第二章 高级数据结构

湖南大学信息科学与工程学院

提纲

2.1 堆

2.2 散列表

提纲

2.1 堆

2.2 散列表

提纲

2.3.1 堆的概述

2.3.2 堆的调整

2.3.3 维护堆的性质

2.3.4 建堆

2.3.5 堆排序算法

2.3.6 优先队列

堆的定义

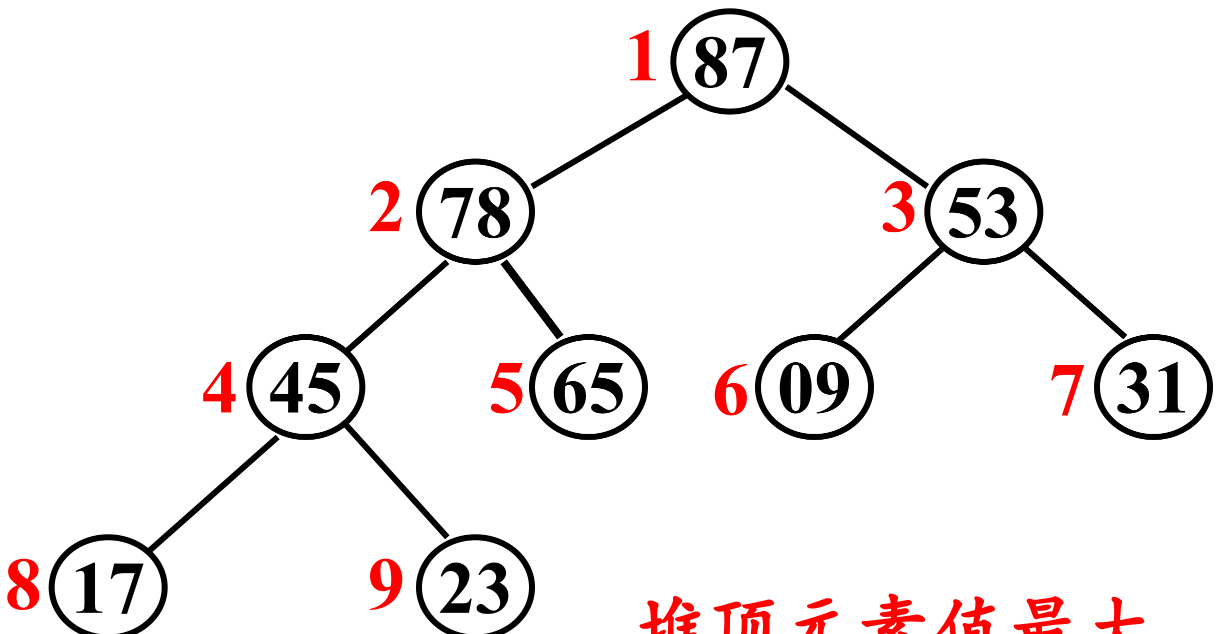
n个元素的序列$\{k_1,k_2,\ldots,k_n\}$当且仅当满足如下关系时，称之为堆（heap）。若满足条件$k_i<=k_{i/2}$且则称大根堆（或最大堆）；若满足条件$k_i>=k_{i/2}$则称小根堆（或最小堆）。

若将和此序列对应的一维数组(即以一维数组作此序列的存储结构)看成是一个完全二叉树，则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。

2.3.1 堆的概述

最大堆

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

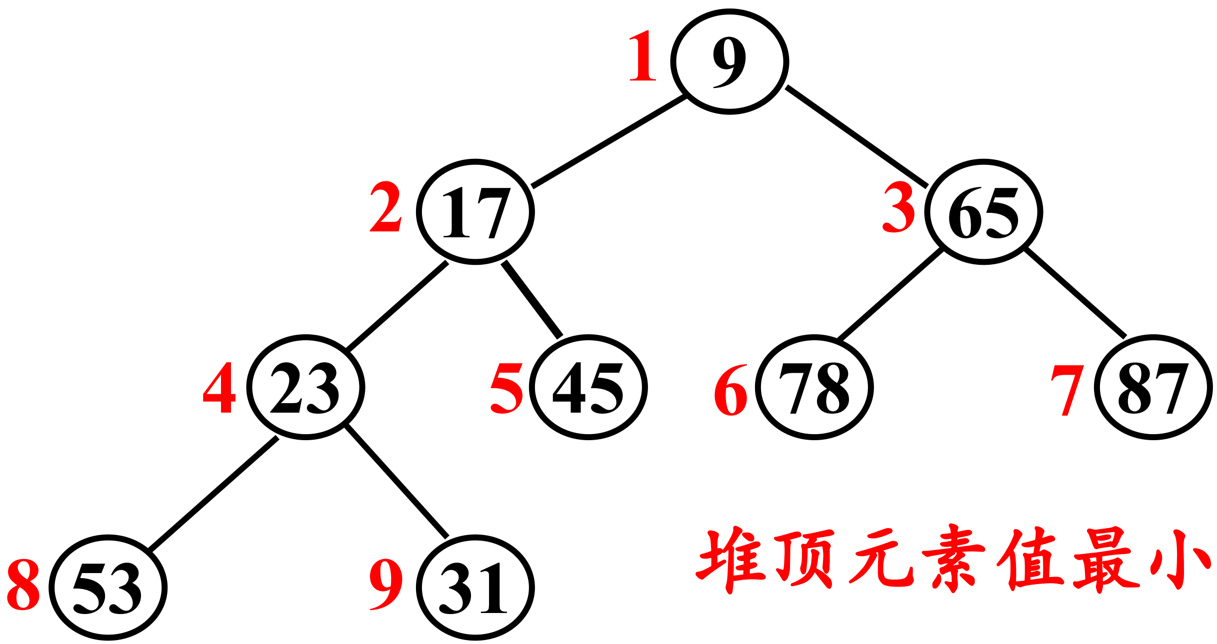


堆顶元素值最大

2.3.1 堆的概述

最小堆

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



堆顶元素值最小

将堆用顺序存储结构来存储，则堆对应一组序列

2.3.1 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

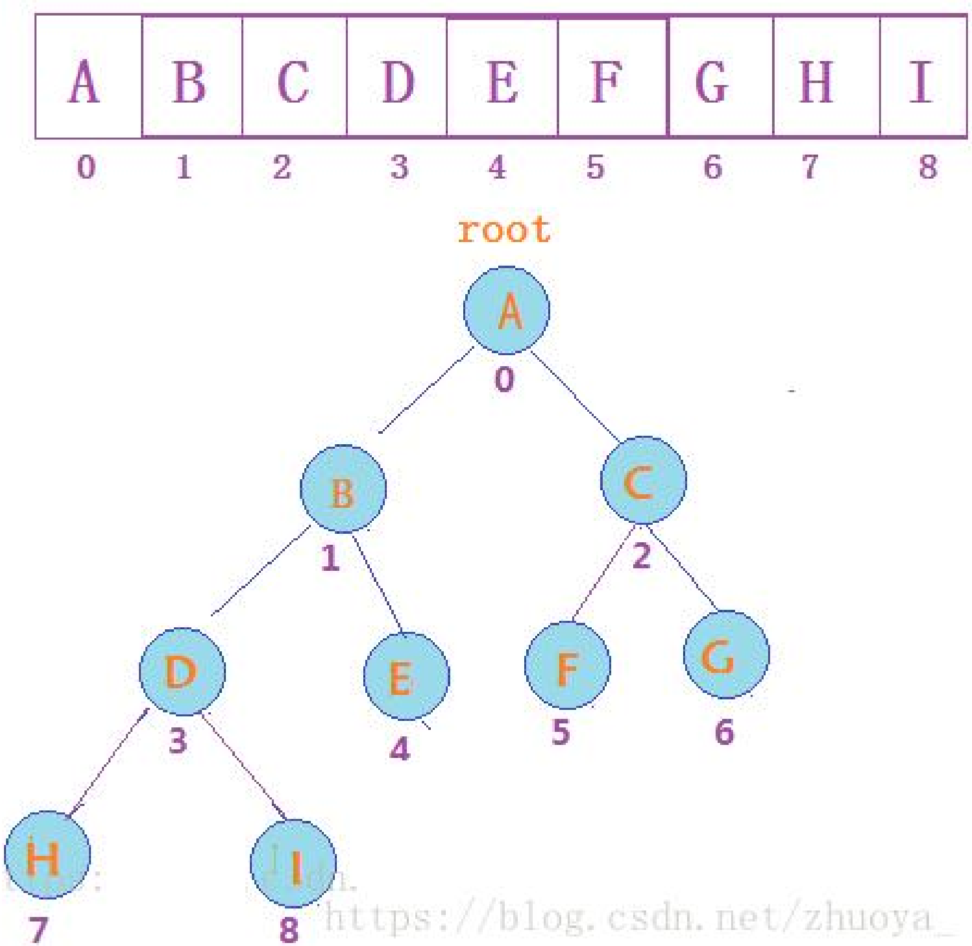
2.3.1 堆的概述

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）

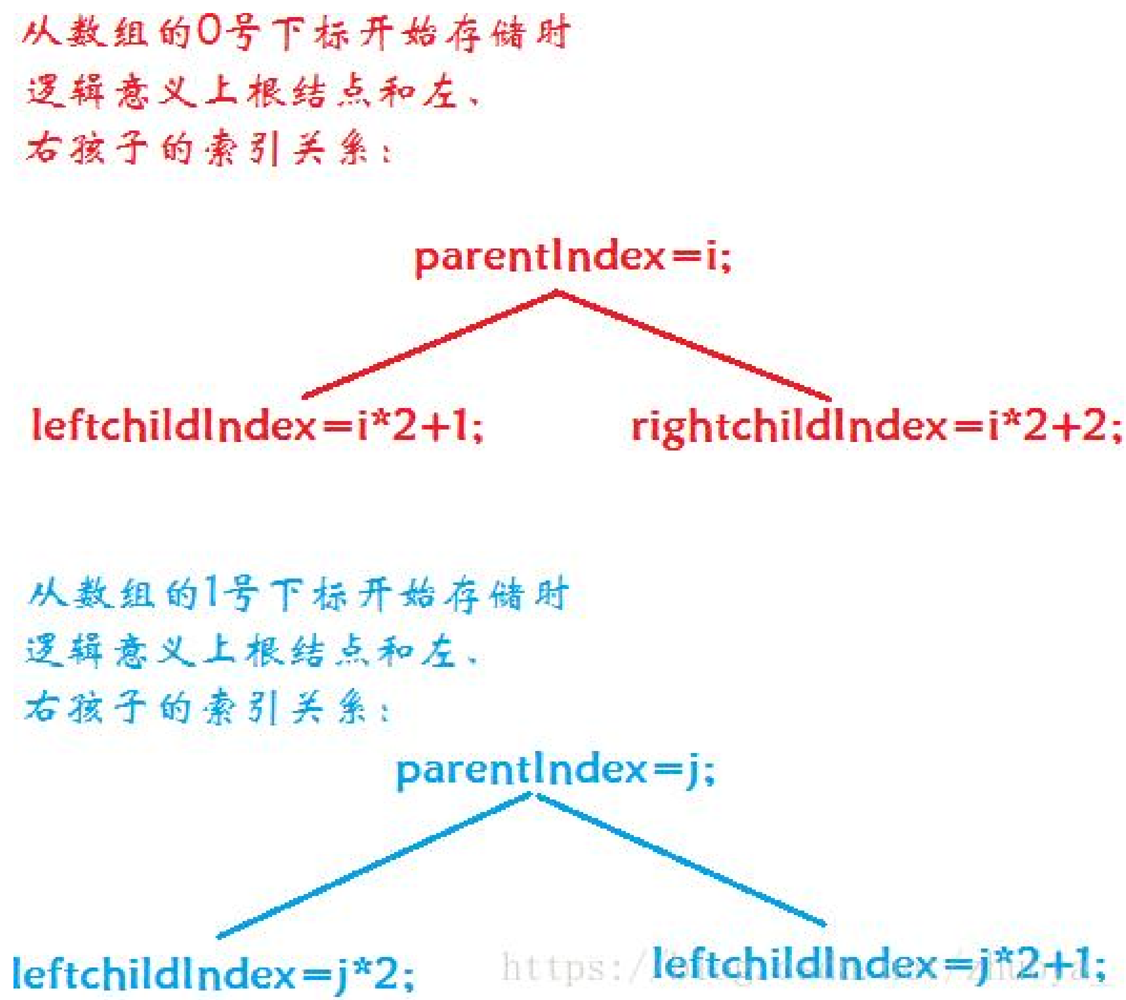
2.3.1 堆的概述

完全二叉树的存储方式：顺序存储

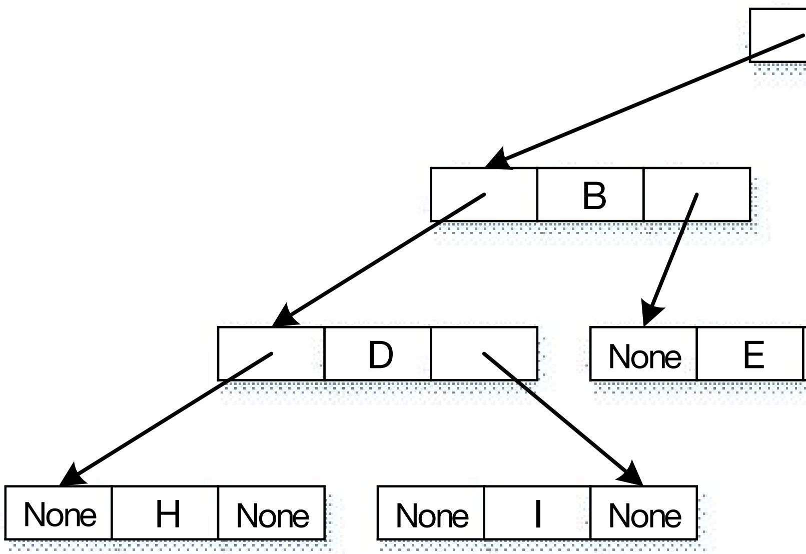


2.3.1 堆的概述

完全二叉树的存储方式：顺序存储



完全二叉树的存储方式：链式存储



提纲

2.3.1 堆的概述

2.3.2 堆的调整

2.3.3 维护堆的性质

2.3.4 建堆

2.3.5 堆排序算法

2.3.6 优先队列

2.3.2 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?

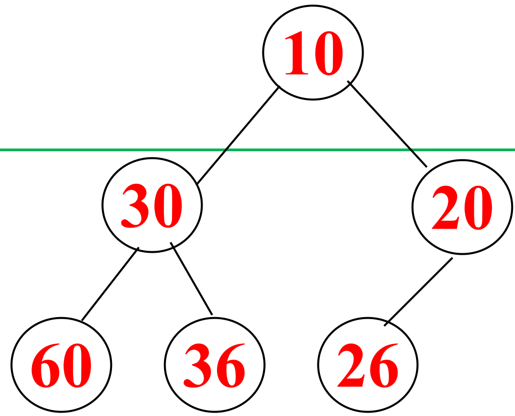
解决方法:

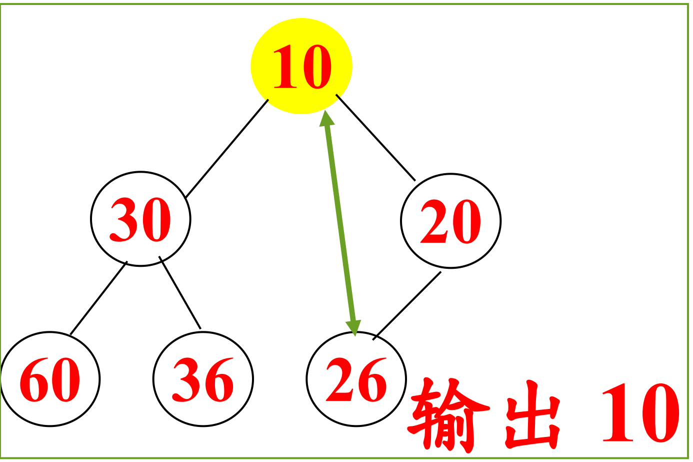
　　输出堆顶元素之后，以堆中最后一个元素替代之；然后将根结点值与左、右子树的根结点值进行比较，并与其中小者进行交换；重复上述操作，直至叶子结点，将得到新的堆，称这个从堆顶至叶子的调整过程为“筛选”

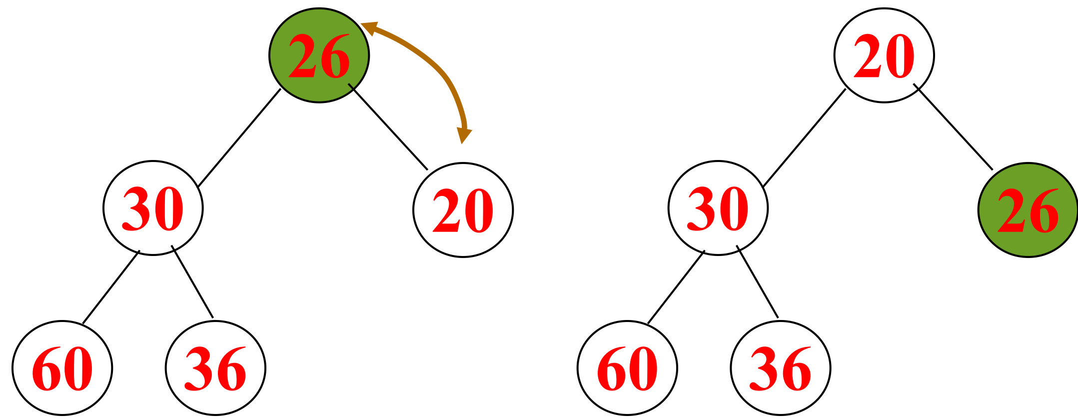
2.3.2 堆的调整

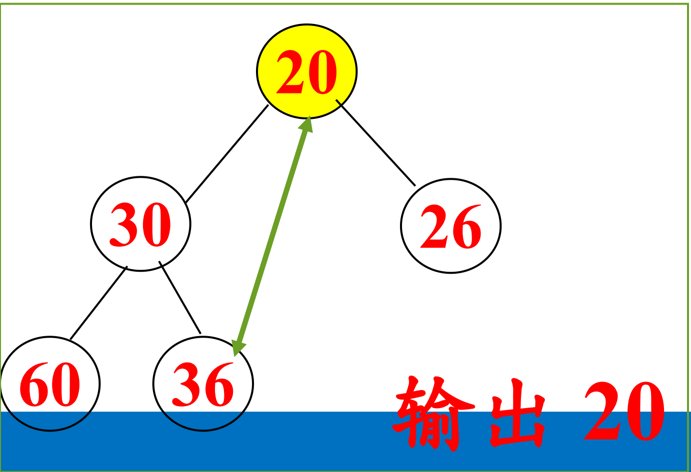
动画演示

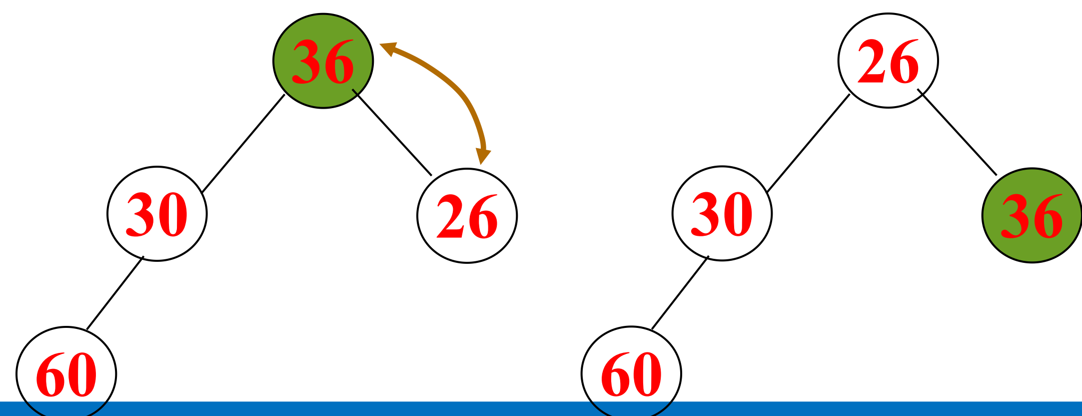
最小堆输出序列

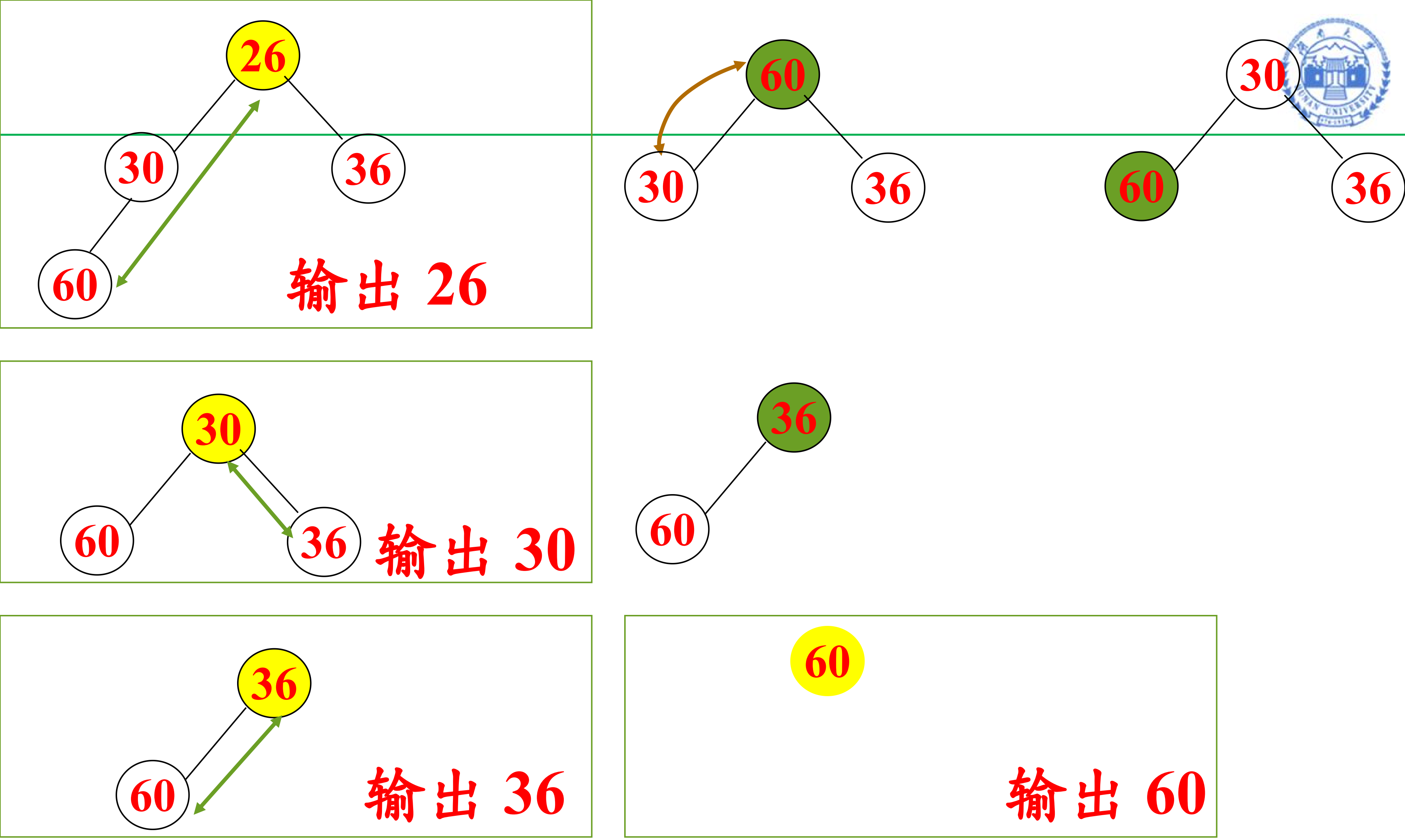

10
30
20
60
36
26


10
30
20
60
36
26
输出 10


26
30
20
60
36
20
30
26
60
36


20
30
26
60
36
输出 20

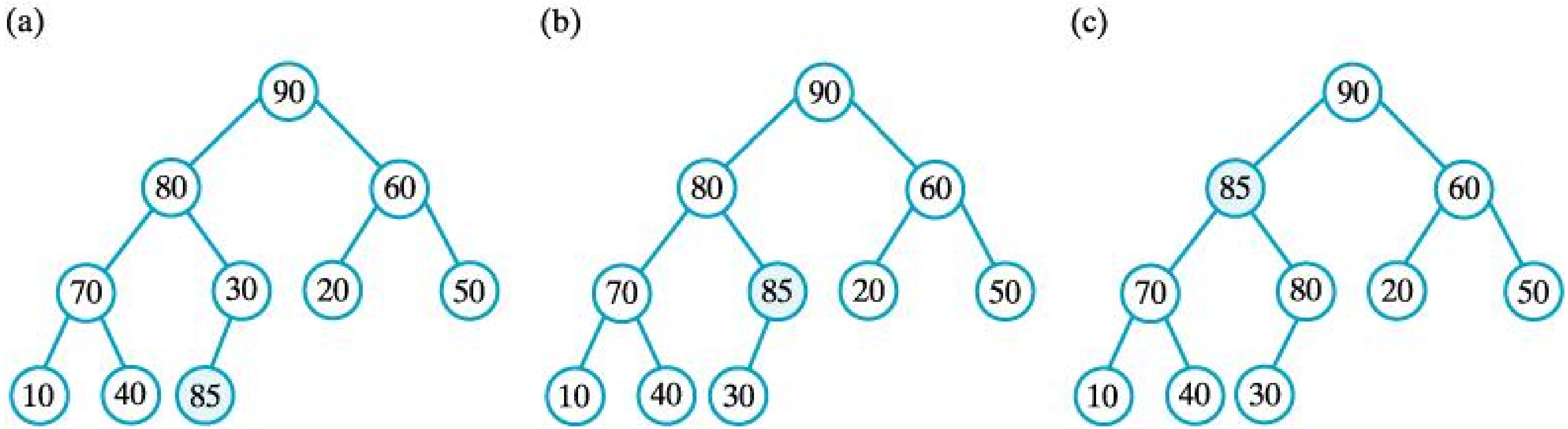

36
30
26
60
26
30
36
60



递增输出序列为：10 20 26 30 36 60

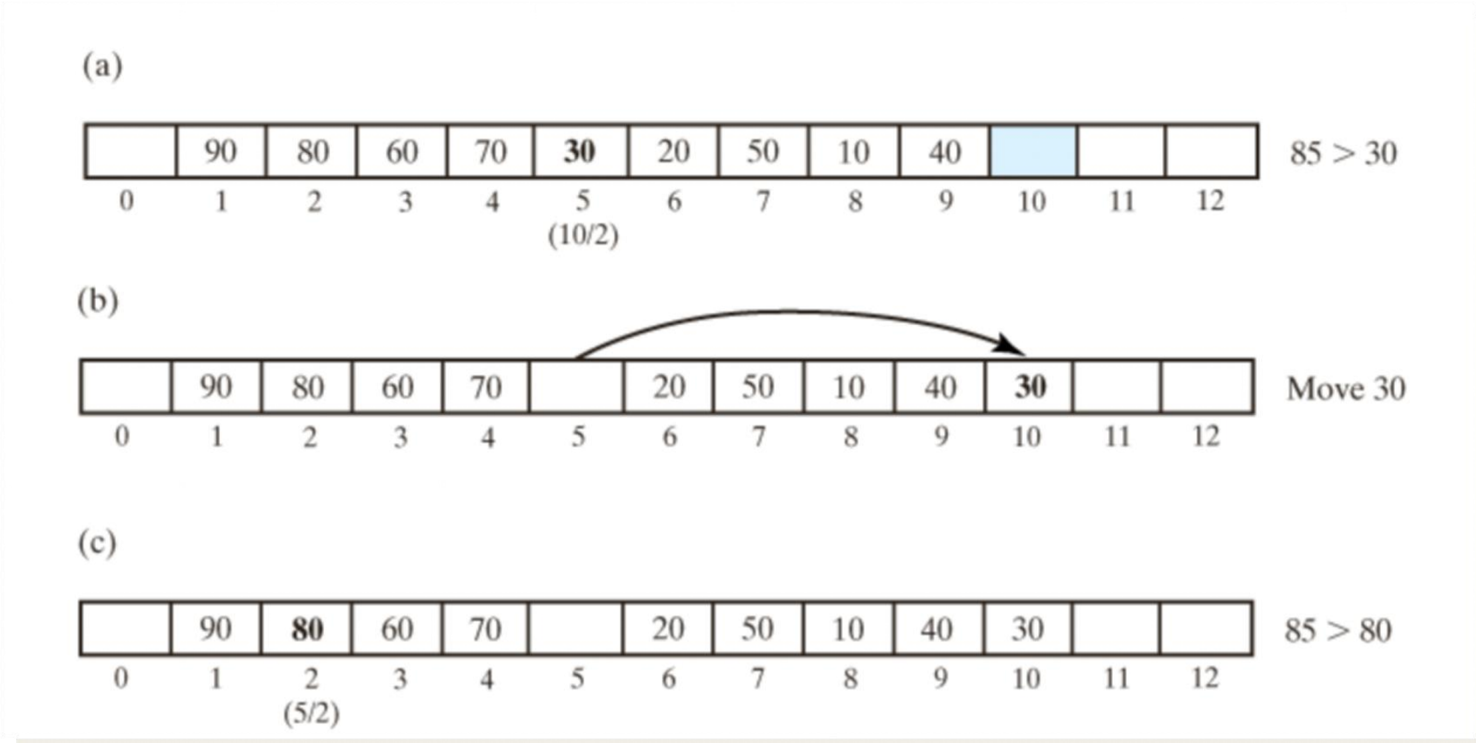
2.3.2 堆的调整

最大堆中加入元素85



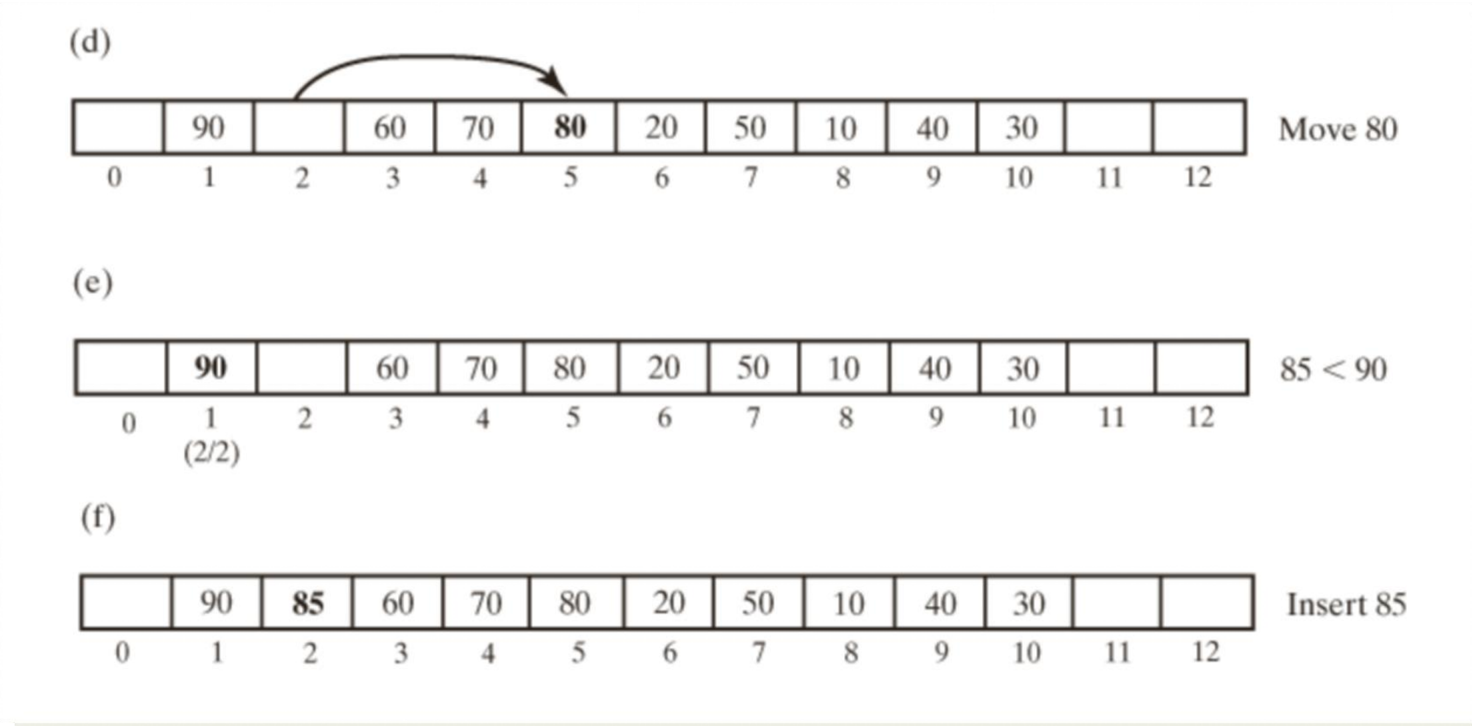
2.3.2 堆的调整

最大堆中加入元素85



2.3.2 堆的调整

最大堆中加入元素85



2.3.2 堆的调整

加入元素时向上调整
删除元素时向下调整

提纲

2.3.1 堆的概述

2.3.2 堆的调整

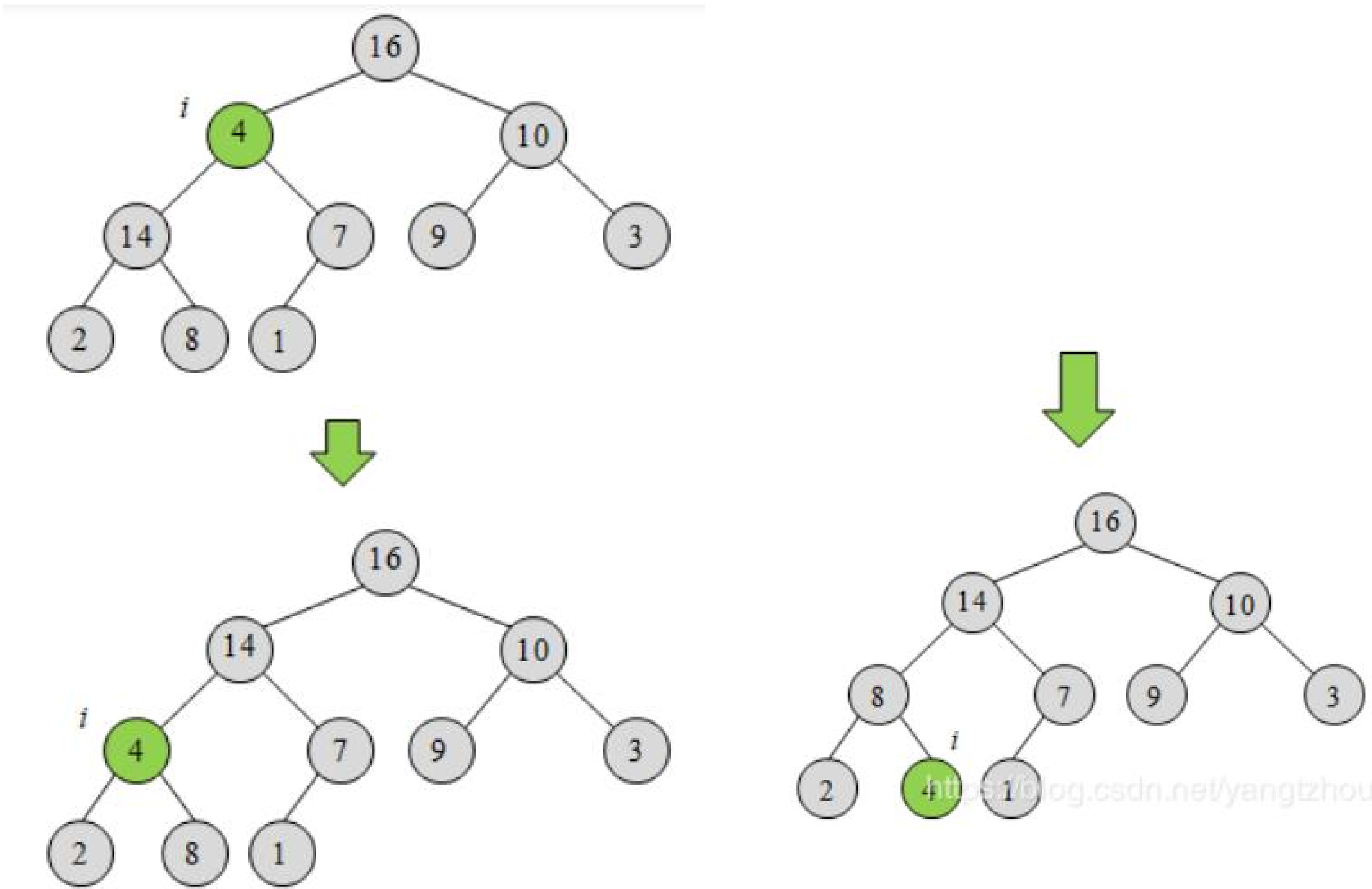
2.3.3 维护堆的性质

2.3.4 建堆

2.3.5 堆排序算法

2.3.6 优先队列

2.3.3 维护堆的性质



2.3.3 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
MAX-HEAPIFY(A, i)
    l = LEFT(i)
    r = RIGHT(i)
    if l ≤ A.heap_size and A[l] > A[i]
        largest = l
    else
        largest = i
    if r ≤ A.heap_size and A[r] > A[largest]
        largest = r
    if largest ≠ i
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```

提纲

2.3.1 堆的概述

2.3.2 堆的调整

2.3.3 维护堆的性质

2.3.4 建堆

2.3.5 堆排序算法

2.3.6 优先队列

2.3.4 建堆

建堆方法:

- ✓ 自底向上
- ✓ 插入建堆

2.3.4 建堆

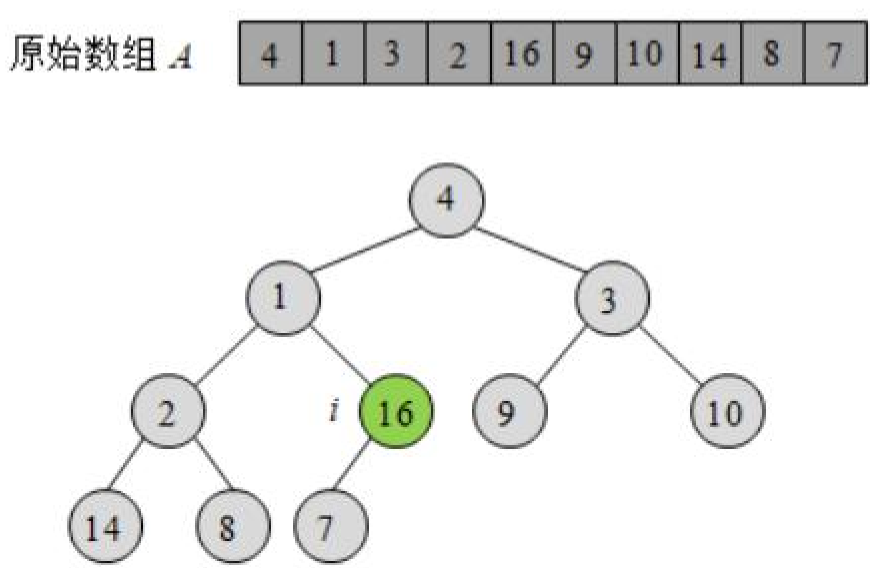
建堆方法：自底向上（迭代实现）

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
BUILD-MAX-HEAP(A)
    A.heap_size = A.length
    for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

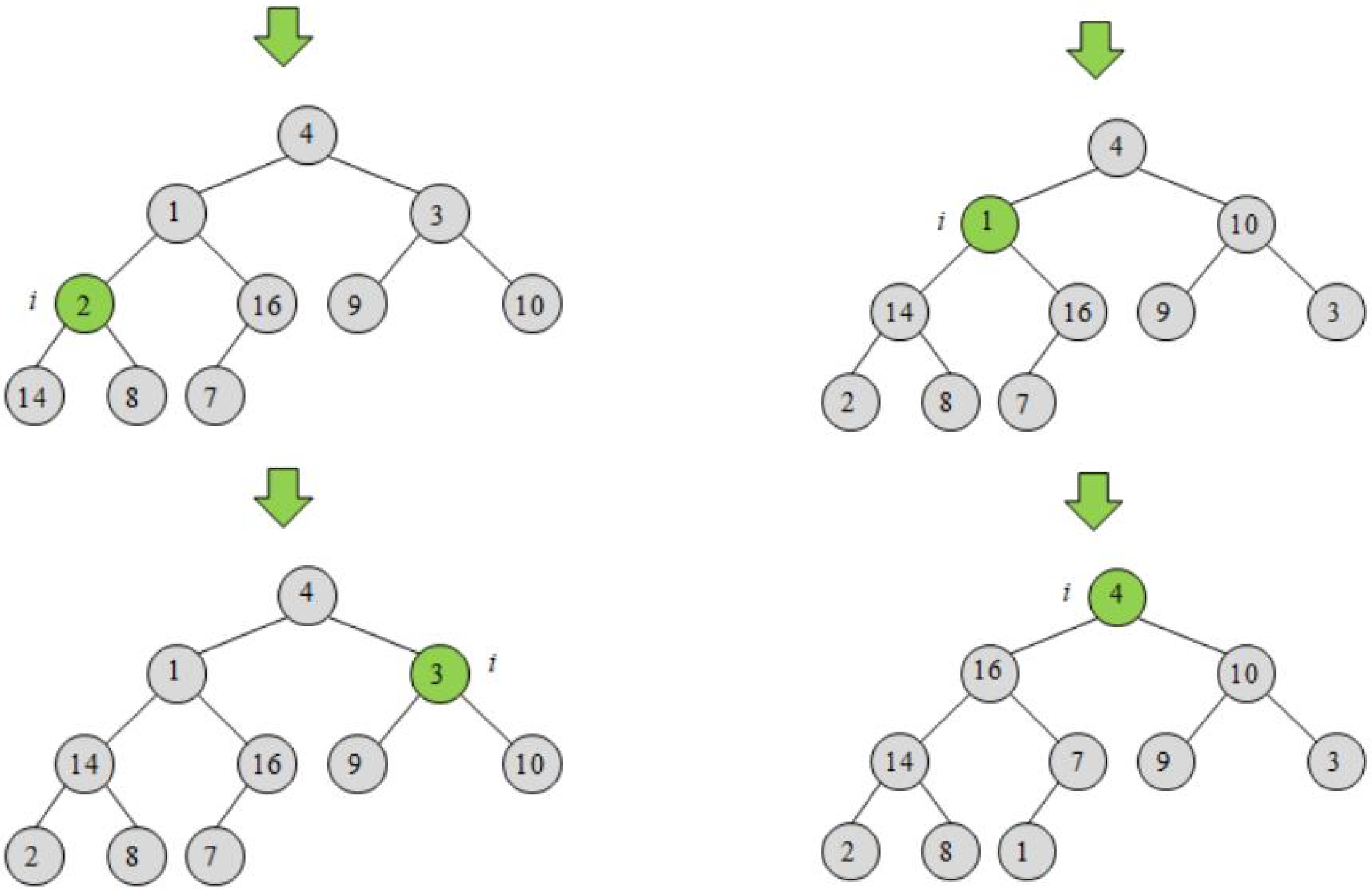
2.3.4 建堆

建堆方法1：自底向上（迭代实现）



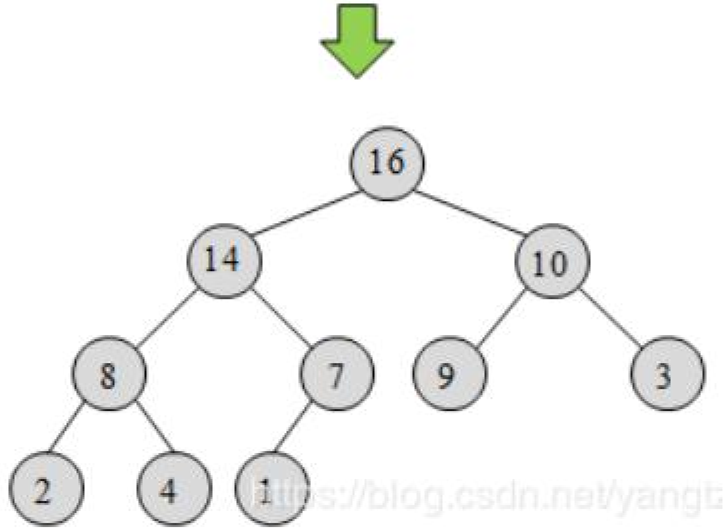
2.3.4 建堆

建堆方法1： 自底向上（迭代实现）



2.3.4 建堆

建堆方法1：自底向上（迭代实现）



2.3.4 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)
```

2.3.4 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)
```

2.3.4 建堆

建堆方法2：插入建堆

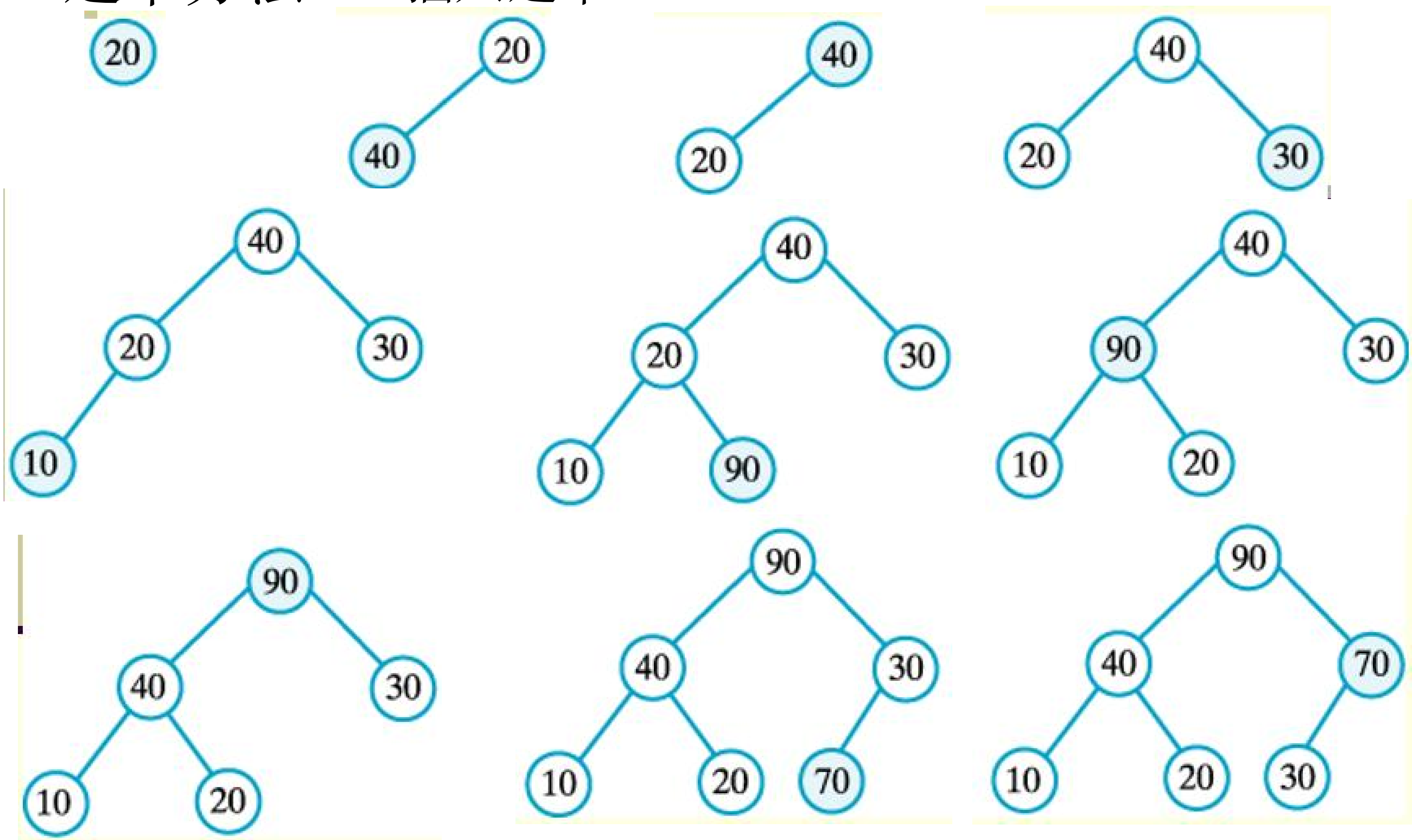
把一个数组看成两部分，左边是堆，右边是还没加入堆的元素，步骤如下：

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

2.3.4 建堆

建堆方法2： 插入建堆



提纲

2.3.1 堆的概述

2.3.2 堆的调整

2.3.3 维护堆的性质

2.3.4 建堆

2.3.5 堆排序算法

2.3.6 优先队列

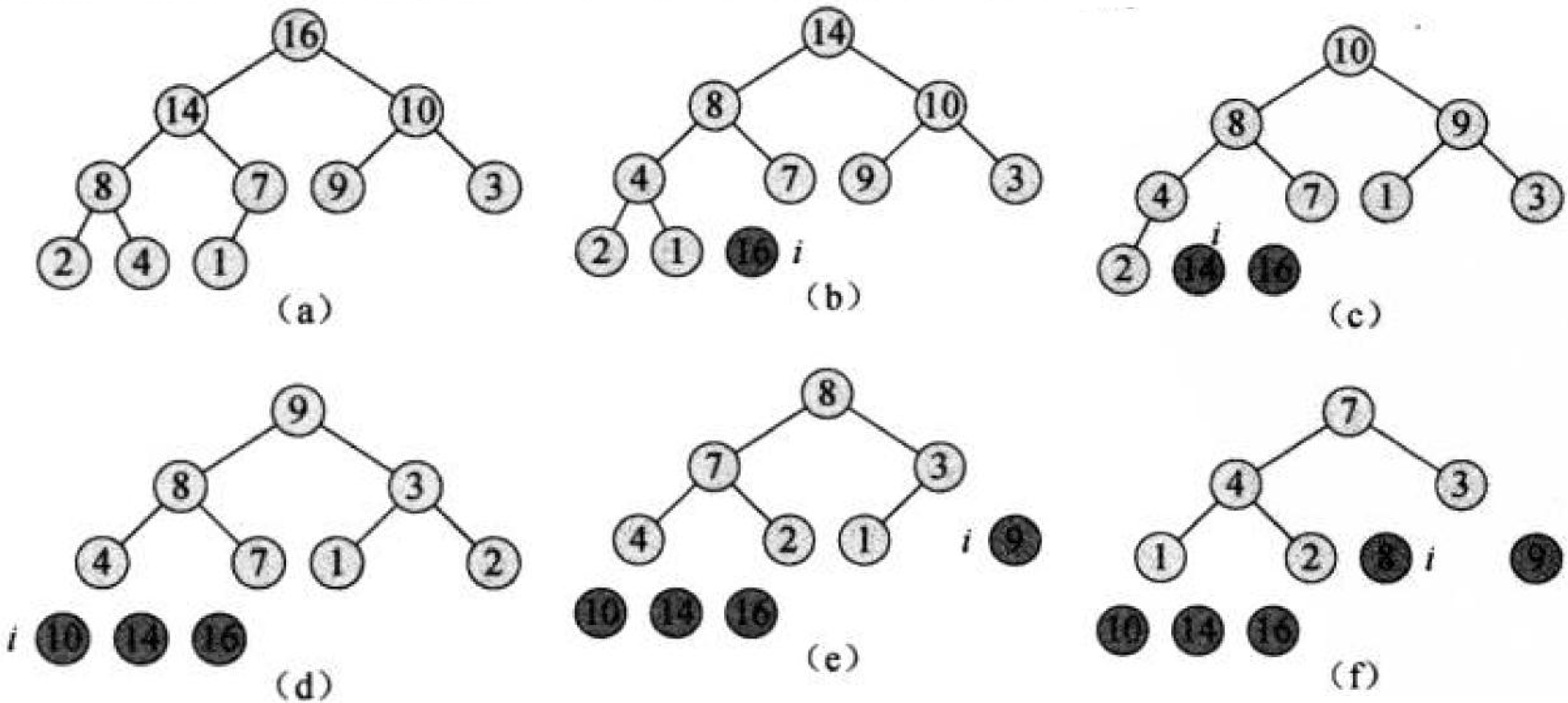
2.3.5 堆排序算法

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**。

2.3.5 堆排序算法

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
        exchange A[1] with A[i]
        A.heap-size = A.heap-size - 1
        MAX-HEAPIFY(A, 1)
```

2.3.5 堆排序算法



16
14
10
8
7
9
3
2
4
1
(a)
14
8
10
4
7
9
3
2
1
16
i
(b)
10
8
9
4
7
1
3
i
2
14
16
(c)
9
8
3
4
7
1
2
i
10
14
16
(d)
8
7
3
4
2
1
i
9
10
14
16
(e)
7
4
3
1
2
8
i
9
10
14
16
(f)

2.3.5 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nlg(n)（递归）

总的时间复杂度为O(n+nlgn)=O(nlgn)

提纲

2.3.1 堆的概述

2.3.2 堆的调整

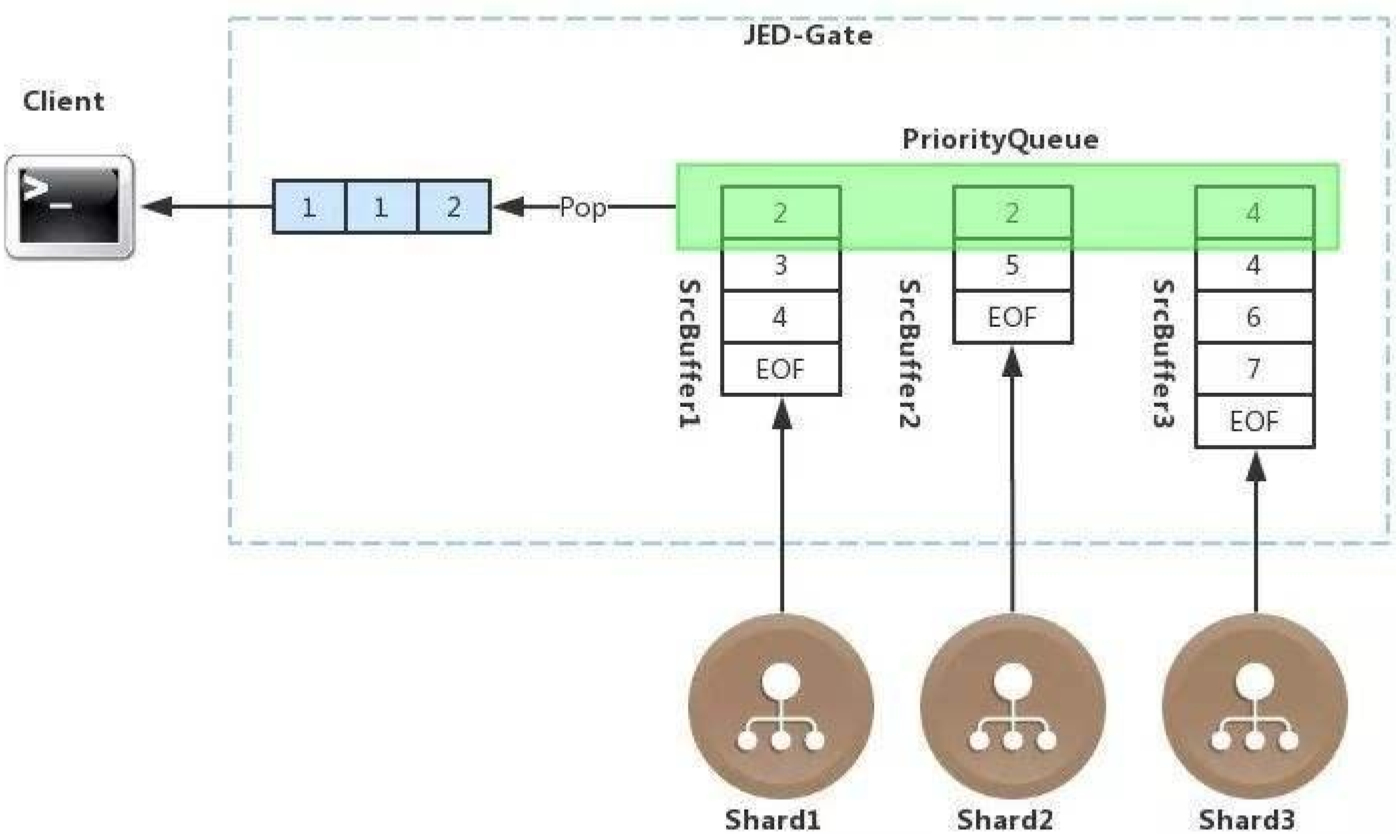
2.3.3 维护堆的性质

2.3.4 建堆

2.3.5 堆排序算法

2.3.6 优先队列

2.3. 6 优先队列



优先级队列有两种：最大优先级队列和最小优先级队列

2.3. 6 优先队列–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。

2.3. 6 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。

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。

2.3. 6 优先队列–最大优先级队列操作实现

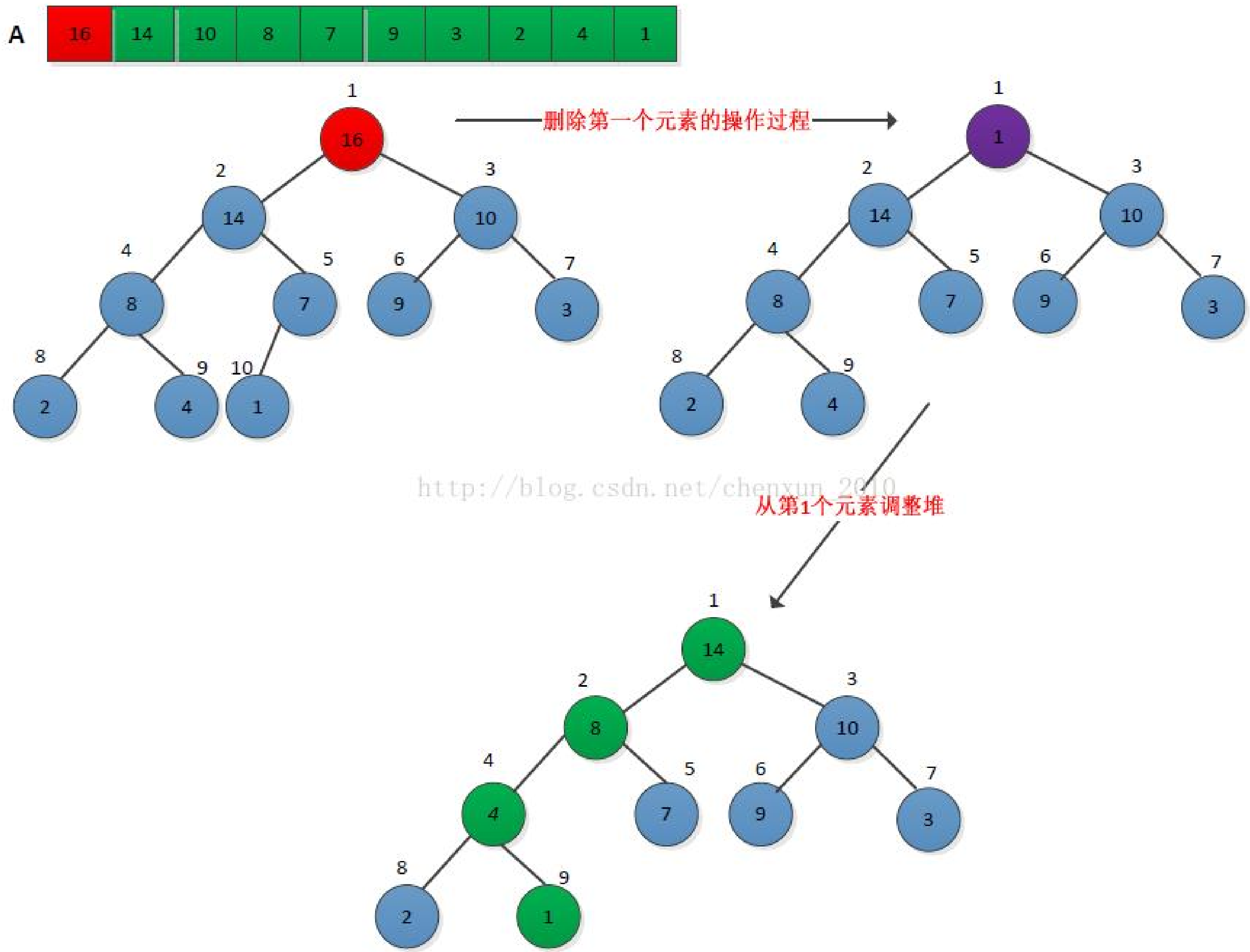
堆实现优先队列：

（1）HEAP_MAXIMUM返回最大堆第一个元素的值。
（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。
（3）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。
（4）MAX_HEAP_INSERT实现INSERT操作，向最大堆中插入新的关键字。

2.3. 6 优先队列–返回第一个元素

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

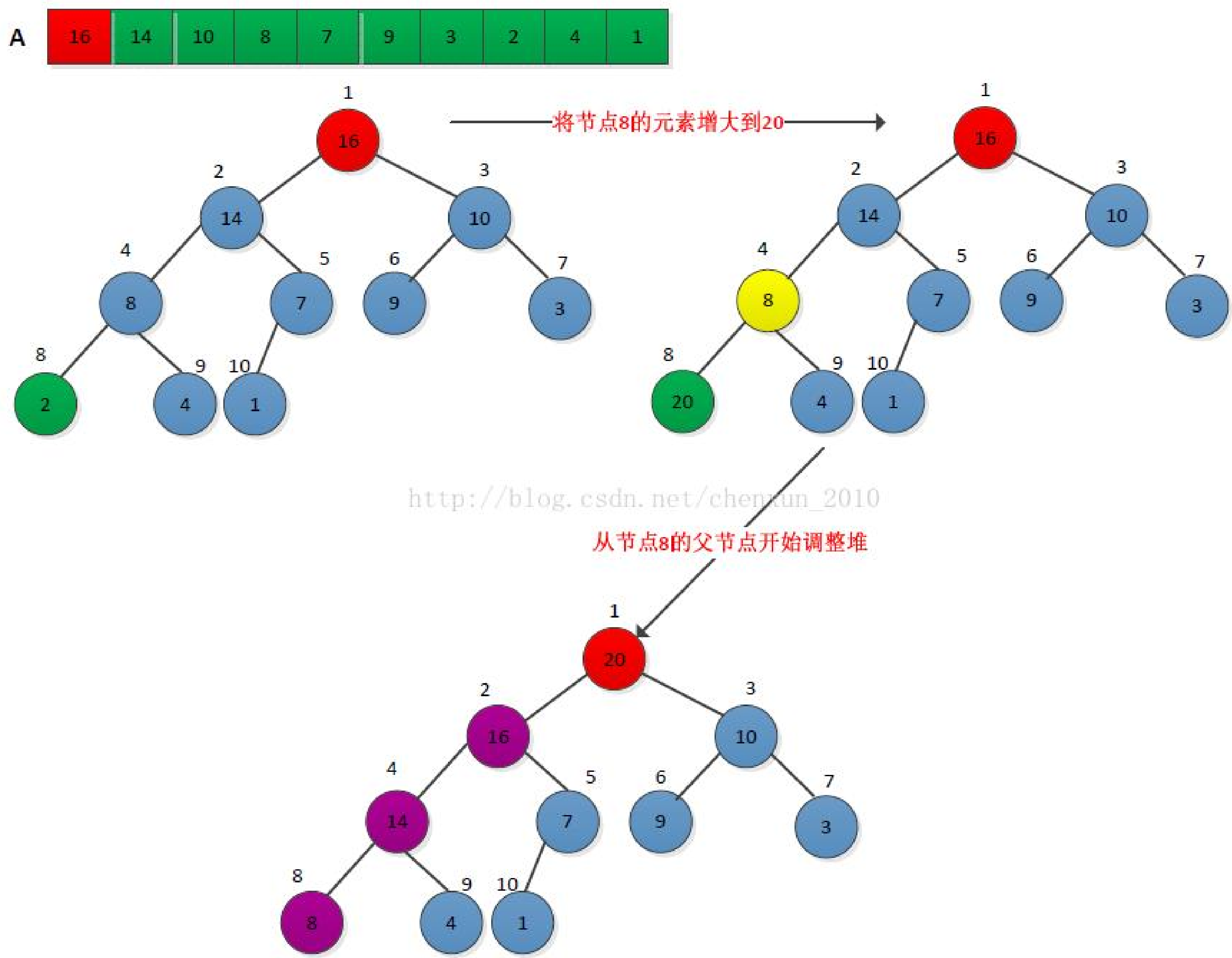
2.3. 6 优先队列–删除操作



2.3. 6 优先队列–删除操作

```
HEAD_EXTRACT_MAX(A)
 if heap_size[A]<1
   ther error
 max = A[1]
 A[1] = A[heap_size[A]];
 heap_size[A] = heap_size[A]-1
 adjust_max_heap(A,1)
 return MAX
```

2.3. 6 优先队列–元素值增加操作



2.3. 6 优先队列–元素值增加操作

```
HEAP_INCREASE_KEY(A,i,key)
   if key < A[i]
     then error
   A[i] = key
  while i>1 && A[PARENT(i)] <A[i]
       do exchange A[i] <-> A[PARENT(i)]
       i = PARENT(i)
```

2.3. 6 优先队列–插入元素操作

```
MAX-HEAP-INSERT(A, key)
    if A.heap_size = A.length
        error "heap is full"
    A.heap_size = A.heap_size + 1
    A[A.heap_size] = -∞
    MAX-HEAP-INCREASE-KEY(A, A.heap_size, key)
```

提纲

2.1 堆

2.2 散列表

提纲

2.1.1 概述

2.1.2 直接寻址

2.1.3 散列表

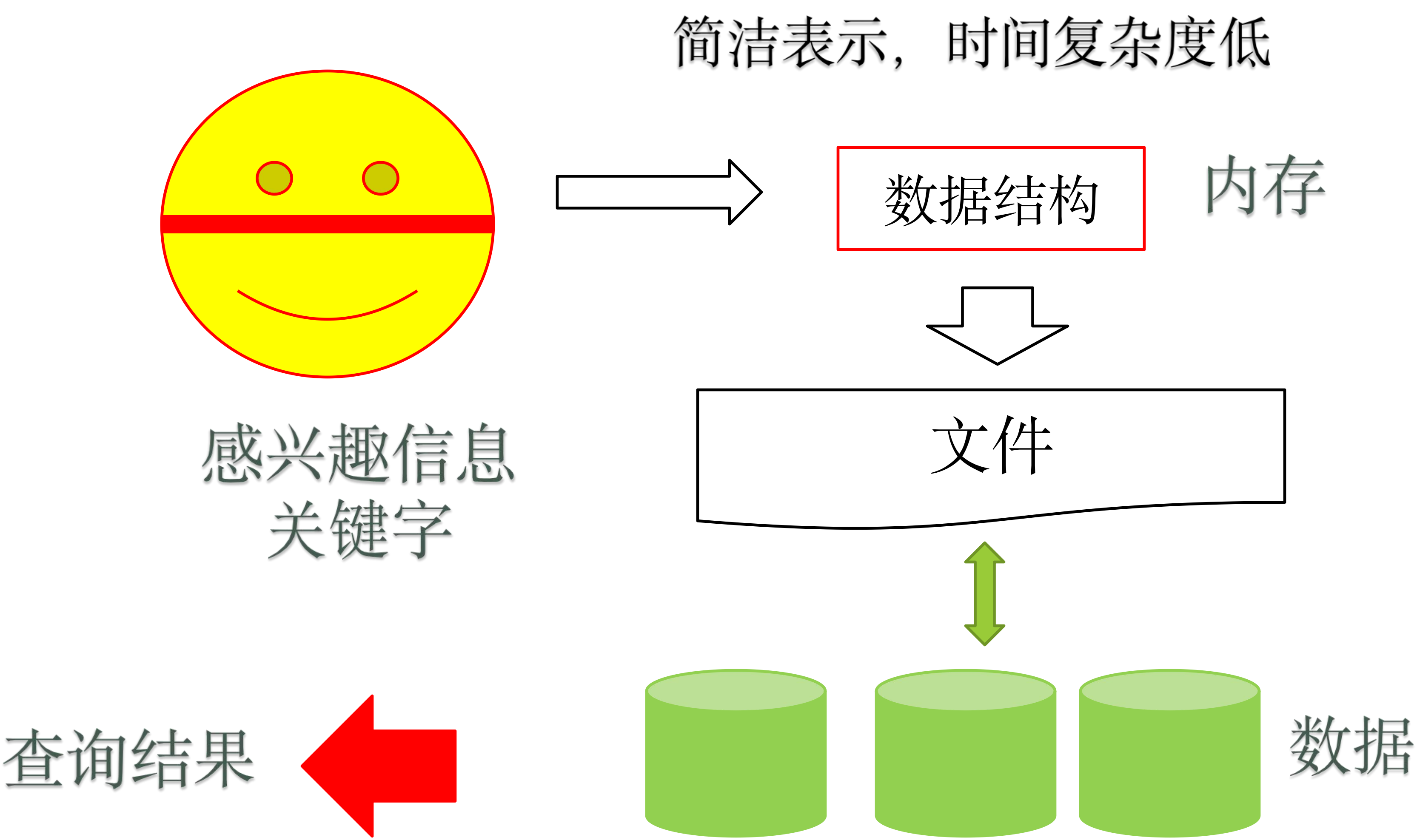
2.1.4 散列函数

2.1.5 开放寻址法

2.1.6 Cuckoo Hash 布谷鸟哈希

2.1.7 布鲁姆过滤器

2.1.1概述



提纲

2.1.1 概述

2.1.2 直接寻址

2.1.3 散列表

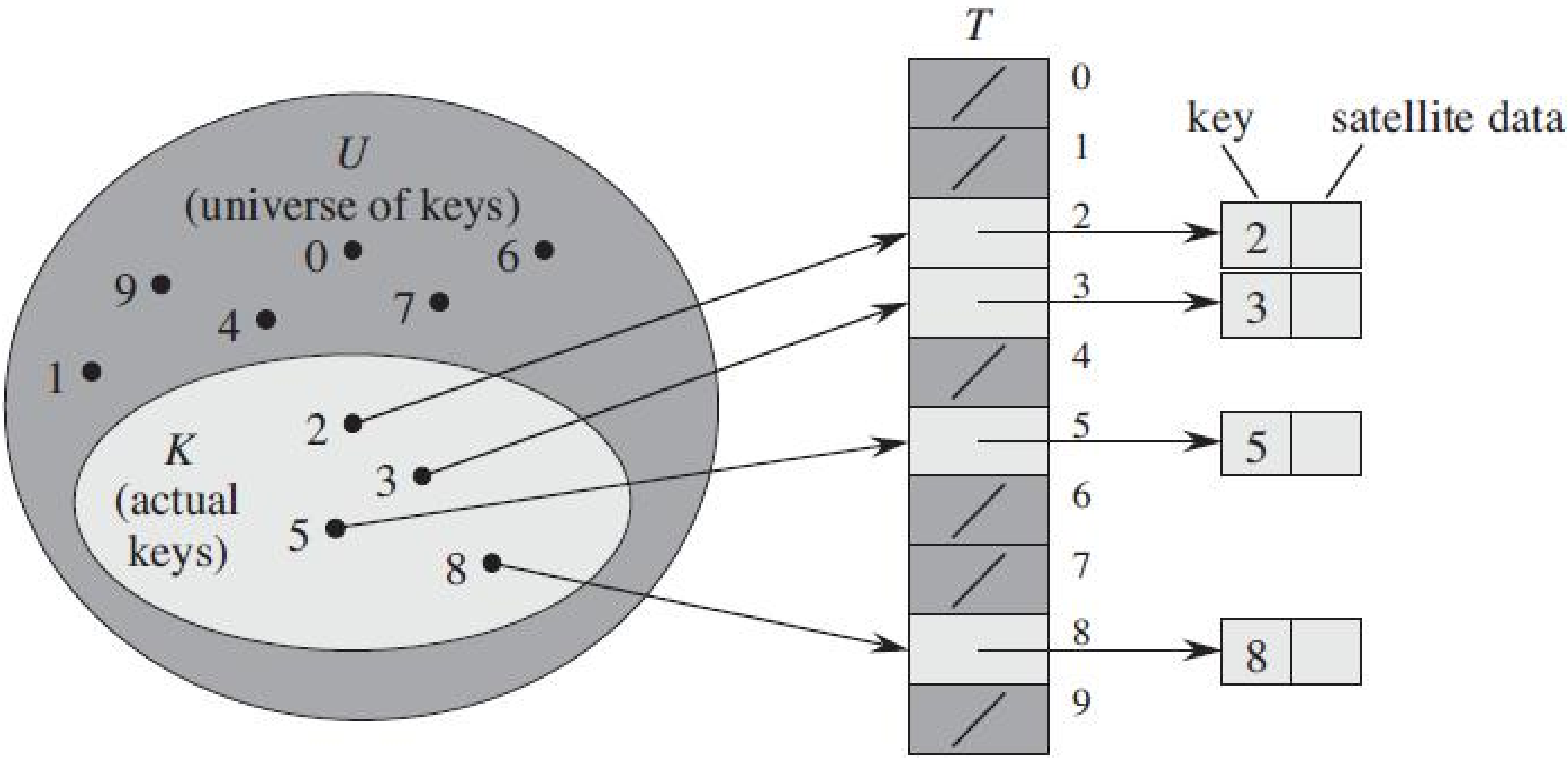
2.1.4 散列函数

2.1.5 开放寻址法

2.1.6 Cuckoo Hash 布谷鸟哈希

2.1.7 布鲁姆过滤器

2.1.2 直接寻址表


T
0
1
2
3
4
5
6
7
8
9
key
satellite data
2
3
5
8
U
(universe of keys)
0
6
9
7
4
1
K
(actual
keys)
2
3
5
8

2.1.2 直接寻址表

DIRECT-ADDRESS-SEARCH(T, k)

1 **return** $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

1 $T[x.key] = x$

DIRECT-ADDRESS-DELETE(T, x)

1 $T[x.key] = \text{NIL}$

2.1.2 直接寻址表

- 举例

key

年龄	人数
1岁	1432
2岁	2318
…	…
100岁	15

T(key-1)

存储地址
0
1
…
99

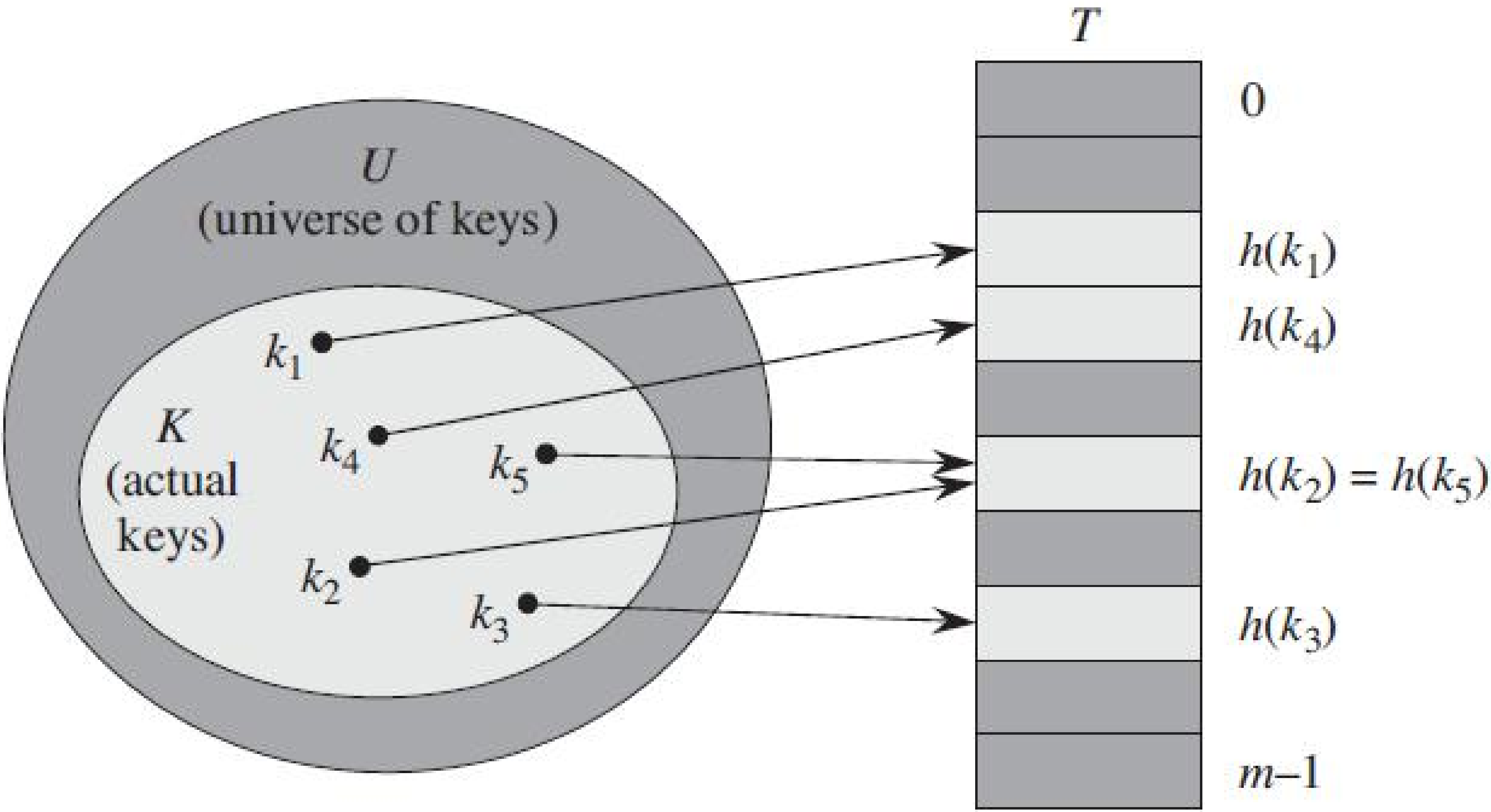
2.1.2 直接寻址表

直接寻址的缺点

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。

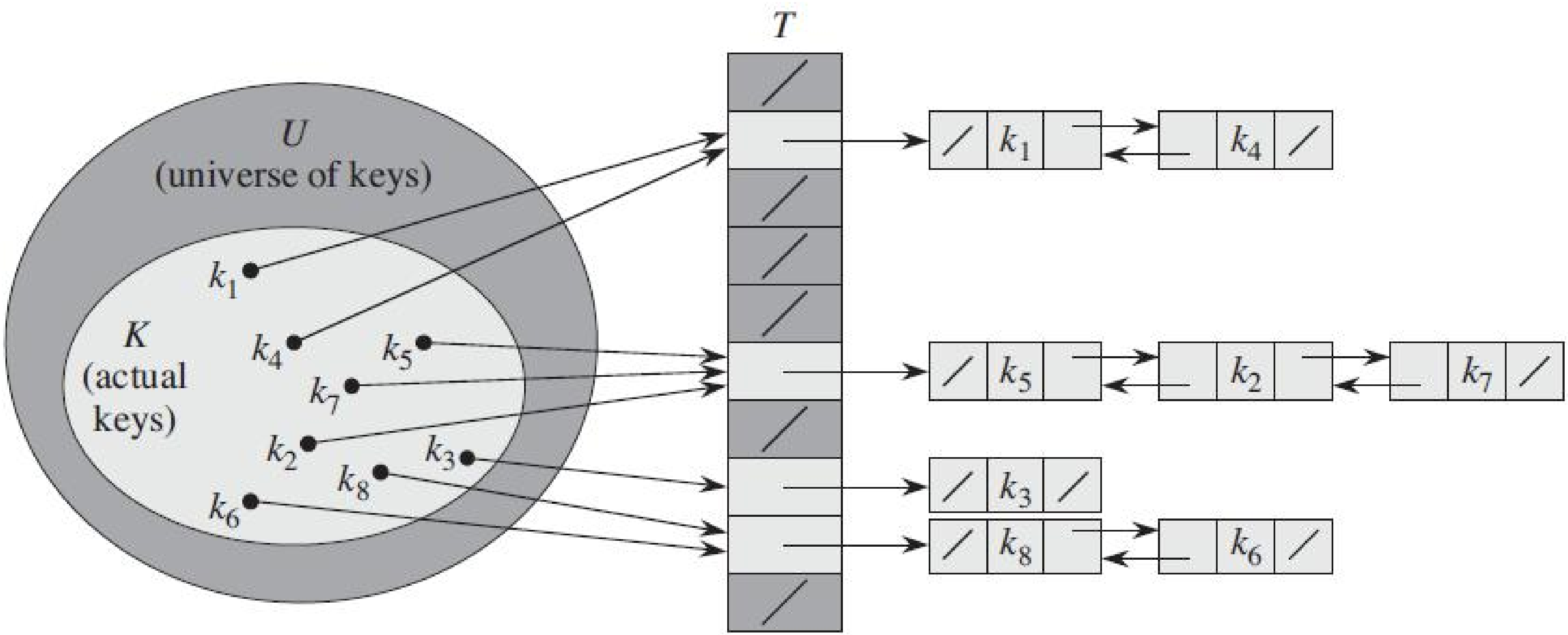
提纲

2.1.3 散列表



T
U
(universe of keys)
K
(actual keys)
k_1
k_4
k_5
k_2
k_3
0
$h(k_1)$
$h(k_4)$
$h(k_2) = h(k_5)$
$h(k_3)$
$m-1$

2.1.3 散列表–链接法（链地址法）



2.1.3 散列表–链接法（链地址法）

CHAINED-HASH-INSERT(T, x)

insert x at the head of list $T[h(x.key)]$

CHAINED-HASH-SEARCH(T, k)

search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE(T, x)

delete x from the list $T[h(x.key)]$

2.1.3 散列表–链接法（链地址法）

链接法散列的分析（算法导论）

①一次不成功查找平均时间分析

　　在查找不成功的情况下，我们需要遍历链表T[j]的每一个元素，而链表T[j]的长度是α，因此需要时间O(α)，加上索引到T(j)的时间O(1)，总时间为Θ (1 + α)。

② 一次成功查找平均时间分析

　　在查找成功的情况下，我们无法准确知道遍历到链表T[j]的何处停止，因此我们只能讨论平均情况。平均时间都为Θ(1+α)

2.1.3 散列表–链接法（链地址法）

链接法散列的分析（数据结构）

表 9.2 不同处理冲突的平均查找长度

处理冲突的方法	平均查找长度	
	查找成功时	查找不成功时
线性探测法	$S_{nl} \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$U_{nl} \approx \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$
二次探测法与双哈希法	$S_{nr} \approx -\frac{1}{\alpha}\ln(1-\alpha)$	$U_{nr} \approx \frac{1}{1-\alpha}$
链地址法	$S_{nc} \approx 1+\frac{\alpha}{2}$	$U_{nc} \approx \alpha+e^{-\alpha}$

2.1

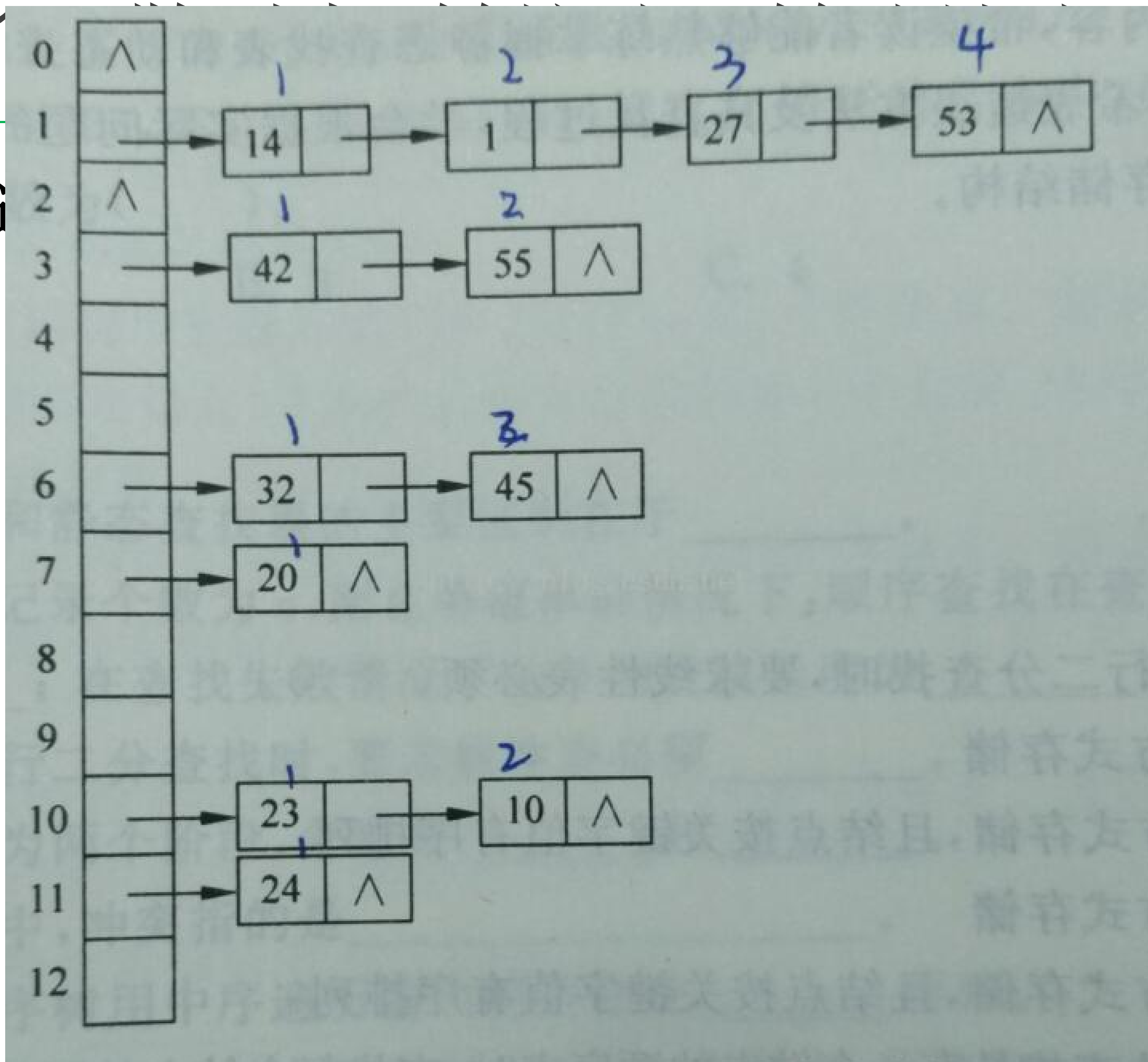


图 9.24　用拉链法解决冲突时的哈希表

提纲

2.1.1 概述

2.1.2 直接寻址

2.1.3 散列表

2.1.4 散列函数

2.1.5 开放寻址法

2.1.6 Cuckoo Hash 布谷鸟哈希

2.1.7 布鲁姆过滤器

2.1.4 散列函数

The 'magic' of hash functions

PHENOMENAL COSMIC POWERS!! → itty bitty living space

惊人的、广大无边的、强势（拉丁神）

微人模式、小的居住空间（拉丁神灯）

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$k_i \in K$，$i=1,2,\ldots,n$，有$0 \le H(k_i) \le m-1$；

2. 对任意$k_i \in K$，$H(k_i)$取[0,m-1]中任一值的概率相等。

2.1.4.1 散列函数:除法散列法

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

A	B	C
十进制(k)	二进制(k)	h(k)(其中m=8)
20	10100	4
98	1100010	2
204	11001100	4
71	1000111	7
67	1000011	3
1234	10011010010	2

2.1.4.1 除法散列法的不足

Problems with division method

对于规则性的键值集合

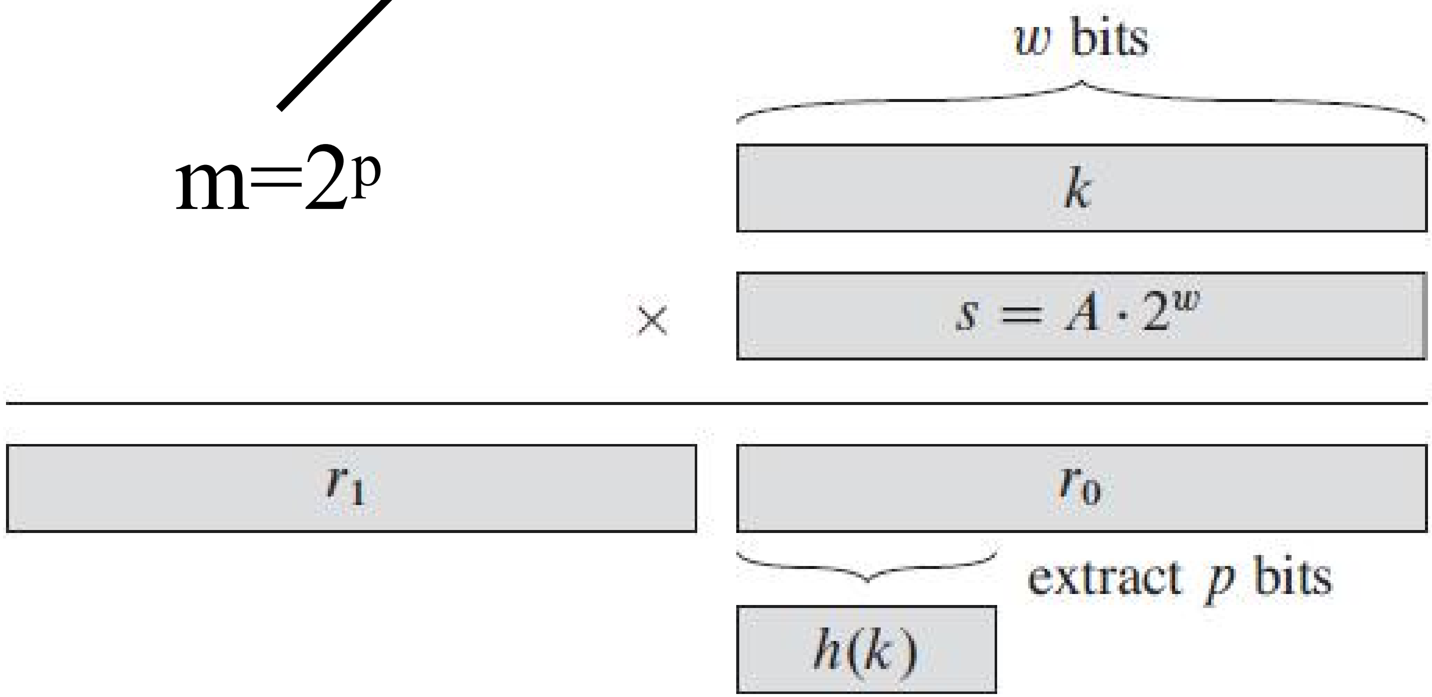
- Regularity
  - Suppose keys are x, 2x, 3x, 4x,…. 假如x与m有个公约数d
  - Suppose x and chosen m have common divisor d
  - Then only use 1/d fraction of table 那么我们将只能有效利用到表格1/d部分空间
    - x series cycles back, leaving d-1 out of d entries blank
    - E.g, m power of 2 and all keys are even, only use half
- So 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）

2.1.4.2 散列函数:乘法散列法

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

$m=2^p$



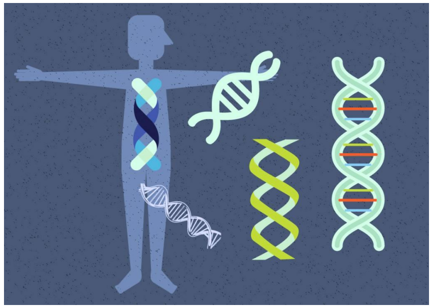
$$A \approx (\sqrt{5} - 1) / 2 = 0.618033988$$

2.1.4.3散列函数:散列值

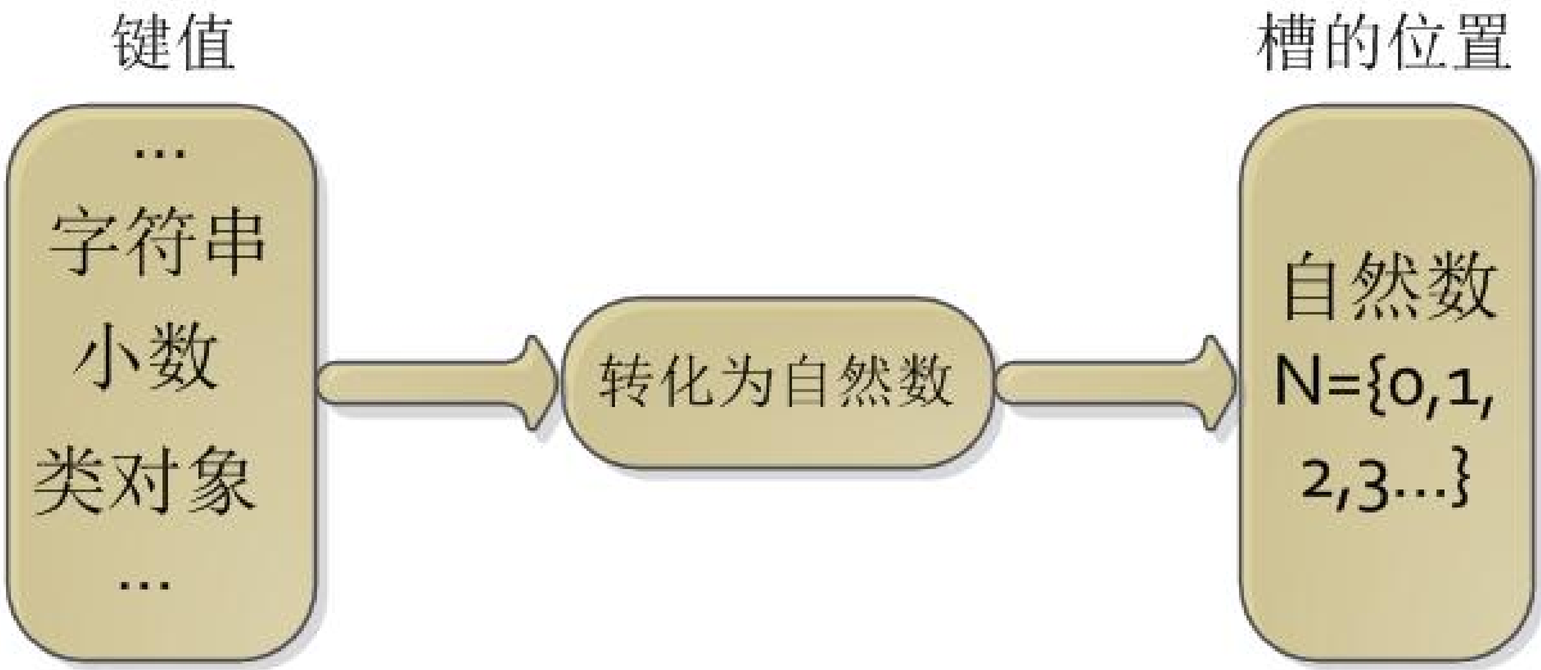
Key不是数字，是字符串呢?

Key=“abcdef”

Hash值



2.1.4.3散列函数:散列值



Java: hashcode
C++: ASCII码相加

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

提纲

2.1.1 概述

2.1.2 直接寻址

2.1.3 散列表

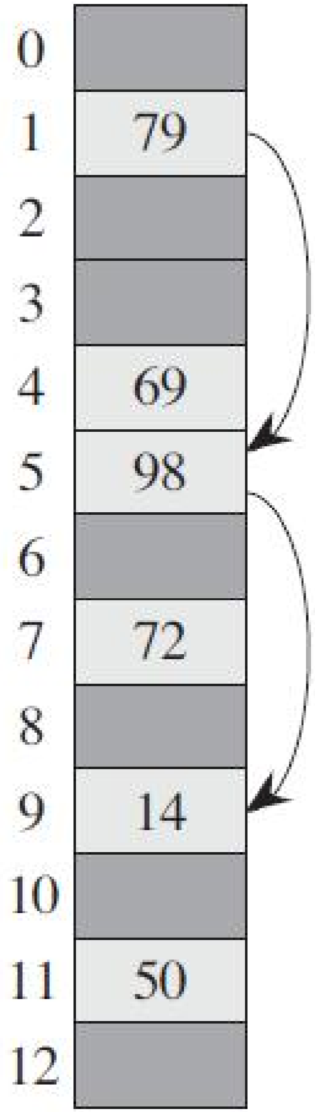
2.1.4 散列函数

2.1.5 开放寻址法

2.1.6 Cuckoo Hash 布谷鸟哈希

2.1.7 布鲁姆过滤器

2.1.5 开放寻址法



2.1.5 开放寻址法

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

2.1.5 开放寻址法

```
HASH-SEARCH(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == k
        return j
    i = i + 1
until T[j] == NIL or i == m
return NIL
```

2.1.5 开放寻址法

```
HASH-DELETE(T,k)
    i=0
    repeat
            j = h(k,i)
            if ( T[j] == k )
                T[j] = DELETED
                return
            i = i + 1
    until T[j] == NIL or i == m
    return
```

2.1.5 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"
```

2.1.5 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

2.1.5 开放寻址法–线性探查

$$h(k, i) = (h'(k) + i) \text{ mode } m$$

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

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

地址 / 操作	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

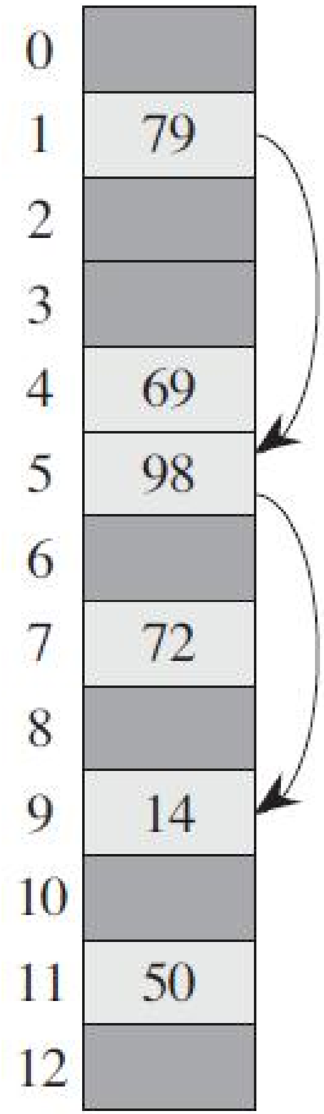
2.1.5 开放寻址法-二次探查

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

2.1.5 开放寻址法–双重散列

$$h(k, i) = (h1(k) + ih2(k)) \bmod m$$

2.1.5 开放寻址法



提纲

2.1.1 概述

2.1.2 直接寻址

2.1.3 散列表

2.1.4 散列函数

2.1.5 开放寻址法

2.1.6 Cuckoo Hash 布谷鸟哈希

2.1.7 布鲁姆过滤器

2.1.6 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。

2.1.6 Cuckoo Hash 布谷鸟哈希

操作：

1）对key值hash，生成两个hash key值，hashk1和hashk2, 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。

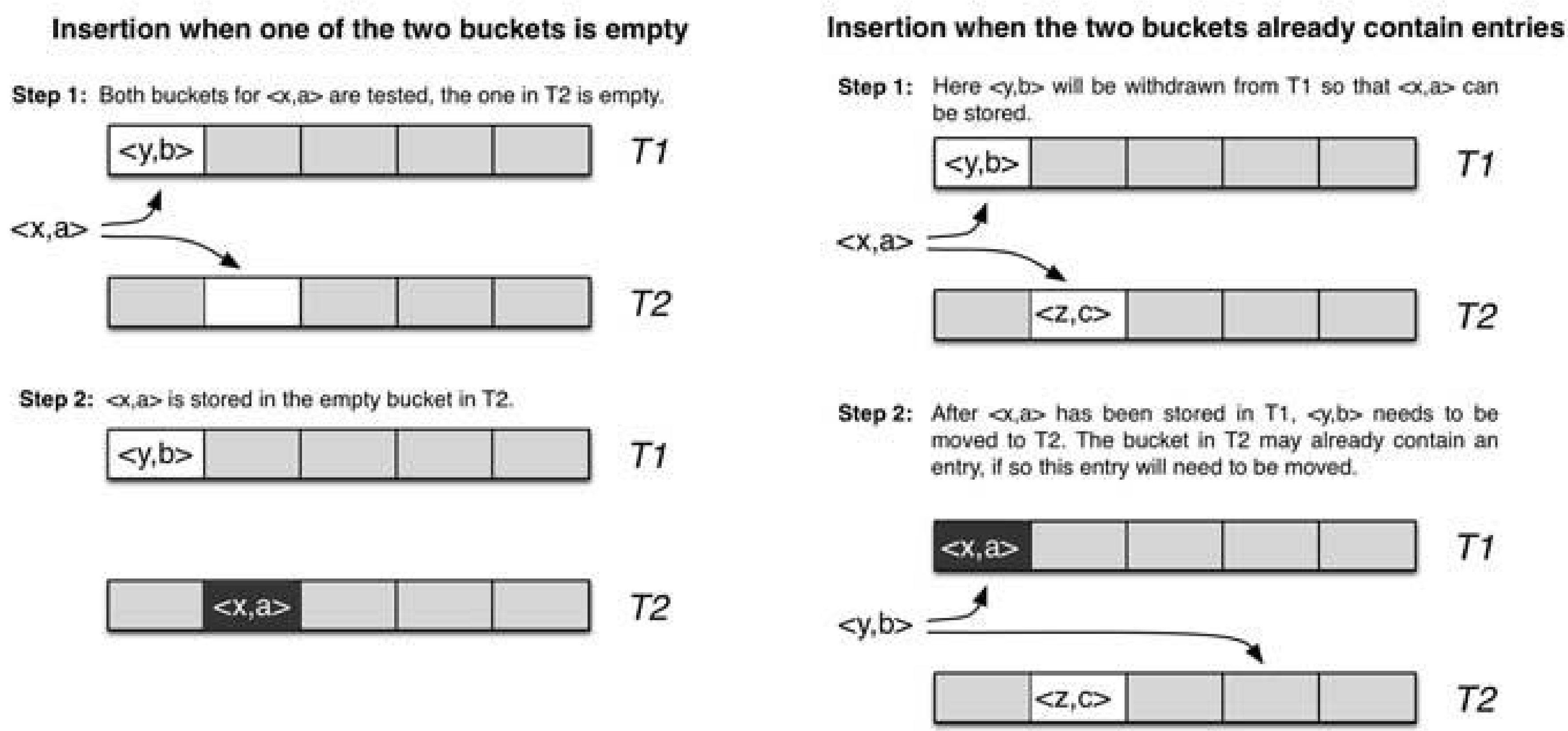
2）否则，任选一个位置，把key值插入，把已经在那个位置的key值踢出来。

3）被踢出来的key值，需要重新插入，直到没有key被踢出为止。

2.1.6 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。

2.1.6 Cuckoo Hash 布谷鸟哈希



2.1.6 Cuckoo Hash 布谷鸟哈希

其他:

1）Cockoo 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）Cockoo 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。

提纲

2.1.1 概述

2.1.2 直接寻址

2.1.3 散列表

2.1.4 散列函数

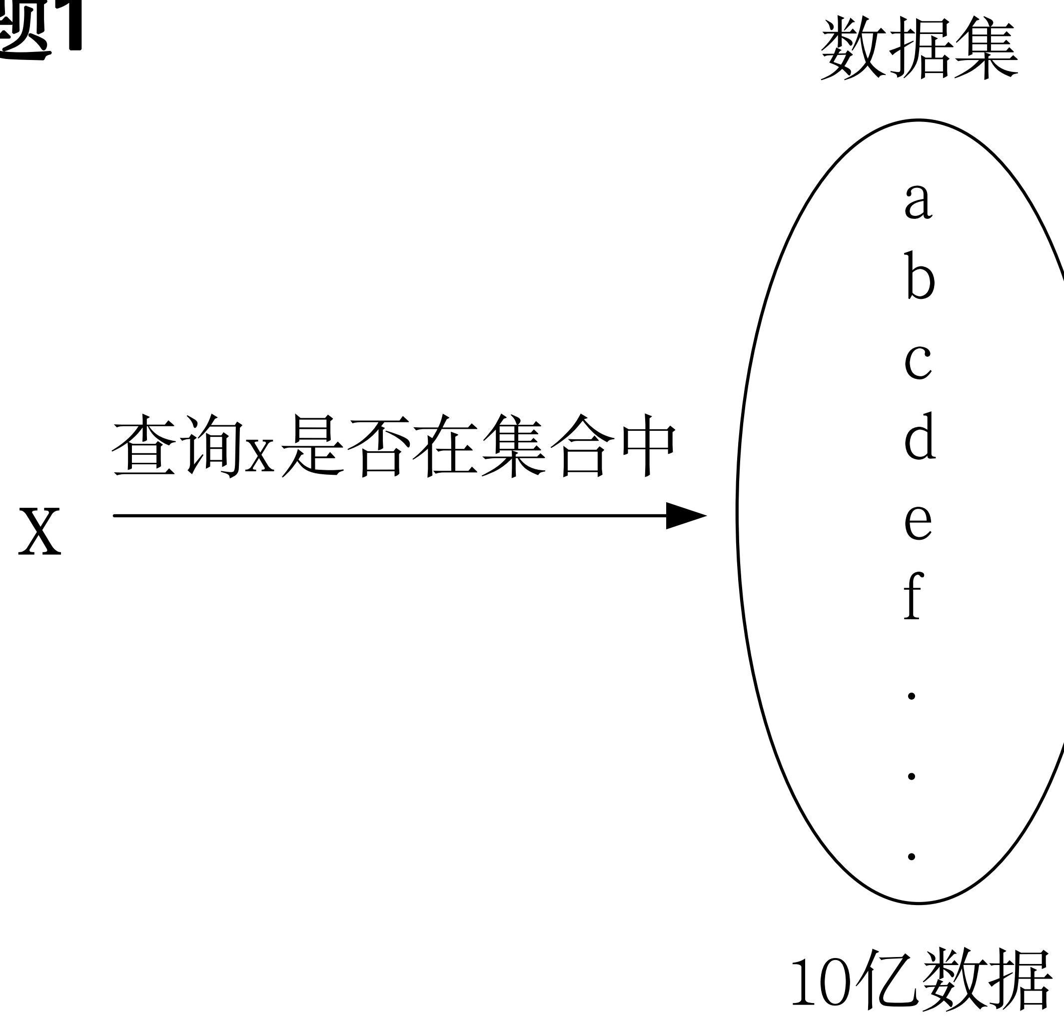
2.1.5 开放寻址法

2.1.6 Cuckoo Hash 布谷鸟哈希

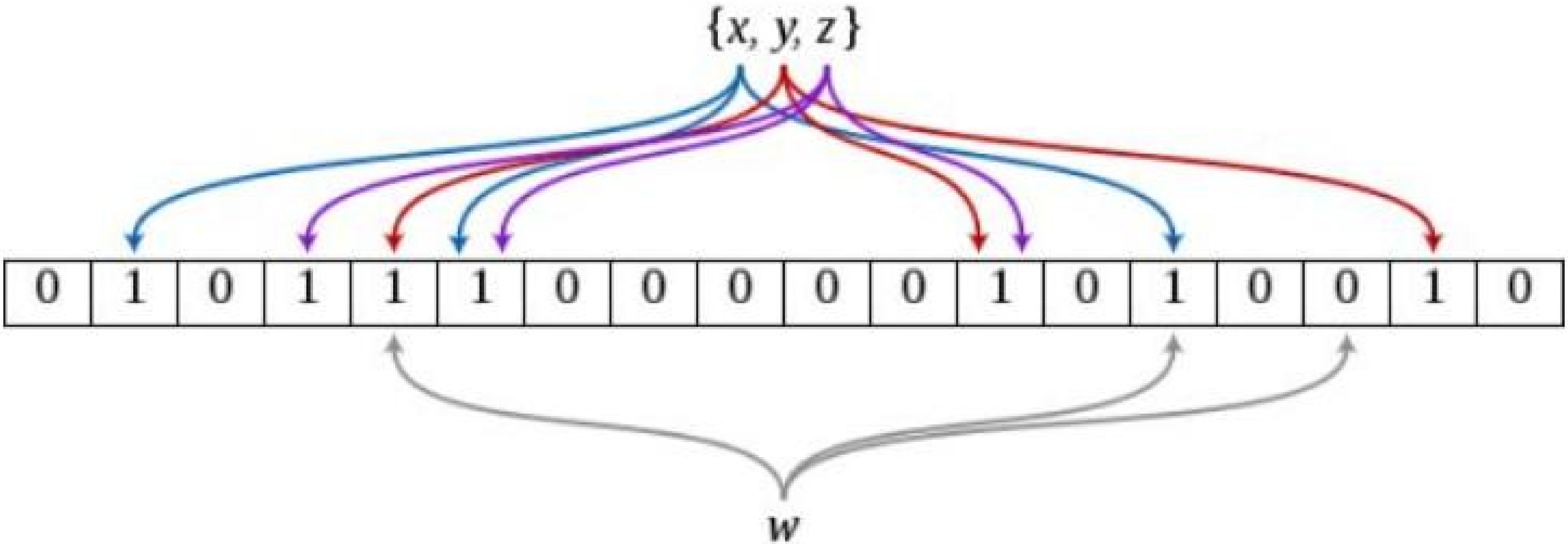
2.1.7 布鲁姆过滤器

2.1.6 布鲁姆过滤器（bloomfilter）

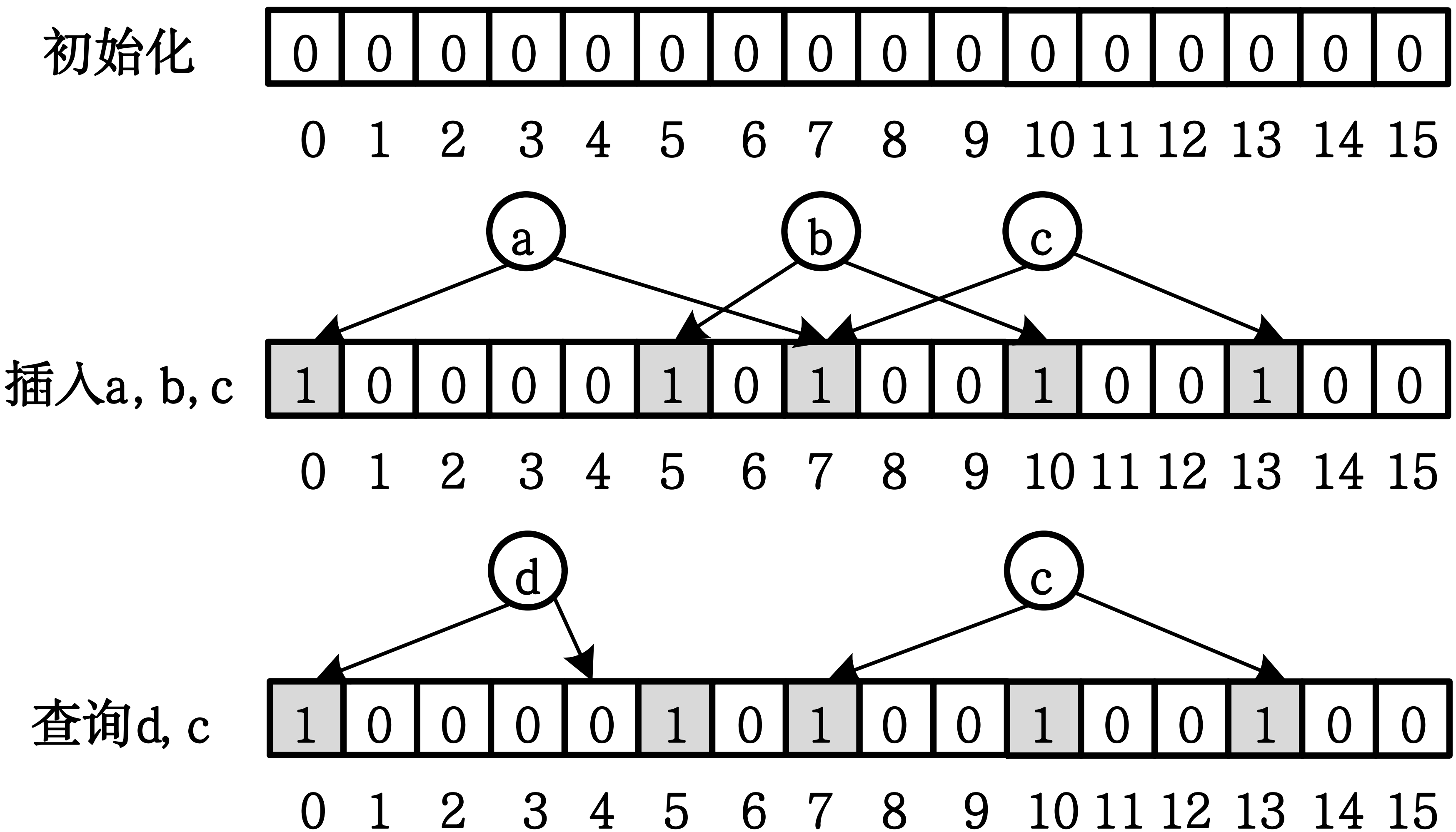
问题1



2.1.6 布鲁姆过滤器（bloomfilter）



2.1.6 布鲁姆过滤器(bloom filter)



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。

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

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

2.1.6 布鲁姆过滤器

误判率分析（假阳性）

(3) 最优分析

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

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