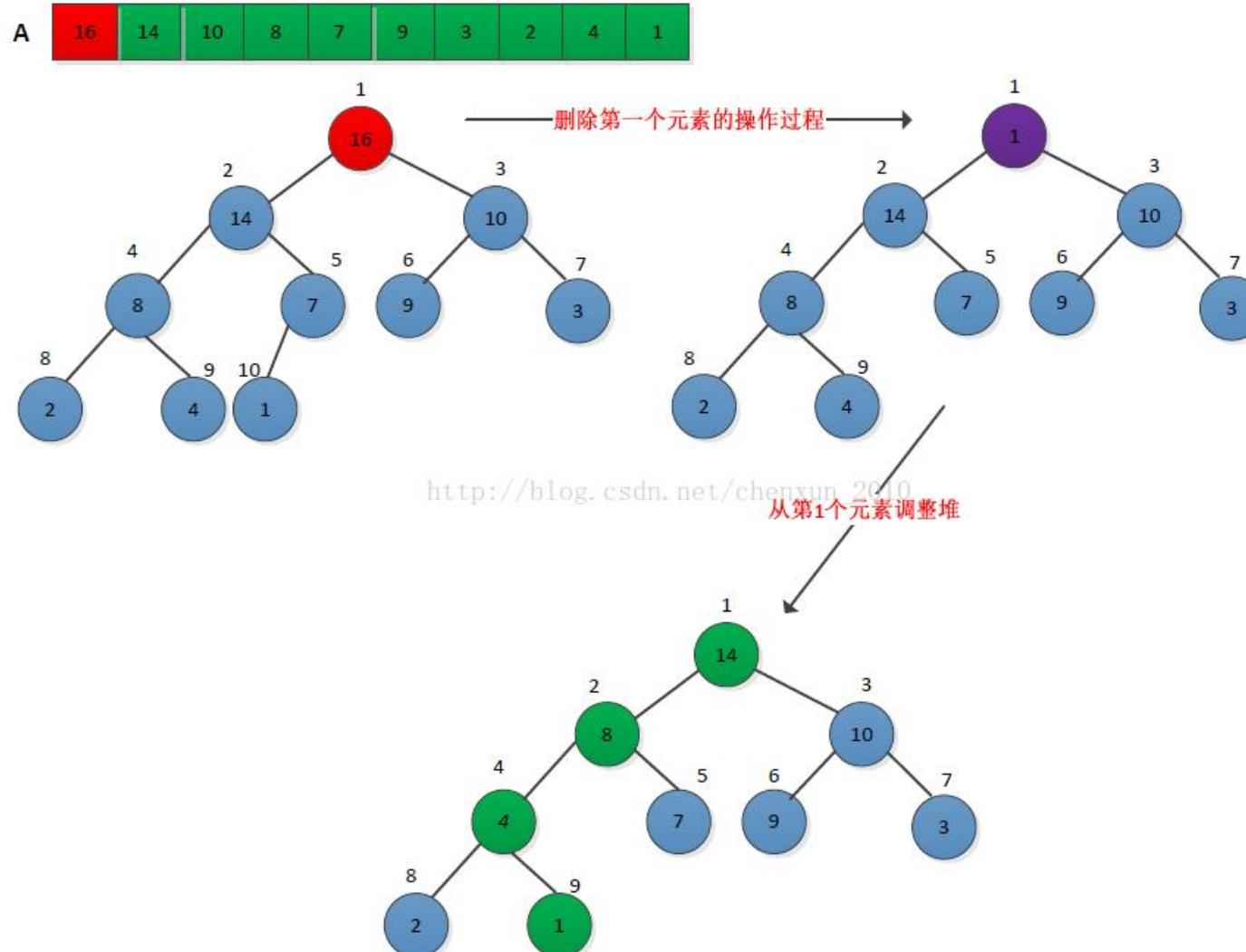
优先队列-返回第一个元素



```
1 MAX-HEAP-MAXIMUM(A)
2   if A.heap_size < 1
3     error "heap underflow"
4   else
5     return A[1]
```



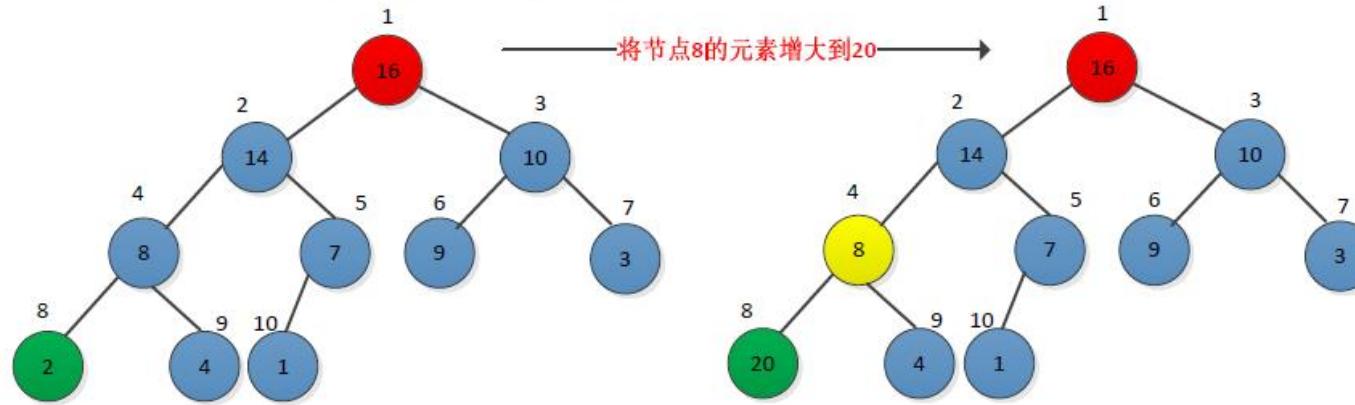
优先队列-删除操作





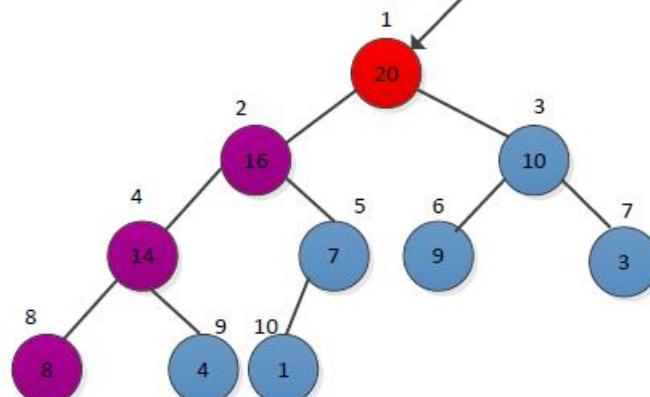
优先队列-元素值增加操作

A	16	14	10	8	7	9	3	2	4	1
---	----	----	----	---	---	---	---	---	---	---



http://blog.csdn.net/chenzun_2010

从节点8的父节点开始调整堆





优先队列-插入元素操作

- 1 MAX-HEAP-INSERT(A,key)
- 2 if A.heap_size = A.length
- 3 error “heap is full”
- 4 A.heap_size = A.heap_size + 1
- 5 A[A.heap_size] = $-\infty$
- 6 MAX-HEAP-INCREASE-KEY(A,A.heap_size,key)



目录

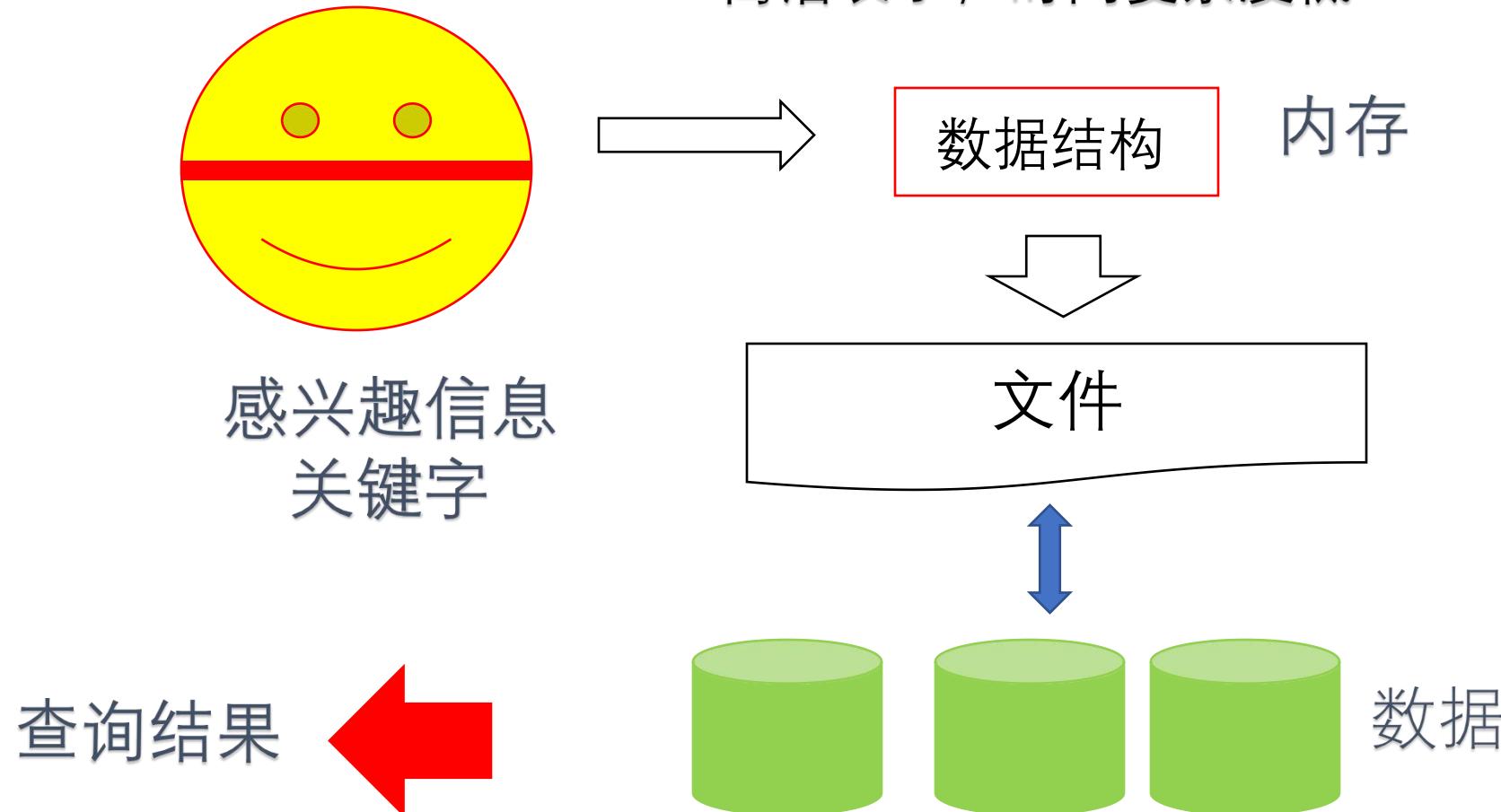
- 第一节 堆**

- 第二节 散列表**



概述

简洁表示，时间复杂度低



快速查找问题



手机上都有基本的通讯功能，有通讯功能必然存在通讯录

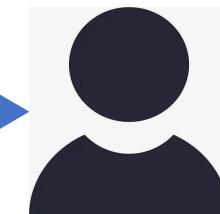
通讯录功能： 输入姓名能映射出电话号码，输入电话号码能映射出姓名



1234 5678



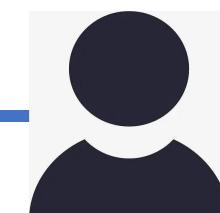
张三



8765 4321



李四





哈希表的作用

哈希表是根据关键码值而直接进行访问的数据结构。它通过把关键码值映射到表中一个位置来访问记录，存放记录的数组叫做哈希表。这个映射函数叫做**哈希函数**。

哈希表的应用场景：

- 网页快速查找信息
- DNA匹配：字符串之间的匹配

课程目标： 哈希表定义以及常用哈希函数的定义和计算，如除法哈希

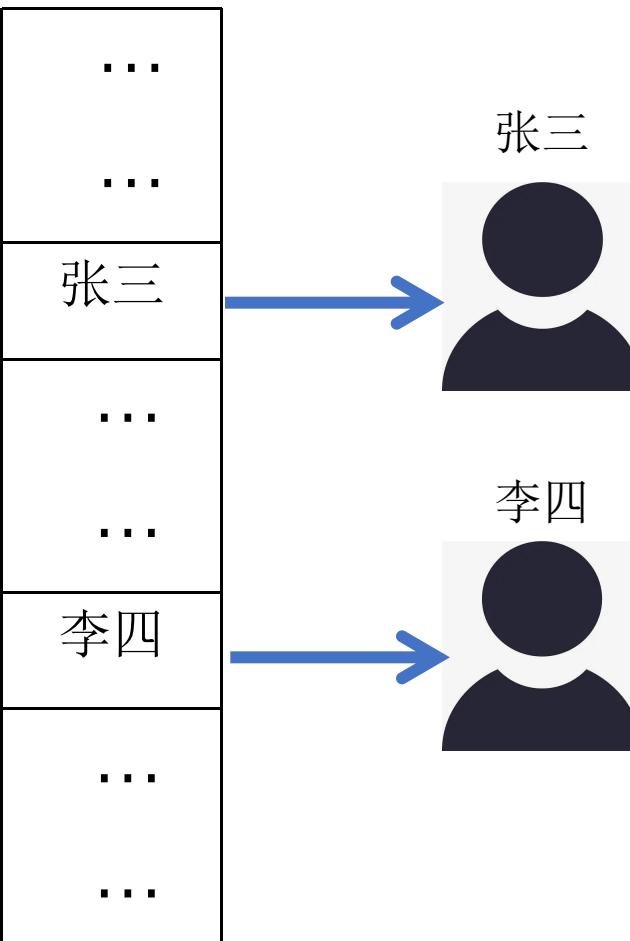


直接寻址

直接寻址：取关键字值为哈希表地址

电 话

1234 5678



8765 4321





直接寻址优缺点

优点：查找速度快

缺点：对于全域较大，但是元素却十分稀疏的情况，使用这种存储方式将浪费大量的存储空间。



哈希函数

为了克服直接寻址技术的缺点，而又保持其快速字典操作的优势，我们可以利用哈希函数 $h: U \rightarrow \{0, 1, 2, \dots, m-1\}$ 来计算关键字 k 所在的位置

哈希函数 $h(k)$ 的作用是将范围较大的关键字映射到一个范围较小的集合中。形式化来说，设关键字集 K 中有 n 个关键字，哈希表长为 m ，哈希函数 h 满足以下要求：

1. 对任意 $k \in K$, 有 $0 \leq h(k) \leq m-1$;
2. 对任意 $k \in K$, $h(k)$ 取 $[0, m-1]$ 中任一值的概率相等。



除法哈希函数

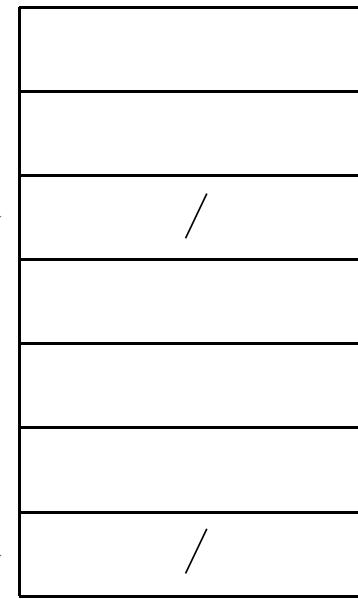
假设m是哈希表大小，通过取关键字值k除以m的余数，将关键字值k映射到哈希表中的一个位置

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

假设m=7

电 话

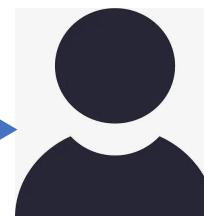
1234 5678



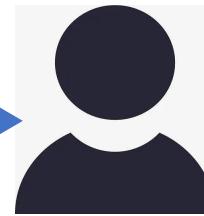
8765 4321



张三



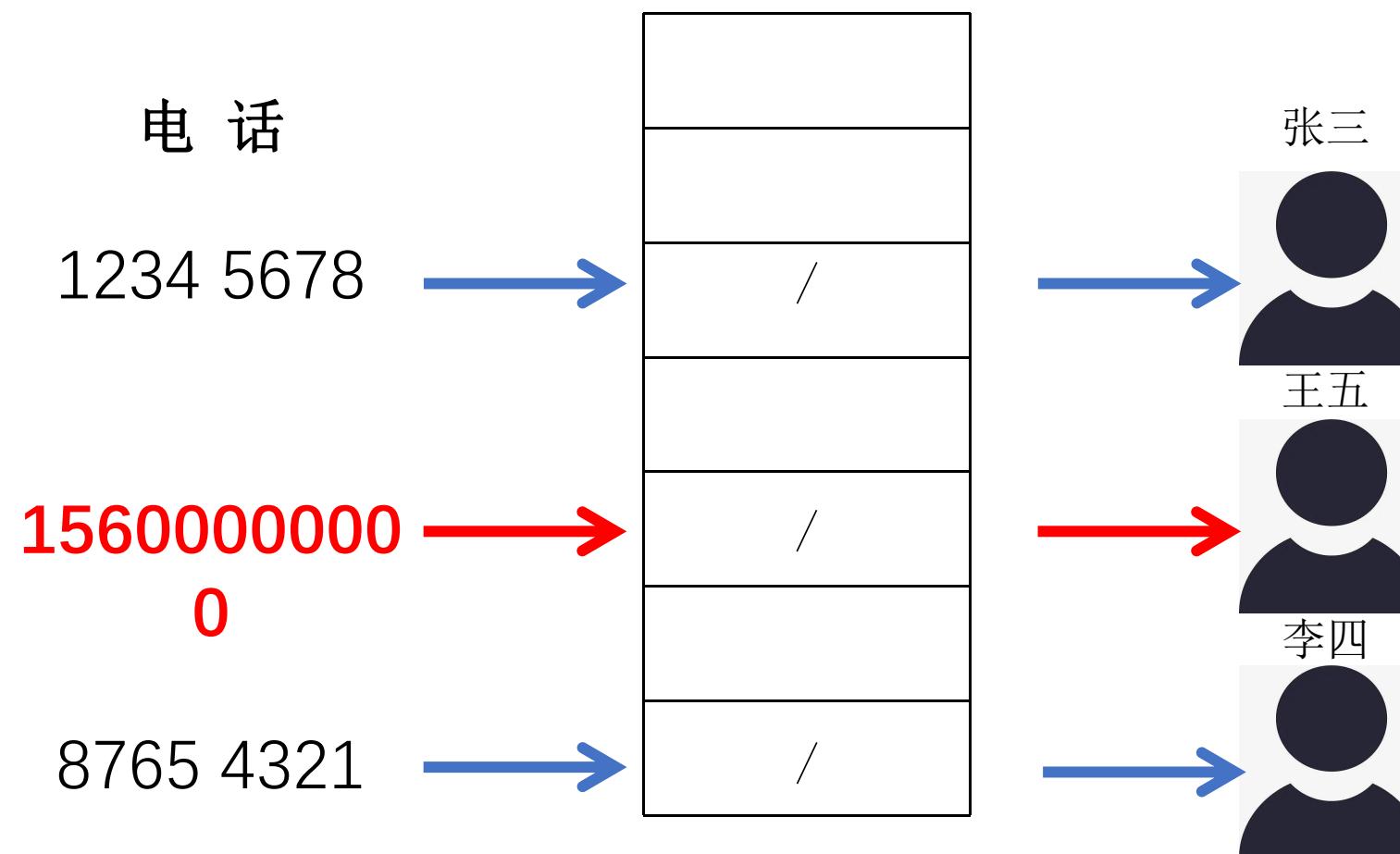
李四





除法哈希函数

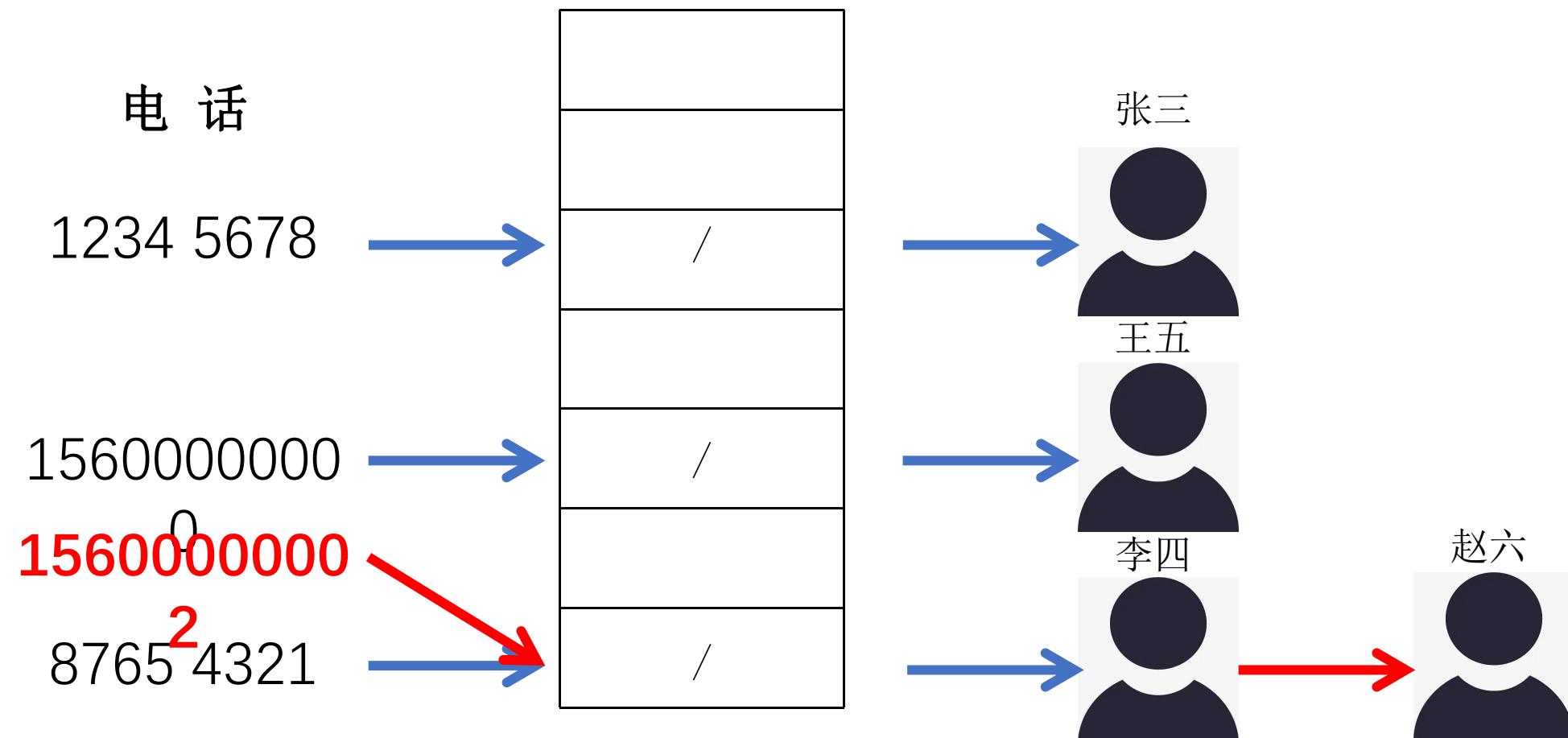
计算电话为15600000000的王五在上述哈希表中位置





除法哈希函数

计算电话为15600000002的赵六在上述哈希表中位置





总结

- **哈希表**通过把关键码值映射到表中一个位置来访问记录，其中的映射函数叫做**哈希函数**
- 常用哈希函数——除法哈希函数



散列函数

构造散列函数的基本方法

◆ 基本要求

设关键字集K中有n个关键字，哈希表长为m，即哈希表地址集为[0,m-1]，则哈希函数H应满足： 1. 对任意 $k_i \in K$, $i=1,2,\dots,n$, 有 $0 \leq H(k_i) \leq m-1$ ；
2. 对任意 $k_i \in K$, $H(k_i)$ 取[0,m-1]中任一值的概率相等。



除法散列法的不足

Problems with division method

对于规则性的键值集合

- Regularity
 - Suppose keys are $x, 2x, 3x, 4x, \dots$ 假如 x 与 m 有个公约数 d
 - Suppose x and chosen m have common divisor d
 - Then only use $1/d$ fraction of table 那么我们将只能有效利用到表格 $1/d$ 部分空间
 - x series cycles back, leaving $d-1$ out of d entries blank
 - E.g., m power of 2 and all keys are even, only use half
- So make m a prime number 所以 m 得是个质数
 - But finding a prime number is hard 但是找个质数很难
 - And now you have to divide (slow)

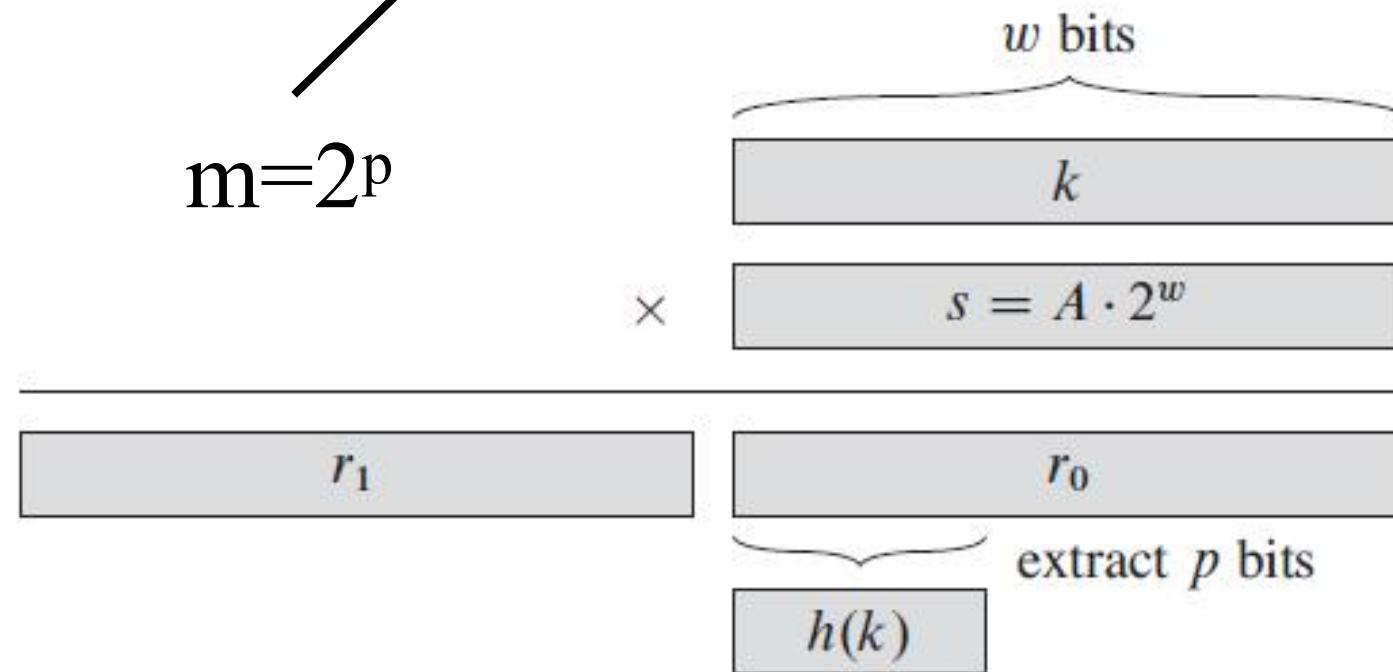
并且你要去相除（比乘法，位移都慢很多）



散列函数:乘法散列法

$$h(k) = \lfloor m (k A \bmod 1) \rfloor$$

$$m=2^p$$



$$A \approx (\sqrt{5} - 1) / 2 = 0.618033988$$



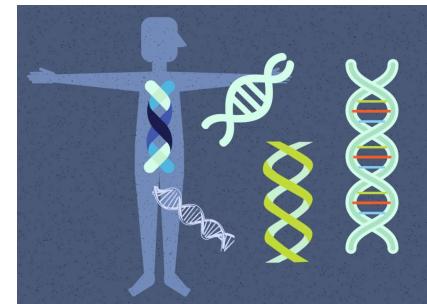
散列函数: 散列值

Key不是数字，是字符串呢？

Key=“abcdef”

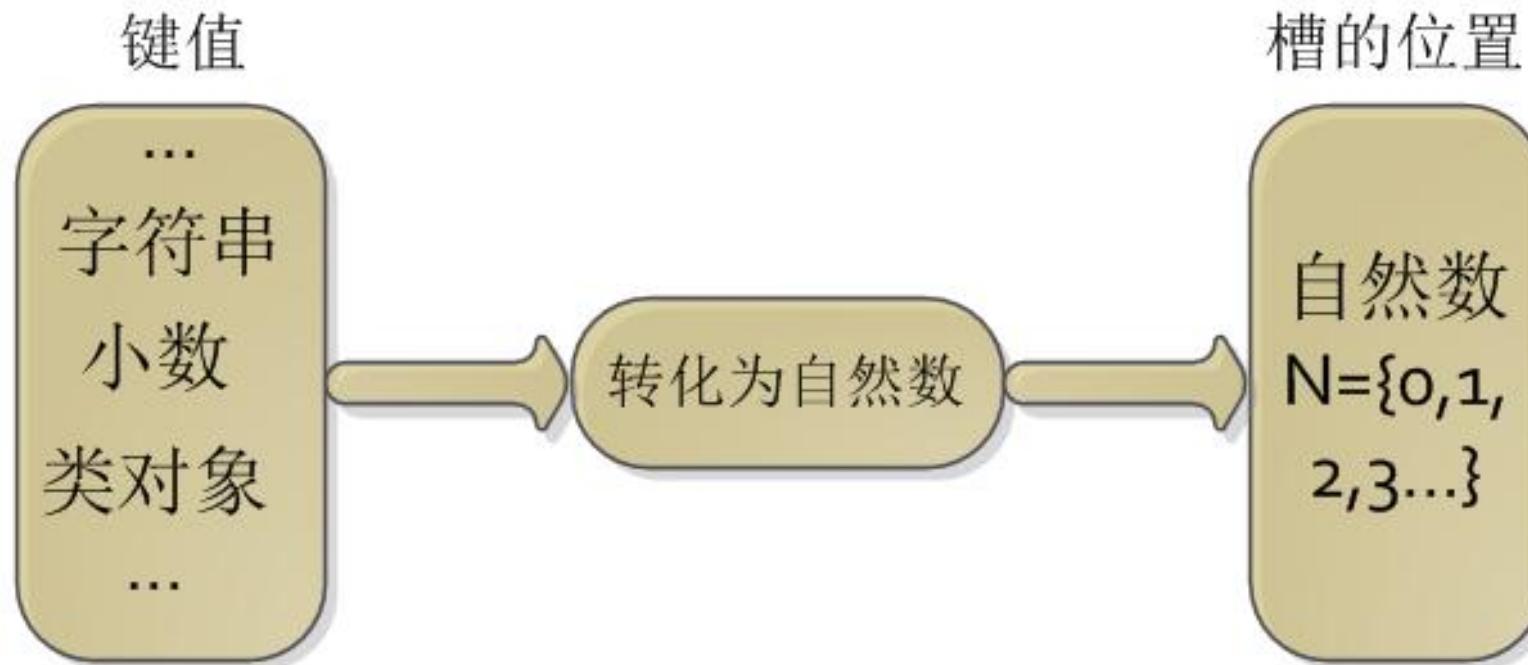


Hash值





散列函数: 散列值

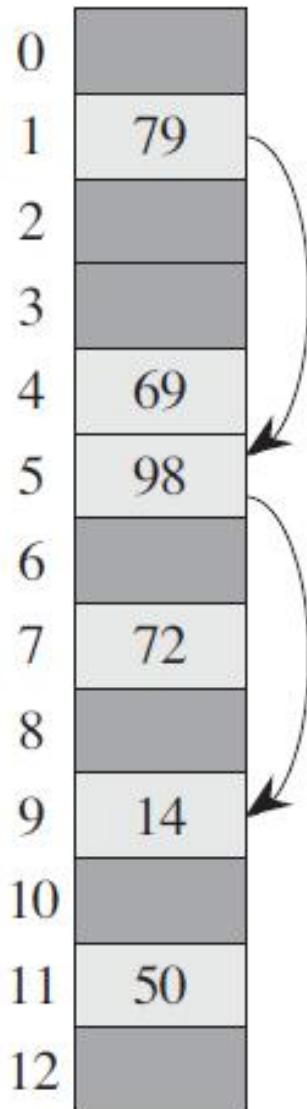


Java: `hashcode`
C++: ASCII码相加

Python:
`Key="abcdef"`
`A=hash(Key)`



开放寻址法





开放寻址法

```
1 HASH-INSERT(T,k)
2   i = 0
3   repeat
4     j = h(k,i)
5     if T[j] == NIL
6       T[j] = k
7       return j
8     else i = i + 1
9   until i == m
10  error "hash table overflow"
```



开放寻址法

```
1 HASH-SEARCH(T,k)
2     i = 0
3     repeat
4         j = h(k,i)
5         if T[j] == k
6             return j
7         i = i + 1
8     until T[j] == NIL or i == m
9     return NIL
```



开放寻址法

HASH-DELETE(T, k)

$i = 0$

repeat

$j = h(k, i)$

if ($T[j] == k$)

$T[j] = \text{DELETED}$

return

$i = i + 1$

until $T[j] == \text{NIL}$ or $i == m$

return



开放寻址法

```
HASH-INSERT(T, k)
    i = 0
    repeat
        j = h(k,i)
        if ( T[j] == NIL or T[j] == DELETED )
            T[j] = k
            return j
        else i = i + 1
    until i == m
    error "hash table overflow"
```



开放寻址法

三种常用技术来计算开放寻址法中的探查序列

- 1.线性探查
- 2.二次探查
- 3.双重散列



开放寻址法-线性探查

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

关键词 (key)	47	7	29	11	9	84	54	20	30
散列地址 h(key)	3	7	7	0	9	7	10	9	8
冲突次数	0	0	1	0	0	3	1	3	6



开放寻址法-线性探查

关键词 (key)	47	7	29	11	9	84	54	20	30
散列地址 h(key)	3	7	7	0	9	7	10	9	8
冲突次数	0	0	1	0	0	3	1	3	6

操作 地址	0	1	2	3	4	5	6	7	8	9	10	11	12	说明
插入47				47										无冲突
插入7				47				7						无冲突
插入29				47				7	29					$d_1 = 1$
插入11	11			47				7	29					无冲突
插入9	11			47				7	29	9				无冲突
插入84	11			47				7	29	9	84			$d_3 = 3$
插入54	11			47				7	29	9	84	54		$d_1 = 1$
插入20	11			47				7	29	9	84	54	20	$d_3 = 3$
插入30	11	30		47				7	29	9	84	54	20	$d_6 = 6$

注意“聚集”现象

http://blog.csdn.net/qq_30091945

开放寻址法-二次探查



$$h(k, i) = (h'(k) + c_1i + c_2i^2) \bmod m$$



开放寻址法-双重散列

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$



Cuckoo Hash 布谷鸟哈希

定义：一种解决 hash 冲突的方法，其目的是使用简单的 hash 函数来提高 hash table 的利用率，同时保证 $O(1)$ 的查询时间。基本思想是使用2个 hash 函数来处理碰撞，从而每个 key 都对应到2个位置。



Cuckoo Hash 布谷鸟哈希

操作：

- 1) 对 key 值 hash，生成两个 hash key 值， hashk_1 和 hashk_2 ，如果对应的两个位置上有一个为空，那么直接把 key 插入即可。
- 2) 否则，任选一个位置，把 key 值插入，把已经在那个位置的 key 值踢出来。
- 3) 被踢出来的 key 值，需要重新插入，直到没有 key 被踢出为止。



Cuckoo Hash 布谷鸟哈希

衍生背景：

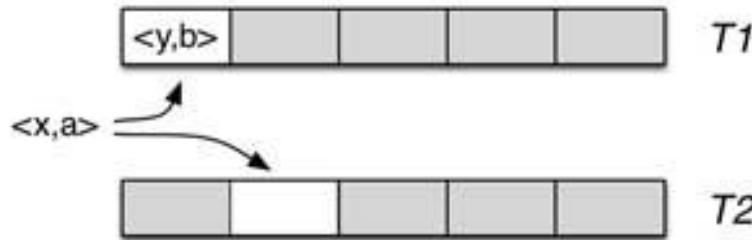
Cuckoo中文名叫布谷鸟，这种鸟有一种即狡猾又贪婪的习性，它不肯自己筑巢，而是把蛋下到别的鸟巢里，而且它的幼鸟又会比别的鸟早出生，布谷幼鸟天生有一种残忍的动作，幼鸟会拼命把未出生的其它鸟蛋挤出窝巢，今后以便独享“养父母”的食物。借助生物学上这一典故，cuckoo hashing处理碰撞的方法，就是把原来占用位置的这个元素踢走，不过被踢出去的元素还要比鸟蛋幸运，因为它还有一个备用位置可以安置，如果备用位置上还有人，再把它踢走，如此往复。直到被踢的次数达到一个上限，才确认哈希表已满，并执行rehash操作。



Cuckoo Hash 布谷鸟哈希

Insertion when one of the two buckets is empty

Step 1: Both buckets for $\langle x, a \rangle$ are tested, the one in T2 is empty.

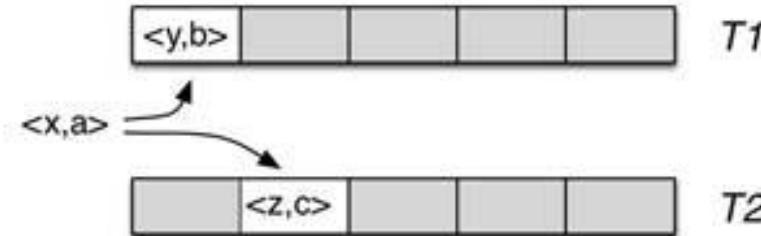


Step 2: $\langle x, a \rangle$ is stored in the empty bucket in T2.

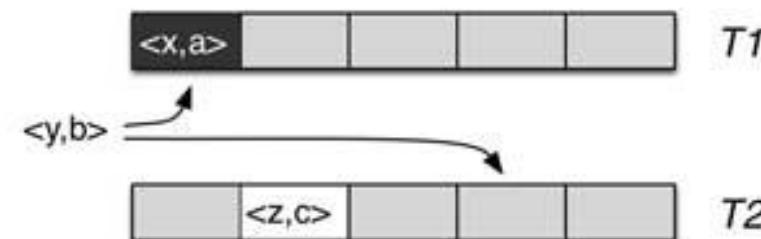


Insertion when the two buckets already contain entries

Step 1: Here $\langle y, b \rangle$ will be withdrawn from T1 so that $\langle x, a \rangle$ can be stored.



Step 2: After $\langle x, a \rangle$ has been stored in T1, $\langle y, b \rangle$ needs to be moved to T2. The bucket in T2 may already contain an entry, if so this entry will need to be moved.





Cuckoo Hash 布谷鸟哈希

其他：

- 1) Cuckoo hash 有两种变形。一种通过增加哈希函数进一步提高空间利用率；另一种是增加哈希表，每个哈希函数对应一个哈希表，每次选择多个张表中空余位置进行放置。三个哈希表可以达到80% 的空间利用率。
- 2) Cuckoo hash 的过程可能因为反复踢出无限循环下去，这时候就需要进行一次循环踢出的限制，超过限制则认为需要添加新的哈希函数。

哈希的扩展应用

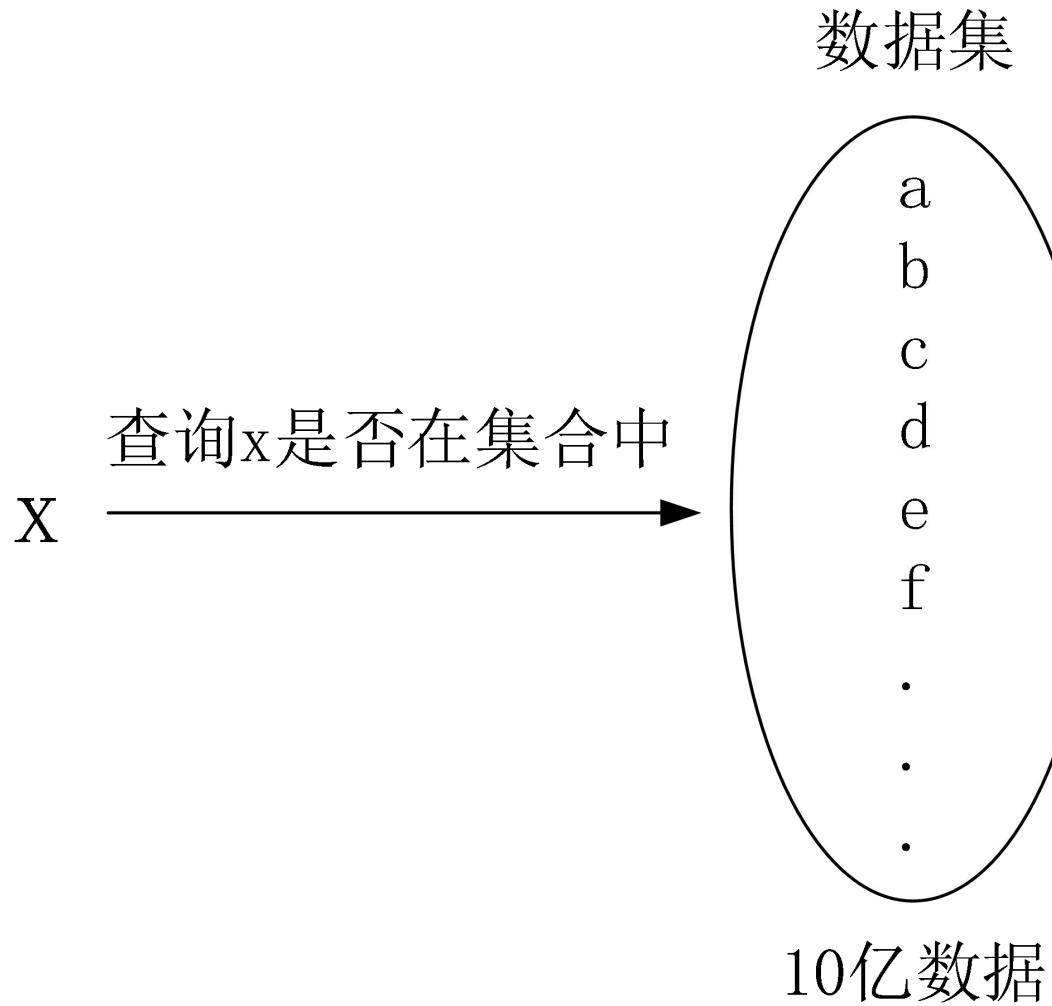


- 布鲁姆过滤器（BloomFilter）
- 最小哈希
- 区块链



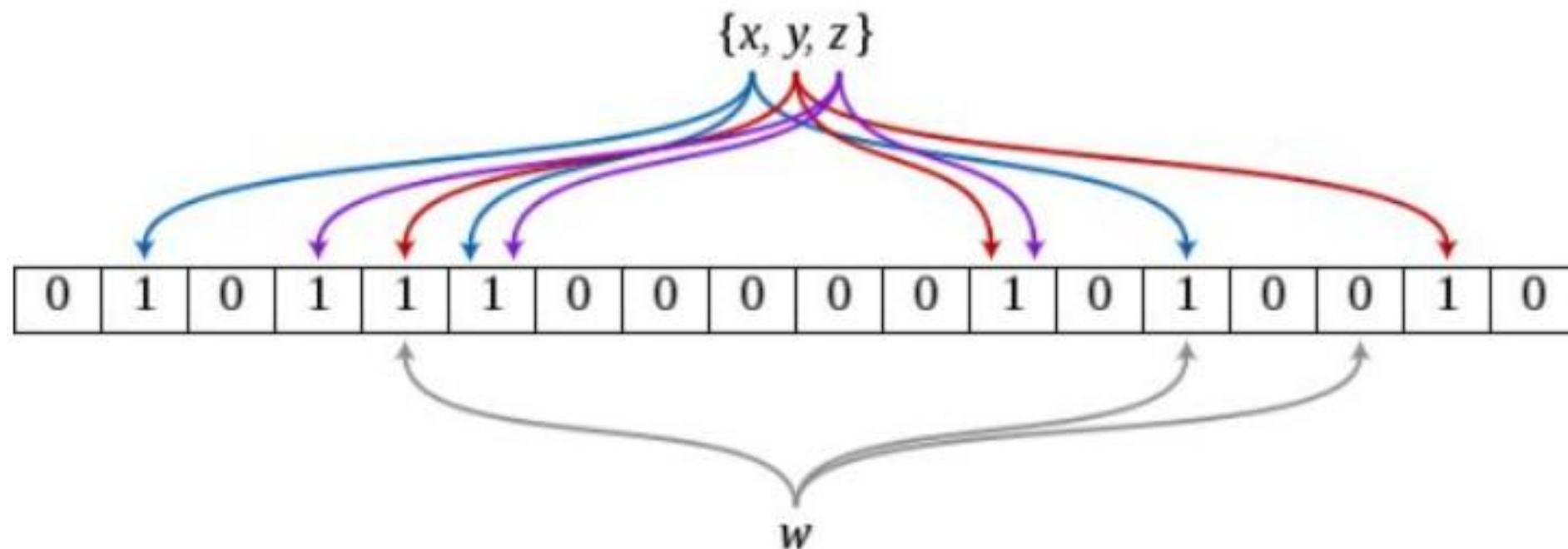
布鲁姆过滤器（bloomfilter）

问题1





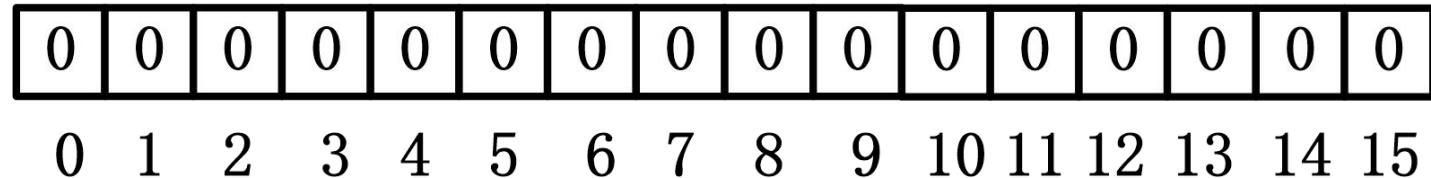
布鲁姆过滤器 (bloomfilter)



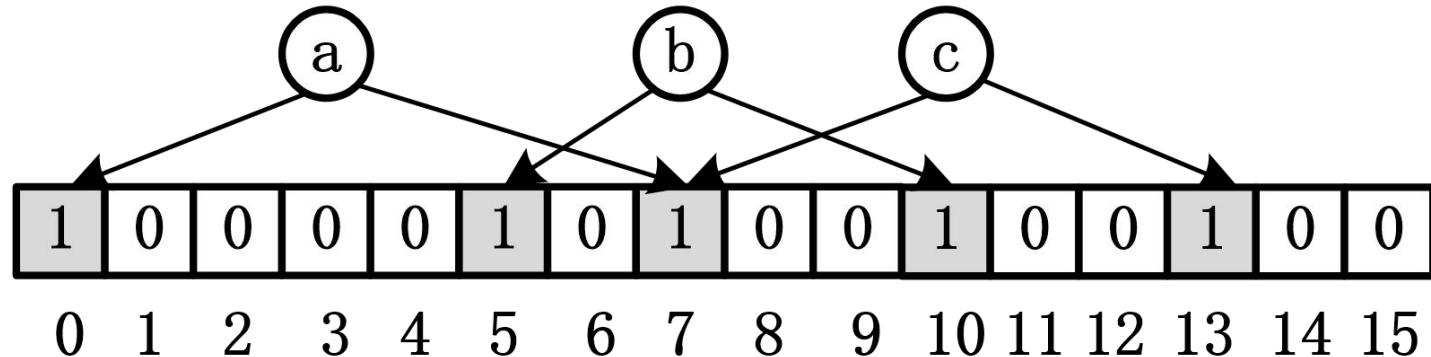


布鲁姆过滤器 (bloomfilter)

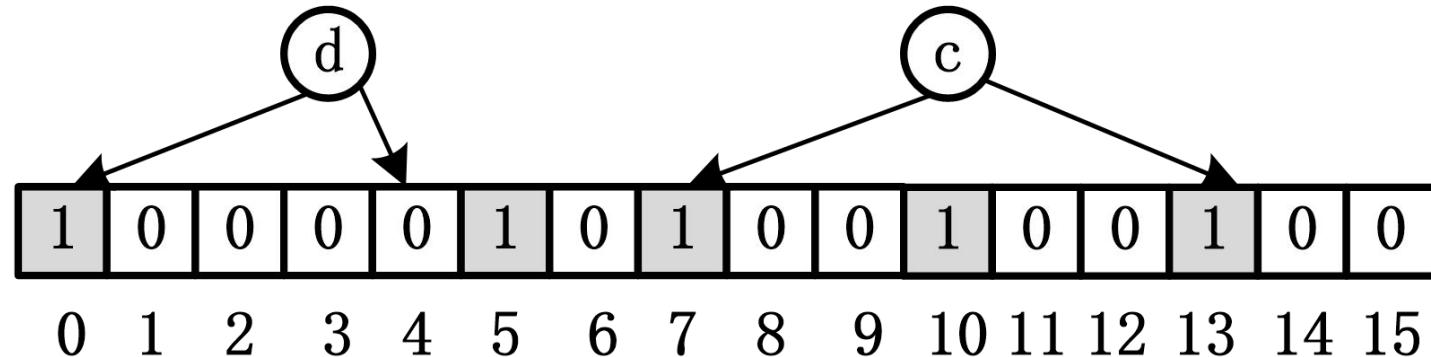
初始化



插入a, b, c



查询d, c





布鲁姆过滤器（bloomfilter）

布鲁姆过滤器算法关键指标

- 空间复杂度
- 时间复杂度
- 误判率（假阳性）

将不属于集合的元素误判断成属于集合中的这种假阳性误判称为一次误判。发生误判的概率称为误判率。



布鲁姆过滤器（bloomfilter）

误判率分析（假阳性）

理论分析，假设k个哈希函数、m位bitset，n个元素之后

$$f_{\text{BF}}(m, k, n) = (1 - e^{-k \cdot n / m})^k$$



布鲁姆过滤器（bloomfilter）

误判率分析（假阳性）

最优分析

$$k = \lceil \ln 2 (m / n) \rceil$$

$$k_{\min} = (\ln 2) \left(\frac{m}{n} \right)$$



布鲁姆过滤器（bloomfilter）

误判率分析（假阳性）

(1) 理论分析，假设k个哈希函数、m位bitset，n个元素之后

$$f_{\text{BF}}(m, k, n) = (1 - e^{-k \cdot n / m})^k$$



最小哈希

MinHash是一种局部敏感哈希技术
可以用来快速估算两个集合的相似度



Jaccard相似度

Jaccard相似度是用来计算集合相似性，也就是距离的一种度量标准

假如有集合A、B，那么 $J(A,B) = |A \cap B| / |A \cup B|$



最小哈希

$h(x)$: 把 x 映射成一个整数的哈希函数

$h_{\min}(S)$: 集合 S 中的元素经过 $h(x)$ 哈希后，具有最小哈希值的元素

那么对集合 A 、 B ， $h_{\min}(A) = h_{\min}(B)$ 成立的条件是 $A \cup B$ 中具有最小哈希值的元素也在 $A \cap B$ 中



最小哈希性质

假设 $h(x)$ 是一个良好的哈希函数，它具有很好的均匀性，能够把不同元素映射成不同的整数。

所以有， $\Pr[h_{\min}(A) = h_{\min}(B)] = J(A, B)$ ，即集合A和B的相似度为集合A、B经过hash后最小哈希值相等的概率。



湖南大學
HUNAN UNIVERSITY

感谢观赏

—— 实事求是 敢为人先 ——