第12章

优先级队列

定义和应用

- 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
- 与FIFO结构的队列不同，优先级队列出队的顺序是由元素优先级决定
  - 航班登机队列
- 相关操作
  - 查找一个元素（top）
  - 插入一个元素（push）
  - 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)
- 在最小优先级队列中，“查找/删除”操作优先级最小的元素
  - 最大优先级队列反之
- 优先级队列中的元素可以有相同的优先级
  - “查找/删除”同优先级元素按任意顺序

抽象数据类型

- 以最大优先级队列为例

```
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
}
```

```
template<class T>
class maxPriorityQueue {
public:
    virtual ~maxPriorityQueue() {}
    virtual bool empty() const = 0;
    virtual int size() const = 0;
    virtual const T& top() = 0;
    virtual void pop() = 0;
    virtual void push(const T& theElement) = 0;
};
```

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)

线性表

- 采用无序线性表描述最大优先级队列
  - 数组描述
    - 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，时间：Θ(n)

线性表

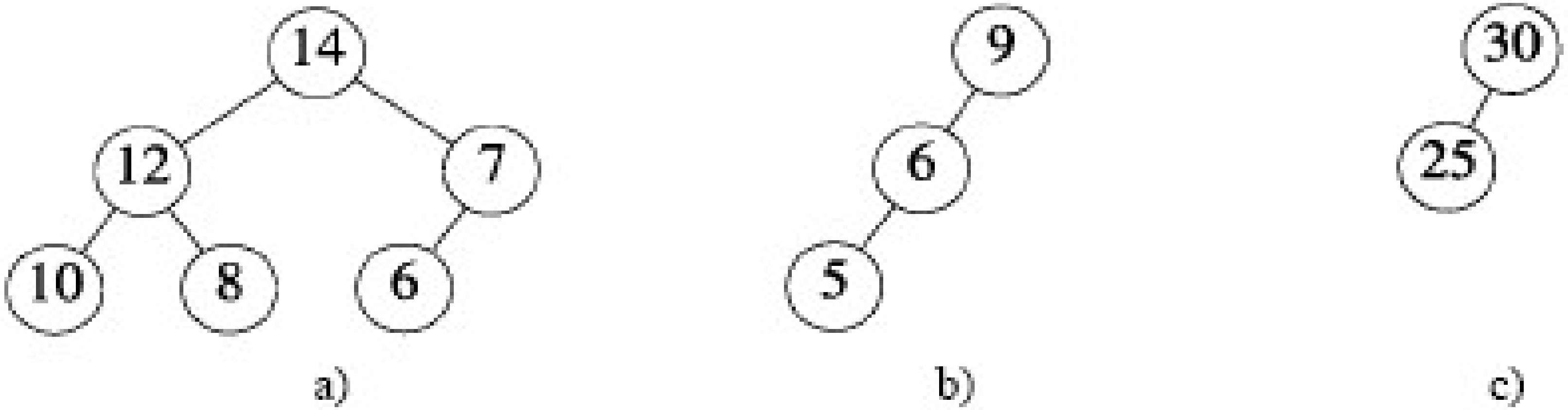
- 采用有序线性表描述最大优先级队列
  - 数组描述
    - 插入：先查找插入元素的位置，时间：O(n)
    - 删除：删除最右元素，时间：$\Theta(1)$
  - 链表描述(按递减次序排列)
    - 插入：先查找插入元素的位置，时间：O(n)
    - 删除：表头删除，时间：$\Theta(1)$

堆(Heaps)

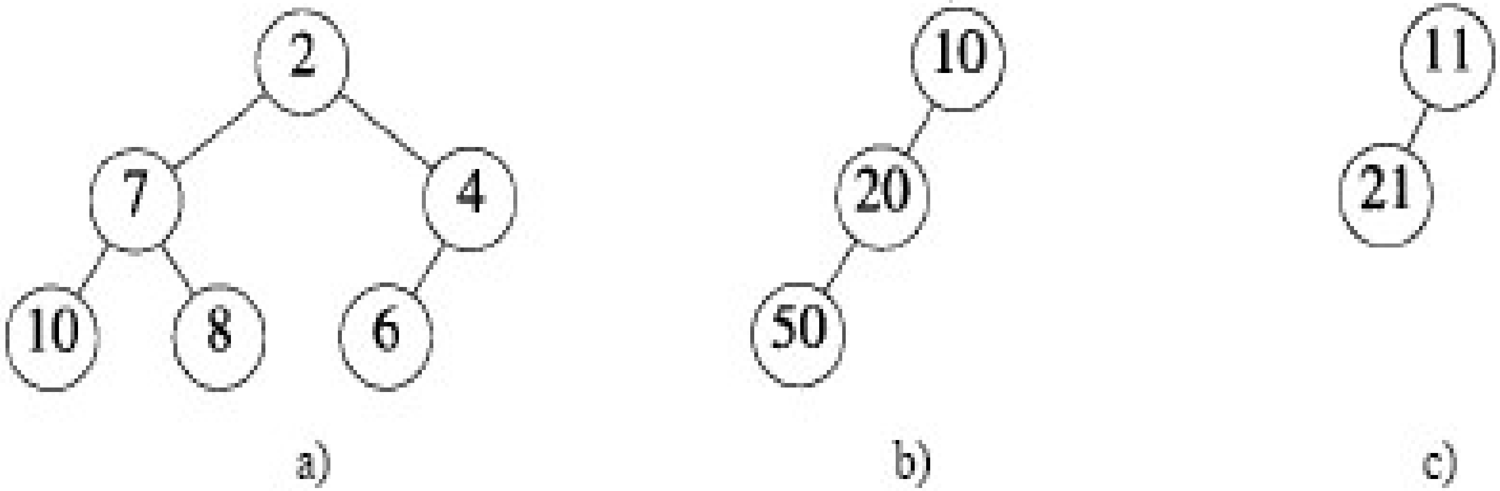
- 大根树(小根树)： 每个节点的值都大于(小于)或等于其子节点(如果有的话)值的树

- 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



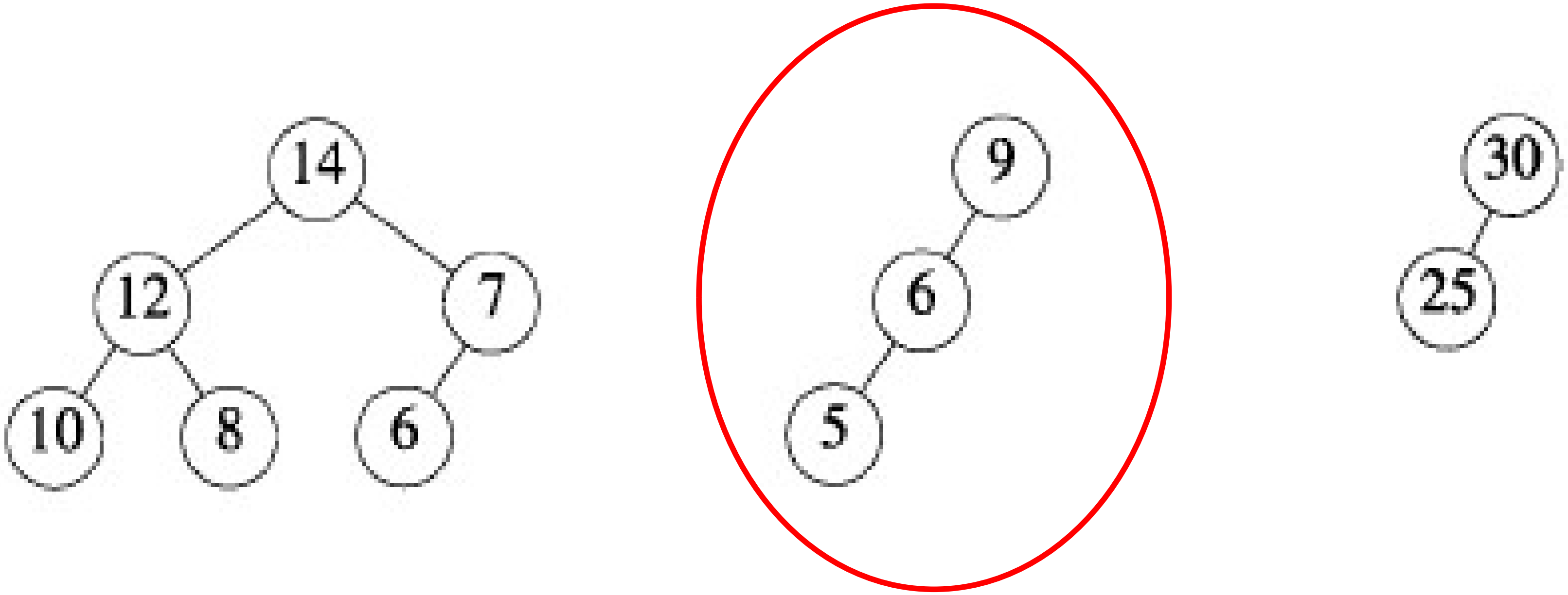
大根树



小根树

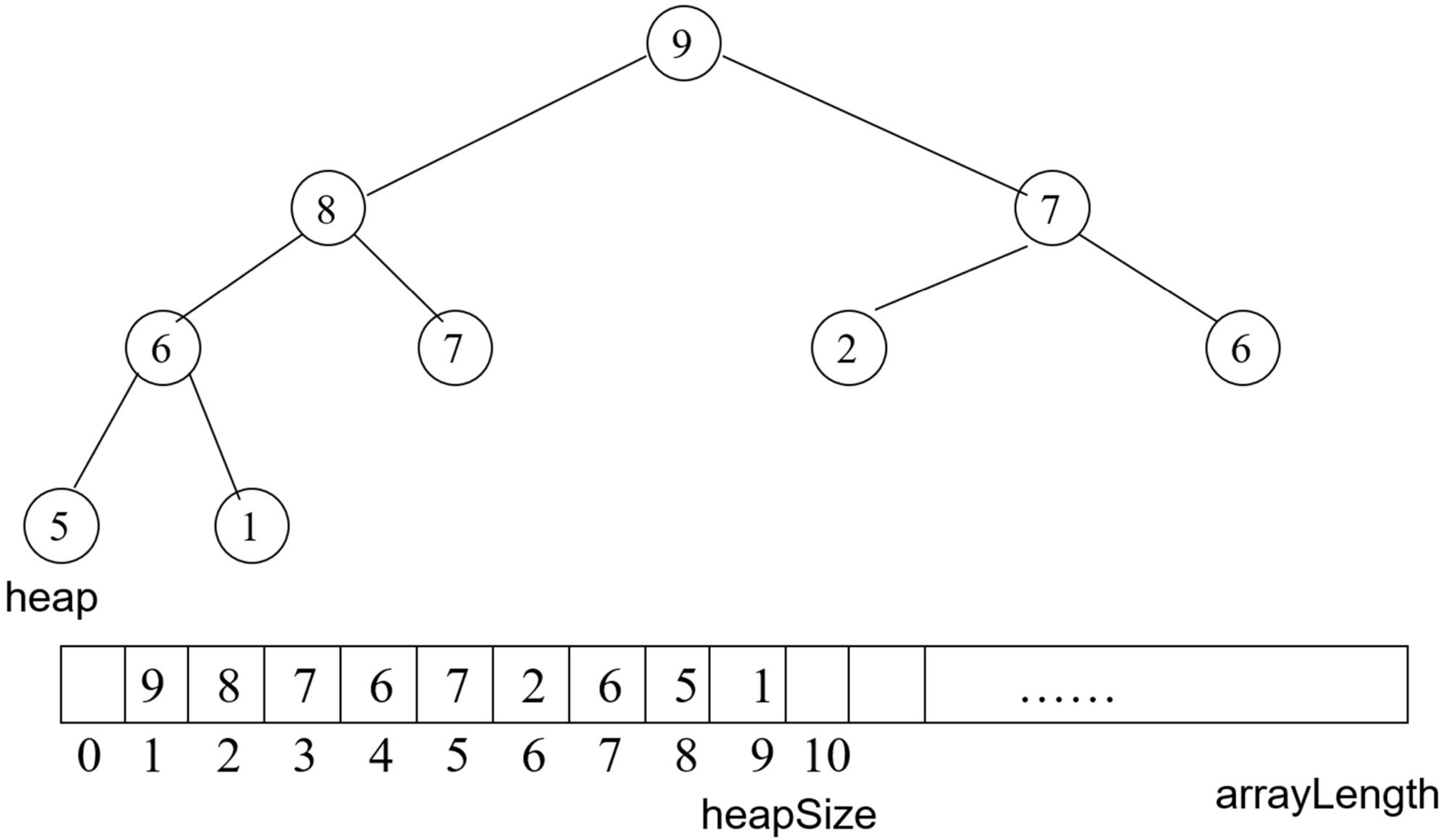
最大堆和最小堆

- 大根堆（小根堆）：既是大根树（小根树），又是完全二叉树
- 又称最大堆或最小堆



堆的描述

- 堆是完全二叉树，可用一维数组有效地描述堆



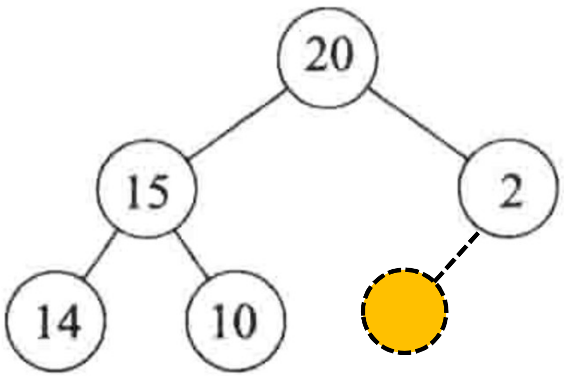
类maxHeap

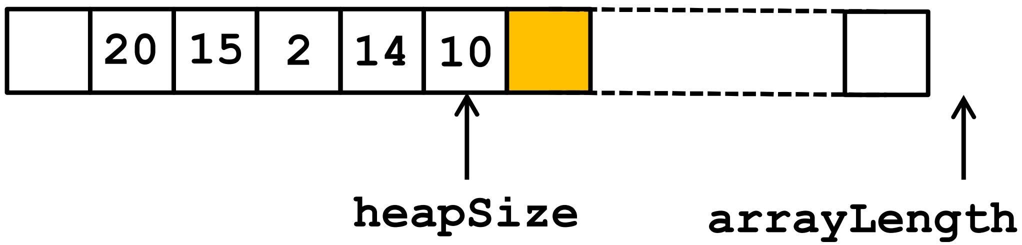
heap		9	8	7	6	7	2	6	5	1			……
	0	1	2	3	4	5	6	7	8	9	10		arrayLength
										heapSize			

```
template<class T>
class maxHeap : public maxPriorityQueue<T> {
private:
    int heapSize;         // 队列中元素的个数
    int arrayLength;      // 数组容量=队列容量+1
    T *heap;              // 元素数组
public:
    maxHeap(int initialCapacity = 10);
    ~maxHeap() { delete [] heap; }
    bool empty() const { return heapSize == 0; }
    int size() const { return heapSize; }
    const T& top() { // 返回最大的元素
        if (heapSize == 0)
            throw queueEmpty();
        return heap[1];
    }
    void pop();
    void push(const T&);
    void initialize(T *, int);
    void deactivateArray() // 让heap指针不再指向数据数组
        { heap = nullptr; arrayLength = heapSize = 0;}
    void output(ostream& out) const;
};
```

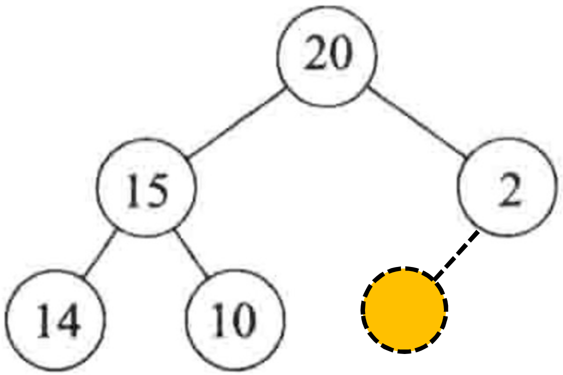
大根堆的插入

- 在大根堆中插入一个新的元素，如何保证还是大根堆?
  - 插入位置在数组末尾

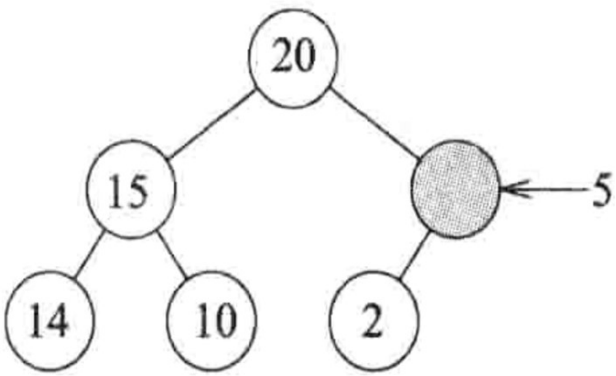




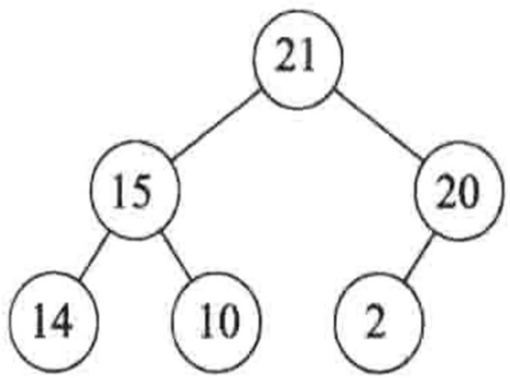
  - 调整元素位置使结果仍为大根堆
    - 从插入位置开始（到根结束），如果父节点比插入节点大，则交换



插入5



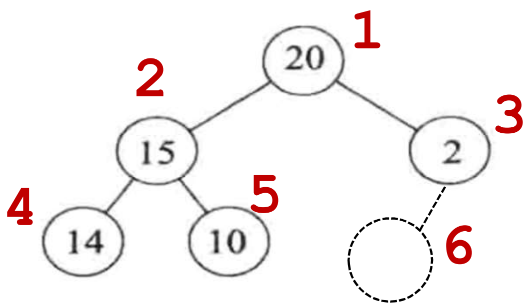
插入21



push方法

```
template<class T>
void maxHeap<T>::push(const T& theElement)
{
    if (heapSize == arrayLength-1) {
        changeLength1D(heap, arrayLength, 2 * arrayLength);
        arrayLength *= 2;
    }

    // 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;
}
```

插入21



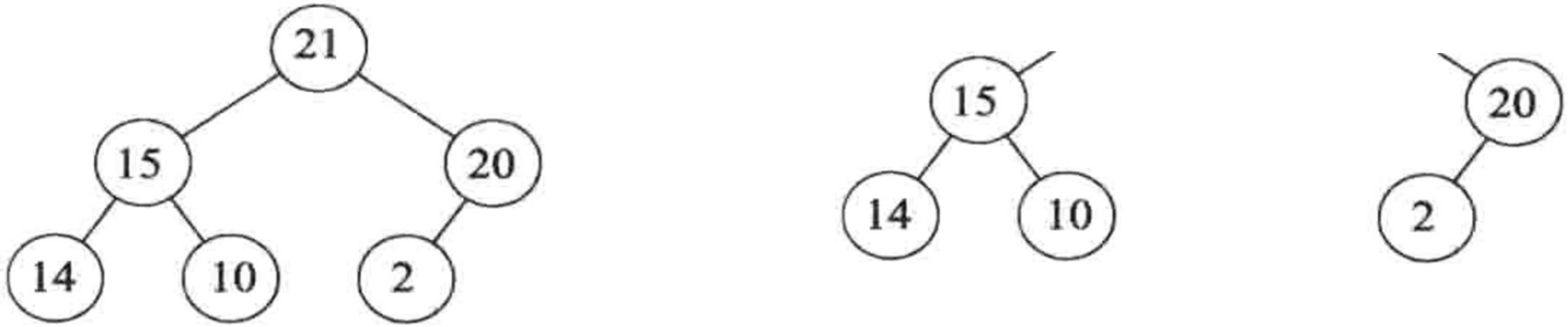
大根堆的插入性能

- 大根堆插入的时间复杂度
  - 每一层的操作复杂度为$\Theta(1)$
  - 操作的层数为大根堆的高度
- 在一个包含了n个元素的大根堆中插入一个元素的时间复杂度为

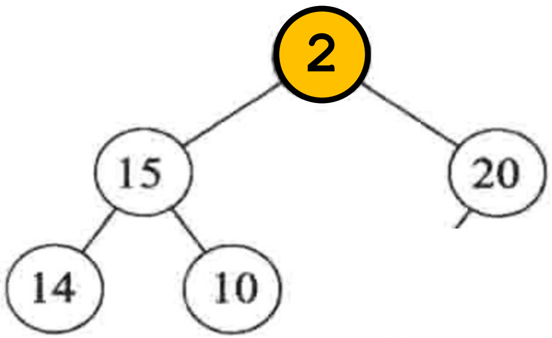
$$O(height) = O(\log_2 n)$$

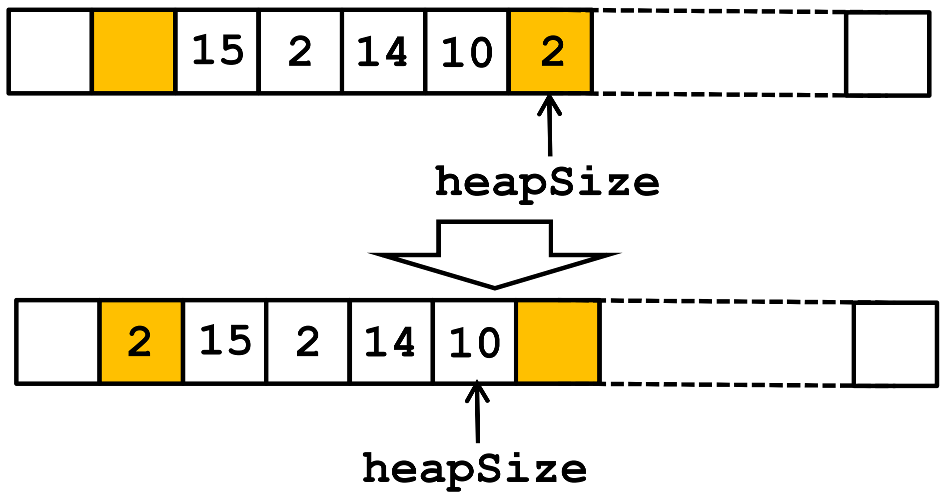
大根堆的删除

- 大根堆的删除操作总是移出根节点
  - 如何重新得到一个大根堆?



- 变回完全二叉树：最后一个元素移至根

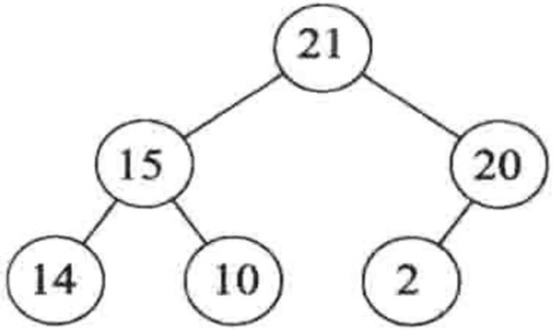


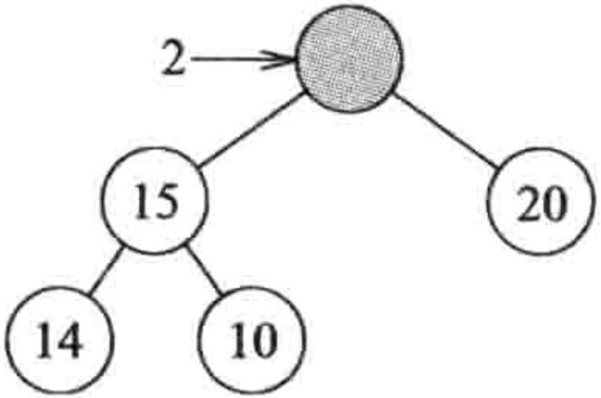


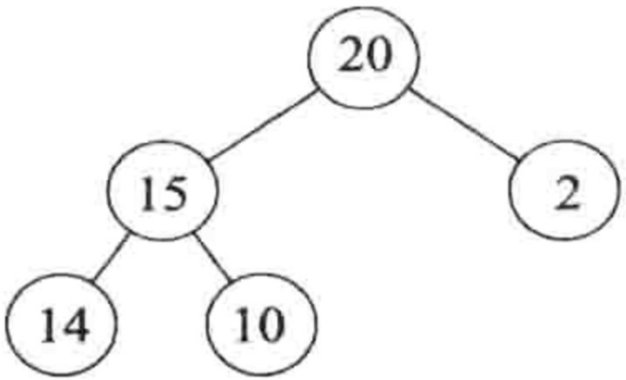
- 重新堆化

大根堆的删除

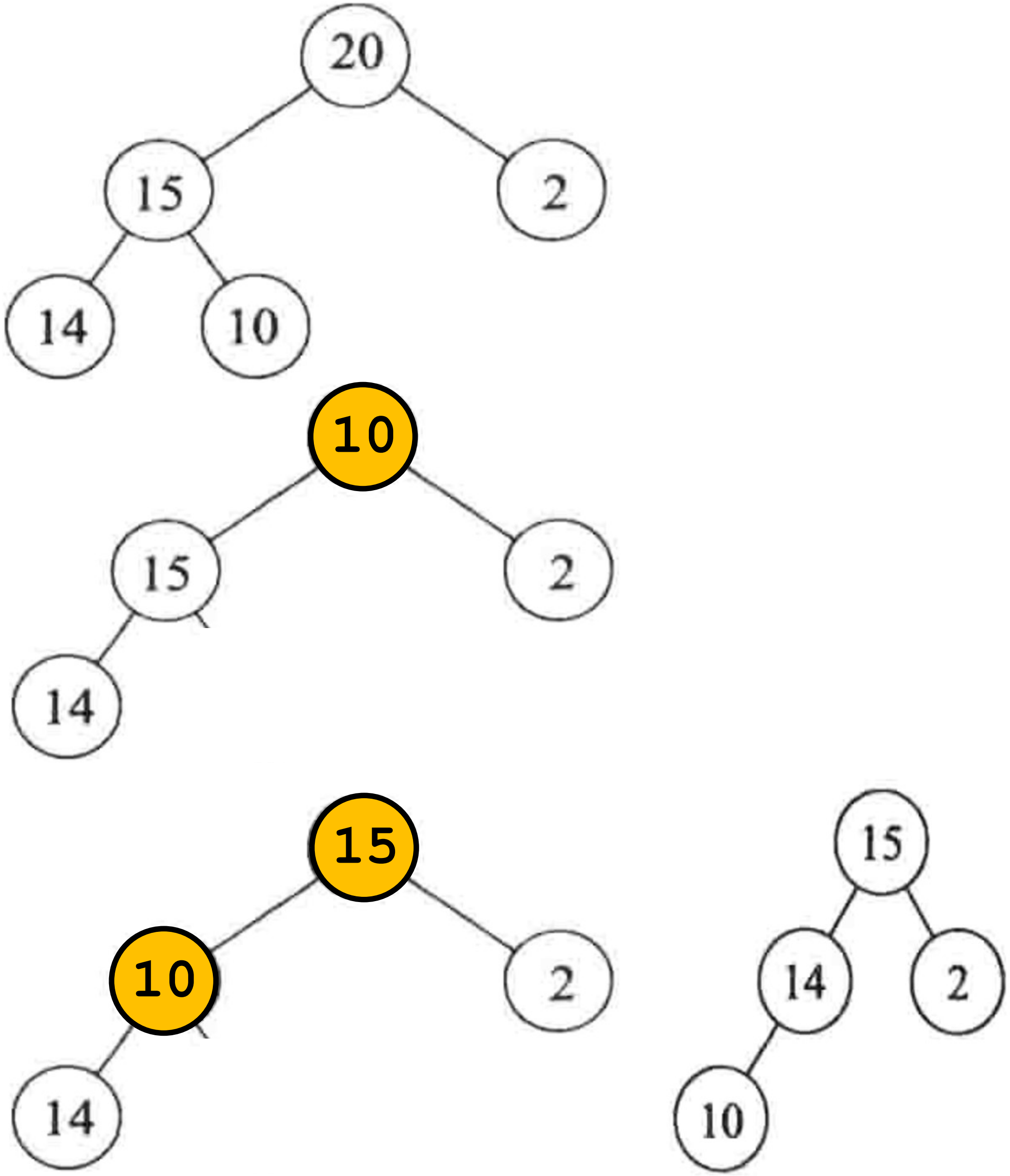
- 删除最大元素21
- 最后一个元素(2)放入根中
  - 不再是堆
- 重新堆化：将根的孩子中较大者(20)与根交换
  - 堆化前，两个子树都为大根堆，交换后只需考虑被交换的子树
  - 沿树调用至：1）孩子节点没有更大的元素；2）抵达叶节点







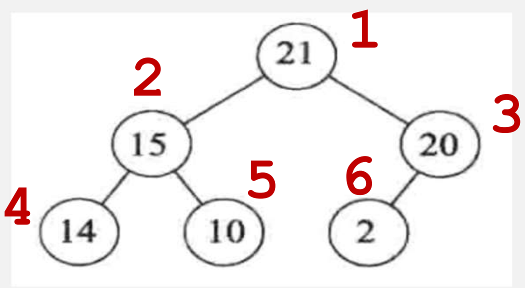
大根堆的删除



- 删除最大元素20
- 10放入根上，不是堆，将根的孩子的大者15与10交换
- 10放入位置2后，不是堆，将位置2的孩子的大者14与10交换
- 抵达叶子节点，结束

方法pop的实现

```
template<class T>
void maxHeap<T>::pop() { // 删除最大的元素，即树根
    if (heapSize == 0)
        throw queueEmpty();

    heap[1].~T();

    // 取出最后一个元素，更新堆大小
    T lastElement = heap[heapSize--];

    // 从根节点开始搜索放置最后一个元素的位置
    int currentNode = 1,
        child = 2;        // 指向currentNode孩子节点中较大的一个
    while (child <= heapSize) { // 抵达叶子节点?
        // 比较两个孩子节点，让child指向更大的一个
        if (child < heapSize && heap[child] < heap[child + 1])
            child++;

        // 最后一个元素是否比两个孩子节点都大?
        if (lastElement >= heap[child])
            break;    // 是

        // 不是
        heap[currentNode] = heap[child]; // 较大的孩子节点上移
        currentNode = child;             // 继续检查被上移的孩子节点的位置
        child *= 2;                      // 更新child指向currentNode的左孩子
    }
    heap[currentNode] = lastElement;     // 找到位置，填入最后一个元素
}
```



大根堆的删除性能

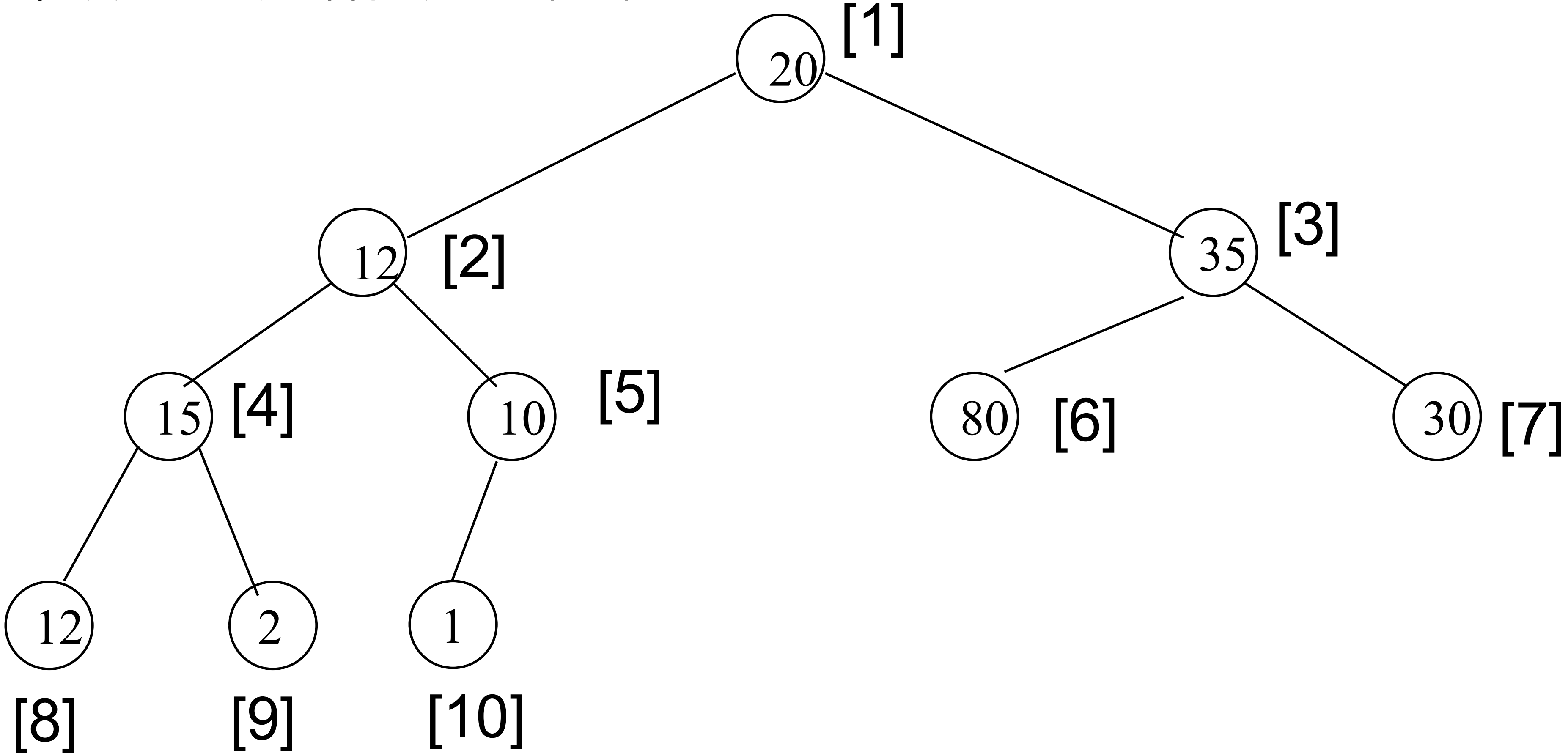
- 大根堆中删除一个元素的时间复杂度
  - 删除的时间复杂度与插入的时间复杂度相同
  - 每一层的操作耗时$\Theta(1)$
  - 操作的层数最多为树的高度
  - 总的时间复杂度为$O(height) = O(\log_2 n)$，n是大根堆中元素的数量

大根堆的初始化

- 将n个元素初始化为一个大根堆
  - 将n个元素依次插入一个空的大根堆，时间复杂度为$O(nlog_2n)$
  - 能否实现更低的时间复杂度?

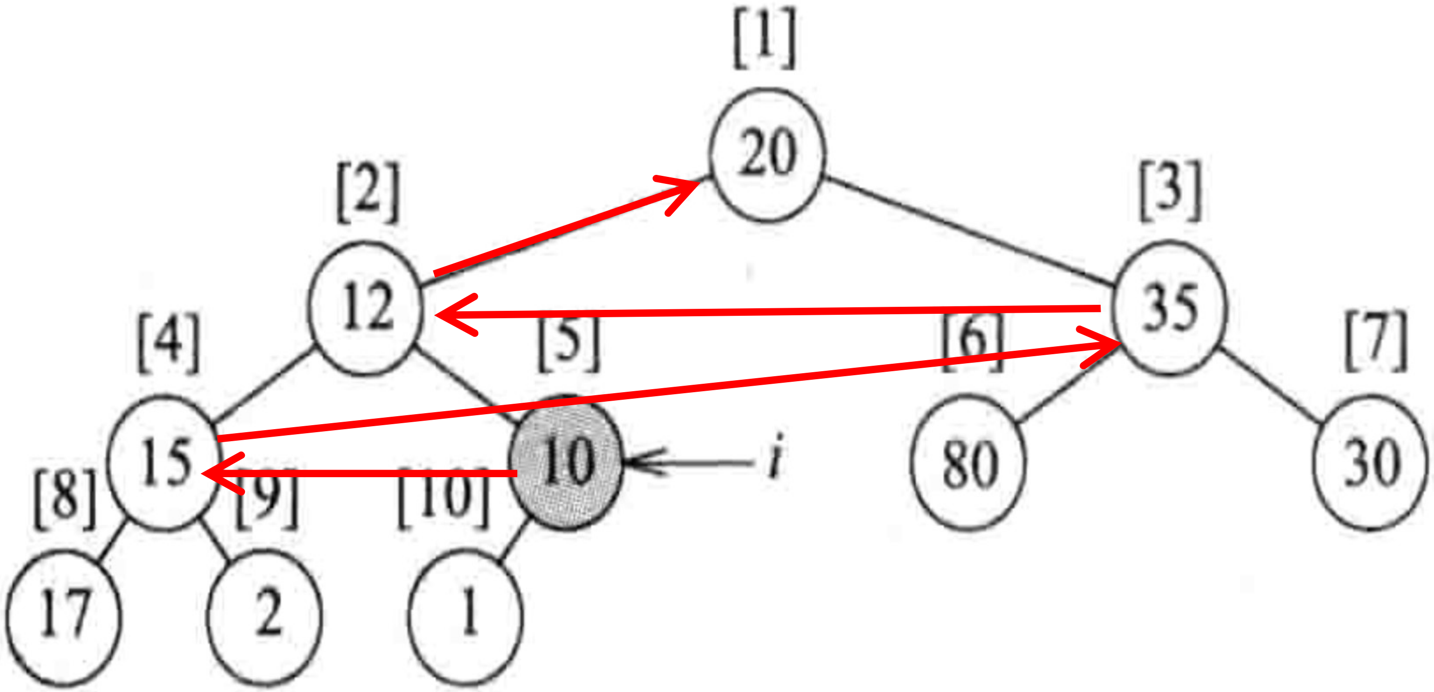
大根堆的初始化

- 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

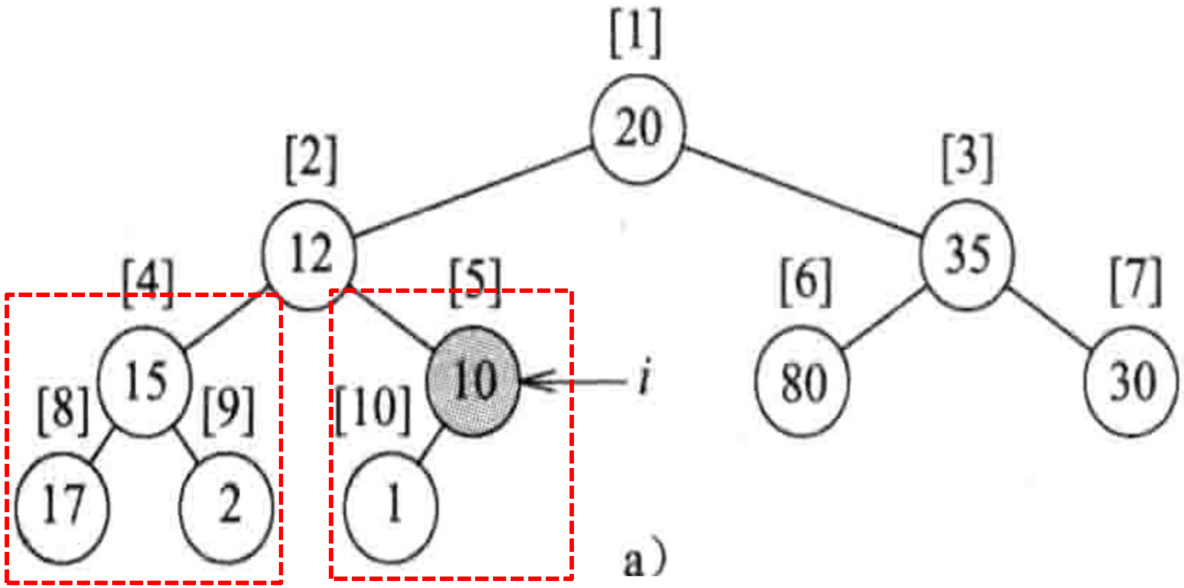


大根堆的初始化

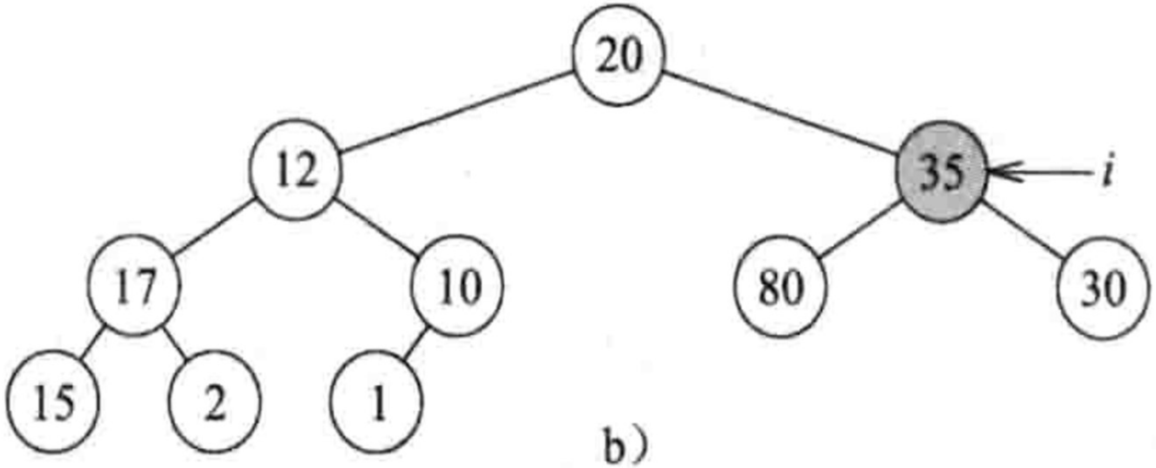
- 从大根堆的最右一个有孩子的节点开始进行堆化操作（同删除中堆化）
  - 叶子节点是最大堆
  - 堆化到上层节点时，其左右子树也为最大堆



大根堆的初始化

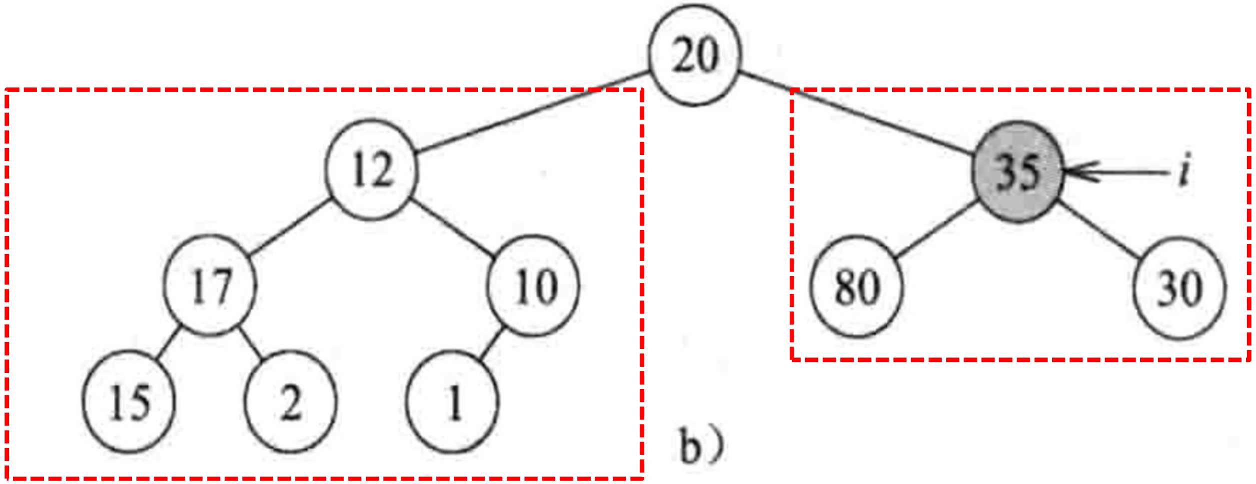


a)

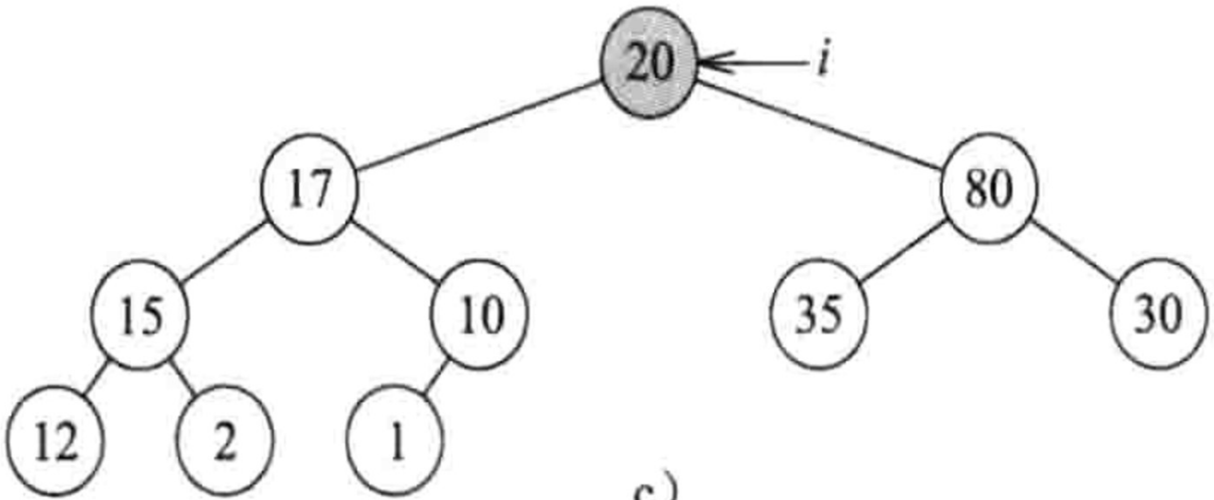


b)

大根堆的初始化

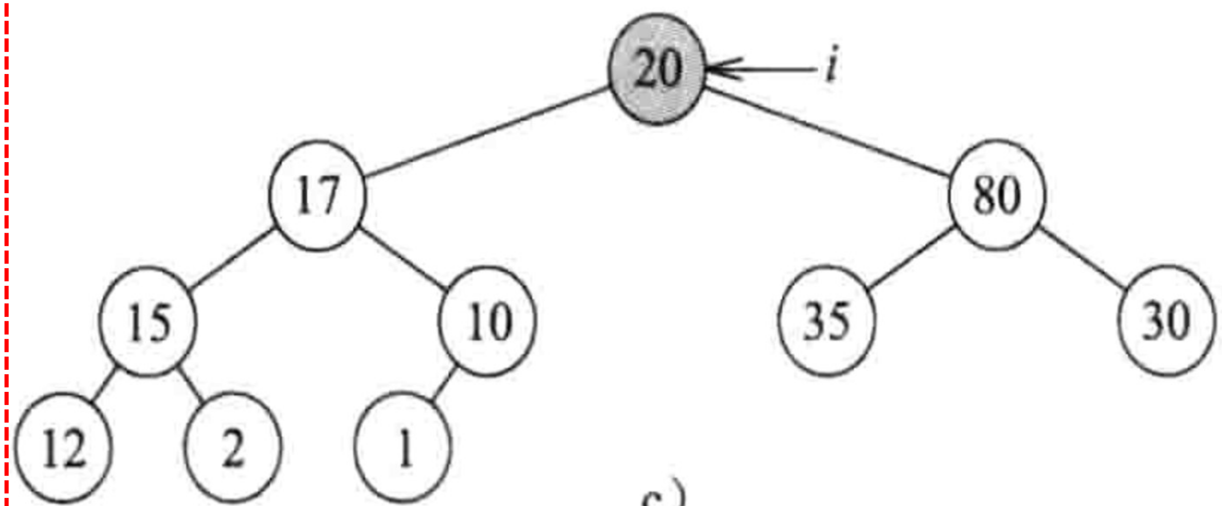


b）

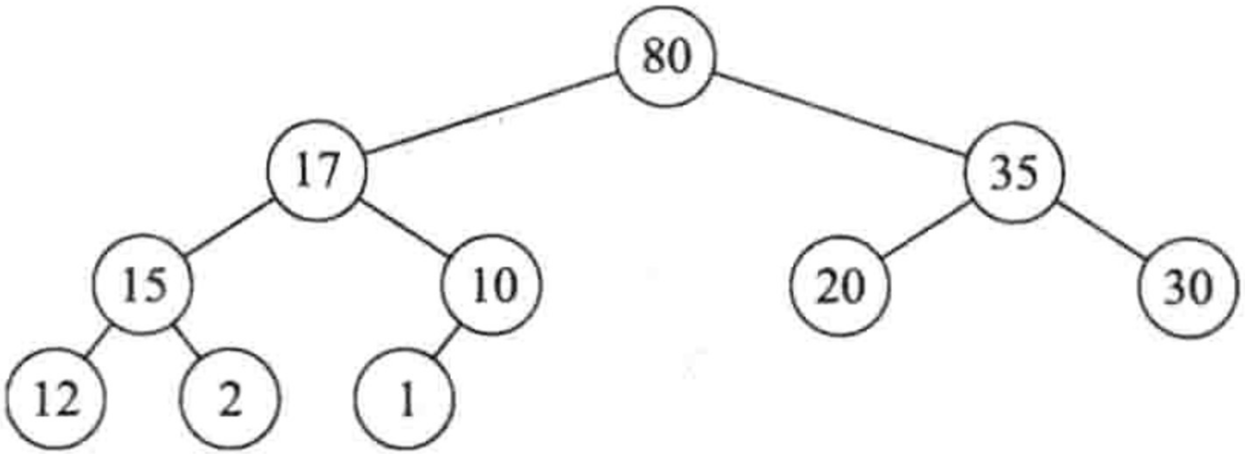


c）

大根堆的初始化

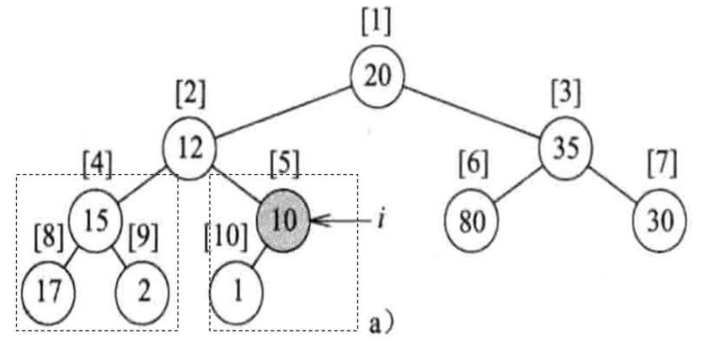


c）



d）

方法initialize实现




```
template<class T>
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，从最后一个有孩子的节点开始
    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; // 准备比较左孩子节点与当前节点
        while (child <= heapSize) {// 子树的堆化进行到叶子时结束
            // 确定两个孩子节点中较大者
            if (child < heapSize && heap[child] < heap[child+1])
                child++;
            // 当前节点是三者中最大的吗
            if (rootElement >= heap[child]) break; // 是，结束该子树的堆化
            // 不是
            heap[child/2] = heap[child]; // 将孩子上移
            child *= 2; // 下移一层，继续堆化
        }
        heap[child/2] = rootElement;
    }
}
```

初始化堆的性能

- 堆的高度h
- 在j层节点的数目 $\leq 2^{j-1}$
- 以j层节点为根的子树的高度 $h-j+1$
- 调整第j层节点的复杂度为 $O(2^{j-1}\cdot(h-j+1))$
- 调整整个树的复杂度为 $O(\Sigma_{j=1}^{h-1}2^{j-1}\cdot(h-j+1))$

$$O\left(\Sigma_{j=1}^{h-1}2^{j-1}\cdot(h-j+1)\right)=$$

$$O\left(2^0\cdot h+2^1\cdot(h-1)+\cdots+2^{h-2}\cdot 2\right)=$$

$$O\left(\Sigma_{k=2}^{h}k2^{h-k}\right)=O\left(2^h\Sigma_{k=2}^{h}\frac{k}{2^k}\right)$$

$$\Sigma_{k=1}^{m}\left(\frac{k}{2^k}\right)=2-\frac{m+2}{2^m}$$

数学归纳法证明

$$O\left(2^h\Sigma_{k=2}^{h}\frac{k}{2^k}\right)=\mathrm{O}\left(2^{\mathrm{h}}\left(2-\frac{\mathrm{h}+2}{2^{\mathrm{h}}}-\frac{1}{2}\right)\right)=$$

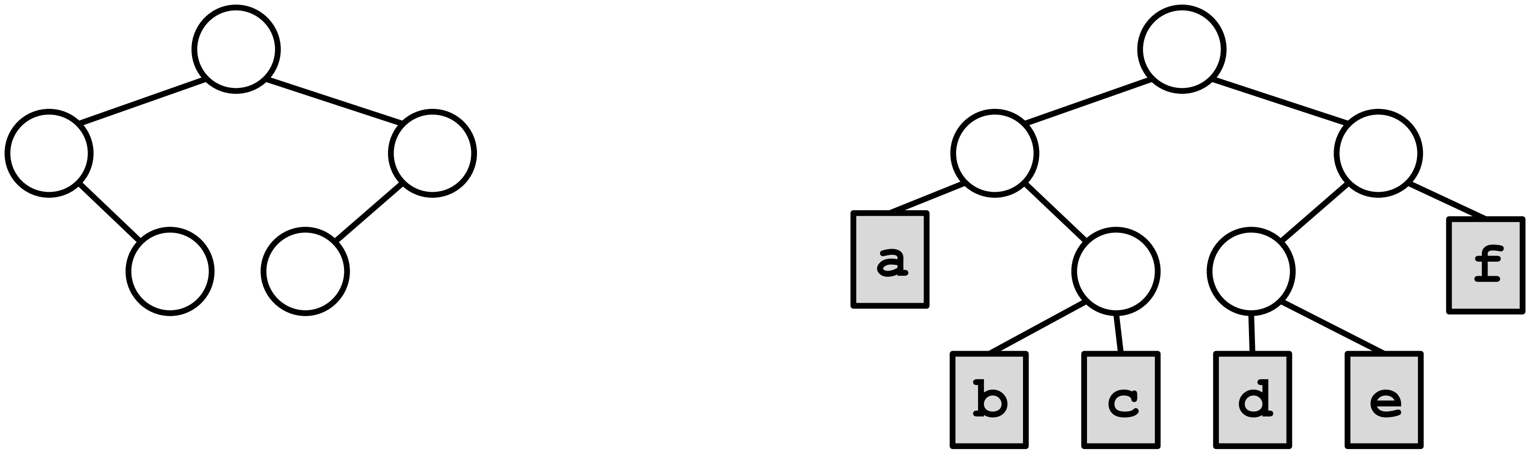
$$O\left(\frac{3}{2}\cdot 2^h-(h+2)\right)=\mathrm{O}(\mathrm{n})$$

左高树

- 最大/最小堆合并操作时复杂度较高
- 左高树(leftist tree)更适合于需要频繁合并的优先队列

扩充二叉树

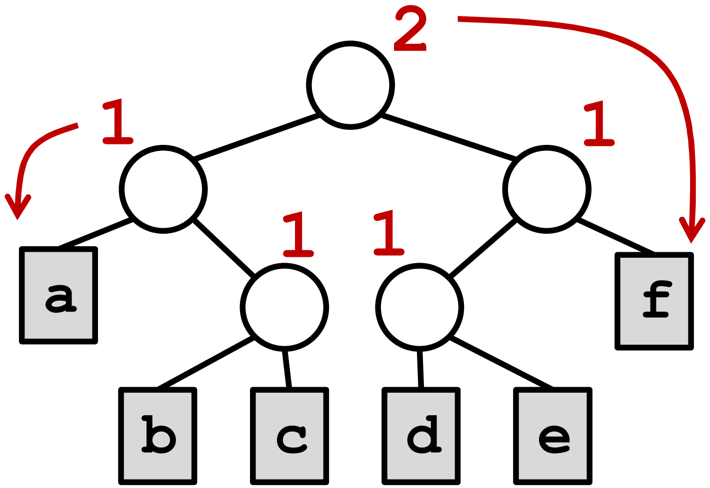
- 使用外部节点(external node)代替空子树
- 树的节点分为内部节点(internal node)和外部节点



- 扩充二叉树(extended binary tree)是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

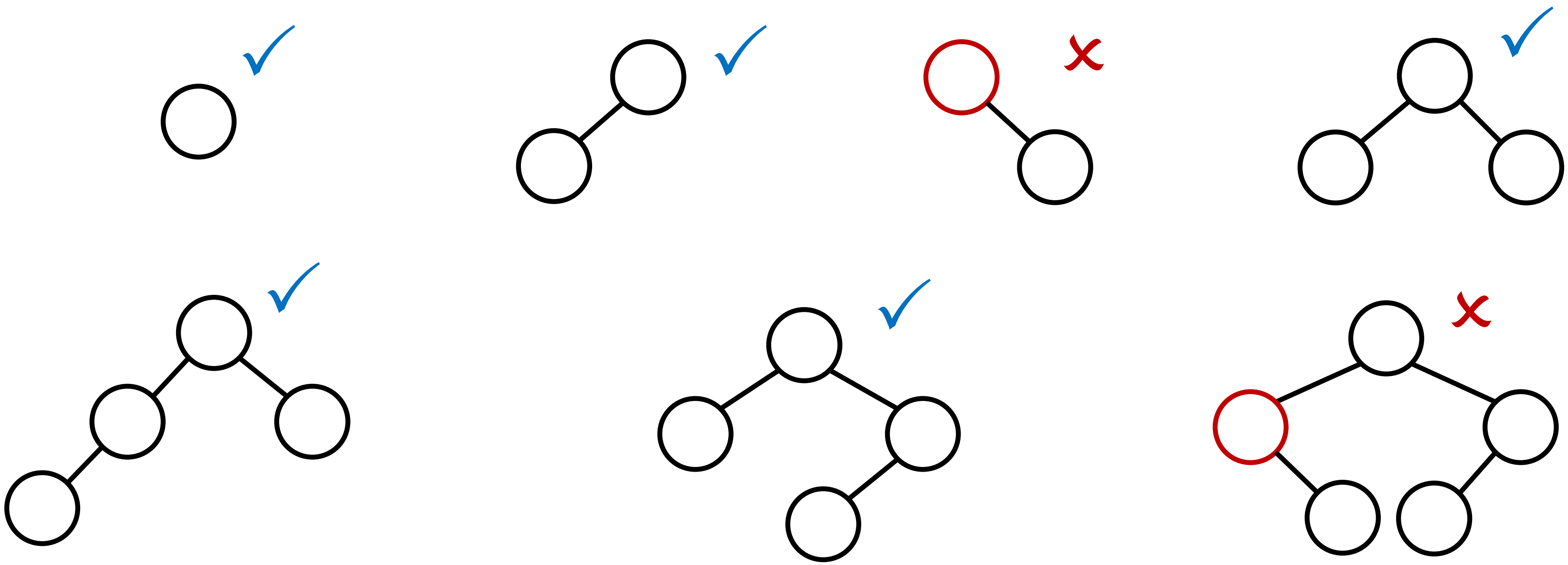


- $s(x) = \min\{s(L), s(R)\} + 1$，其中$L$和$R$分别为$x$的左、右孩子
  - 外部节点的s值为0，即$s(a) = 0$

高度优先左高树HBLT

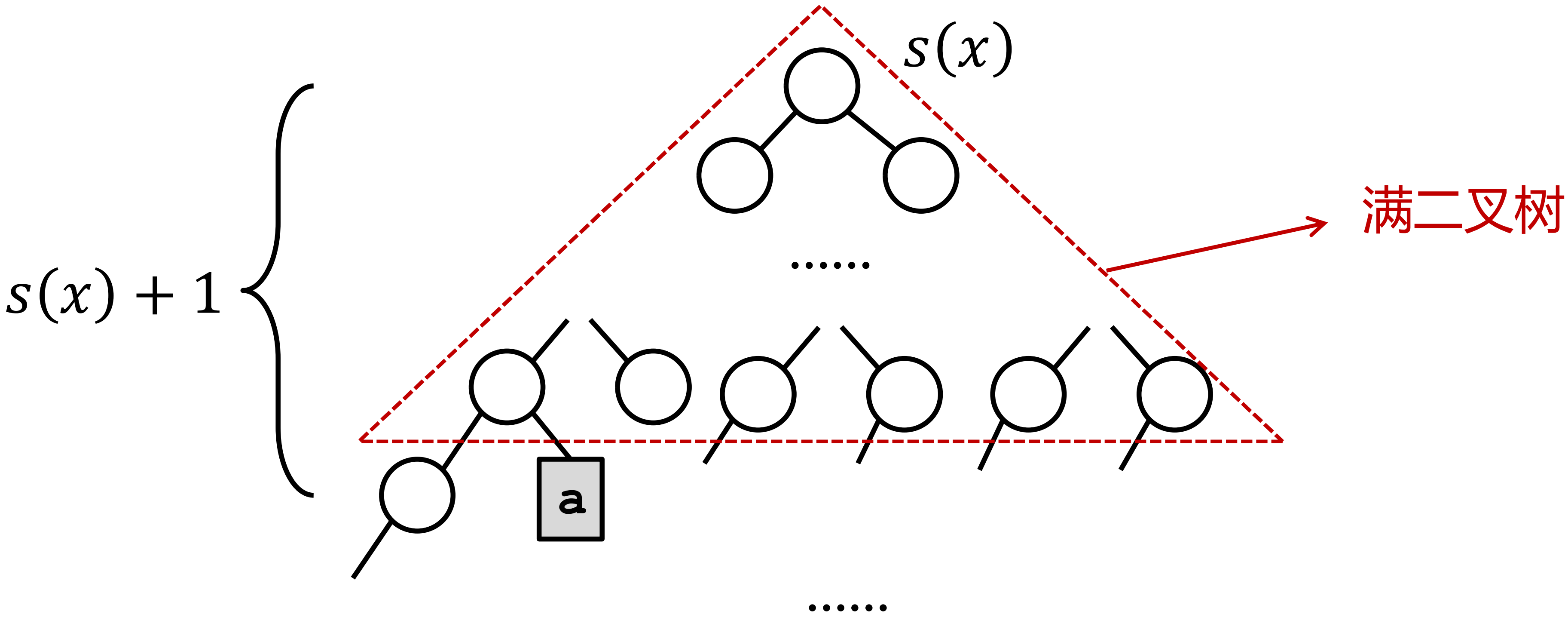
- 高度优先左高树(HBLT)：当且仅当一颗二叉树的任何一个内部节点，其左孩子的s值大于等于其右孩子的s值时，该二叉树为高度优先左高树
  - **height-biased leftist tree**

- 对任意一个内部节点x，有$s(L) \geq s(R)$



HBLT相关性质

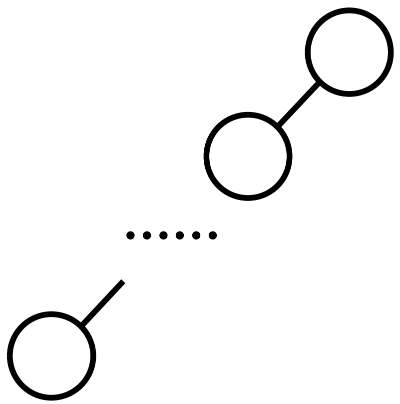
- 令x为HBLT的一个内部节点，则有
  - 1）以x为根的子树的节点数目至少有$2^{s(x)}-1$

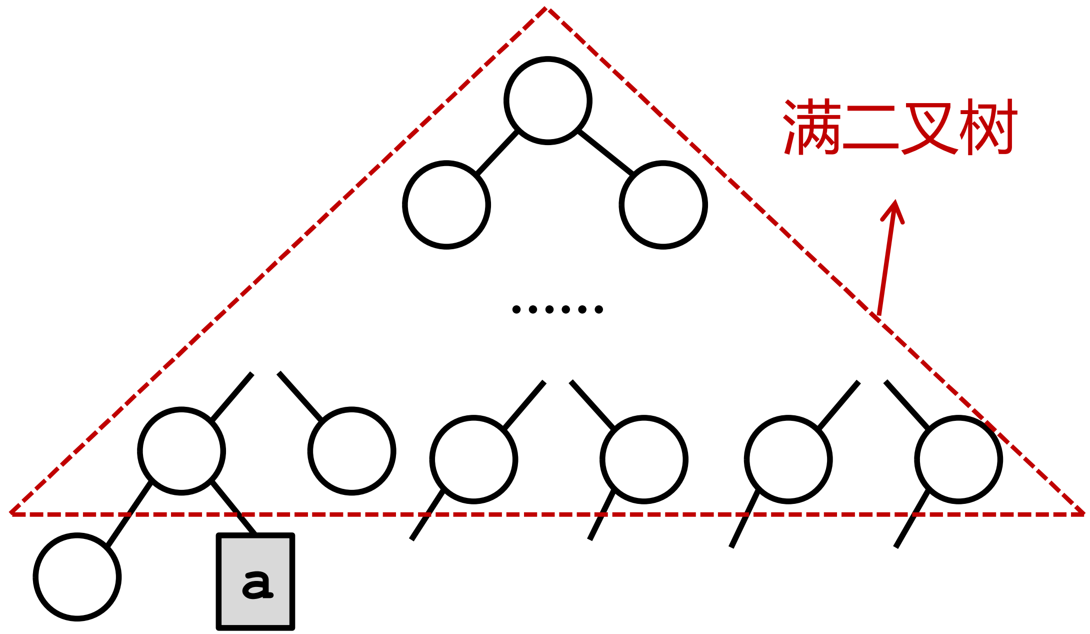


- 证明：
  - 根据定义，外部节点出现在子树的第$s(x)+1$层
  - 前$s(x)$层是一个满二叉树，故有$2^{s(x)}-1$个节点

HBLT相关性质

- 令x为HBLT的一个内部节点，则有
  - 1）以x为根的子树的节点数目至少有$2^{s(x)}-1$
  - 2）若以x为根的子树有m个节点，则$s(x)$最多为$\log_2(m+1)$



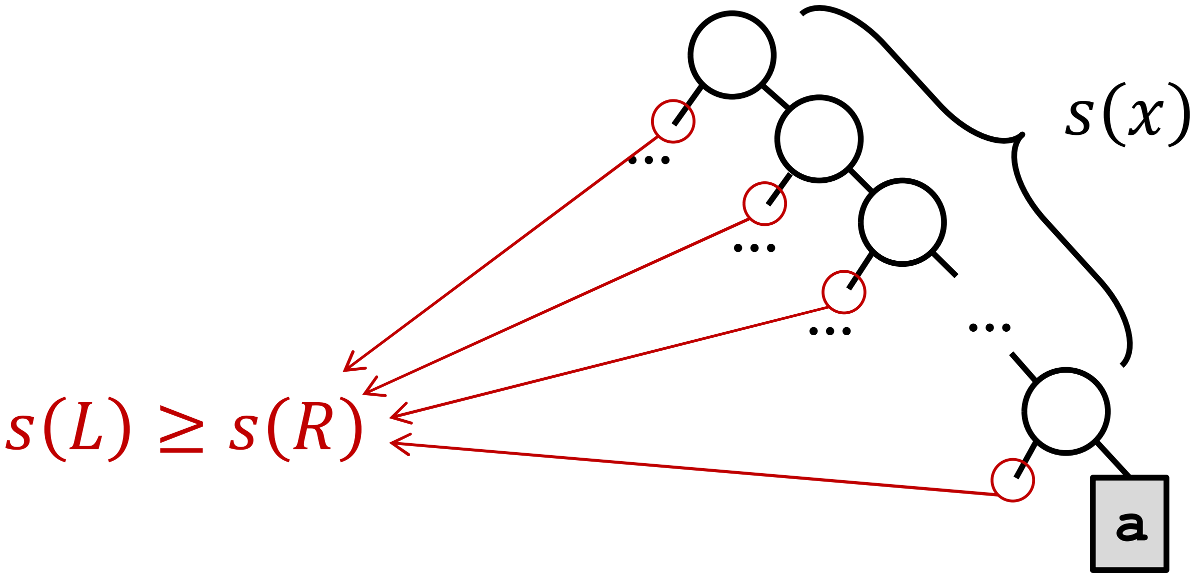


$s(x)$最少时$s(x)=1$

$s(x)$最多时$s(x)=\log_2(m+1)$

HBLT相关性质

- 令x为HBLT的一个内部节点，则有
  - 1）以x为根的子树的节点数目至少有$2^{s(x)}-1$
  - 2）若以x为根的子树有m个节点，则$s(x)$最多为$\log_2(m+1)$
  - 3）从x到一外部节点的最右路径（即从x开始沿右孩子移动的路径）的长度为$s(x)$

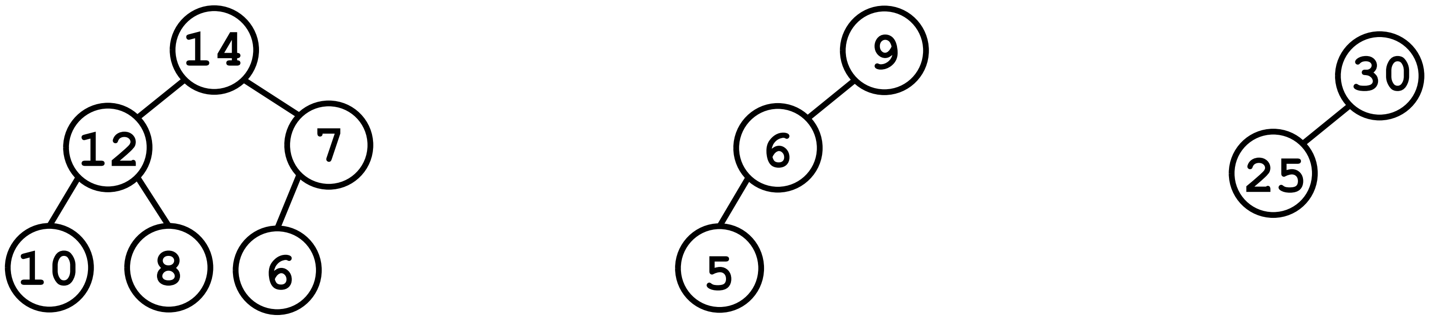


如果最右路径大于$s(x)$，节点x的s值必然更大

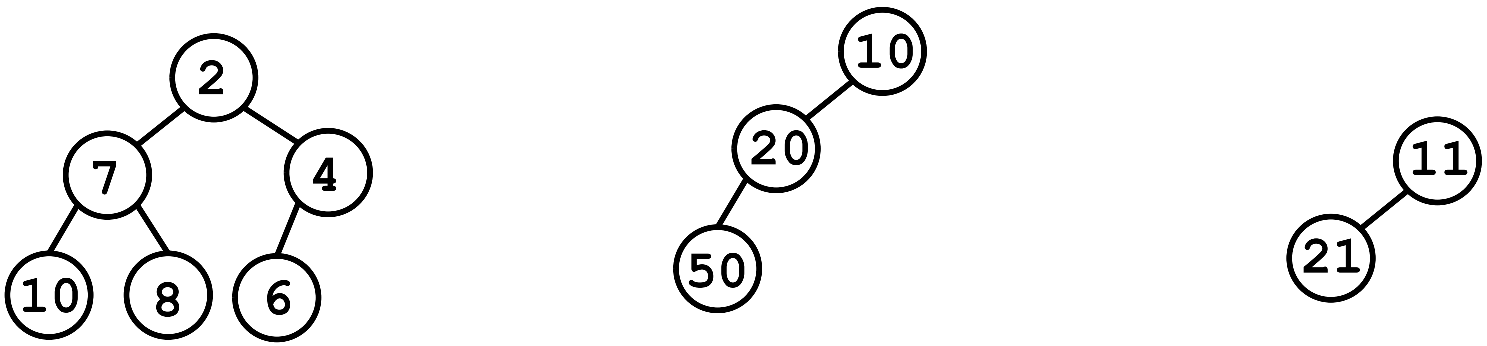
最大/最小HBLT

- 若一颗HBLT同时还是大/小根树，则称为最大/最小高度优先左高树(max/min HBLT)

max HBLT



min HBLT



重量优先左高树

- 重量优先左高树(WBLT)：当且仅当任一内部节点的左孩子的w值大于等于右孩子的w值时，一颗二叉树称为重量优先左高树(weight-biased leftist tree)

- $w(x)$计算x节点为根的子树的内部节点数目

- $w(x) = w(L) + w(R) + 1$
  - $w(x) = 0$，如果x为外部节点

- 若一颗WBLT同时还是大/小根树，则称为最大/最小重量优先左高树(max/min WBLT)

最大HBLT的插入和删除

- 插入和删除一个节点后，结果仍为最大HBLT
  - 插入和删除操作均基于HBLT的合并操作
- 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
- 删除：删除根节点（最大元素），合并根节点的左子树和右子树
  - 最大HBLT根节点的左子树和右子树均为最大HBLT

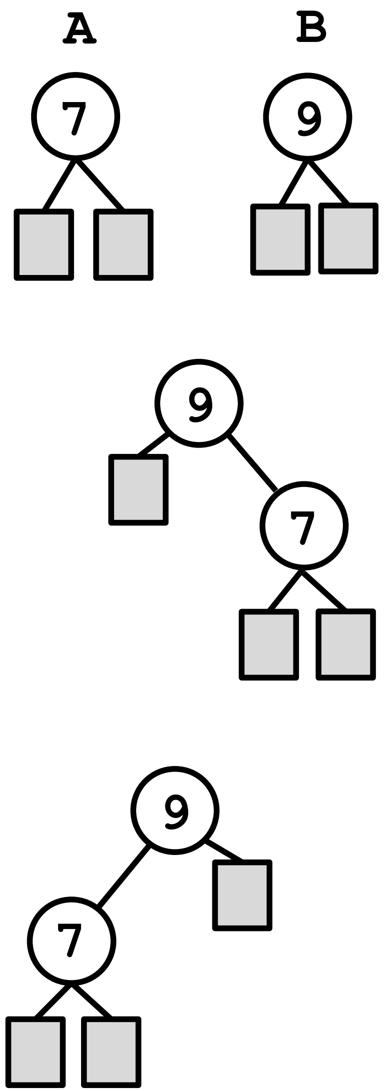
最大HBLT的合并操作

- 合并两个最大HBLT树，其结果仍为最大HBLT树

- 递归合并过程
  - 挑选两棵树A、B中根节点中更大的A作为新树的根节点
  - 将A的左子树作为新树的左子树
  - 将A的右子树与B进行合并操作，结果作为新树的右子树
    - A的右子树仍为最大HBLT
  - 如果新树中左子树的*s*值比右子树的*s*值更小，则交换

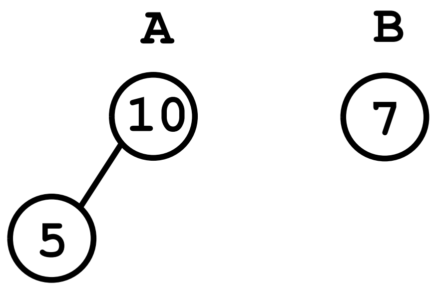
- 合并时，若A、B有一个为外部节点，合并结果为另一个

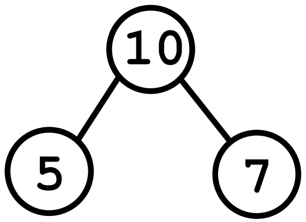
最大HBLT的合并操作



- B的根节点9更大，所以9作为新树根节点
- B的左子树为外部节点，作为新左子树，其s值为0
- B的右子树为外部节点，与A合并。由于其中一个为外部节点，所以结果为A。结果的s值为1，该合并结果作为新右子树
- 因为右子树的s更大，交换左右子树

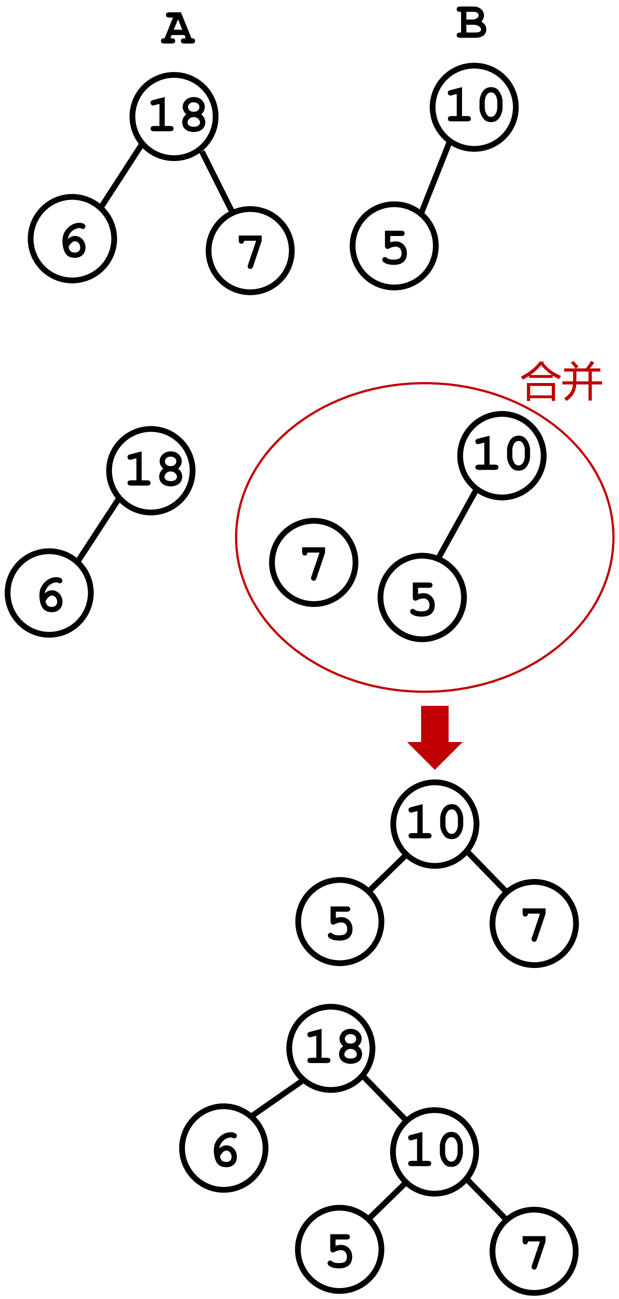
最大HBLT的合并操作



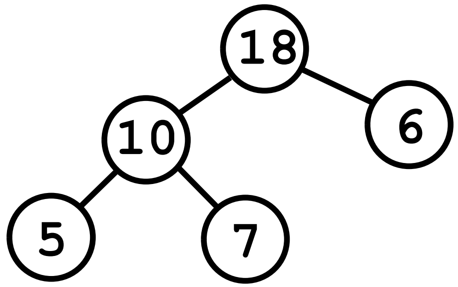


- A的根节点10更大，所以10作为新树根节点
- A的左子树作为新左子树，其*s*值为1
- A的右子树为外部节点，与B合并。由于其中一个为外部节点，所以结果为B。结果的*s*值为1，该合并结果作为新右子树
- 因为左子树的*s*值等于右子树的*s*值，完成合并

最大HBLT的合并操作



- A的根节点18更大，所以18作为新树根节点
- A的左子树作为新左子树，其s值为1
- A的右子树与B合并
  - 10作为根节点
  - 左子树为新左子树，s值为1
  - 右子树外部节点与7合并
  - 合并后新右子树s值为1，与新左子树一致，合并完成
- 合并后新右子树s值为2，比新左子树大，交换



最大HBLT的合并操作

- 最大HBLT合并操作的时间复杂度
  - 合并总是发生在右子树
  - 每次合并有一棵树的高度比上一层递归时减1

- 合并两个元素数量为m，n的最大HBLT的时间复杂度为$O(\log m + \log n) = O(\log mn)$

初始化最大HBLT

- 方法1：将n个元素插入到最初为空的最大HBLT中

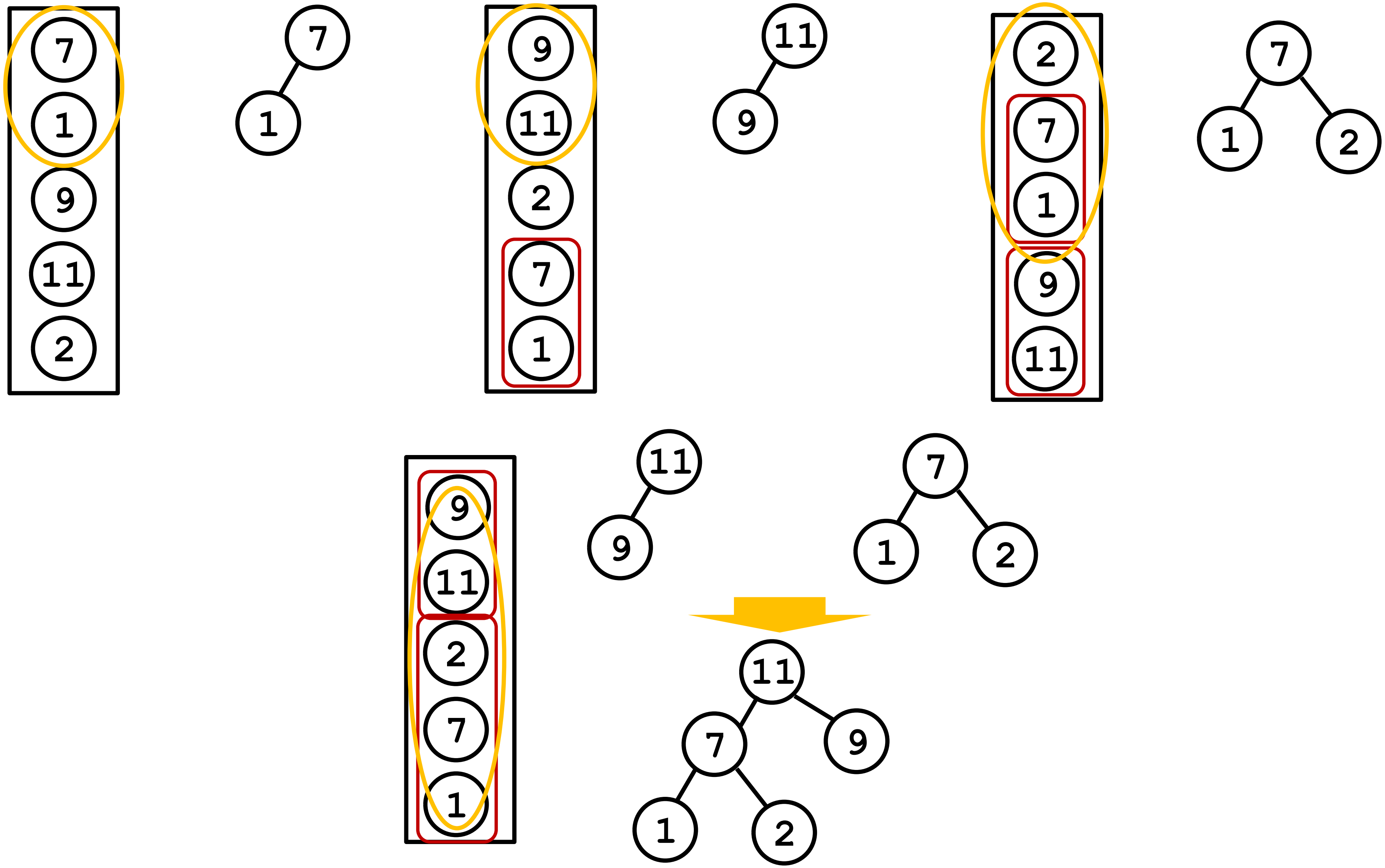
- 顺次插入n个元素的复杂度为$O(n\log n)$

初始化最大HBLT

- 方法2：
  - 创建n个最大HBLT，每个包含一个不同的元素，并将n个树插入一个FIFO队列
  - 从队列删除两个最大HBLT，将其合并后插入队列
  - 重复上一步直至剩下最后一个最大HBLT

初始化最大HBLT

- 初始化元素7、 1、 9、 11、 2



初始化最大HBLT

- 时间复杂度$O(n)$
  - 合并 $\frac{n}{2}$ 对包含1个元素的最大HBLT
  - 合并 $\frac{n}{4}$ 对包含2个元素的最大HBLT
  - 合并 $\frac{n}{8}$ 对包含4个元素的最大HBLT
  - ……
  - 合并 $\frac{n}{2^{i+1}}$ 对包含2^i个元素的最大HBLT，每次$O(2i)$
    - $O\left(\log\left(2^i \cdot 2^i\right)\right)$
  - 总的复杂度
  - $O\left(1 \cdot \frac{n}{2} + 2 \cdot \frac{n}{4} + \cdots + (2i) \cdot \frac{n}{2^{i+1}}\right) = O\left(n\Sigma\left(\frac{i}{2^i}\right)\right) = O(n)$

$\Sigma\left(\frac{i}{2^i}\right) \leq 2$

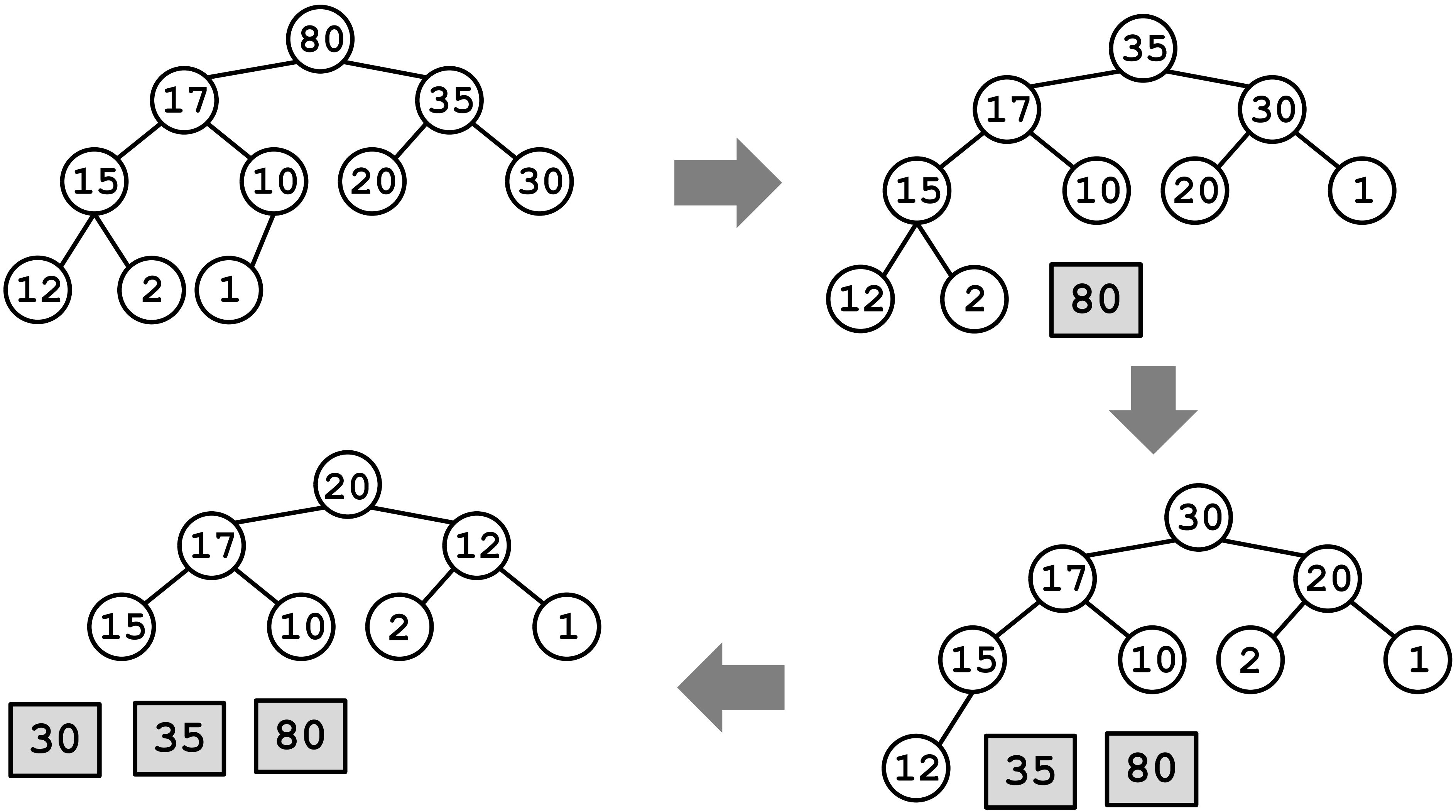
堆排序

- 可以利用堆来实现n个元素的排序，元素的值即为其优先级

- 堆排序算法:
  - 1）将要排序的n个元素初始化为一个大/小根堆
  - 2）依次删除元素

例

- A=[-, 20, 12, 35, 15, 10, 80, 30, 17, 2, 1]



堆排序实现

```
template <class T>
void heapSort(T a[], int n) {
    // 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; // 最后大小为1的大根堆不用操作
    }
    heap.deactivateArray(); // 置空堆指针，从堆的析构函数中保存数组a
}
```

堆排序的时间复杂度

- 对n个元素进行堆排序
  - 初始化时间：$\Theta(n)$
  - 每次删除的时间：$O(\log n)$
  - 共n个元素，总时间为：$O(n\log n)$

霍夫曼编码

- 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，根据不同的符号在一段文字中出现的频率来进行编码

霍夫曼编码

- 假设长度1000的文本由4种字符a，x，u，z组成
  - aaxuaxaxz……
- 若每个字符用一个字节(8-bit)储存
  - 1000字节，8000位
- 每个字符用相同位数的编码
  - 一共四种字符，需要2位编码（$\log_2 n$）
    - a: 00, x: 01, u: 10, z:11
  - 2000位

霍夫曼编码

- 霍夫曼编码根据不同符号在一段文字中的频率来设计压缩编码
  - 频率(frequency): 一个字符出现的次数
- aaxuaxz
  - 频率: a:3, x:2, u:1, z:1
- 霍夫曼编码通过减少高频率字符的码长、增加低频率字符的码长来增加压缩率
  - 只减少高频字符码长为导致无法解码

霍夫曼编码

- 对aaxuaxz进行编码

频率：a:3 x:2 u:1 z:1
码表：a:0 x:10 u:110 z:111

a a x u a x z

0 0 10 110 0 10 111

13-bit < 14-bit

- 当频率相差变大时，编码收益会更高

霍夫曼编码

- 解码过程

码表：a:00 x:01 u:10 z:11

00000110000111

a a x u a x z

每次读入固定长度码字

码表：a:0 x:10 u:110 z:111

0010110010111

a a x u a x z

读入单个码字，如果码表中没有对应的码字，继续读入下一个

霍夫曼编码

- 解码过程

  码表：a:0 x:1 u:00 z:01

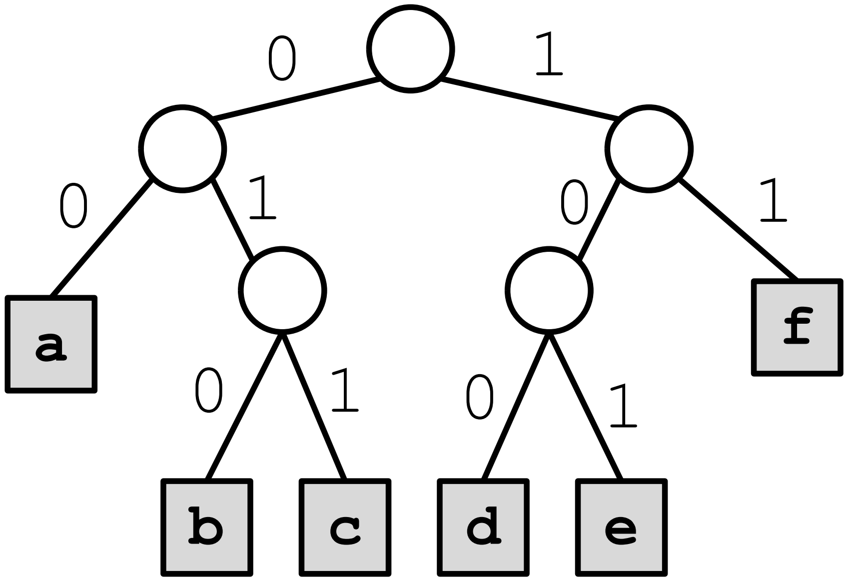
```
a a x u a x z
001000101
a? u?
```

如果仅减少高频字符码长，会导致无法解码

- 若要成功解码可变长编码，编码需要满足没有任何一个代码是另一个代码的前缀
  - 使用短码会导致部分长码无法使用：0导致01无法使用

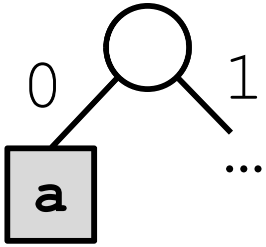
霍夫曼编码

- 霍夫曼编码使用扩充二叉树
  - 外部节点对应被编码的字符



a: 00
b: 010
c: 011
d: 100
e: 101
f: 11

- 保证没有一个编码是另一个编码的前缀



霍夫曼树

- 扩充二叉树的加权外部路径长度(weighted external path length, WEP)为

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

- 二叉树具有n个外部节点
- $L(i)$是从根到达外部节点i的路径长度（边数）
- $F(i)$是外部节点i的权值

霍夫曼树

- 扩充二叉树的加权外部路径长度(weighted external path length, WEP)为

$$WEP = \Sigma_{i=1}^{n}\big(L(i)\cdot F(i)\big)$$

- 对于一个霍夫曼编码码本转换的二叉树，$L(i)$是单个码字长度，$F(i)$是码字频率，WEP就是该码本压缩后编码串的期望长度
- 霍夫曼树就是对于给定频率具有最小WEP的二叉树

霍夫曼编码压缩过程

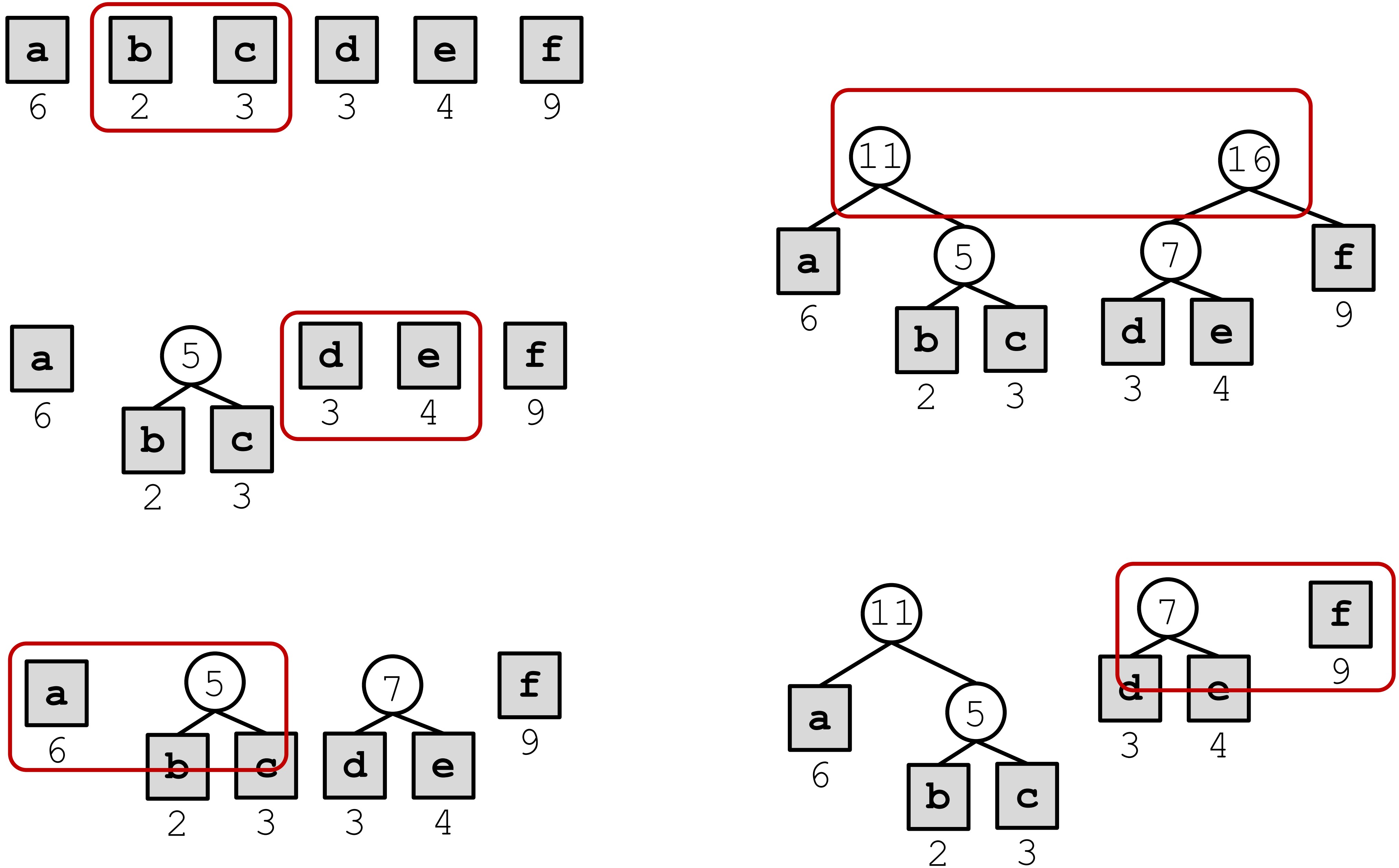
- 使用霍夫曼编码压缩一段文本过程
  - 1）确定不同字符的出现频率
  - 2）建立具有最小加权外部路径的二叉树，即霍夫曼树
    - 树的外部节点为字符，其权重为对应的出现频率
  - 3）遍历从根到所有外部节点的路径得到每个字符的编码
    - 所有左路径为0，右路径为1；或反之
  - 4）使用得到的编码来代替字符串中的字符

- 如何构造霍夫曼树?

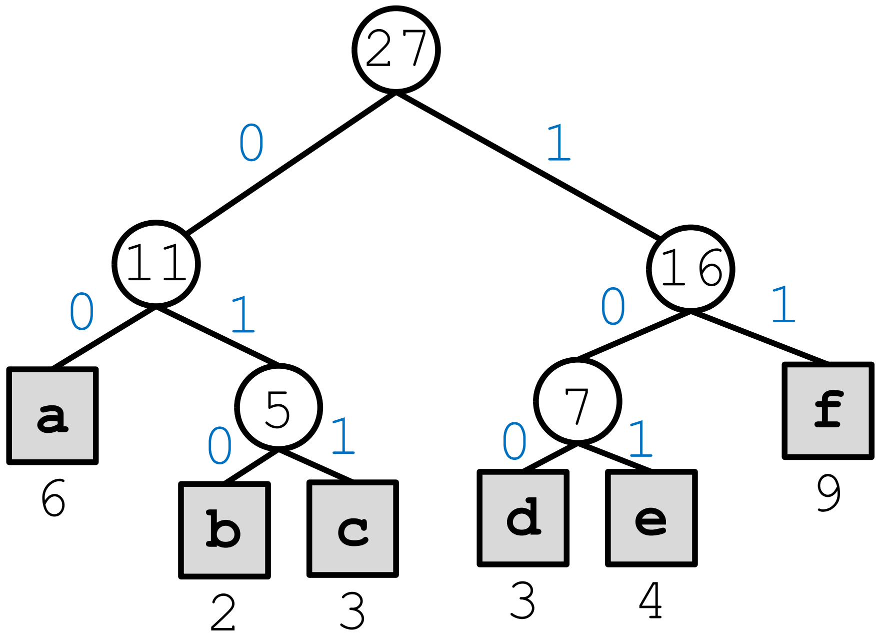
构造霍夫曼树

- 构造霍夫曼树的过程：
  - 1）初始化二叉树集合，每个二叉树仅包含一个外部节点，代表不同字符，且其权重为字符的出现频率
  - 2）合并两个具有最小权重的二叉树
    - 根节点为一个新节点
    - 两个二叉树分别作为新根节点的左右孩子
    - 赋予新树新的权重，其值为两个孩子节点的和
  - 3）重复上一步直至仅剩最后一棵树

构造霍夫曼树示例


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

构造霍夫曼树示例



a: 00
b: 010
c: 011
d: 100
e: 101
f: 11

- 小根堆用于组织构建霍夫曼树过程中的中间结果
  - 每个小根堆节点的元素为一棵树

构造霍夫曼树的实现

```
template<class T>
class  huffmanNode { // 霍夫曼树中的节点
    // 基于n个数据的权重构造霍夫曼树
    friend linkedBinaryTree<int> *huffmanTree(T w[], int n);
private:
    linkedBinaryTree<int> *tree;
    T weight;
public:
    operator T() const  { return weight; }
};
```

- linkedBinaryTree中包含一个元素，指向子树的两个指针
  - 权重weight没有放在二叉树节点中

构造霍夫曼树的实现

```
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(int 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)