第12章

优先级队列

本章内容

- **12.1 定义和应用**
- **12.2 抽象数据类型**
- 12.3 线性表
- **12.4 堆**
- 12.5 左高树
- **12.6 应用**
  - 12.6.1 **堆排序**
  - 12.6.2 机器调度
  - 12.6.3 **霍夫曼编码**

12.1 定义和应用

- 优先级队列
  - 优先级队列(priority queue)是0个或多个元素的集合，每个元素都有一个**优先级或值**。
  - 与FIFO结构的队列不同，优先级队列中元素出队列的顺序由元素的优先级决定。
  - 从优先级队列中删除元素是根据优先级高或低的次序，而不是元素进入队列的次序.
  - 对优先级队列执行的操作有:
    - 1)查找一个元素(top)
    - 2)插入一个新元素(push)
    - 3)删除一个元素(pop)

优先级队列

- 两种优先级队列:
  - 最小优先级队列(Min priority queue)
  - 最大优先级队列(Max priority queue)
- 在**最小优先级队列(Min priority queue)**中，“查找/删除”操作用来“查找/删除”**优先级最小**的元素;
- 在**最大优先级队列(Max priority queue)**中，“查找/删除”操作用来“查找/删除”**优先级最大**的元素。
- 优先级队列中的元素可以有相同的优先级。

12.2 抽象数据类型

抽象数据类型 *MaxPriorityQueue*{

实例

有限个元素集合，每个元素都有一个优先级

操作

empty()：判断优先级队列是否为空，为空时返回true

Size()：返回队列中的元素数目

top()：返回优先级最大的元素

pop()：删除优先级最大的元素

push(*x*)：插入元素*x*

}

- 最小优先级队列的抽象数据类型描述?

优先级队列的描述

- 优先级队列的描述
  - 线性表
  - 堆(Heaps)
  - 左高树(Leftist trees)

12.2 线性表

- 采用**无序线性表**描述最大优先级队列
  - 数组描述(利用公式Location(i)=i-1)
    - 插入：表的右端末尾执行，时间: Θ(1)；
    - 删除:查找优先级最大的元素，时间: Θ(n)；
  - 链表描述
    - 插入：在链头执行，时间: Θ(1)；
    - 删除： 查找优先级最大的元素， Θ(n)；

线性表

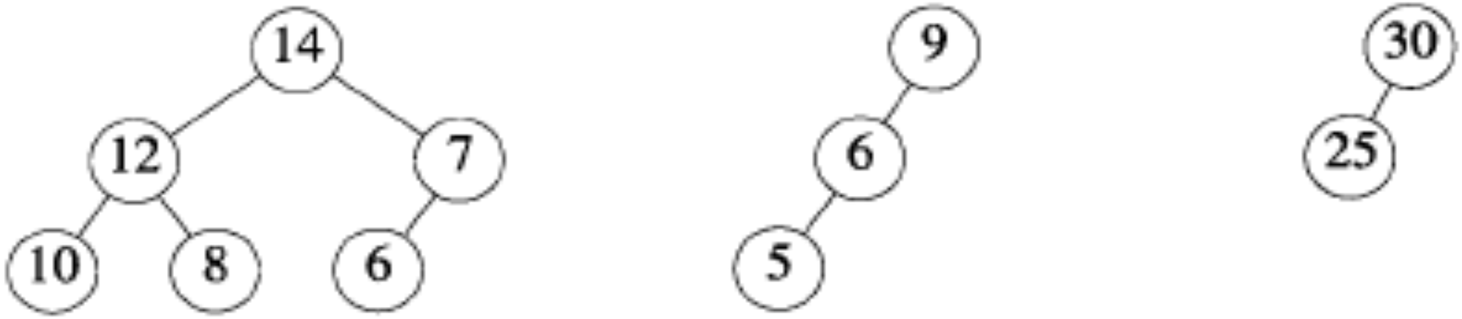
- 采用**有序线性表**描述最大优先级队列
  - 数组描述(利用公式Location(i)=i-1),元素按递增次序排列)
    - 插入：先查找插入元素的位置，时间: O(n)；
    - 删除： 删除最右元素，时间: Θ(1)；
  - 链表描述(按递减次序排列)
    - 插入：先查找插入元素的位置，时间: O(n)；
    - 删除: 表头删除，时间: Θ(1)；

12.4 堆(Heaps)

- 12.4.1 定义
- 定义[大根树(小根树)] 每个节点的值都大于(小于)或等于其子节点(如果有的话)值的树。

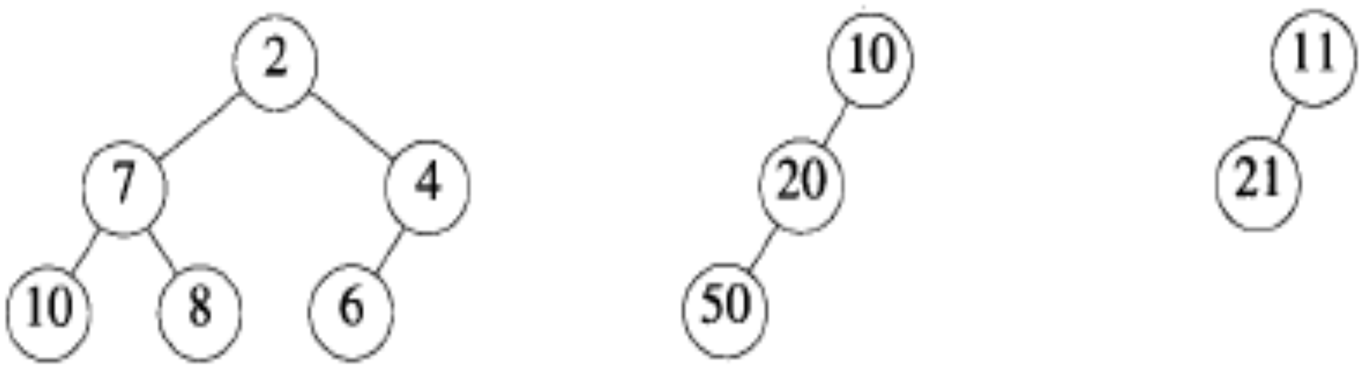
- 大根树(max tree):又称最大树
- 小根树(min tree):又称最小树

- 大根树或小根树节点的子节点个数可以大于2。



a) b) c)

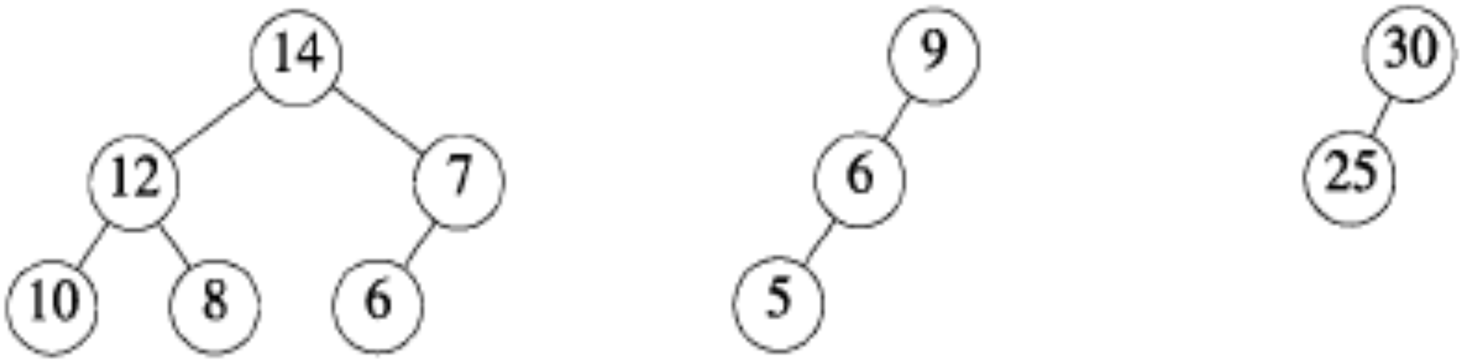
大根树



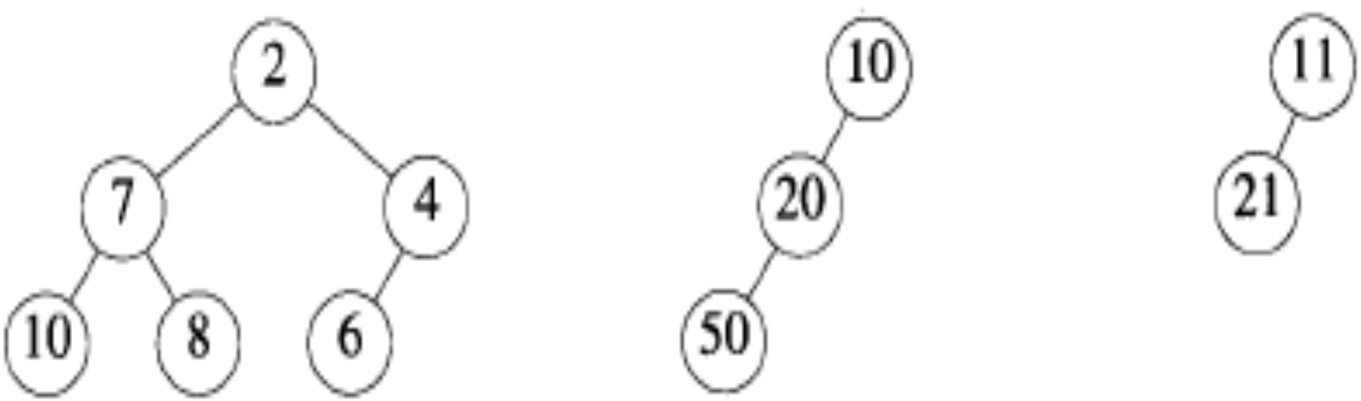
a) b) c)

小根树

- 定义[大根堆(小根堆)]： 既是大根树(小根树)，又是完全二叉树。
- 大根堆(max heap):又称最大堆， 小根堆(min heap):又称最小堆。

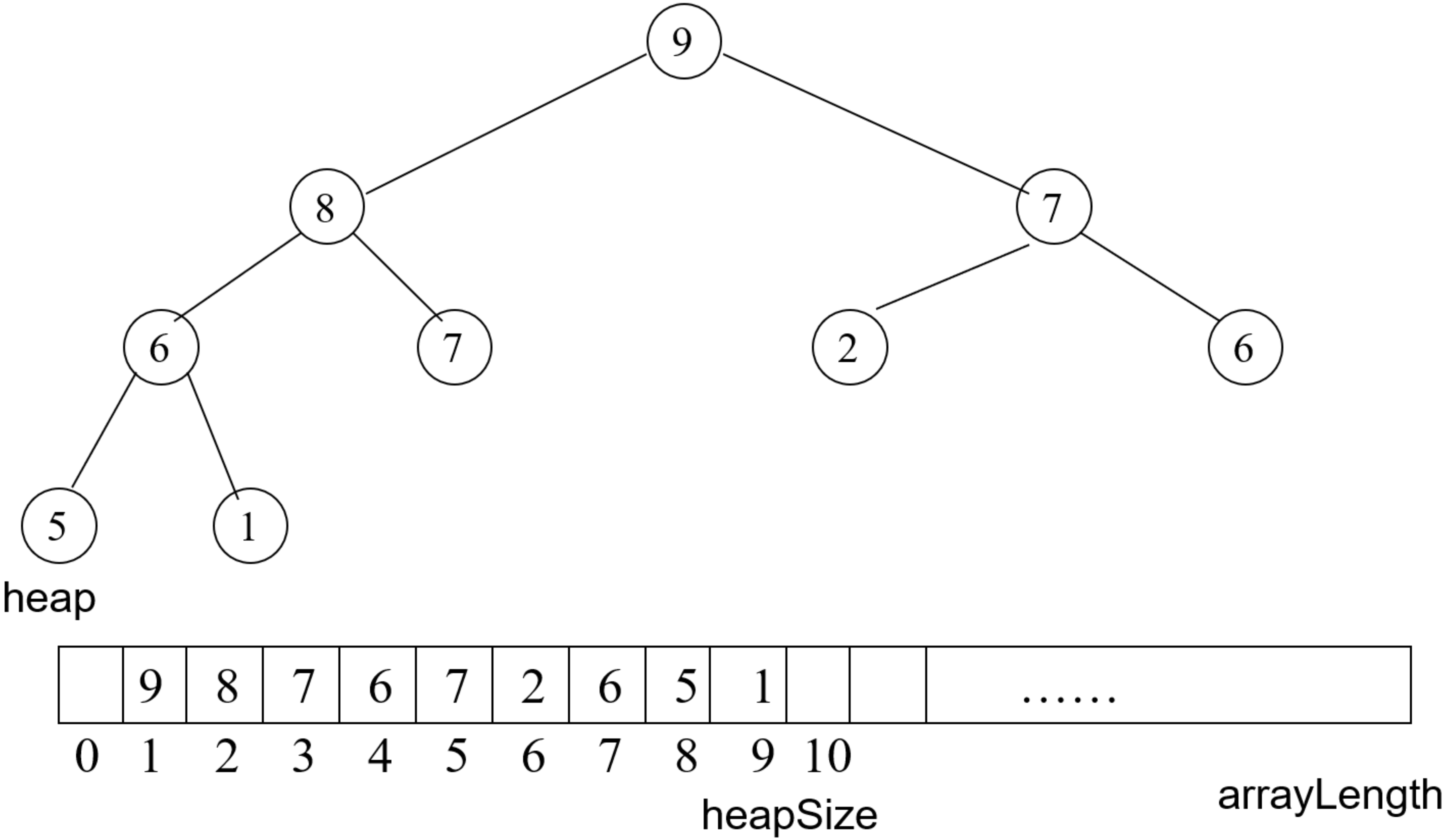


a) b) c)



a) b) c)

- **堆的描述：** 堆是完全二叉树，可用一维数组有效地描述堆.



类maxHeap

heap

	9	8	7	6	7	2	6	5	1			……
0	1	2	3	4	5	6	7	8	9	10		

heapSize

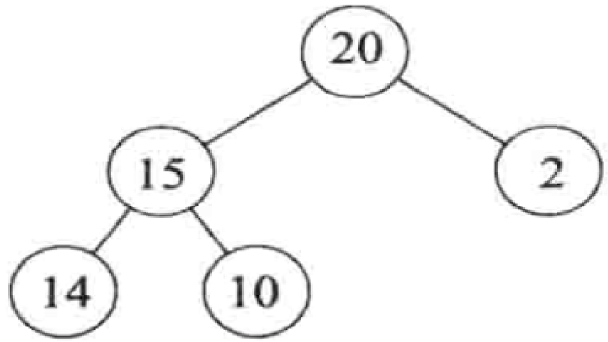
arrayLength

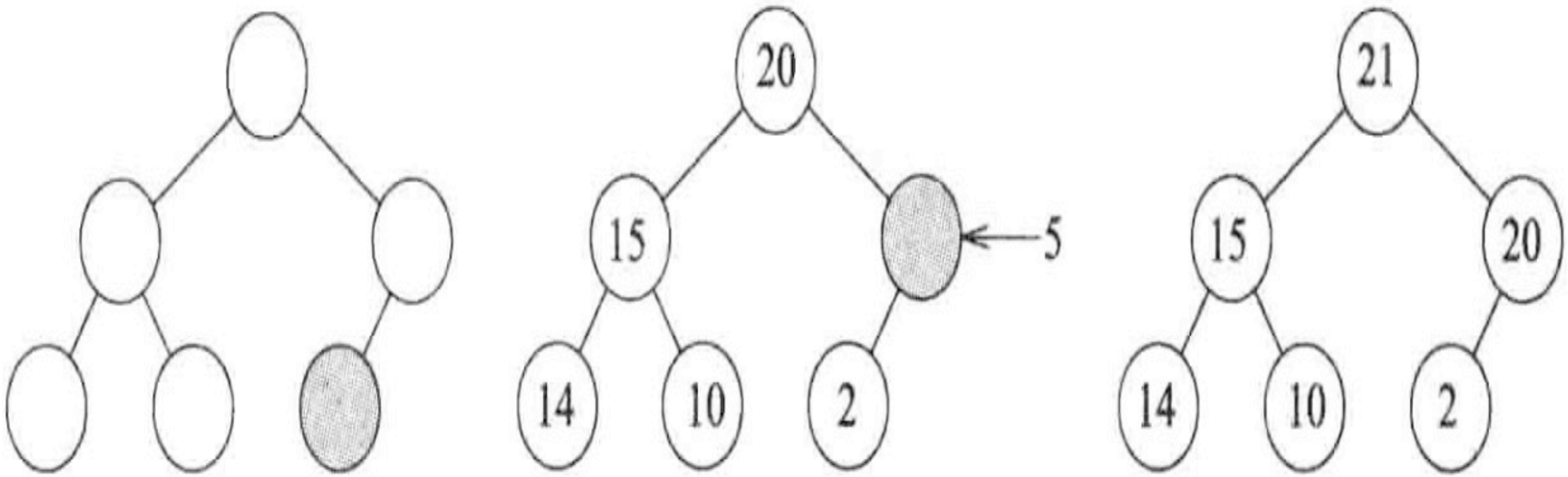
- 数据成员：
  - T *heap; // 元素数组
  - int arrayLength; //数组的容量
  - int heapSize; //堆中的元素个数
  - 方法：
  - empty()
  - *Size*()
  - *top*()
  - *pop*()
  - *push*(*x*)

方法empty,Size,top

- 方法empty
  - 判断heapSize是否为0.
- 方法*Size*
  - 返回heapSize的值
- 方法*top*
  - 如果 堆为空(heapSize==0)
  - 则　抛出异常queueEmpty
  - 否则　返回 heap[1]

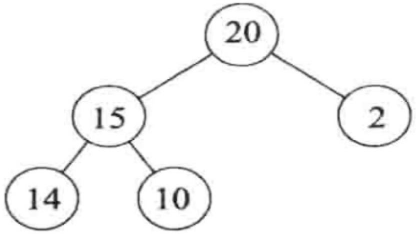
12.4.2 大根堆的插入





- 插入新元素5
- 插入新元素21

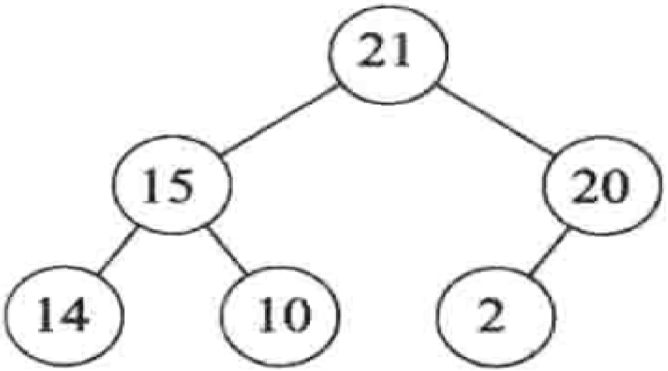
```
template<class T>
maxHeap<T>& maxHeap<T>::push(const T& theElement)
{// 把theElement 插入到大根堆中
  if  (heapSize = = arrayLength-1) // 没有足够空间
    ……//数组长度加倍
  //为theElement寻找应插入位置
  // currentNode 从新的叶节点开始，并沿着树上升
  int  currentNode = ++heapSize;
  while (currentNode != 1 &&
          theElement > heap[currentNode/2])
    {//不能够把theElement放入heap[currentNode]
     heap[currentNode] = heap[currentNode/2]; // 将元素下移
     currentNode /= 2; // 移向父节点
     }
heap[currentNode] = theElement;

}
```



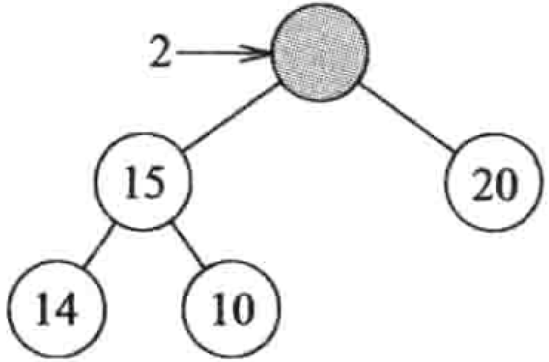
大根堆的插入时间复杂性

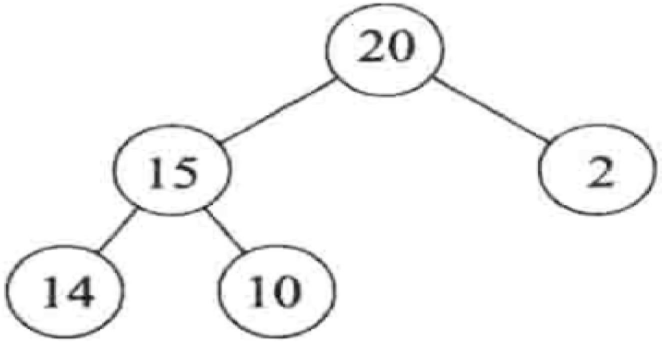
- 插入的时间复杂性:
  - 每一层的工作，耗时：$\Theta(1)$
  - 实现插入策略的时间复杂性:
  - O(height) = $O(log_2 n)$, (n 是堆的大小)

12.4.3 大根堆的删除

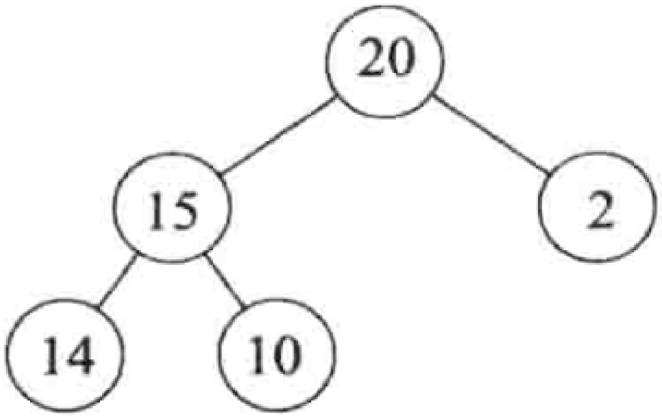


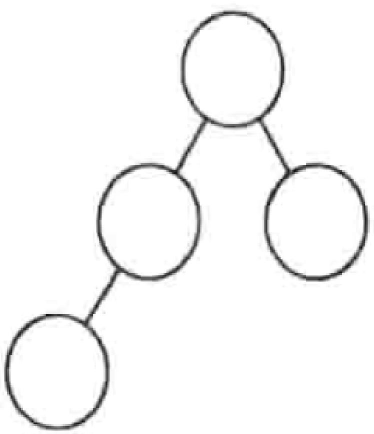
- 删除最大元素21：
- 2放入根上，不是堆，
- 将根的孩子的大者20放到根上；2放到位置3上，是堆；

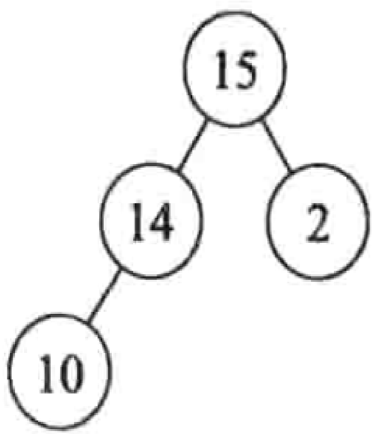




大根堆的删除

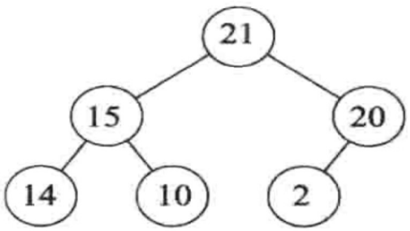






- 删除最大元素20
- 10放入根上，不是堆，将根的孩子的大者15放到根上；
- 10放入位置2上，不是堆，将位置2的孩子的大者14放到位置2上； 10放入位置4上

```
template<class T>
maxHeap<T>& maxHeap<T>::pop()
{// 从堆中删除最大元素
   if (heapSize = = 0)  throw queueEmpty(); // 检查堆是否为空
     heap[1].~T(); // 删除最大元素
     // 删除最后一个元素，然后重新构造堆
     T  lastElement = heap[heapSize--]; // 最后一个元素
     // 从根开始，为lastElement 寻找合适的位置
     int currentNode= 1, // 堆的当前节点
     child = 2; // currentNode的孩子
     while (child <= heapSize)
          {if (child < heapSize && heap[child] < heap[child+1]) child++;
            // heap[child] 是currentNode的大孩子
           //可以把lastElement 放入heap[currentNode]吗?
           if (lastElement >= heap[child])  break; //可以
           //不可以
          heap[currentNode] = heap[child]; // 将孩子child上移
          currentNode = child; // currentNode下移一层
          child *= 2;
          }
     heap[currentNode] = lastElement;

}
```



大根堆的删除性能

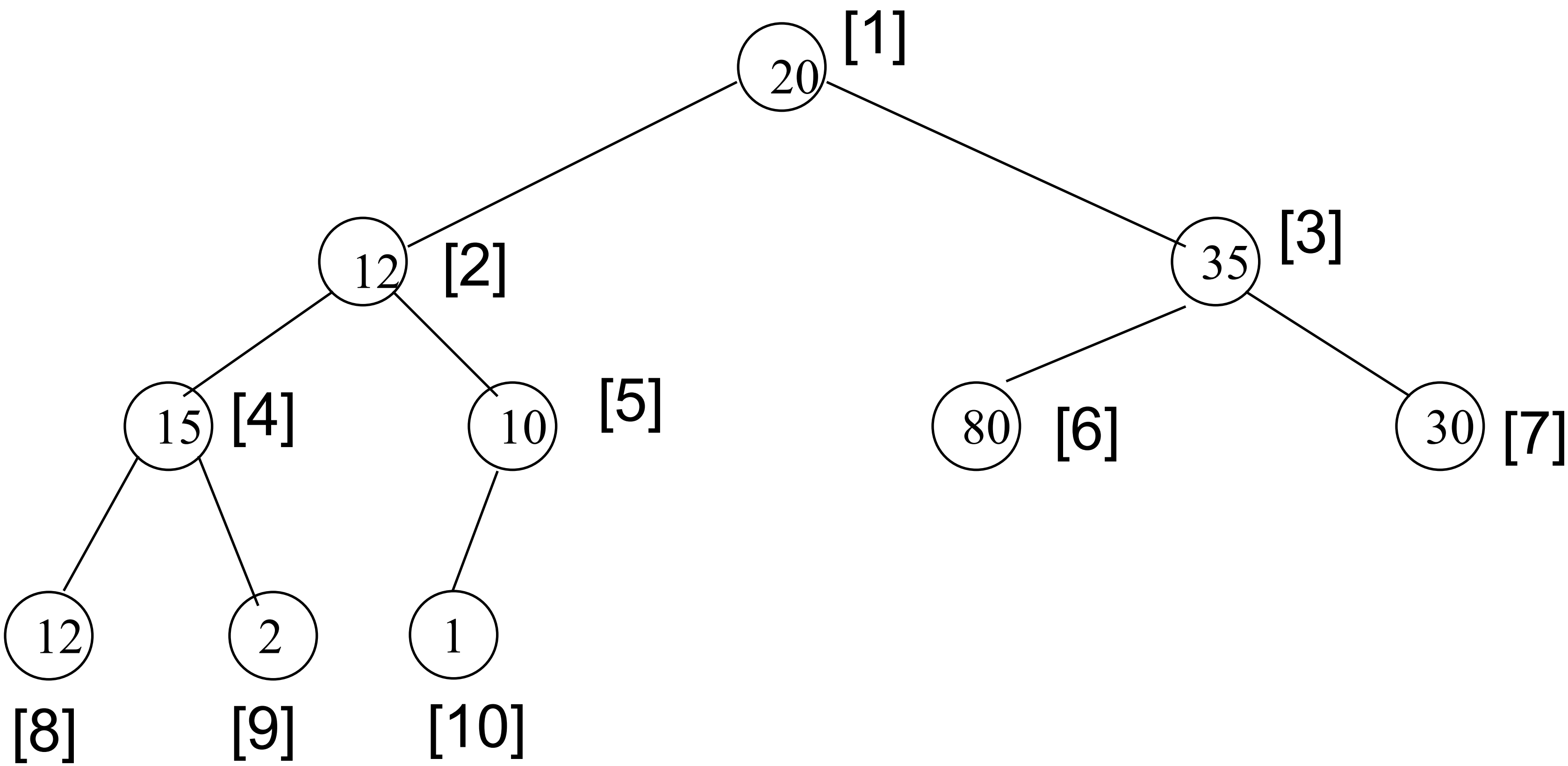
- 删除的时间复杂性:
  - 删除的时间复杂性与插入的时间复杂性相同.
  - 每一层的工作，耗时：$\Theta(1)$
  - 实现删除策略的时间复杂性:
  - O(height) = $O(\log_2 n)$, (n 是堆的大小)

12.4.4 大根堆的初始化

- 1. 通过在初始化为空的堆中执行n次插入操作来构造。
  - $O(n\log_2 n)$
- 2. 通过必要时对树进行调整的方法构造。
  - $\Theta(n)$

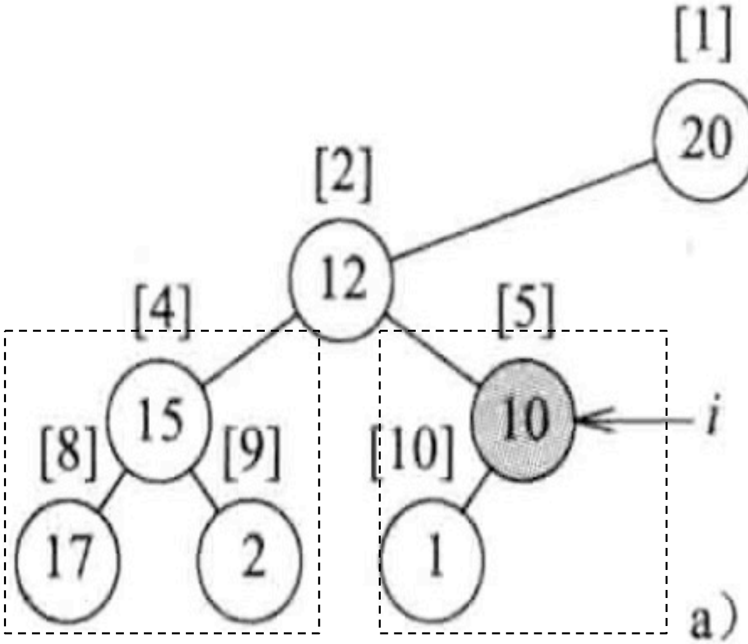
大根堆的初始化

- 例: 数组a[1..10]= [20,12,35,15,10,80,30,12,2,1]
- 将数组a初始化为大根堆.

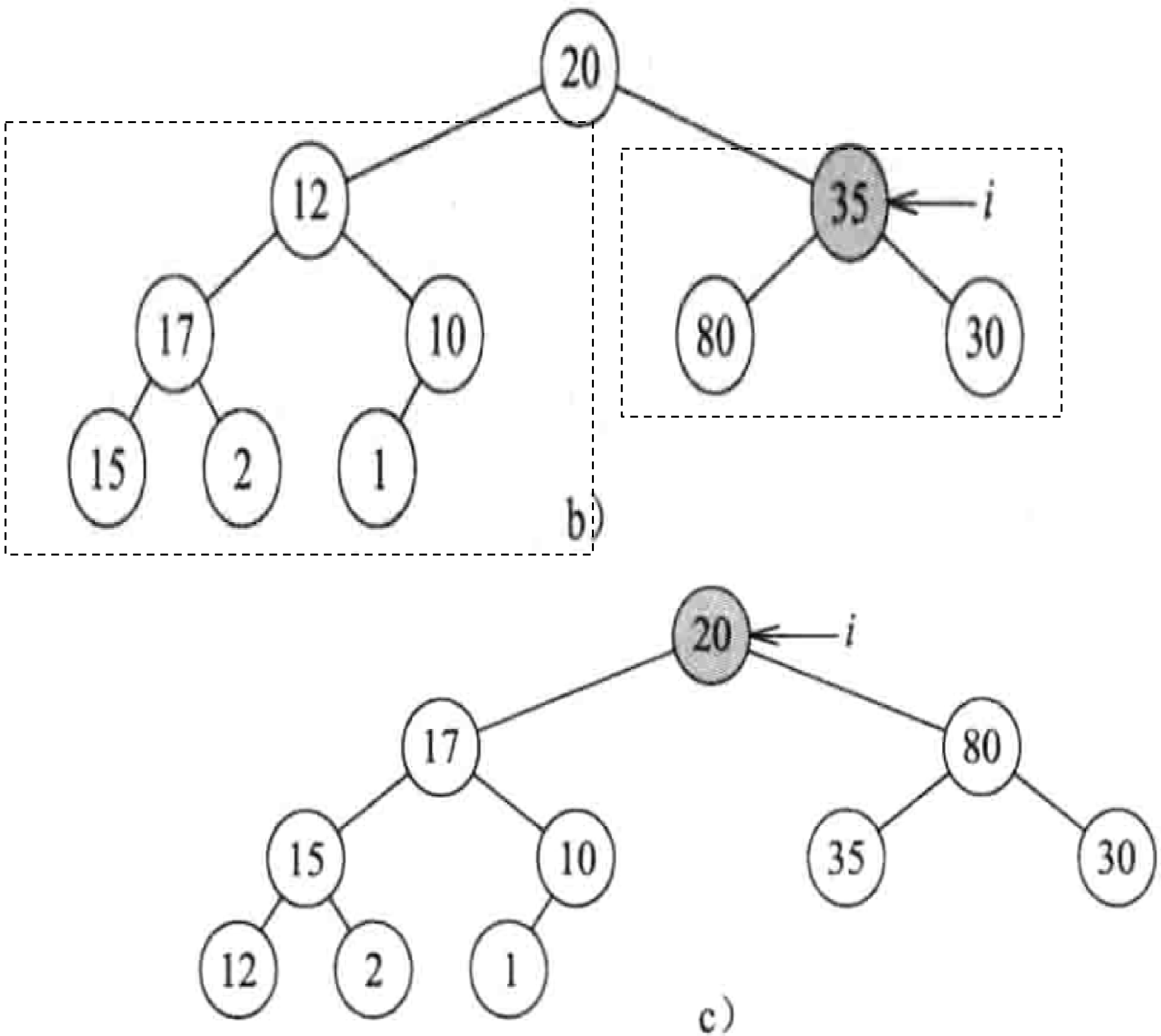


大根堆的初始化

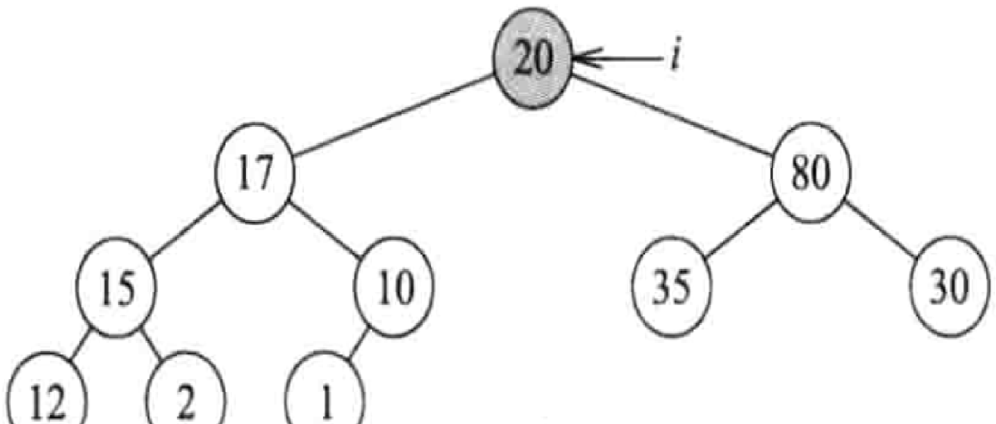
- 从数组中最右一个有孩子的节点开始.



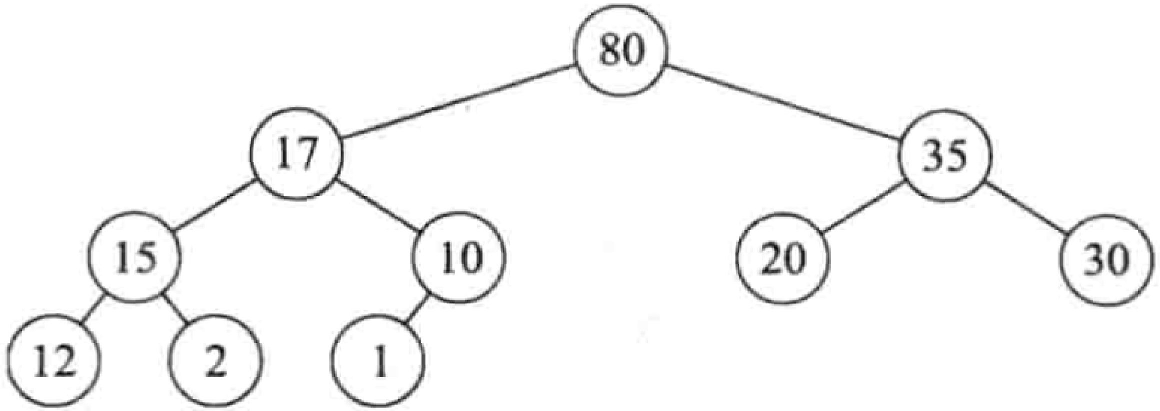
大根堆的初始化



大根堆的初始化

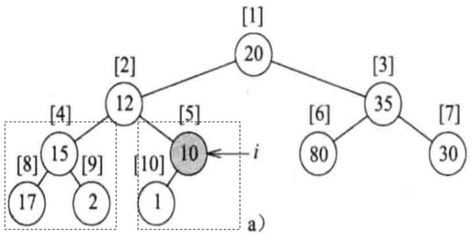


c)



d)

```
void maxHeap<T>::initialize(T *theHeap, int theSize)
{// 在数组theHeap[1.. theSize]中建大根堆.
    delete []heap;  heap = theHeap;
    heapSize = theSize;
    // 堆化
    for (int root = heapSize/2; root >= 1; root--) {
        T rootElement = heap[root]; // 子树的根
        // 寻找放置rootElement的位置
        int child = 2*root; // child的父节点是元素rootElement的位置
        while (child <= heapSize) {
            // heap[child] 应是兄弟中较大者
            if (child < heapSize && heap[child] < heap[child+1])
               child++;
            //能把rootElement放入heap[child/2]吗?
            if (rootElement >= heap[child]) break; // 能
            // 不能
            heap[child/2] = heap[child]; // 将孩子上移
            child *= 2; // 下移一层
            }
        heap[child/2] = rootElement;
    }
}
```



初始化堆的时间复杂性

- 堆的高度 = h.
- 在j层节点的数目 <= 2^{j-1}.
- 以j层节点为根的子树的高度 = h-j+1
- 调整(或重构)以j层节点为根的子树： O(h-j+1).
- 调整(或重构)所有以j层节点为根的子树：

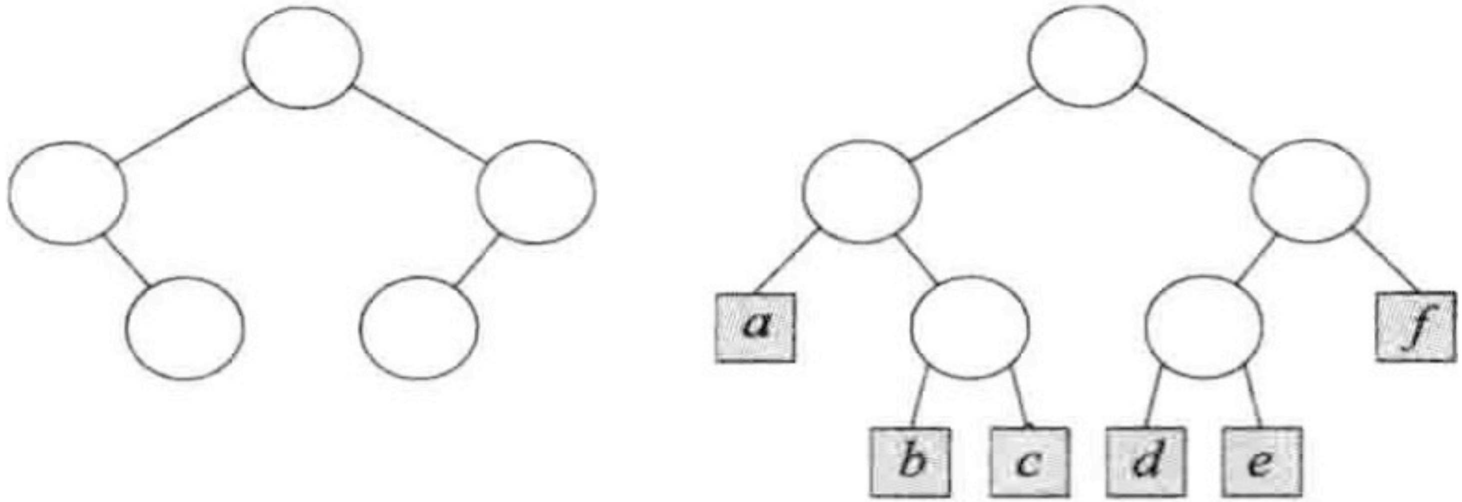
  $<= 2^{j-1}(h-j+1) = t(j).$
- 总的时间： t(1) + t(2) + … + t(h-1)
- $= O(2^h)$
- = O(n).

12.5 左高树(Leftist Trees)

- 堆结构是一种隐式数据结构(implicit data structure)。时间效率和空间利用率都很高。
  - 没有明确的指针或其他数据能够用来重塑这种结构
- 左高树是一种适合于实现优先队列的链表结构。
- 左高树适合于优先队列的某些应用
  - 合并两个优先队列或多个长度不同的队列
- 左高树
  - 高度优先左高树(HBLT— height-biased leftist tree)
    - 最大/最小HBLT
  - 重量优先左高树(WBLT—weight-biased leftist tree)
    - 最大/最小WBLT

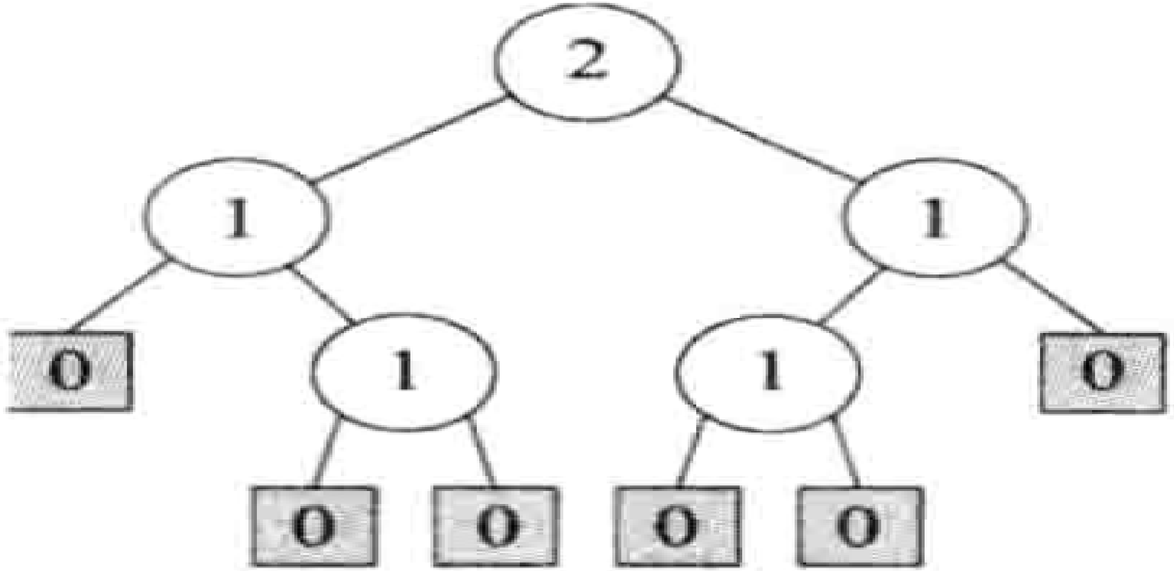
扩充二叉树(Extended Binary Tree)

- 外部节点(External node) –代替树中的空子树的节点。
- 内部节点(Internal node) – 其余节点。
- 扩充二叉树(Extended binary tree) –增加了外部节点的二叉树.



函数 s( )

- 对扩充二叉树中的任意节点*x*，令*s(x)*为从节点*x*到它的子树的外部节点的所有路径中最短的一条路径长度。



s()的特性

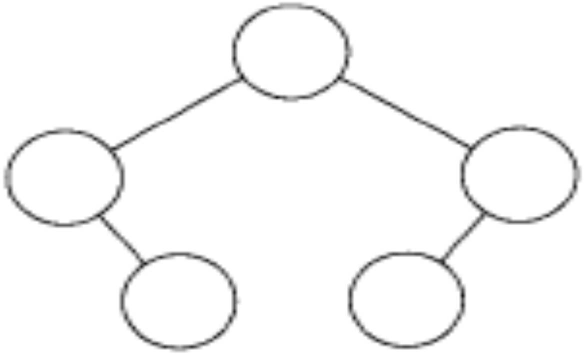
- 根据*s*(*x*)的定义可知：
- 若x是外部节点，则 s(x) = 0.

- 否则,
  - s(x) = min {s(L), s(R)} + 1
  - 其中*L*与*R*分别为x的左右孩子。

高度优先左高树 (HBLT)

- 定义[高度优先左高树] 当且仅当一棵二叉树的任何一个内部节点，其**左孩子的*s*值**大于等于**右孩子的*s*值**时，该二叉树为高度优先左高树(height-biased leftist tree, HBLT)。
- 对每一个内部节点 x,
  - s(L) >= s(R),其中*L*与*R*分别为x的左右孩子。

- 图所示的二叉树是 HBLT?



a)

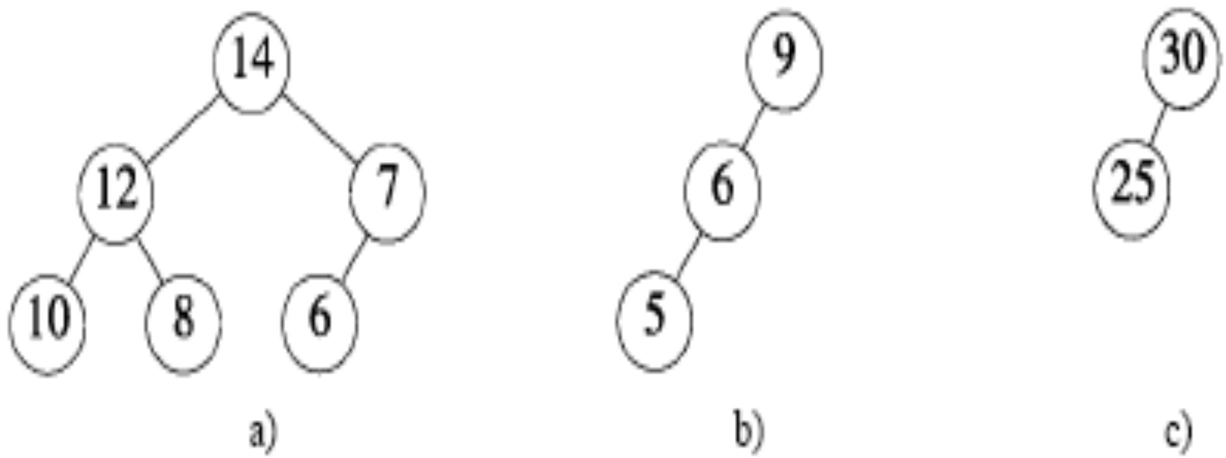
高度优先左高树 (HBLT)

- 定理12-1：令*x*为一个HBLT的内部节点，则
  - (a)以*x*为根的子树的节点数目至少$2^{s(x)}-1$
  - (b)若以*x*为根的子树有*m*个节点，则s(x)最多为$\log_2(m+1)$
  - (c)通过最右路径(即，此路径是从*x*开始沿右孩子移动)从*x*到达外部节点的路径长度为s(x) .
- 证明：
  - (a)根据*s(x)*的定义，从*x*节点往下第*s(x)*-1层没有外部节点(否则*x*的*s*值将更小)。
  - 以*x*为根的子树中1~*s*(*x*)层全是内部节点。共$2^{s(x)}-1$

  (b)有(a)知$m\geq 2^{s(x)}-1$；所以$s(x)\leq\log_2(m+1)$

最大/最小HBLT

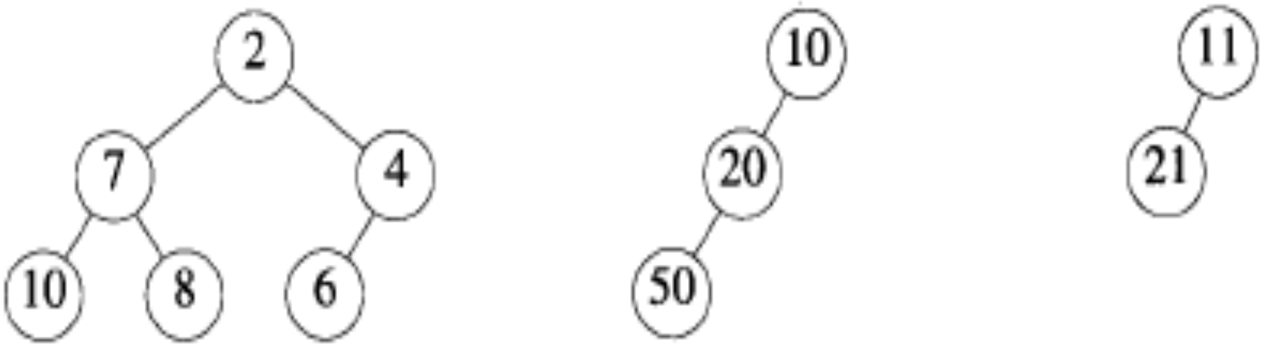
- 定义[最大HBLT]：即同时又是最大树的HBLT；



–是最大HBLT吗?
–是！

最大/最小HBLT

- 定义[最小HBLT] 即同时又是最小树的HBLT 。



a)　　b)　　c)

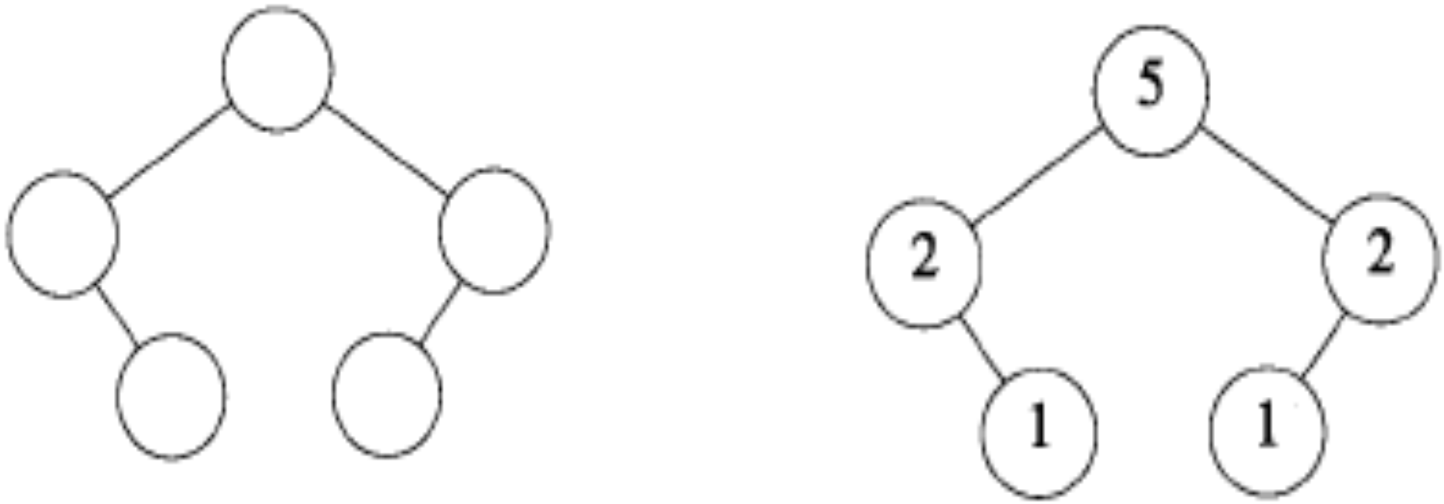
–是最小HBLT吗?

–是!

重量优先左高树 (WBLT)

- 定义x的重量w(x), 为以x为根的子树的内部节点数目。
- →若x 是外部节点， 则w(x) = 0
- →若x为内部节点， 其重量为其孩子节点的重量之和加1。
- 定义[重量优先左高树](WBLT—weight-biased leftist tree)： 当且仅当一棵二叉树的任何一个内部节点，其**左孩子的*w*值**大于等于**右孩子的*w*值**时。
- [最大(小)WBLT] 该二叉树为重量优先左高树即同时又是最大(小)树的WBLT。

重量优先左高树 (WBLT)



a)

- 是 WBLT吗?

最大HBLT的操作

- 12.5.2 最大HBLT的插入
- 12.5.3 最大HBLT的删除
- 12.5.4 两棵最大HBLT的合并
- 12.5.5 初始化最大HBLT

最大HBLT的插入

- 创建一棵含有插入元素的单元素最大HBLT，与被插入的最大HBLT合并；

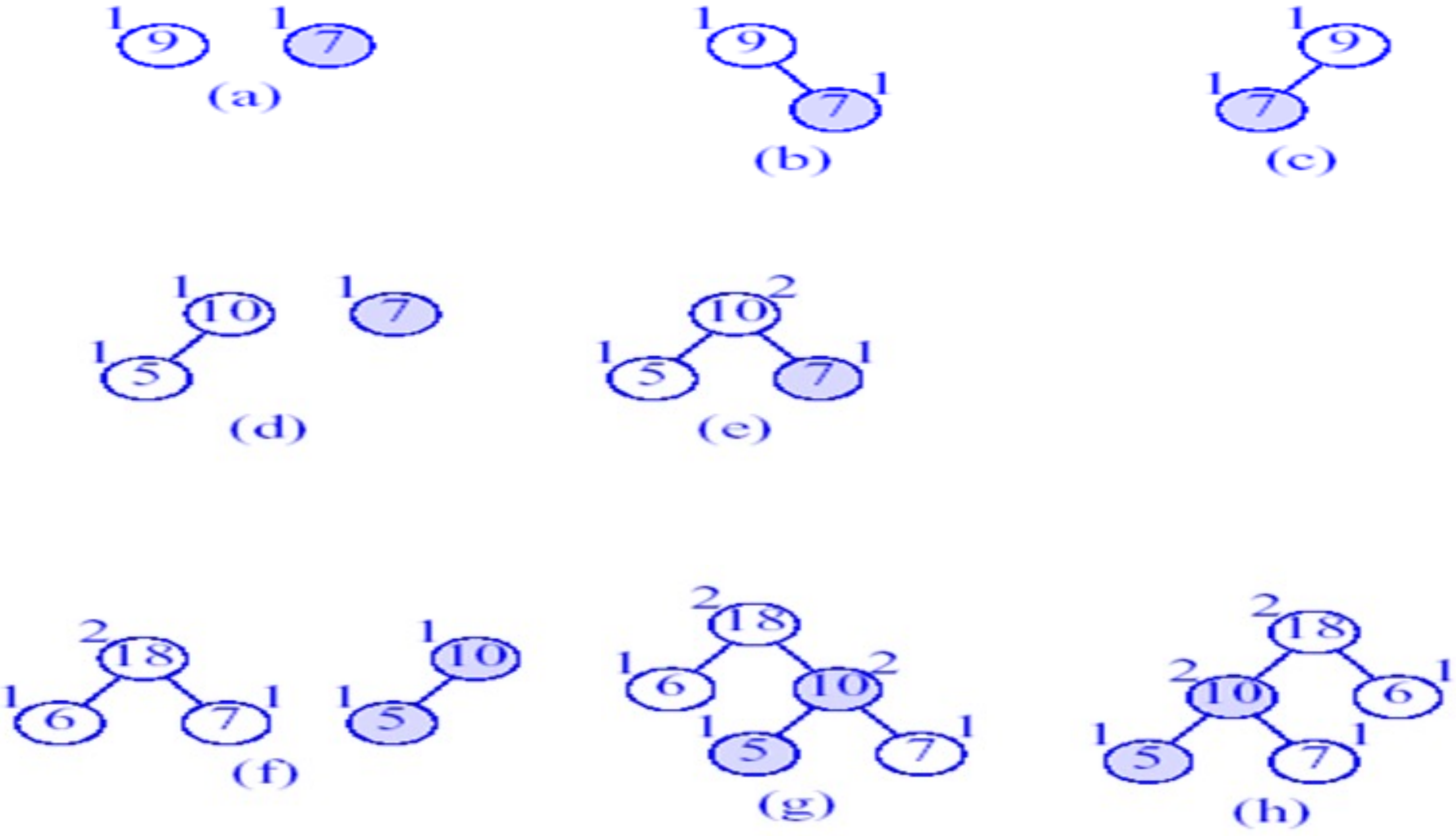
最大HBLT的删除

- 删除根节点.
- 合并根节点的左子树和右子树（都是最大HBLT）。

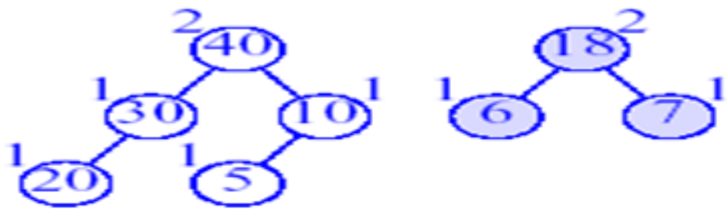
合并两棵最大HBLT

- *A*、*B*：需要合并的两棵最大HBLT
- *A*、*B*其中一个为空，将另一个作为合并的结果；
- *A*、*B*均不为空
  - 比较*A*、*B*的根元素，较大者为合并后的HBLT的根
  - 设*A*具有较大的根，*A*的左子树为*AL*，*A*的右子树*AR*
  - *A*的右子树*AR*与*B*合并的结果：*C*
  - *A*与*B*合并的结果：以*A*的根为根，*AL*与*C*为左右子树的最大HBLT
  - 如果AL的*s*值小于*C*的*s*值，则*C*为左子树，否则*L*为左子树

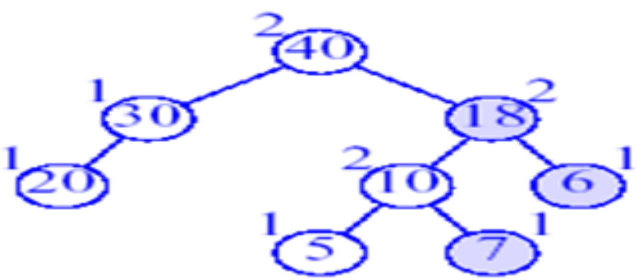
合并最大HBLT



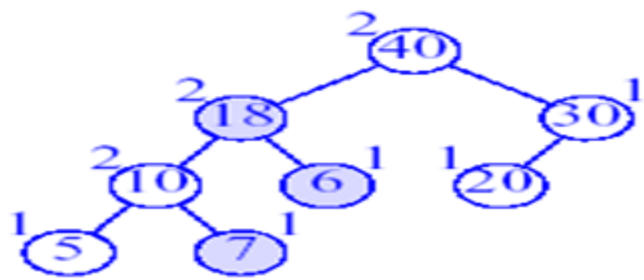
合并最大HBLT



(i)



(j)



(k)

初始化最大HBLT

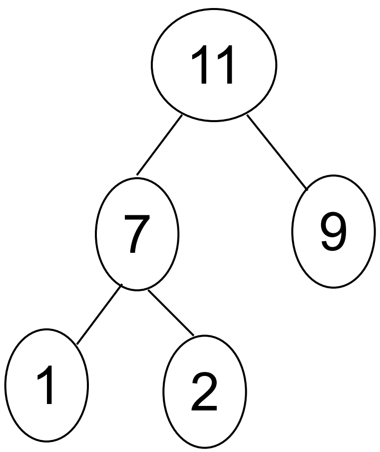
- 方法1.
  - 通过将*n* 个元素插入到最初为空的最大HBLT中来对其进行初始化.
- 时间为:O(nlogn)

12.5.5 初始化最大HBLT

- 方法2.
  - 1、创建*n* 个最大HBLT， 每个树中包含一个元素，这*n* 棵树排成一个FIFO队列
  - 2、从队列中依次删除两个最大HBLT， 将其合并，然后再加入队列末尾。
  - 重复第2步，直到最后只有一棵最大HBLT 。
- 时间复杂性：O(n)

12.5.5 初始化最大HBLT

- 元素:7,1, 9, 11,2



12.5.5 初始化最大HBLT

- 时间复杂性
  - 为简单起见，假设n是2的幂次方。
  - 合并n / 2对具有1个元素的最大HBLT，每次合并O(1)。
  - 合并n/ 4对含有2个元素的最大HBLT，每次合并O(2)。
  - 合并n/ 8对含有4个元素的最大HBLT，每次合并O(3)。
  - …… 。
  - 合并两棵含2^i个元素的最大HBLT，每次合并O(i+1)，
  - …… 。
  - 合并1对含n/2 (2^{k-1})个元素的最大HBLT,每次合并O(k)
  - 因此总时间为：
  - O(n/2+2*(n/4)+3*(n/8)+…+k*(n/n)) = O(n)

12.6 应用

- **12.6.1 堆排序**
- 12.6.2 机器调度
- **12.6.3 霍夫曼编码**

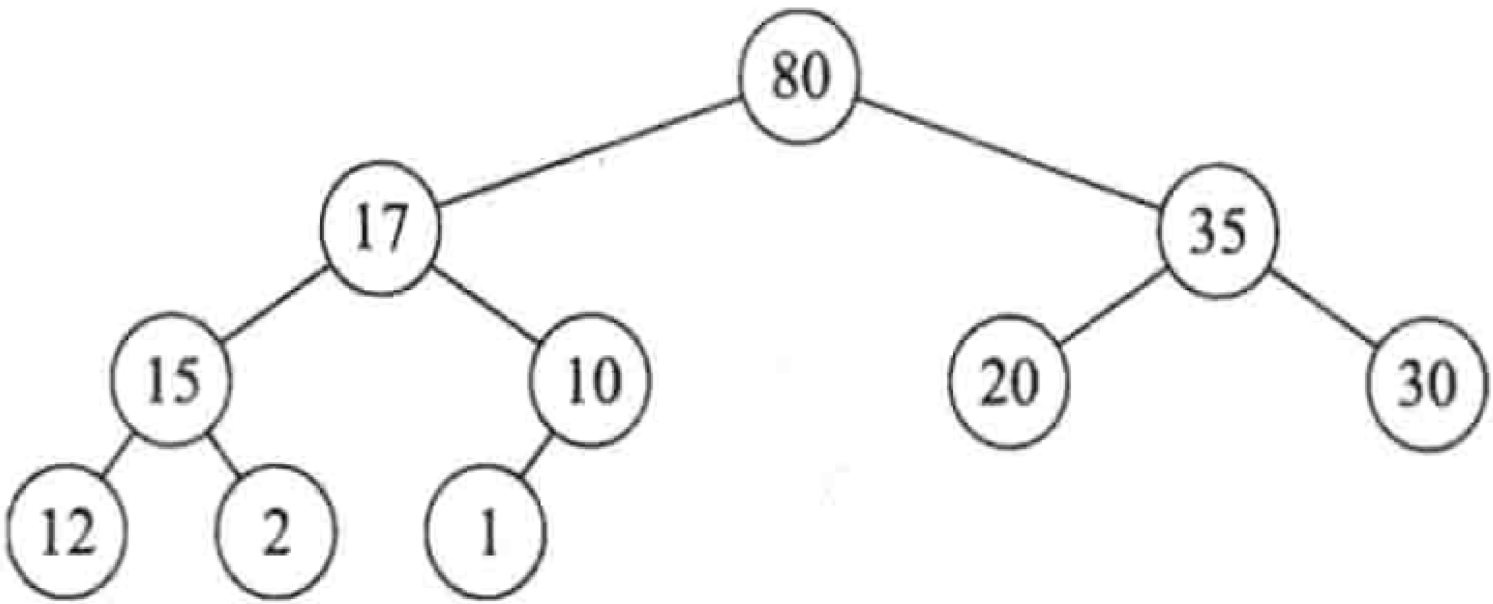
12.6.1 堆排序(Heap Sort)

- 可利用堆来实现n个元素的排序，每个元素的关键值为优先级。

- 堆排序算法:
  - 将要排序的n个元素初始化为一个大(小)根堆.
  - 每次从堆中提取(即删除)元素。
    - 如果使用大根堆,各元素将按非递增次序排列。
    - 如果使用小根堆,各元素将按非递减次序排列。

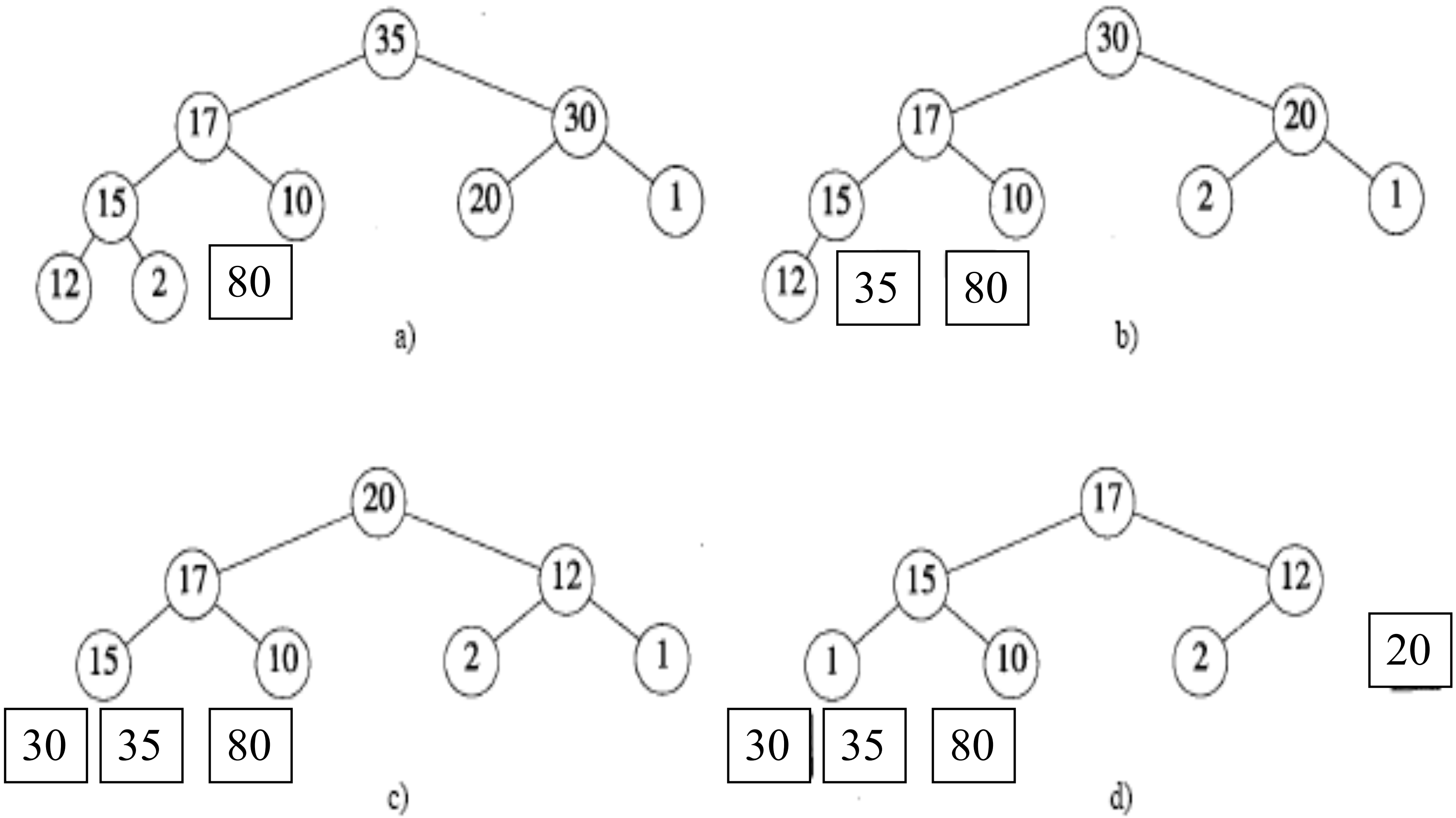
例：堆排序

- A=[-，20，12，35，15，10，80，30，17，2，1]
- 初始化大根堆



d）

例：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)
{// 利用堆排序算法对数组a[1:n] 进行排序

//在数组上创建一个大根堆
maxHeap<T> heap(1);
heap.initialize(a, n);
// 从大根堆中逐个抽取元素
for (int i = n-1; i >= 1; i--) {
     T x=heap.top();
     heap.pop();
     a[i+1] = x;
     }
heap.deactivateArray(); // 从堆的析构函数中保存数组a
}
```

堆排序的时间复杂性分析

- 对 *n* 个元素进行排序：
  - 初始化所需要的时间为：$\Theta(n)$
  - 每次删除所需要的时间为： $O(\log n)$
  - 共有n个元素，总时间为： $\mathbf{O(n\log n)}$

12.6.3 霍夫曼编码(Huffman Codes)

- 基于LZW算法的文本压缩工具，利用了字符串在文本中重复出现的规律。
- 霍夫曼编码(Huffman code)是另外一种文本压缩算法，它根据不同符号在一段文字中的相对出现频率来进行压缩编码。

霍夫曼编码

- 假设:
  - 文本是由a,x, u, z组成的字符串(aaxuaxaxz……);
  - 这个字符串的长度为:1000
- 每个字符用一个字节来存储.
  - 字符串: 1000 个字节(8000位).
- 每个字符的编码具有相同的位数(两位)
  - a:00; x:01; u: 10; z:11
  - 字符串: 2000位.

霍夫曼编码

- 编码表:
  - 采用如下格式来存储:
  - 符号个数，代码1，符号1，代码2，符号2，……
- 每个字符的编码长度= $\lceil \log_2(\text{符号个数}) \rceil$
-
- aaxuaxz ⇔ 00000110000111

(编码总长度:14bits)

霍夫曼编码

- 霍夫曼编码根据不同符号在一段文字中的**相对出现频率**来设计压缩编码。
- **频率(frequency)**：一个字符出现的次数称为频率．
- aaxuaxz:
  - 频率:　a:3　x:2　u:1　z:1

霍夫曼编码

- 编码： a:0   x:10   u:110   z:111
- aaxuaxz: 0010110010111
  - 编码总长度:13bits (<14bits)。当频率相差大时这种差别会更为明显。
- 当每个字符出现的频率有很大变化时，可以通过**可变长的编码**来降低编码串的长度。
- 0010110010111 ⇒ aaxuaxz

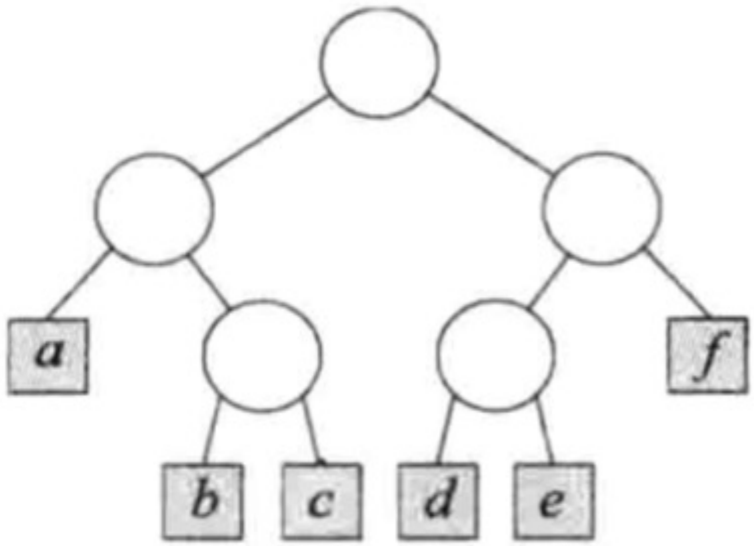
霍夫曼编码

- 编码： a:0   x:01   u:001   z:111
- aaxuaxz: 0001001001111
- 0001001001111
  - ⇒ ？
  - 无法译码

- 若使用可变长编码，编码需要满足：
  - **没有任何一个代码是另一代码的前缀**

霍夫曼编码

- 霍夫曼编码使用扩充二叉树(外部节点对应于字符串中被编码的字符)



a: 00

b: 010

c: 011

d: 100

e: 101

f: 11

霍夫曼树(Huffman Trees)

- 扩充二叉树(外部节点标记为1..n)的**加权外部路径长度**(Weighted External Path length):

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

- **L(i)** —从根到达外部节点*i* 的路径长度(即路径的边数);
- **F(i)** —外部节点*i* 的权值(weight) .
- 如果F(i)是字符串中被编码的字符的频率, WEP 就是**压缩编码串的长度**.
- **霍夫曼树**：对于给定的频率具有<u>**最小加权外部路径长度 的二叉树**</u>。

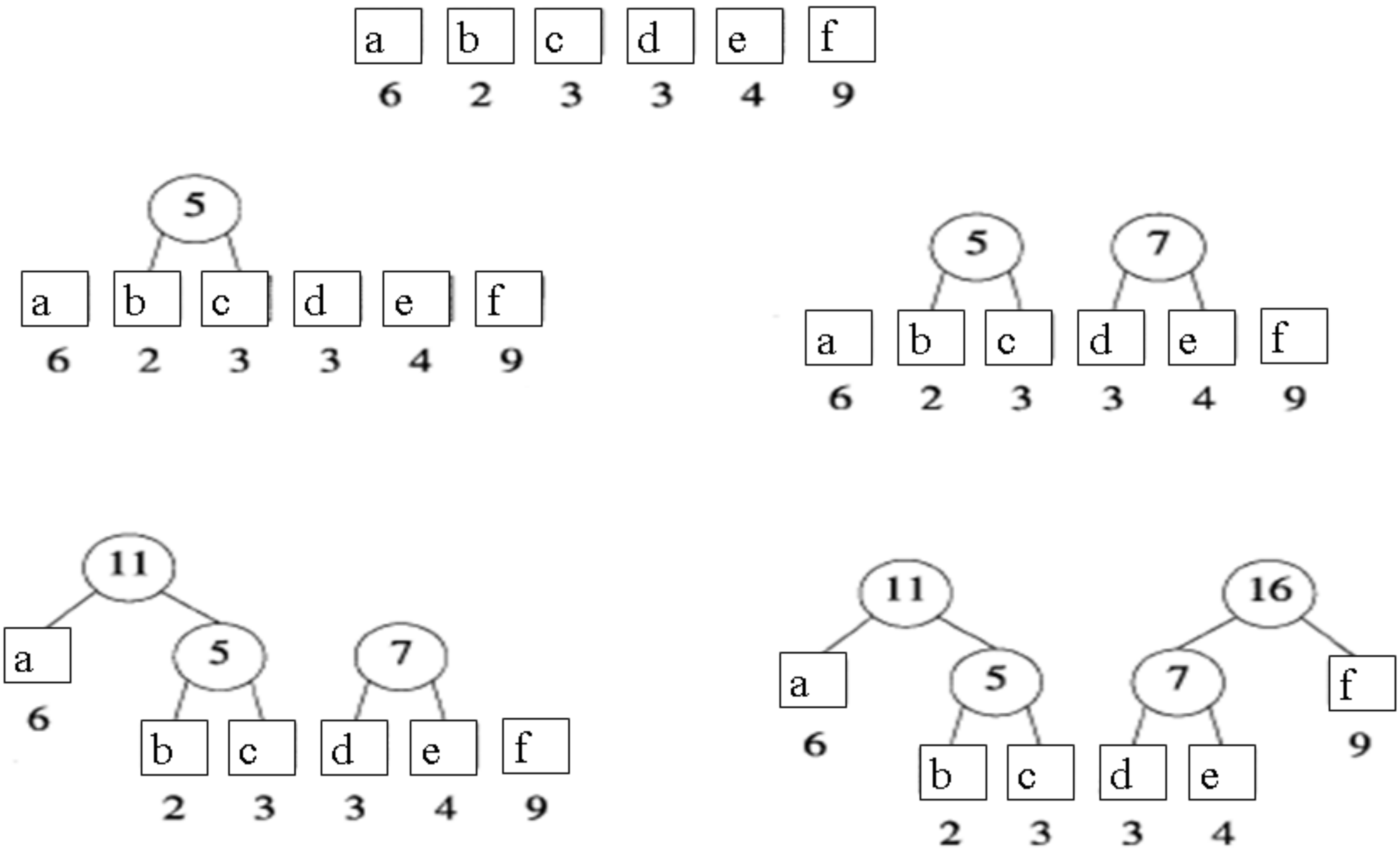
利用霍夫曼编码进行文本压缩编码

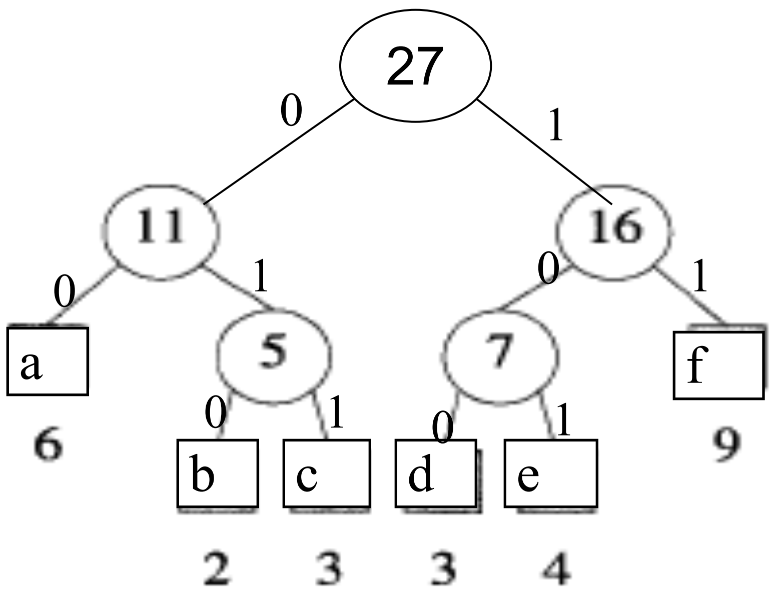
- 利用霍夫曼编码对字符串或一段文本进行压缩编码：
  - 1.确定不同字符的频率。
  - 2. 建立具有最小加权外部路径的二叉树(即霍夫曼树)，树的外部节点用字符串中的字符表示，外部节点的权值(weight)即为该字符的频率。
  - 3.遍历从根到外部节点的路径得到每个字符的编码。
  - 4.用字符的编码来代替字符串中的字符。

构造霍夫曼树

- 构造霍夫曼树：
- 1. 初始化二叉树集合，每个二叉树含一个外部节点，每个外部节点代表字符串中一个不同的字符.
- 2. 从集合中选择两棵具有最小权值的二叉树，并把它们合并成一棵新的二叉树。合并方法是把这两棵二叉树分别作为左右子树，然后增加一个新的根节点。新二叉树的权值为两棵子树的权值之和。
- 重复第2步，直到仅剩下一棵树为止。

构造霍夫曼树示例


a b c d e f
6 2 3 3 4 9
5
a b c d e f
6 2 3 3 4 9
5 7
a b c d e f
6 2 3 3 4 9
11
a
6
5 7
b c d e f
2 3 3 4 9
11 16
a
6
5 7
f
9
b c d e
2 3 3 4



a: 00

b: 010

c: 011

d: 100

e: 101

f: 11

构造霍夫曼树的实现

- 二叉树使用链表描述(linkedBinaryTree<int>)
  - 二叉树节点(成员: leftChild,element,rightChild)中, element为int类型
    - 外部节点中element值是字符的标识，用1..n表示;
    - 内部节点中element值为0;

- linkedBinaryTree<T> 中合并两棵二叉树的方法

```
linkedBinaryTree<T> ::makeTree(T& element,
                               linkedBinaryTree<T>& left,
                               linkedBinaryTree<T>& right)
```

  - 创建一个二叉树,element作为根节点中元素,left作为左子树,right作为右子树

构造霍夫曼树的实现

- 小根堆的元素类型huffmanNode<T>：

```
template<class T>
class  huffmanNode<T>
{friend linkedBinaryTree<int> * huffmanTree(T[], int);
 public:
    operator T() const  {return weight;}//向类型T转换
 private:
     linkedBinaryTree<int> *tree;
     T  weight;
};
```

```
template <class T>
linkedBinaryTree <int>* HuffmanTree(T weight[], int n)
{// 用权值weight [1:n]构造霍夫曼树，n>=1
// 创建一组单节点树hNode数组
huffmanNode<T> *hNode = new huffmanNode<T> [n+1];
linkedBinaryTree<int> emptyTree;
for (int i = 1; i <= n; i++)
    {hNode[i].weight = weight[i];
     hNode[i].tree = new linkedBinaryTree<int>;
     hNode[i].tree ->makeTree(i,emptyTree,emptyTree);
    }

// 将一组单节点树hNode [1:n]变成一个小根堆
minHeap <huffmanNode<T>> heap(1);
heap.initialize(hNode, n);
```

```
//不断从最小堆中取出两棵树合并成一棵放入,直到剩下一棵
huffmanNode <T> w, x, y;
linkedBinaryTree<int> *z;
for(i=1; i<n; i++)
  {//从最小堆中选出两棵权值最小的树
x=heap.top(); heap.pop();
y=heap.top(); heap.pop();
//合并成一棵树w,放入堆
z= new linkedBinaryTree<int>;
z ->makeTree(0, *x.tree, *y.tree);
w.weight = x.weight + y.weight; w.tree=z;
heap.push(w);
delete x.tree;
delete y.tree;
}
return heap.top().tree;
}
```

Ch12 作业

- P320 练习26
- P321 练习40.(1) (2)