第12章　优先队列

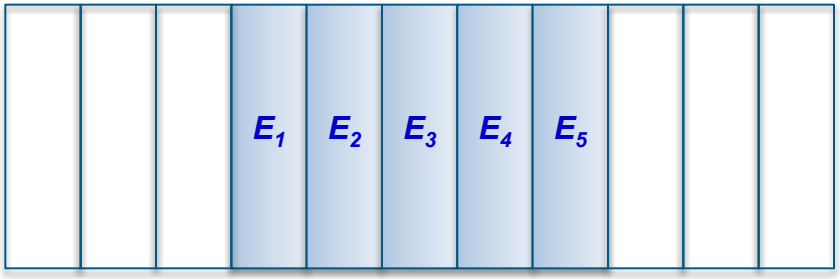
计算机学院

主要内容

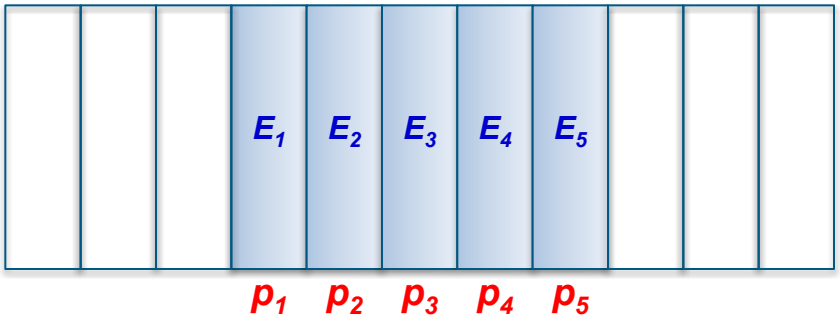
- 优先队列ADT
- 堆及堆排序
- 霍夫曼编码

队列→优先队列

- 队列是一种一维表结构
  - 按照入队次序出队



- 优先队列建立在树形结构上
  - 按照优先级出队



优先队列

- 是0个或多个元素的集合
  - 每个元素都有一个优先权或值
  - 两个元素可以有相同的权值
  - 其上的操作包括：插入、删除、查找
  - 最大优先队列，最小优先队列

优先队列ADT---最大

抽象数据类型*MaxPriorityQueue*{

实例

有限的元素集合，每个元素都有一个优先级

操作

Create()：创建一个空的优先队列

Size()：返回队列中的元素数目

Max()：返回具有最大优先级的元素 → 查

Insert(*x*)：将*x*插入队列 → 增

DeleteMax(*x*)：从队列中删除具有最大优先级的元素，并将该元素返回至*x* → 删

}

例：机器服务收费(天河一号、南开之星)

- 用户付费相同，但所需服务时间不同
  - 用户→最小优先队列，优先级——时间
  - 新用户→加入优先队列
  - 最大收益：机器可用→最少服务时间用户获得服务
- 用户所需时间相同，愿意支付费用不同
  - 支付费用作为优先级
  - 机器可用→交费最多的用户最先得到服务

例：排队

- 东方之珠K歌
  - 普通客户
  - 银卡客户
  - 金卡客户

- 交通银行办理业务
  - 普通客户
  - VIP客户
  - 南开大学财务处

优先队列的思考

- 在某种程度上类似于排序列表
  - 优先级反映元素的“大小”
  - 给定某最大/最小优先队列，求其出队次序类似于将某列表按从大到小/从小到大排序

- **优先队列的线性表表示法正是基于这一思想**

线性表描述最大优先队列

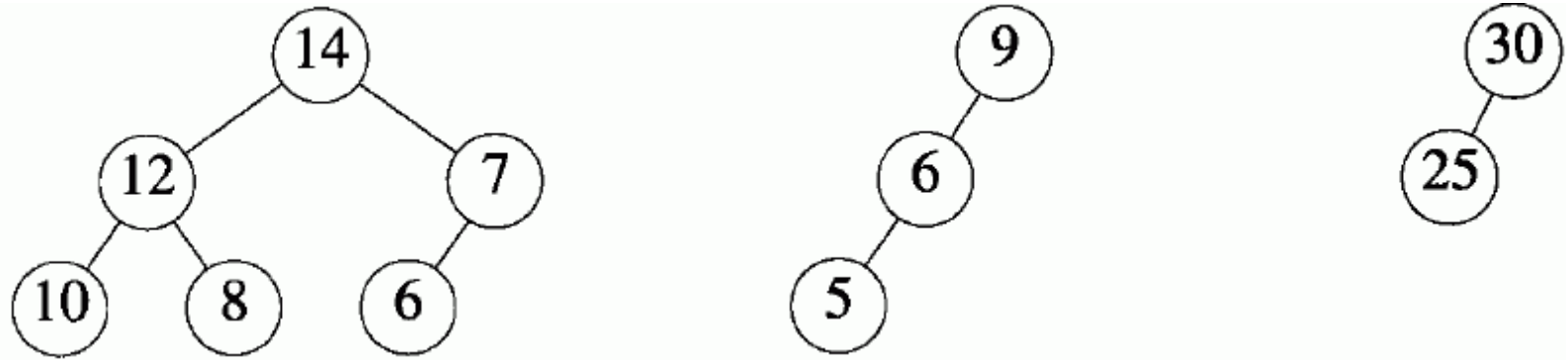
- 无序线性表——最简单
  - 公式化描述
    - 插入操作：表尾，Θ(1)
    - 删除操作：查找最大优先级元素， Θ(n)
  - 链表描述
    - 插入操作：链头， Θ(1)；删除操作，Θ(n)
- 有序列表
  - 公式化：递增，链表：递减
  - 删除：Θ(1)；插入：Θ(n)

主要内容

- 优先队列ADT
- **堆及堆排序**
- 霍夫曼编码

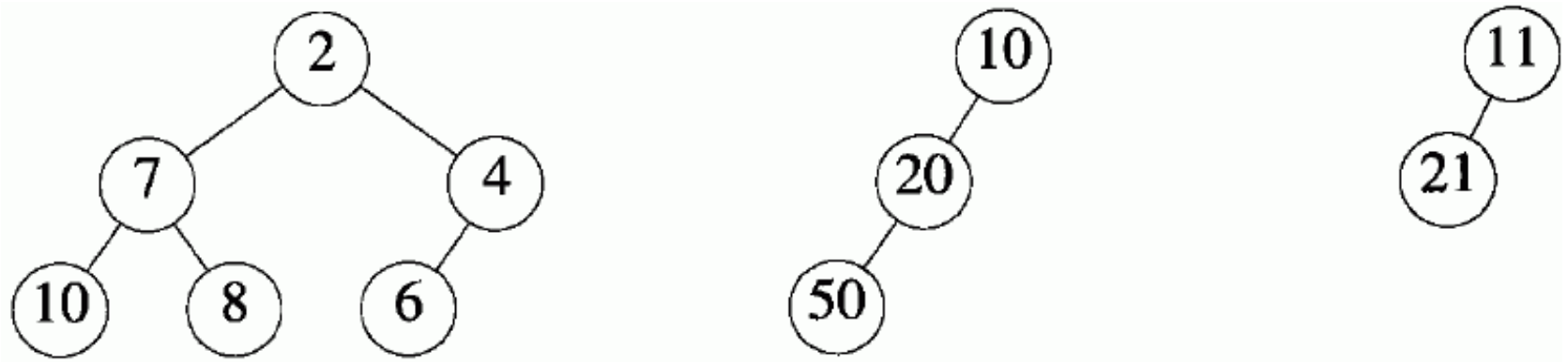
H1. 堆及堆排序

- **最大树**：每个节点的值都大于或等于其子节点（若存在）值的树



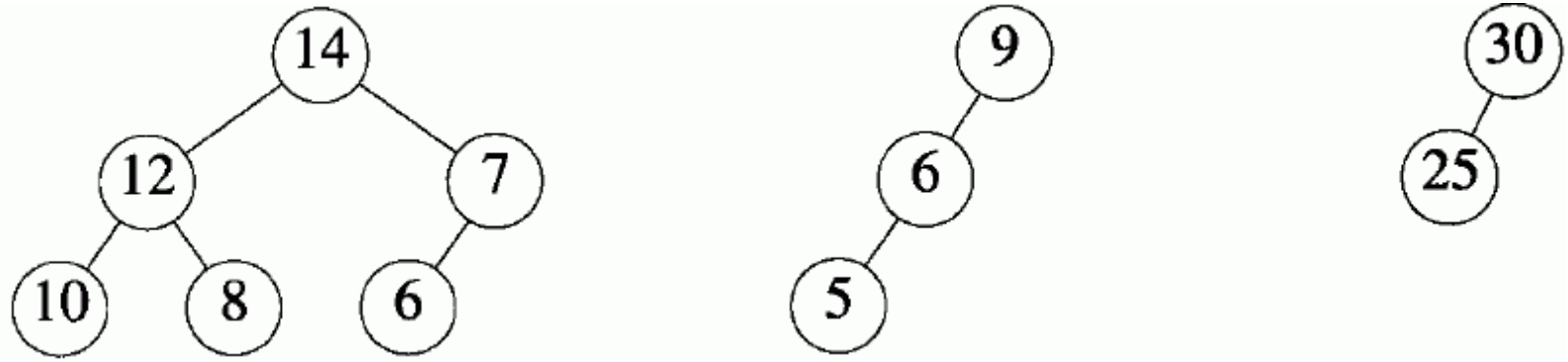
最小树示例

- **最小树**：每个节点的值都小于或等于其子节点（若存在）值的树



堆的定义

- **最大堆：**
  - 是一棵最大树
  - 同时是一棵完全二叉树



是　　不是　　是

堆的描述

- 特殊的完全二叉树→一维数组有效描述
- 父子节点位置关系——简单公式
  - 堆节点从1开始编号，则节点i的左孩子是2i、右孩子是2i+1
  - 堆节点从0开始编号，则节点i的左孩子是2i+1、右孩子是2i+2
- n个元素，高度$\log_2(n+1)$

MaxHeap类

```
template<class T>
class MaxHeap {
  public:
    MaxHeap(int MaxHeapSize = 10);
    ~MaxHeap() {delete [] heap;}
    int Size() const {return CurrentSize;}
    T Max() {          //查
       if (CurrentSize == 0)
            throw OutOfBounds();
       return heap[1];
     }
```

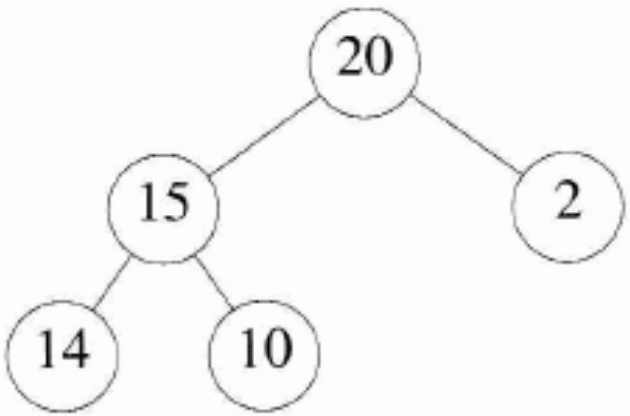
MaxHeap类

```
    MaxHeap<T>& Insert(const T& x); //增
    MaxHeap<T>& DeleteMax(T& x);   //删
    void Initialize(T a[], int size, int ArraySize);
private:
    int CurrentSize, MaxSize;
    T *heap;  // element array
};
```

构造函数

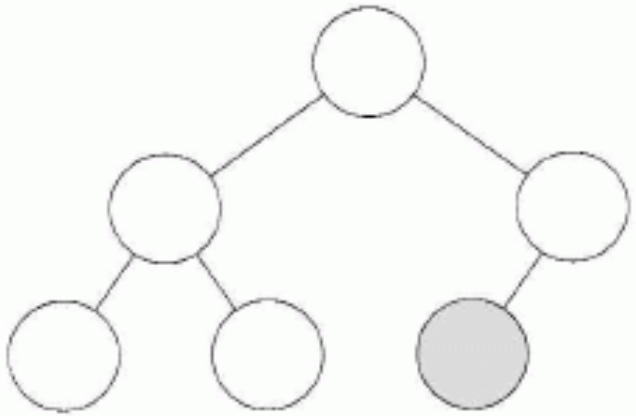
```
template<class T>
MaxHeap<T>::MaxHeap(int MaxHeapSize)
{// Max heap constructor.
   MaxSize = MaxHeapSize;
   heap = new T[MaxSize+1];
   CurrentSize = 0;
}
```

最大堆的插入操作



完全二叉树
插入后必然形如右图

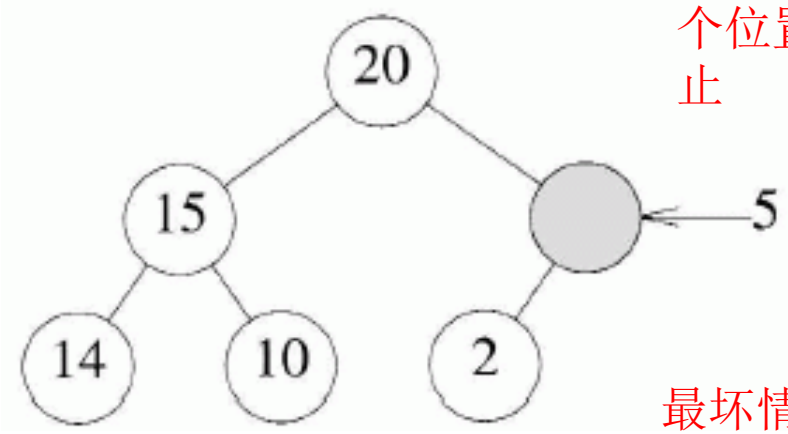
但新元素放在阴影位置，可能不符合堆的特性

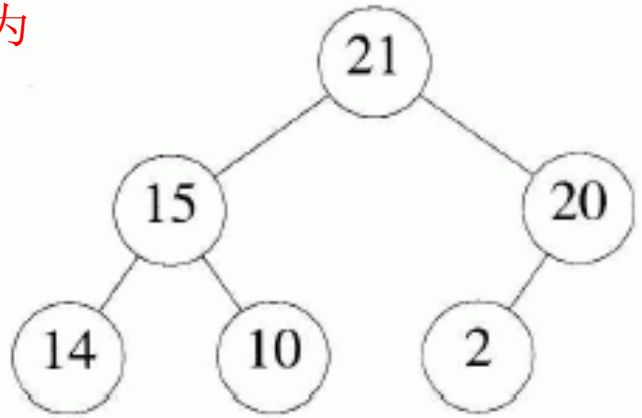


插入5

上移重整堆

新元素向上移动（与父节点交换）
重复此过程，直到到达某个位置时符合堆的特性为止





最坏情况$O(\log_2 n)$

插入函数

```
template<class T>
MaxHeap<T>& MaxHeap<T>::Insert(const
  T& x)
{// Insert x into the max heap.
   if (CurrentSize == MaxSize)
      throw NoMem(); // no space

   // 寻找新元素x的位置
   // i——初始为新叶节点的位置，逐层向上，寻找最
  终位置
   int i = ++CurrentSize;
```

插入函数（续）

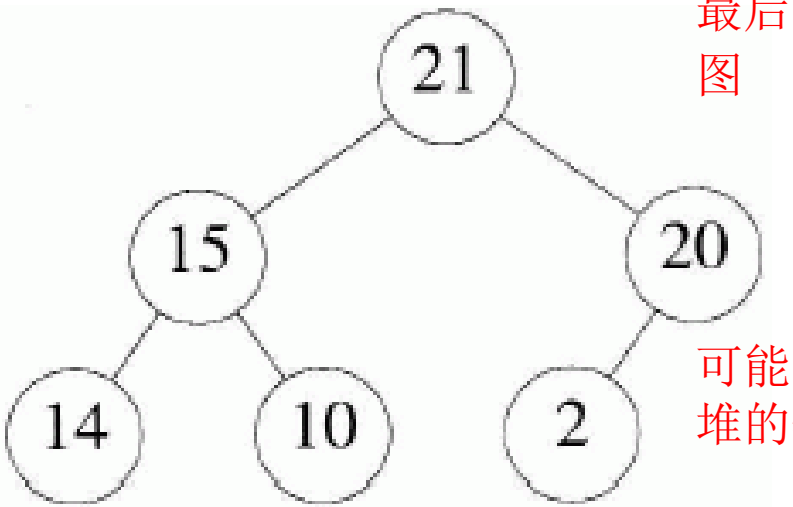
```
    while (i != 1 && x > heap[i/2]) {
     // i不是根节点，且其值大于父节点的值，需要继续调整
      heap[i] = heap[i/2]; // 父节点下降
      i /= 2;              // 继续向上，搜寻正确位置
      }

  heap[i] = x;
  return *this;
}
```

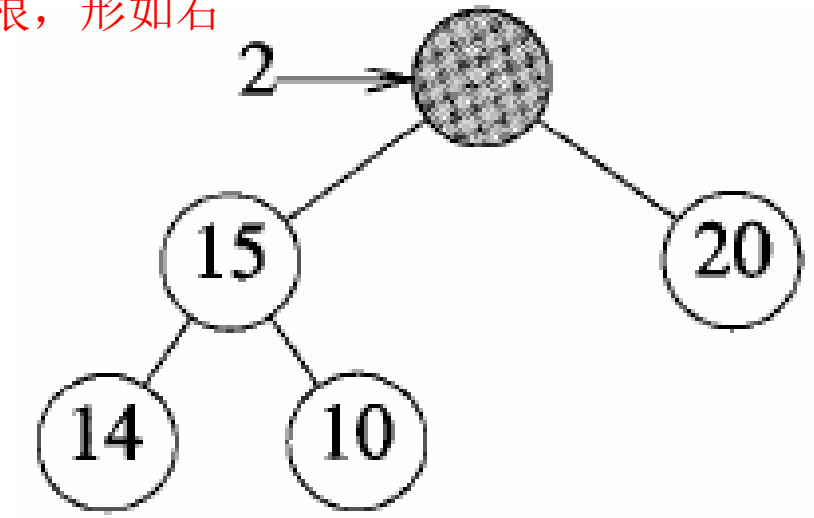
O(logn)

删除操作

删除根－最大优先级保持完全二叉树结构
最后一个元素→根，形如右图



可能不符合堆的特性

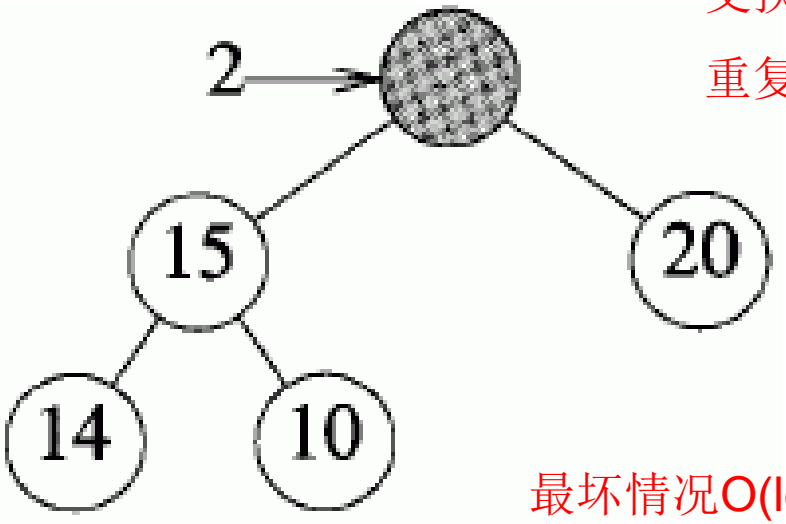


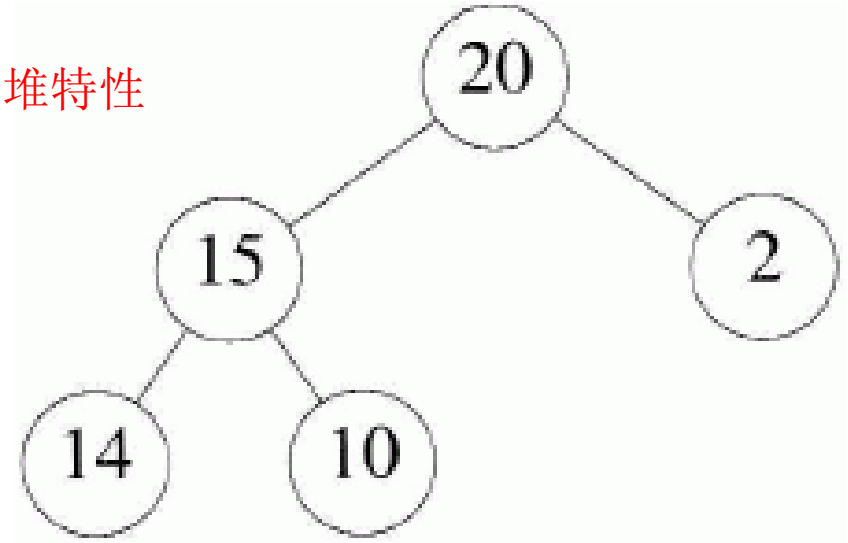
删除21

下降过程

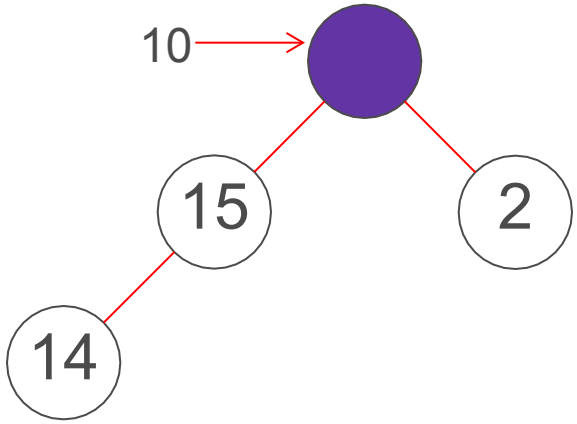
选取子节点中较大者与父节点交换

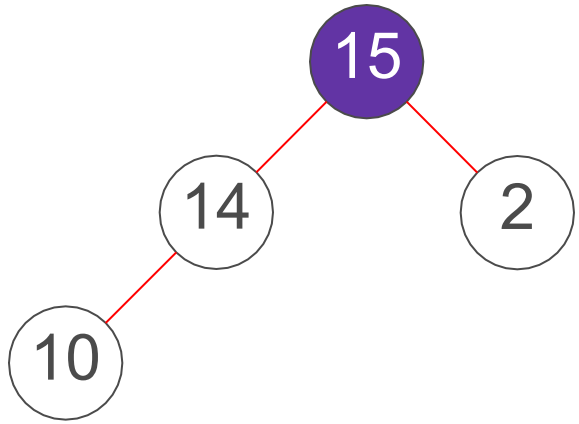
重复，直至符合堆特性





最坏情况$O(\log_2 n)$





删除函数

```
template<class T>
MaxHeap<T>& MaxHeap<T>::DeleteMax(T&
  x)
{// Set x to max element and delete max
  element from heap.
   // check if heap is empty
   if (CurrentSize == 0)
      throw OutOfBounds(); // empty

   x = heap[1]; // 删除最大元素
```

删除函数（续）

```
// 重整堆
T y = heap[CurrentSize--]; // 取最后一个节点，从根开始
重整

// find place for y starting at root
int i = 1,  // current node of heap
    ci = 2; // child of i

while (ci <= CurrentSize) {
   // 使ci指向i的两个孩子中较大者
   if (ci < CurrentSize && heap[ci] < heap[ci+1])
       ci++;
```

删除函数（续）

```
// y的值大于等于孩子节点吗？
    if (y >= heap[ci]) break;   // 是，i就是y的正确位置，
  退出

    // 否，需要继续向下，重整堆
    heap[i] = heap[ci]; // 大于父节点的孩子节点上升
    i = ci;             // 向下一层，继续搜索正确位置
    ci *= 2;
    }
  heap[i] = y;

  return *this;
}
```

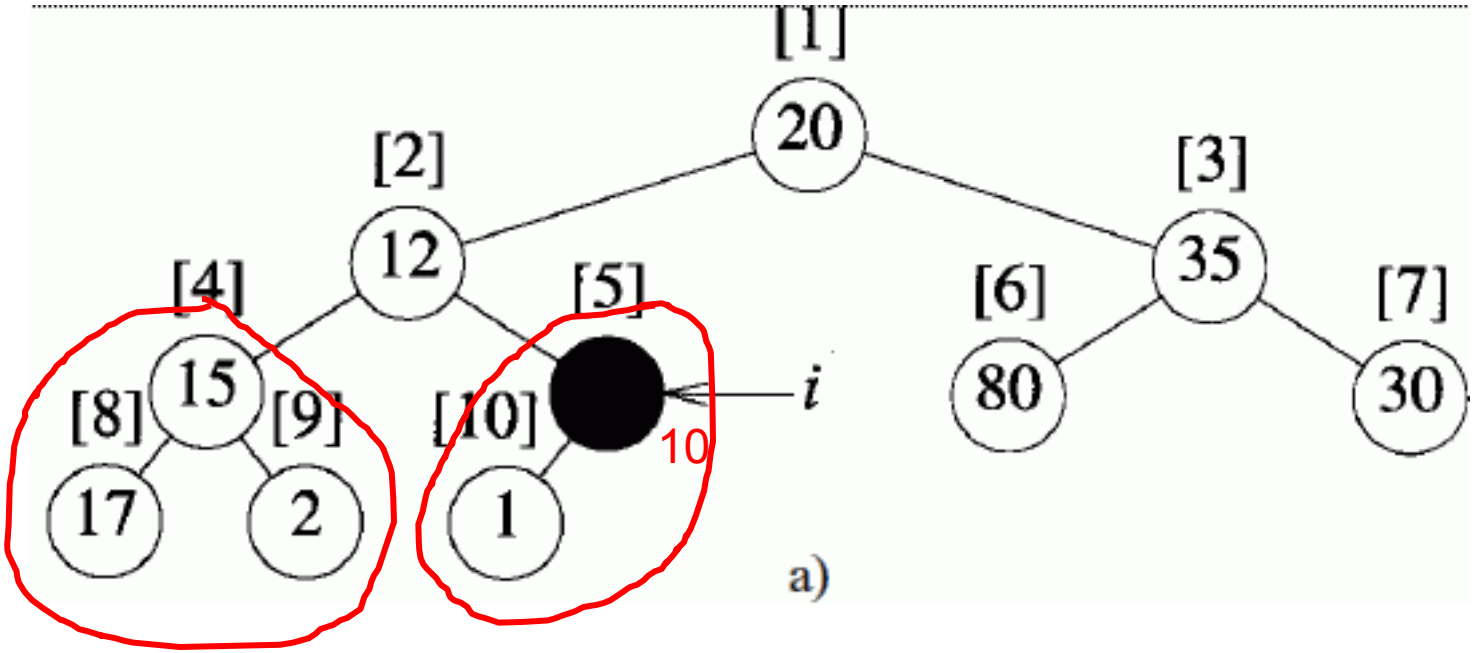
最大堆的创建

- n个元素——如何构成堆？
- 思路一：
  - 空堆，n次插入——O(nlogn)

最大堆的创建

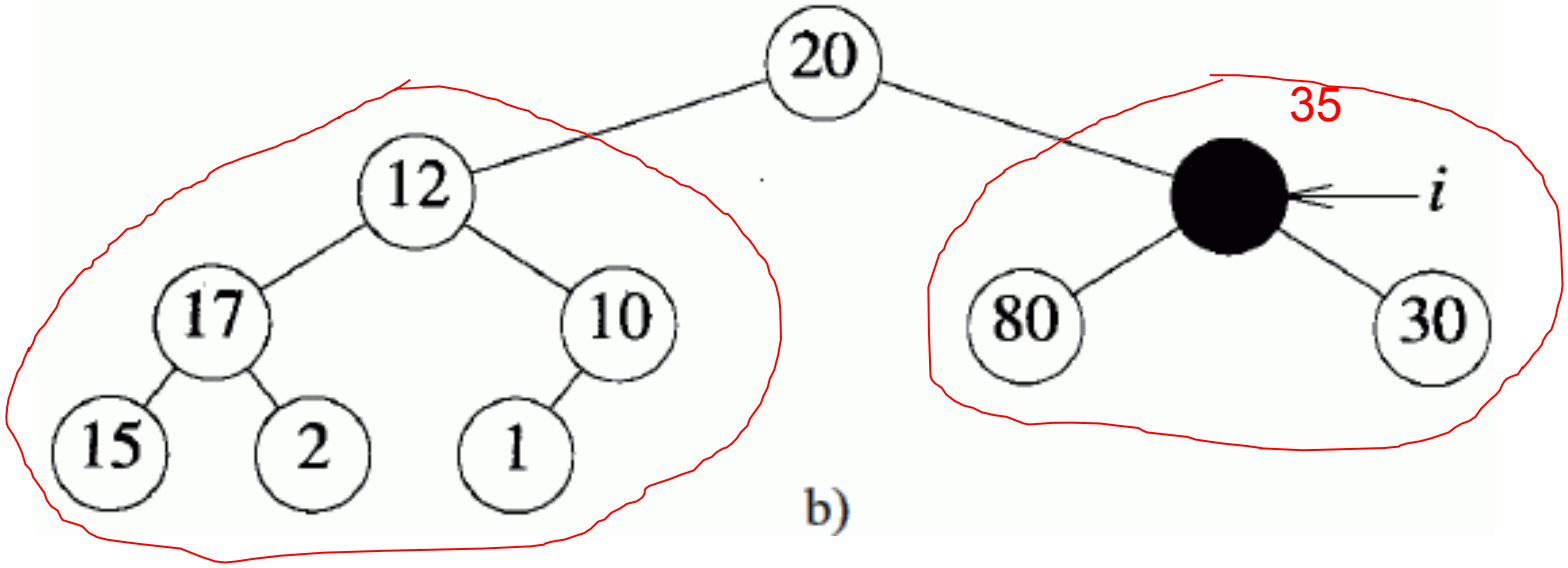
- n个元素——如何构成堆？
- 思路二：
  - 更好的方法，Θ(n)：n个元素构成完全二叉树，可能不是堆，整理它
    - 从最后一个内部节点（n/2）开始到根节点，将每个节点i的子树（完全二叉树）整理为堆
    - 重要特性——i的两个子树都已经是堆了
      - 刚开始的i直至上一层，其子树均为单节点，肯定是
      - 更高层节点，子树已经重整过——必然是堆！
    - ➔整理堆——下降过程！

建堆实例（思路二）



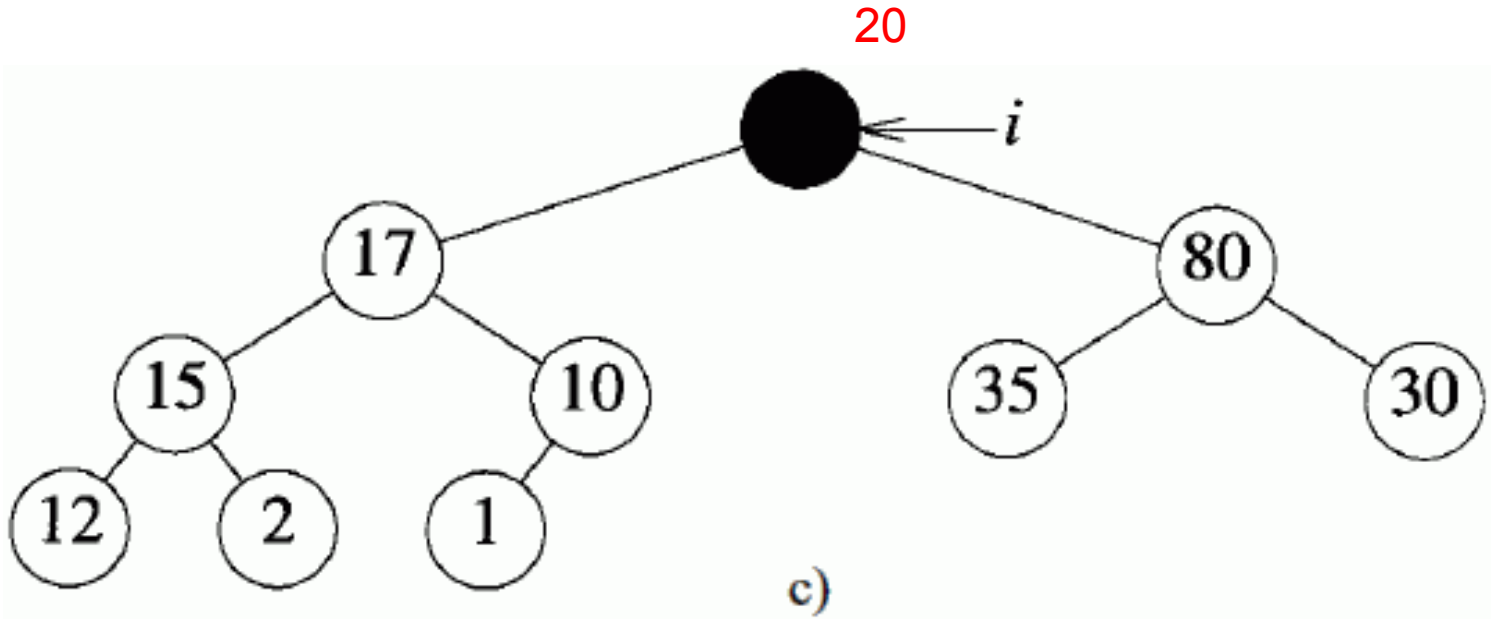
step1：先形成一棵完全二叉树，一般来说还不是堆
step2：从编号为i = 10/2的节点开始，检查以其为根的子树是否为堆？是！
step3：检查编号为4的节点子树是否为堆？不是！调整为堆！

建堆实例（续）



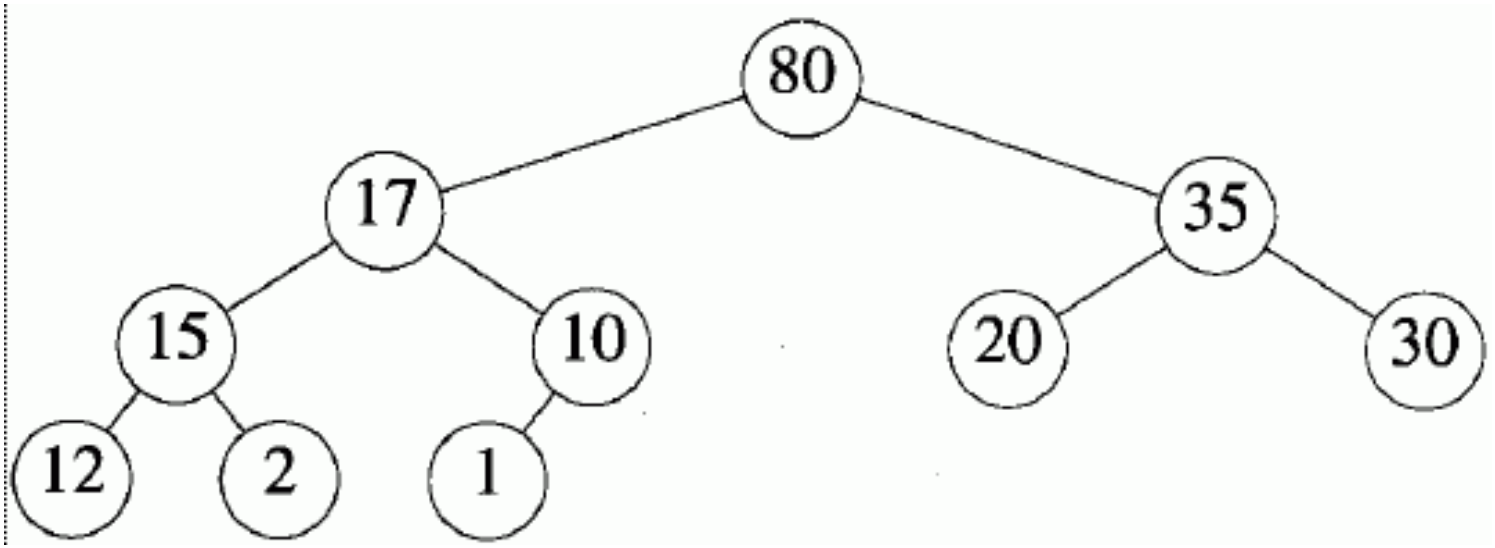
step4：检查编号为3的节点子树是否为堆？不是！调整为堆！
step5：检查编号为2的节点子树是否为堆？不是！调整为堆！

建堆实例（续）



step6：最后检查编号为1的节点子树是否为堆？不是！调整为堆！
step7：建堆成功！

建堆实例（续）



建堆函数

```
template<class T>
void MaxHeap<T>::Initialize(T a[], int size,
                            int ArraySize)
{// Initialize max heap to array a.
   delete [] heap;
   heap = a;
   CurrentSize = size;
   MaxSize = ArraySize;
```

建堆函数（续）

```
// 从最后一个内部节点开始，一直到根，对每个子树进行堆
重整
for (int i = CurrentSize/2; i >= 1; i--) {
  T y = heap[i]; // 子树根节点元素
  // find place to put y
  int c = 2*i; // parent of c is target
               // location for y

  while (c <= CurrentSize) {
    // heap[c] should be larger sibling
    if (c < CurrentSize &&
        heap[c] < heap[c+1]) c++;
```

建堆函数（续）

```
        // can we put y in heap[c/2]?
        if (y >= heap[c]) break;  // yes

        // no
        heap[c/2] = heap[c]; // move child up
        c *= 2; // move down a level
        }
    heap[c/2] = y;
    }
}
```

每棵子树的调整完全类似于删除操作

建堆复杂性分析

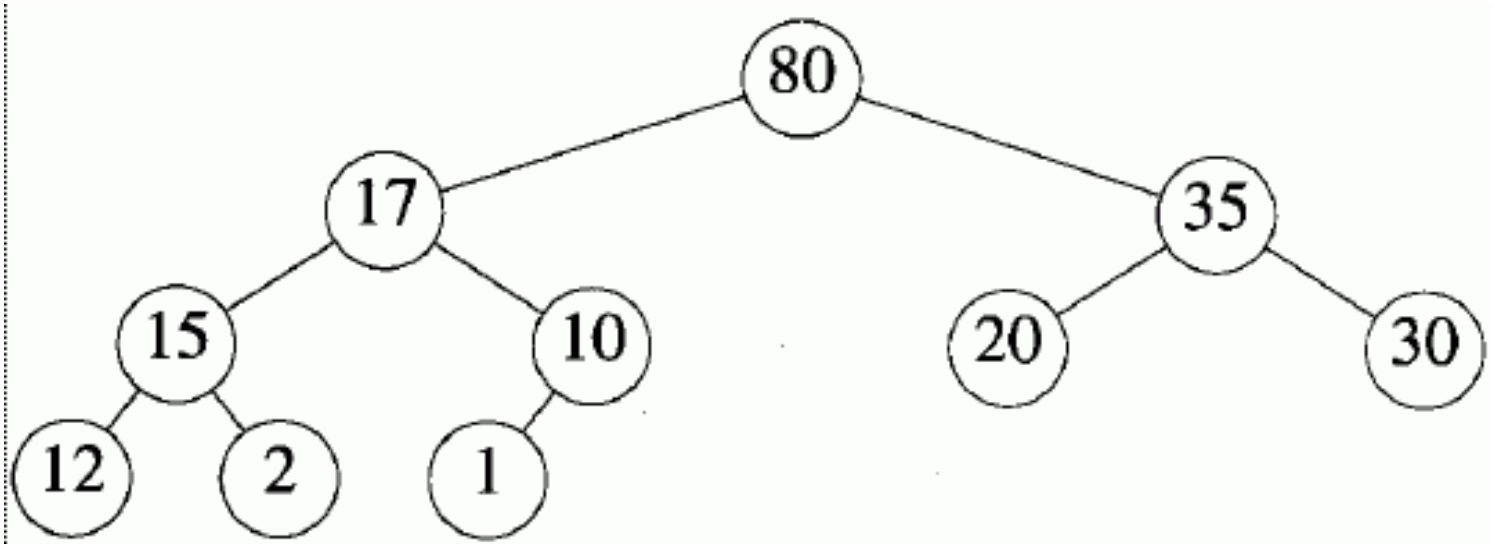
- 内层循环O(logn)，n/2次，O(nlogn)
- 实际上，Θ(n)
- 每次内层循环O(h_i)
  - h_i，子树高度，取值2～h
  - 第j层节点数2^{j-1}，以它们为根子树高度h-j+1
  - 所以建堆时间

$$O(\sum_{j=1}^{h-1} 2^{j-1}(h-j+1)) = O(\sum_{k=2}^{h} k2^{h-k}) = O(2^h \sum_{k=2}^{h} \frac{k}{2^k}) = O(2^h) = O(n)$$

  - 且循环执行n/2→Ω(n)→Θ(n)

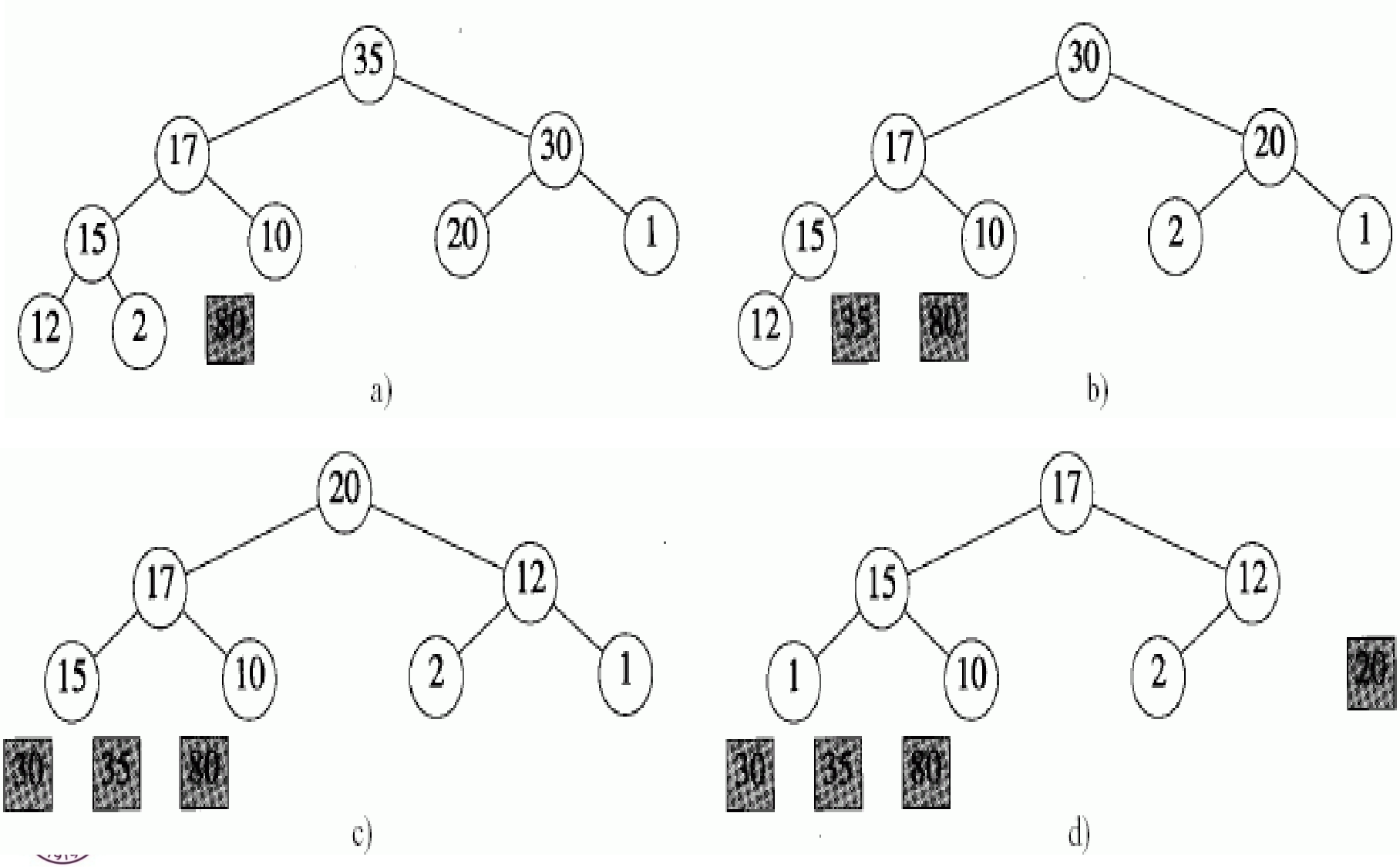
堆排序

- n个元素——建最大堆
- 重复：删除元素——放入末尾——重整
- 假定初始序列是
  [20, 12, 35, 15, 10, 80, 30, 17, 2, 1]



初始最大堆

堆排序


35
17
30
15
10
20
1
12
2
80
a)
30
17
20
15
10
2
1
12
35
80
b)
20
17
12
15
10
2
1
30
35
80
c)
17
15
12
1
10
2
20
30
35
80
d)

堆排序实现

```
template <class T>
void HeapSort(T a[], int n)
{// Sort a[1:n] using the heap sort method.
   // create a max heap of the elements
   MaxHeap<T> H(1);
   H.Initialize(a,n,n);      // 建堆，直接使用a作为
  堆的空间
```

堆排序实现

```
  // extract one by one from the max heap
  T x;
  for (int i = n-1; i >= 1; i--) {
    H.DeleteMax(x);
    a[i+1] = x;
    }

  // 将堆设置为空，但不释放空间——不销毁a
  H.Deactivate();
}
```

堆排序复杂性分析

- step1：建最大堆：O(n)
- step2：每次删除一个元素，共删n次：O(nlogn)
- 所以总的时间代价是O(nlogn)
- 同时，空间代价是O(1)

- **综合以上，明显优于选择、冒泡、插入等**

例题

- 有以下关键字：28，72，97，63，4，53，84，32，61，52，使用堆排序方法将所给关键字排成升序序列，给出排序过程。要求画出初始堆，每输出一个元素，画出剩余元素组成的新堆。

H1小结

- 堆是表达优先队列的一种方式
- 堆的插入：O(*log*n)从下向上调整
- 堆的删除：O(*log*n)从上向下调整
- 堆的初始化：O(n)从后向前遍历、自上而下调整
- 堆排序：先建最大堆、再依次删除
- 堆排序时间已达最优

主要内容

- 优先队列ADT
- 堆及堆排序
- **霍夫曼编码**

文本压缩

- 问题提出
  - 文本信息是一种基本数据类型
  - 需要找到存储文本信息以及有效地在计算机之间传递它们的方法
- 常用方法
  - 关键字编码
  - 行程长度编码
  - 赫夫曼编码

关键字编码

- 基本思想
  - 在文本中有一些常见词汇
  - the，and，which，that，what
  - 如果这些单词占用更少的空间，文档就会减小
  - 即使每个单词节省的空间有限，但是整个文档节省的总空间仍可能非常可观
- 定义
  - 用单个字符代替常用的单词

编码实例

单词	符号
as	^
the	~
and	+
that	$
must	&
well	%
these	#

文本段落

The human body is composed of many independent systems, such as the circulatory system, the respiratory system, and the reproductive system. Not only must all systems work independently, they must interact and cooperate as well. Overall health is a function of the well-being of separate systems, as well as how these separate systems work in concert.

共有字符352个

编码后段落

The human body is composed of many independent systems, such ^ ~ circulatory system, ~ respiratory system, + ~ reproductive system. Not only & all systems work independently, they & interact + cooperate ^ %. Overall health is a function of ~ %-being of separate systems, ^ % ^ how # separate systems work in concert.

共有字符317个，节省字符35个

压缩率=(317/352)*100%≈90.1%

行程长度编码

- 定义
  - 迭代编码
  - 把一系列重复字符替换为它们重复出现的次数
  - 常用于一些大规模数据流中
- 编码规则
  - 重复字符的序列用标志字符，后面加重复字符和说明字符重复次数的数字替换

AAAAAAA

*A7

解码规则

- 标志字符说明这三个字符的序列应该被解码为相应的重复字符串，其他文本则按照常规处理

*n5*x9ccc*h6 some other text *k8eee

nnnnnxxxxxxxxxccchhhhhh some other text kkkkkkkkeee

原始文本字符51个，编码串字符35个，节省16个

压缩率=(35/51)*100%≈68.6%

H2. 霍夫曼编码

- Huffman code：一种文本压缩算法
  - 考虑字符的出现频率进行编码
- 4个字符a, u, x, z组成的文本
  - 每个字符一个字节：1000字节，8000位
  - 每个字符两位(a:00， x:01， u:10， z:11)：2000位
  - 保存编码表
    - 符号个数，代码1，符号1，代码2，符号2，...—48位
  - 压缩比8000/2048=3.9

霍夫曼编码方法

- *aaxuaxz*
  - 码长两位：000001110000111，14位
  - a，x，u，z出现频率：3，2，1，1
  - 可变长度编码：频率高的短码，低的长码
  - a:0，x:10，u:110，z:111
  - 0010110010111，13位——稍好于原方法
  - 若1000个字符，频率：996，2，1，1——等长编码2000位，变长编码1006位

解码方法

- 如何解码？
- 任何一个编码都不是其他编码的前缀

a	0
x	00
u	01
z	001

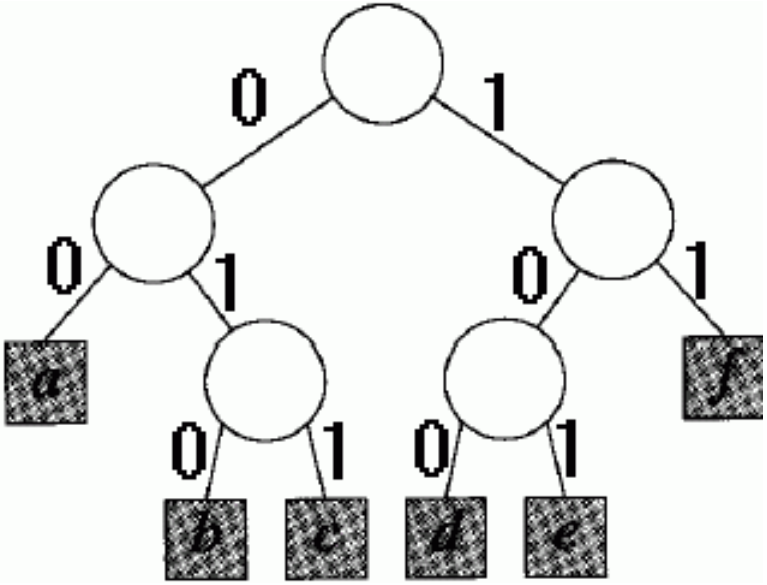
00010100100101

- 检测前缀即可

算法

- 扩充二叉树
  - 外部节点——符号，边——左标0，右标1
  - 根→外部节点路径
  - 编码后长度

2*F(a)+ 3*F(b)+
3*F(c)+3*F(d)+
3*F(e)+ 2*F(f)



算法（续）

- 扩充二叉树外部节点1，2，...，n，则压缩码长度为：

$$WEP = \sum_{i=1}^{n} L(i) * F(i)$$

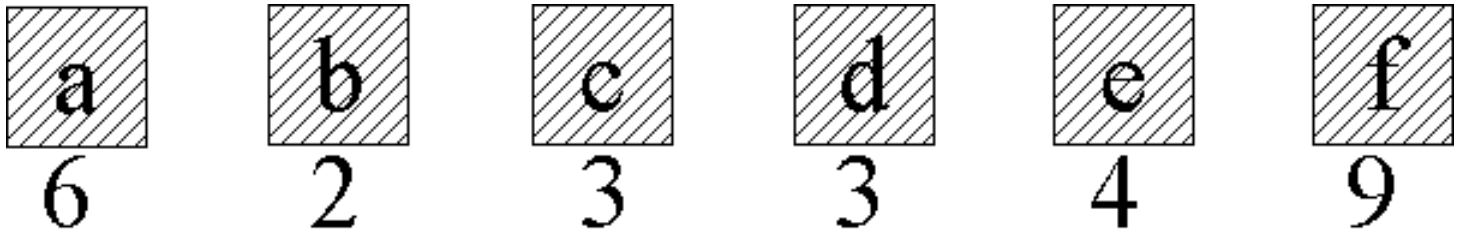
- L(i)：外部节点i的路径长度
- WEP：加权外部路径长度
- 霍夫曼树：最短压缩码←WEP最小

算法（续）

- 霍夫曼编码方法
  1. 获取不同字符的频率
  2. 构造霍夫曼树
  3. 遍历根到外部节点路径➔每个字符编码
  4. 用编码替换文本中的字符

构造霍夫曼树

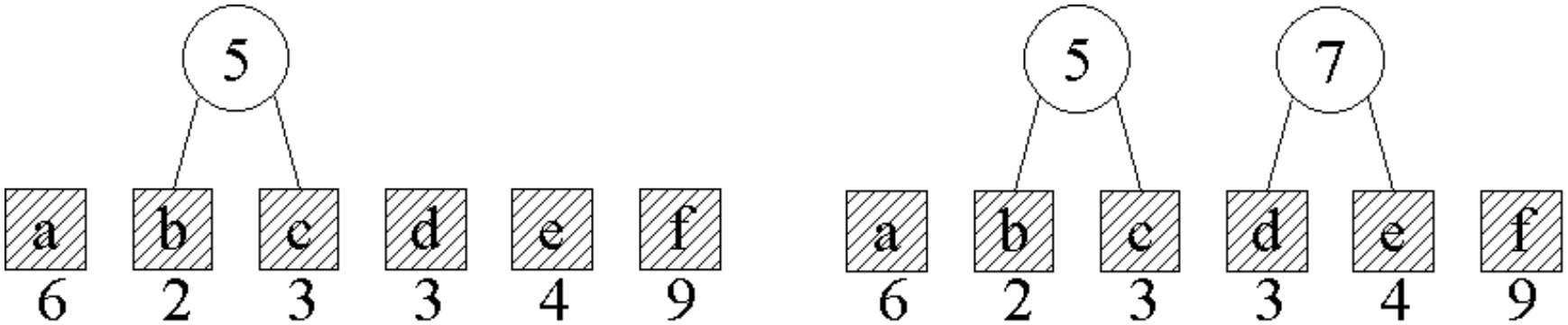
初始：单节点二叉树集合
树的权重——字符频率



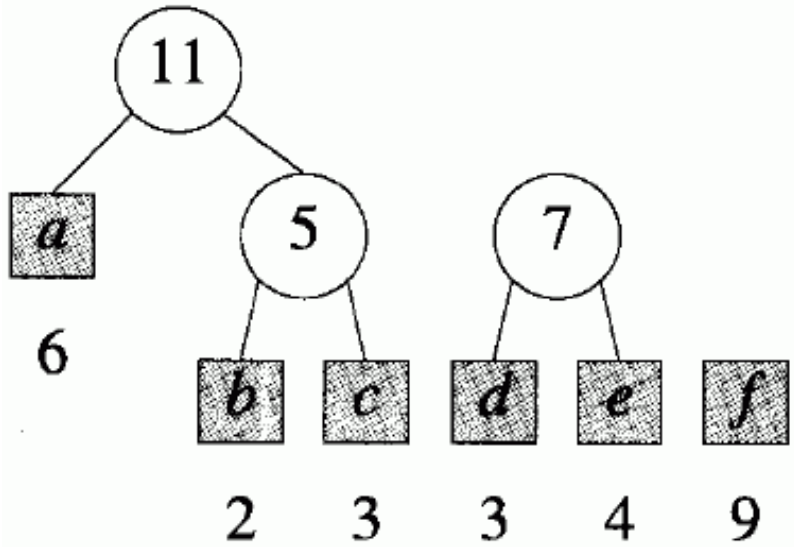
构造霍夫曼树

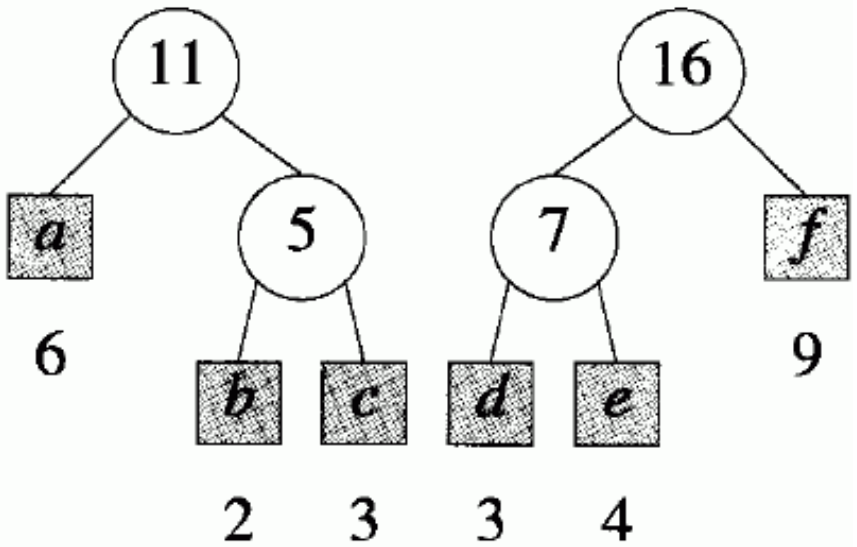
选择两个w值最小的树合并，权重相加作为新的权重

重复，直至只剩一棵树



构造霍夫曼树





Huffman类

```
template<class T>
class  Huffman {
   friend BinaryTree<int> HuffmanTree(T [],
   int);
   public:
      operator T () const {return weight;}
   private:
      BinaryTree<int> tree;
      T weight;
};
```

霍夫曼树构造函数

```
template <class T>
BinaryTree<int> HuffmanTree(T a[], int n)
{// Generate Huffman tree with weights a[1:n].
   // create an array of single node trees
   Huffman<T> *w = new Huffman<T> [n+1];
   BinaryTree<int> z, zero;
   for (int i = 1; i <= n; i++) {
      z.MakeTree(i, zero, zero);
      w[i].weight = a[i];
      w[i].tree = z;
      }

   // make array into a min heap
   MinHeap<Huffman<T> > H(1);
   H.Initialize(w,n,n);
```

霍夫曼树构造函数（续）

```
  // repeatedly combine trees from heap
  Huffman<T> x, y;
  for (i = 1; i < n; i++) {
    H.DeleteMin(x);
    H.DeleteMin(y);
    z.MakeTree(0, x.tree, y.tree);
    x.weight += y.weight; x.tree = z;
    H.Insert(x);
    }

  H.DeleteMin(x); // final tree
  H.Deactivate();
  delete [] w;
  return x.tree;
}
```

H2小结

- 霍夫曼树是一棵二叉树
  - 叶节点是具有不同权值的元素
  - 其他节点仅用于计算，没有实际意义
  - 从根到叶的路径（左0右1）即是该叶的编码

- 霍夫曼编码的基本思想
  - 让权值高的叶节点尽量靠近根，这样它的路径就能尽量短

思考

- 在一棵霍夫曼树中
  - 度为0的节点有几个?
  - 度为1的节点有几个?
  - 度为2的节点有几个?

本章结束