



堆的前言

◆ 目标：在一个队列中取出元素的最大(最小)的值

数组：

插入-元素直接放入尾部 $\Theta(1)$

删除-查找最大(最小)值 $\Theta(n)$

删除需要移动的元素 $O(n)$

链表：

插入-链表头部 $\Theta(1)$

删除-查找最大(最小)值 $\Theta(n)$

删去元素 $\Theta(1)$

有序数组：

插入-找到位置， $O(n)$ 或者 $O(n\log n)$

移动后续所有元素并插入新数据 $O(n)$

删除-删除最后一个元素 $\Theta(1)$

有序链表：

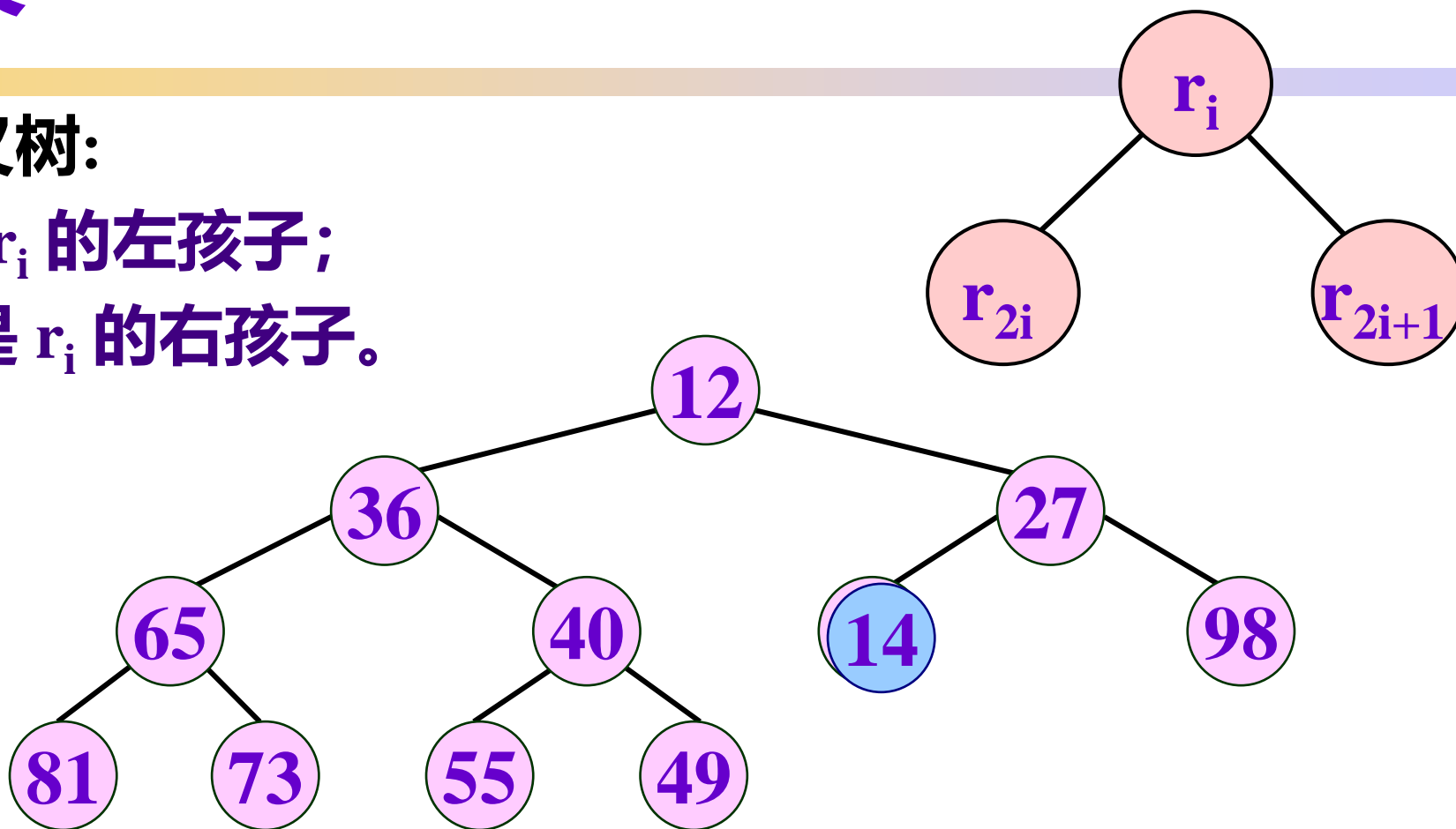
插入-找到 $O(n)$ ，插入 $\Theta(1)$ 。删除- $\Theta(1)$



堆的定义

◆ 完全二叉树:

- || r_{2i} 是 r_i 的左孩子;
- || r_{2i+1} 是 r_i 的右孩子。

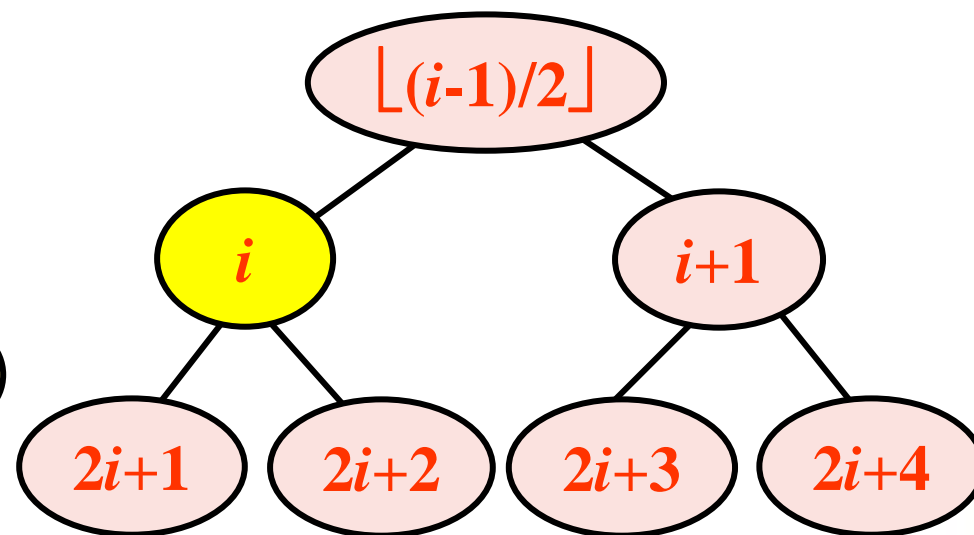
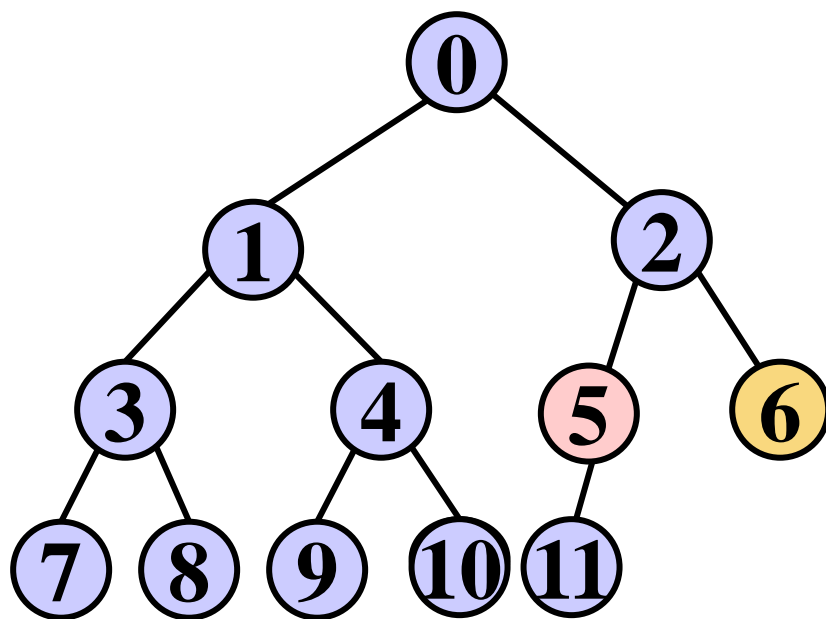


堆: $R[11]\{12, 36, 27, 65, 40, 34, 98, 81, 73, 55, 49\}$

$R[11]\{12, 36, 27, 65, 40, 14, 98, 81, 73, 55, 49\}$ 不是堆

5.2.2 二叉树的性质 - 5

- ◆ 完全二叉树的性质
- ◆ 如果根结点从0开始编号，则结点*i*与其父结点和子结点的对应关系是：



堆的定义

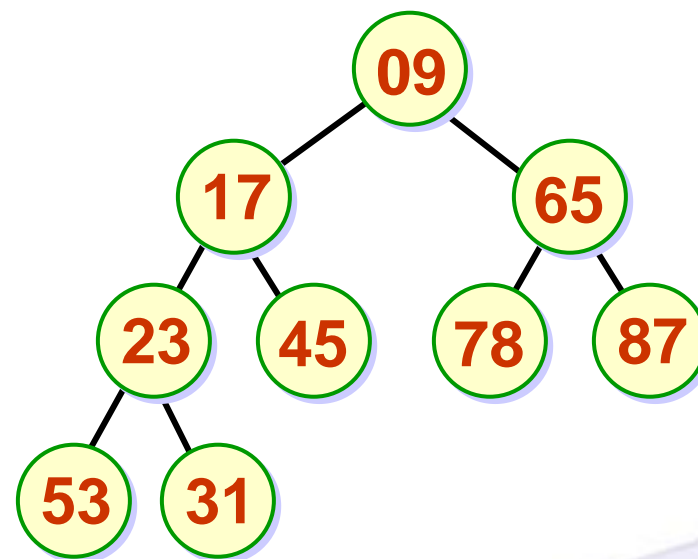
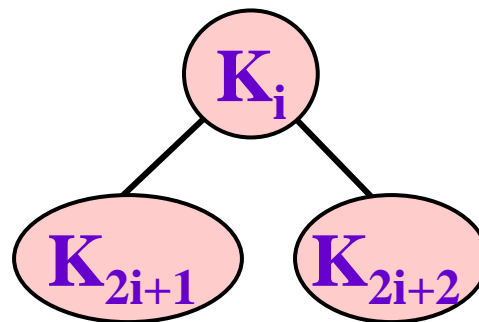
◆ 小根堆

$$\text{¶ } K_i \leq K_{2i+1}$$

$$\text{¶ } K_i \leq K_{2i+2}$$

完全二叉树的顺序表示

0	1	2	3	4	5	6	7	8
09	17	65	23	45	78	87	53	31





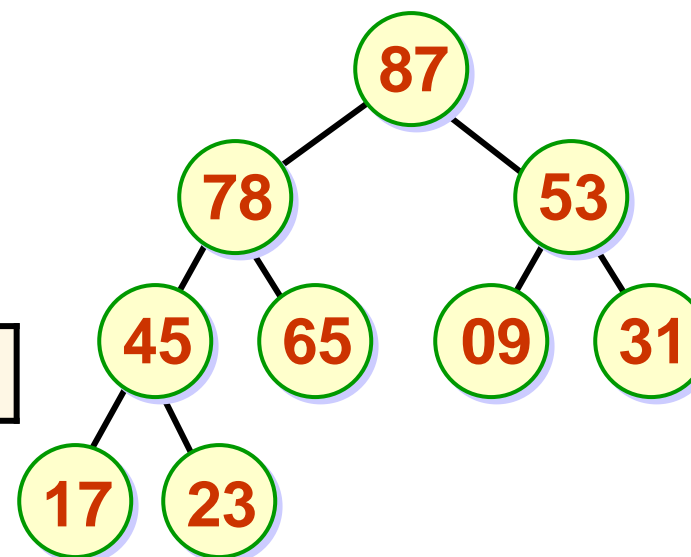
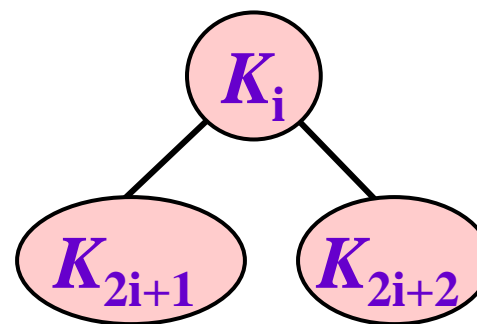
◆ 大根堆

$$\P K_i \geq K_{2i+1}$$

$$\P K_i \geq K_{2i+2}$$

完全二叉树的顺序表示

0	1	2	3	4	5	6	7	8
09	17	65	23	45	78	87	53	31





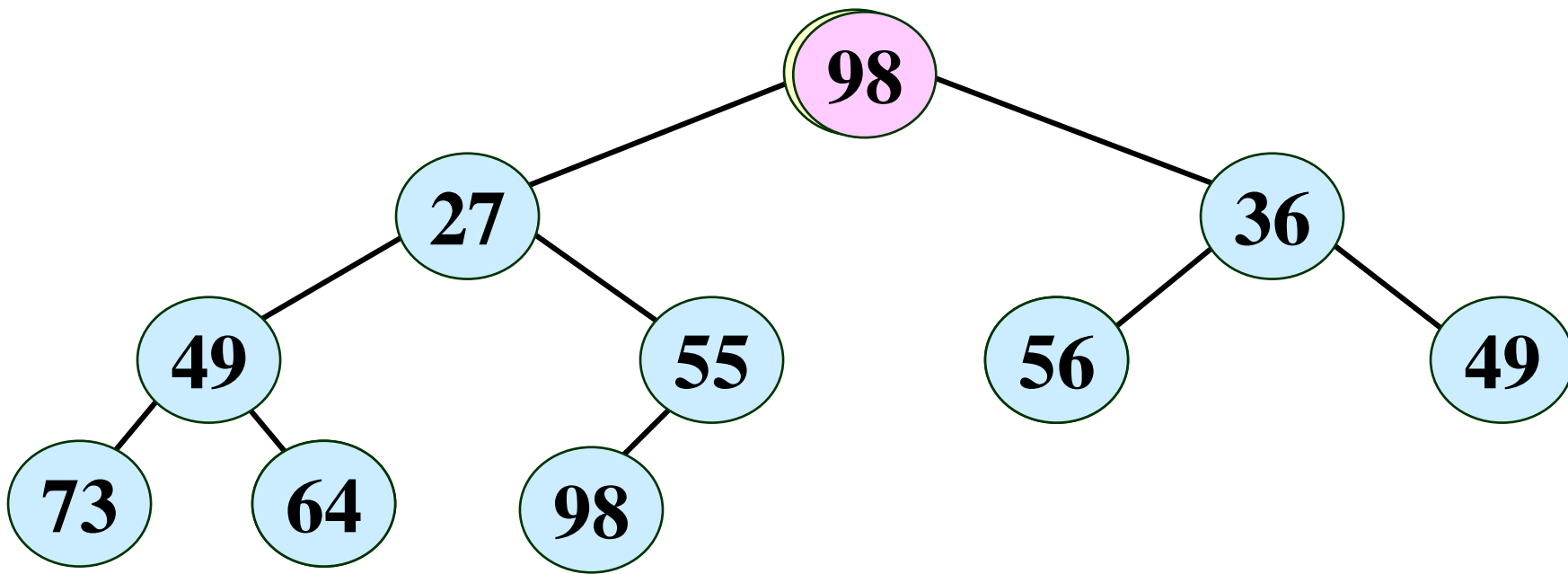
堆的定义

```
#define heapSize 100           //堆的最大元素个数
typedef int HElemType;        //堆中元素的数据类
typedef struct {               //堆结构的定义
    HElemType elem[heapSize];  //堆存储数组
    int curSize;               //当前元素个数
} minHeap;
```



调整堆

◆ 即假设H.elem[i..m]中记录的关键字除 elem[i] 之外均满足堆的特征

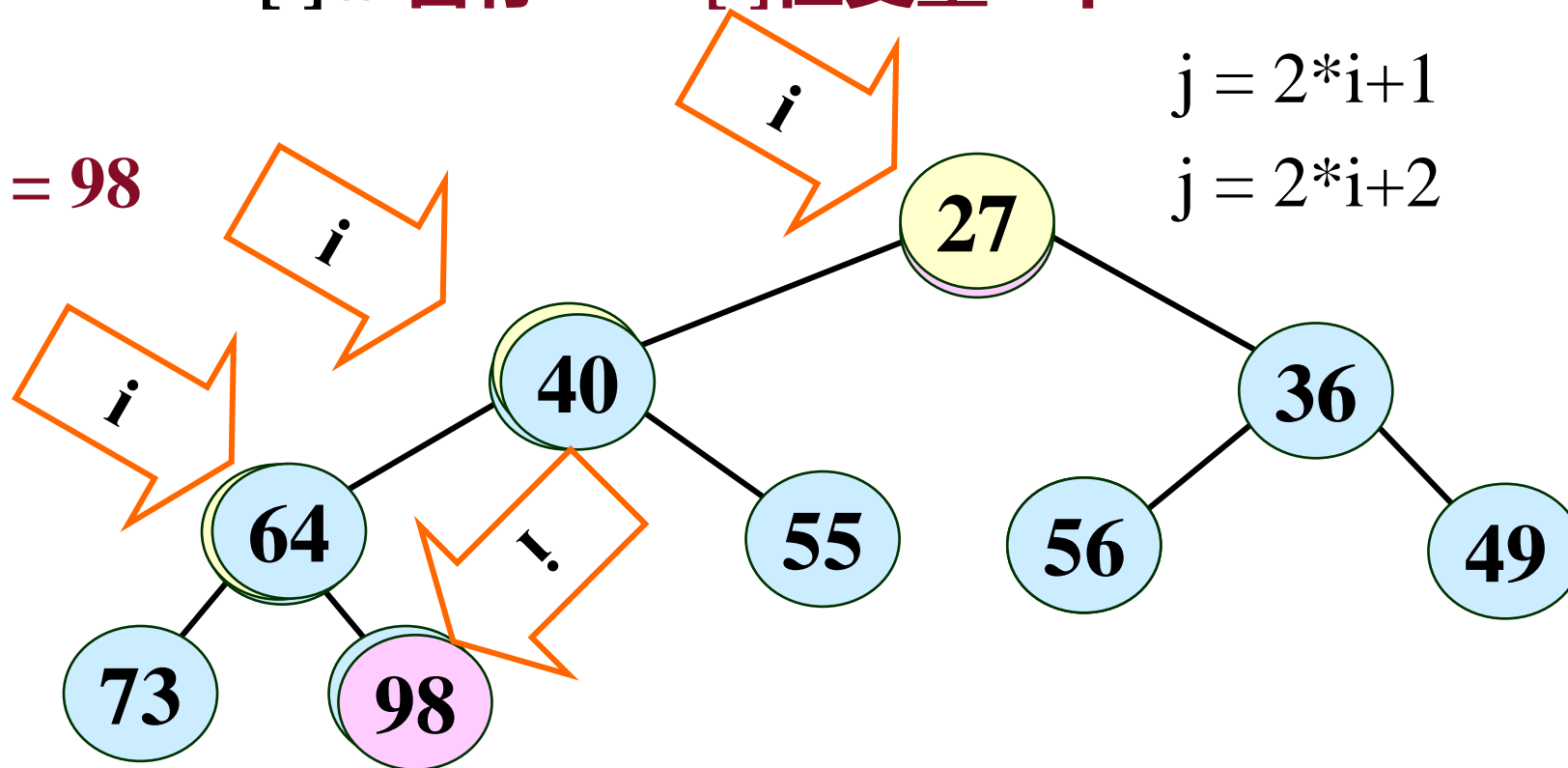


堆: R[10]{98,36,27,49,55,56,49,73,64}{12}

调整堆

`temp = H.elem[i]` // 暂存 `elem[i]` 在变量 `rc` 中

`temp = 98`



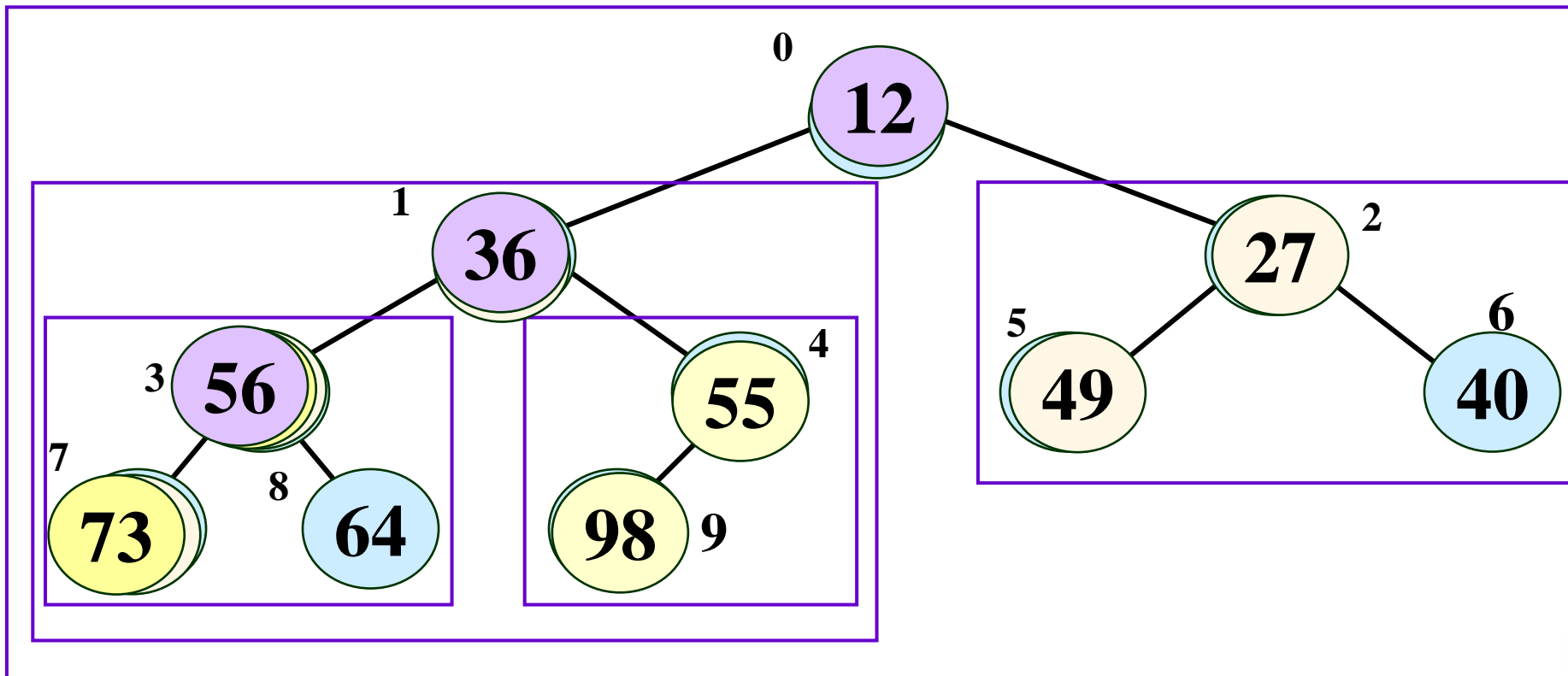
`H.elem[i] = temp`

调整需要多次从上向下的交换，称为筛选

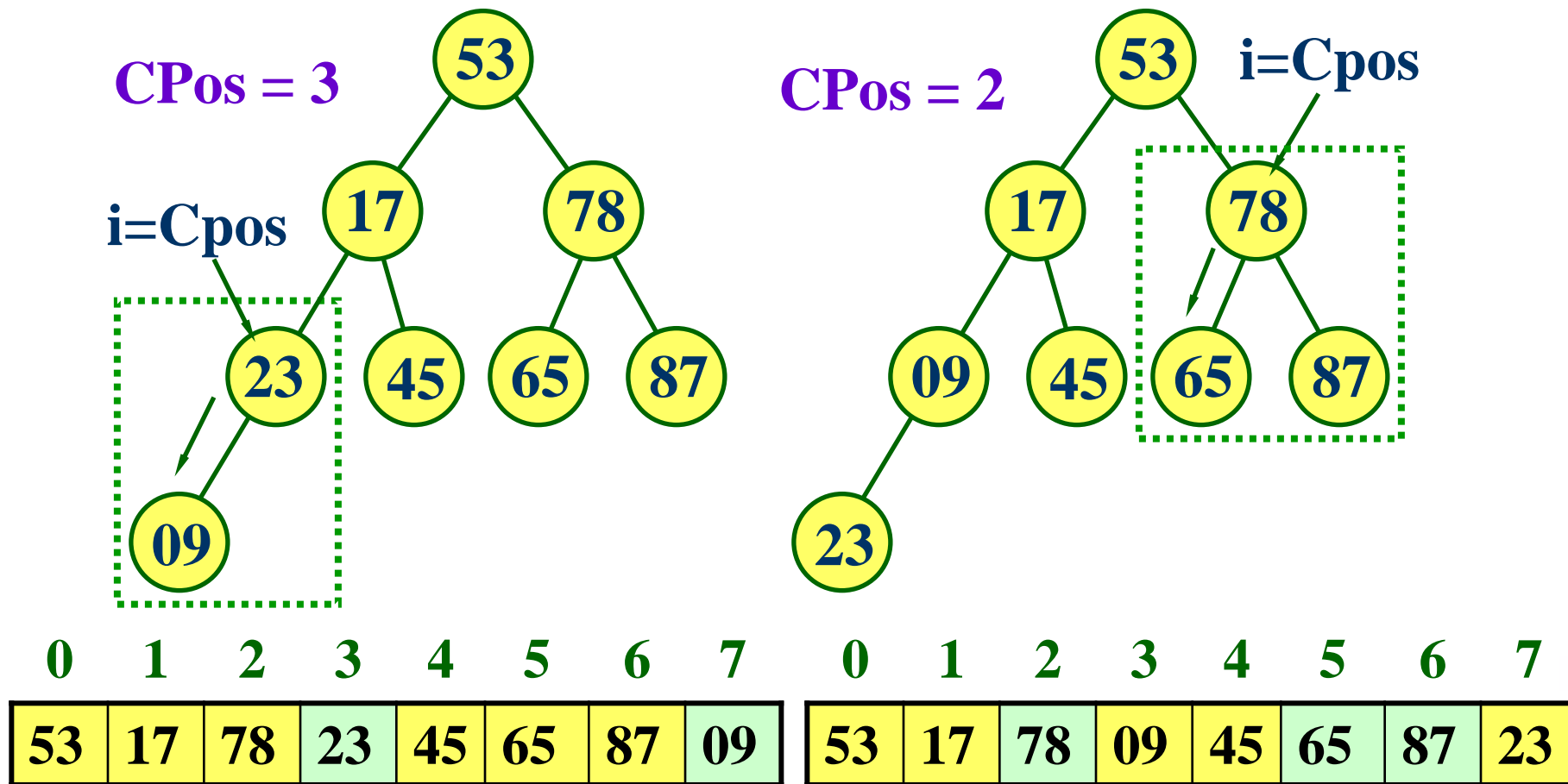
小根堆的向下筛选算法

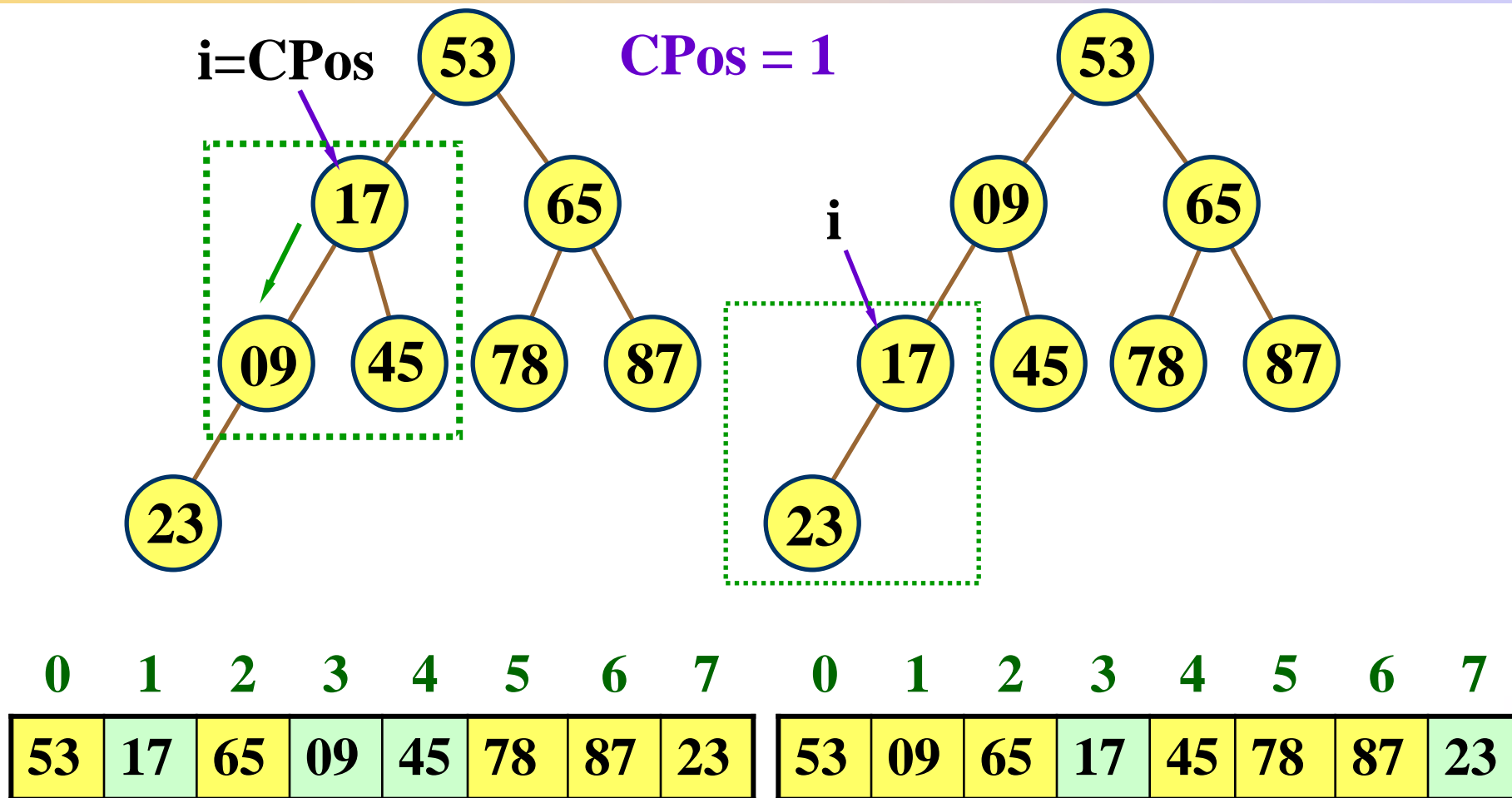
```
void siftDown ( minHeap& H, int i, int m ) {  
    //从结点i开始到m为止, 自上向下比较, 将一个集合局  
    //部调整为小根堆  
    HElemType temp = H.elem[i];  
    for ( int j = 2*i+1; j <= m; j = 2*j+1 ) {  
        if ( j < m && H.elem[j] > H.elem[j+1] ) j++; //左右比较  
        if ( temp <= H.elem[j] ) break; //小则不做调整  
        else { H.elem[i] = H.elem[j]; i = j; } //小者上移  
    }  
    H.elem[i] = temp; //回放temp中暂存的元素  
}
```

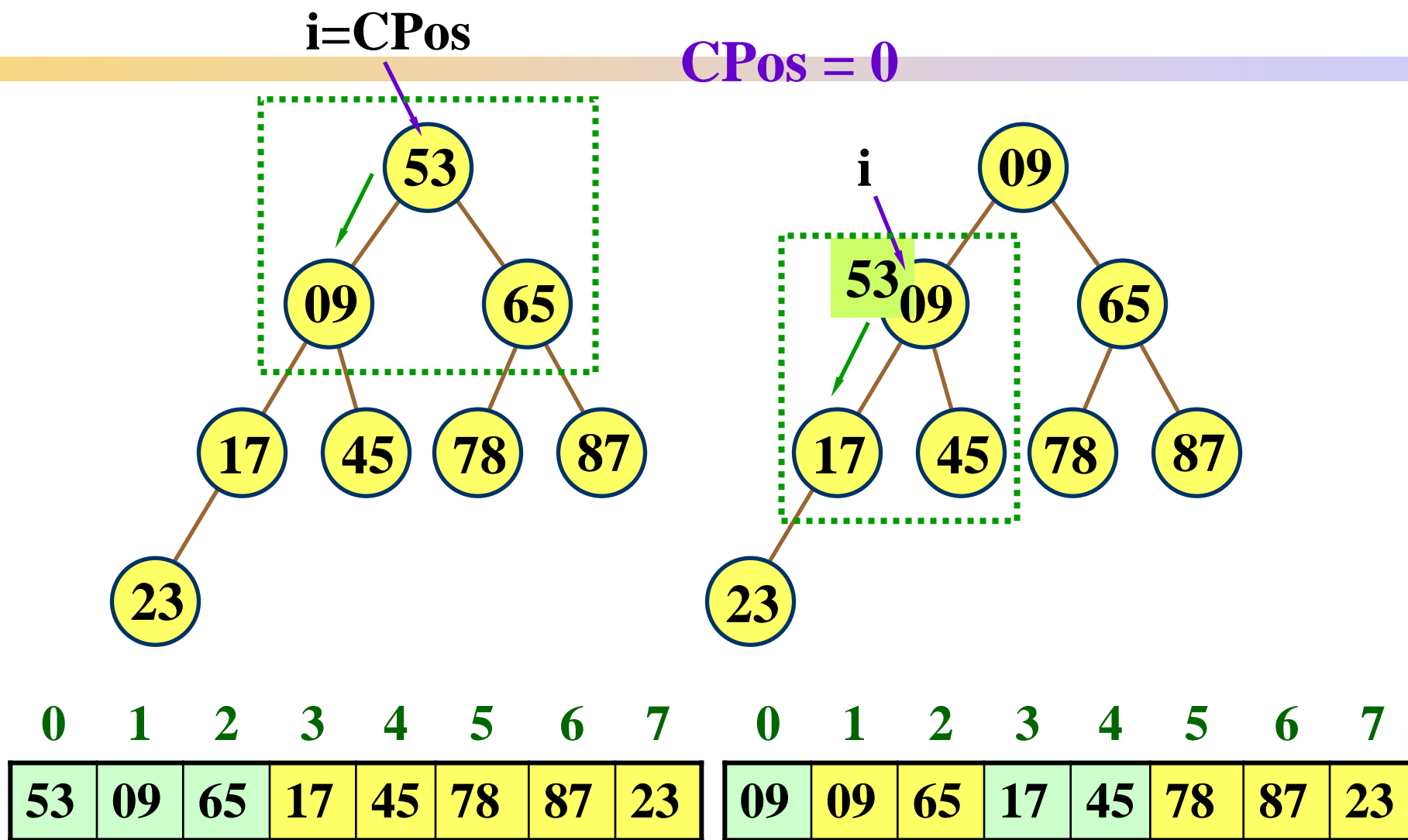
建堆 小根堆

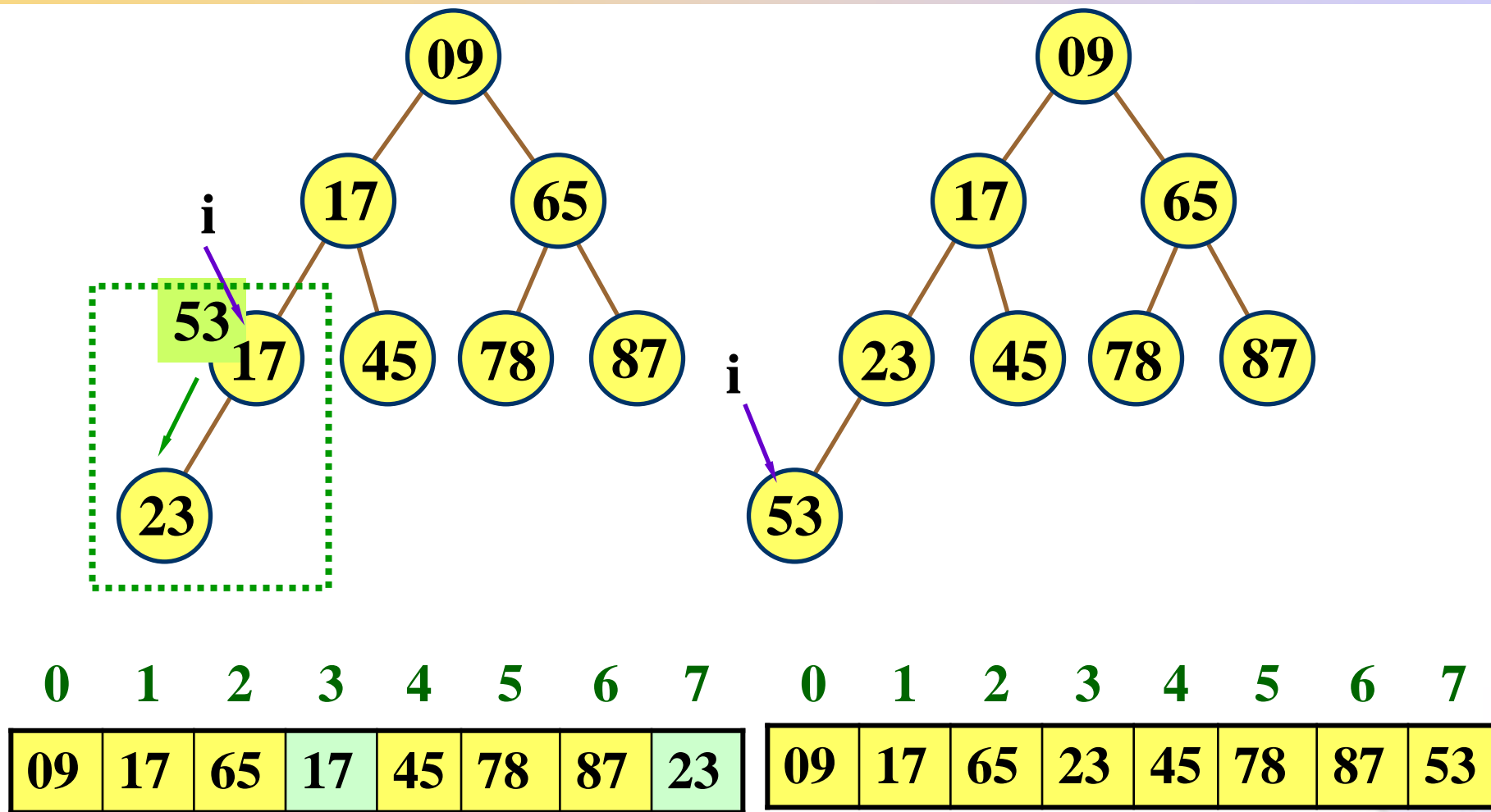


◆ 自下向上逐步调整为小根堆的过程










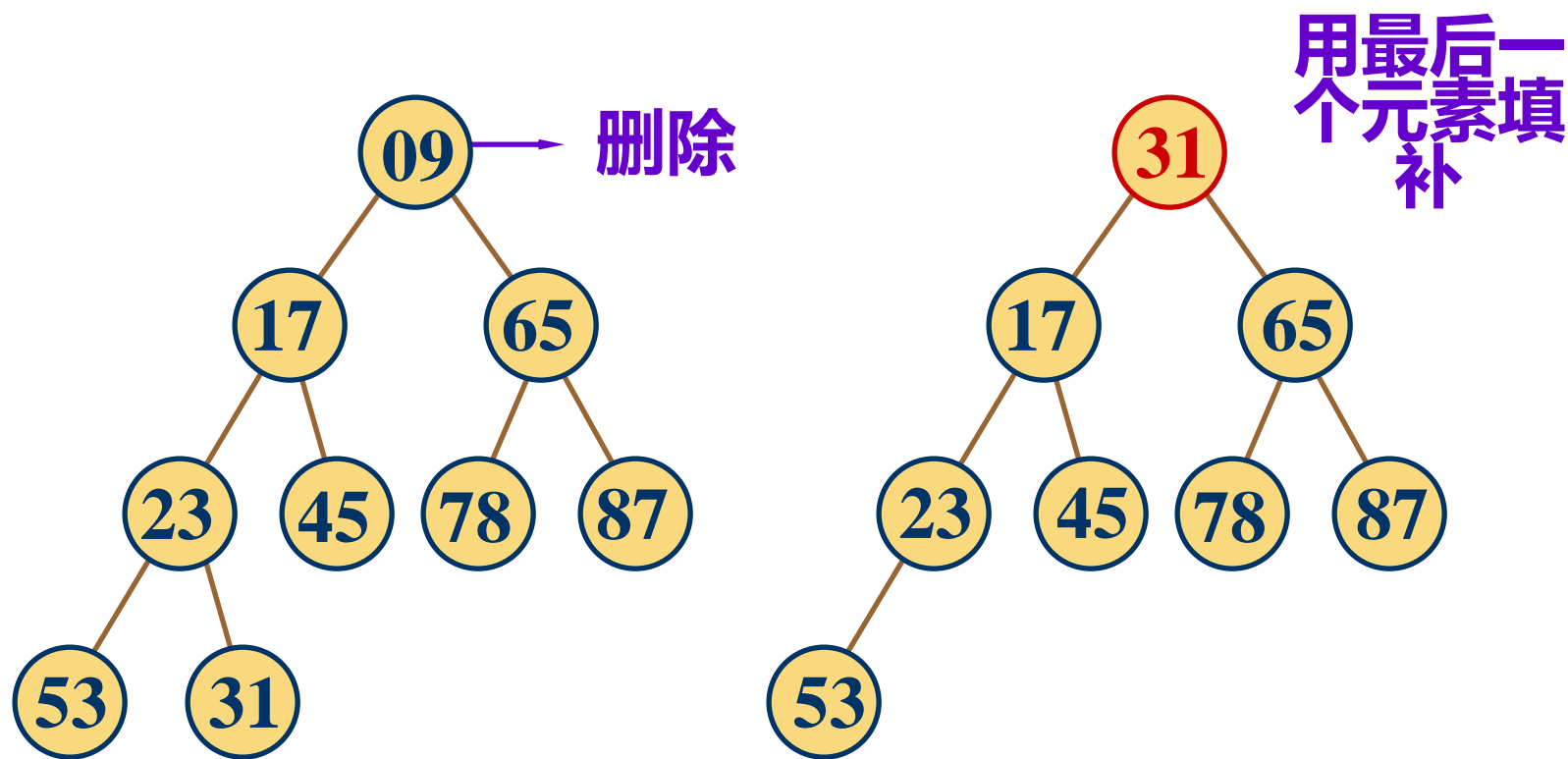


小根堆的建立

```
void creatMinHeap ( minHeap& H,  
    HElemType arr[ ], int n ) {  
    //将一个数组从局部到整体，自下向上调整为小根堆  
    for ( int i = 0; i < n; i++ ) H.elem[i] = arr[i];    //复制  
    H.curSize = n;  
    for ( i = (H.curSize-2)/2; i >= 0; i-- )  
        //自底向上逐步扩大小根堆  
        siftDown ( H, i, H.curSize-1 );  
        //局部自上向下筛选  
};
```

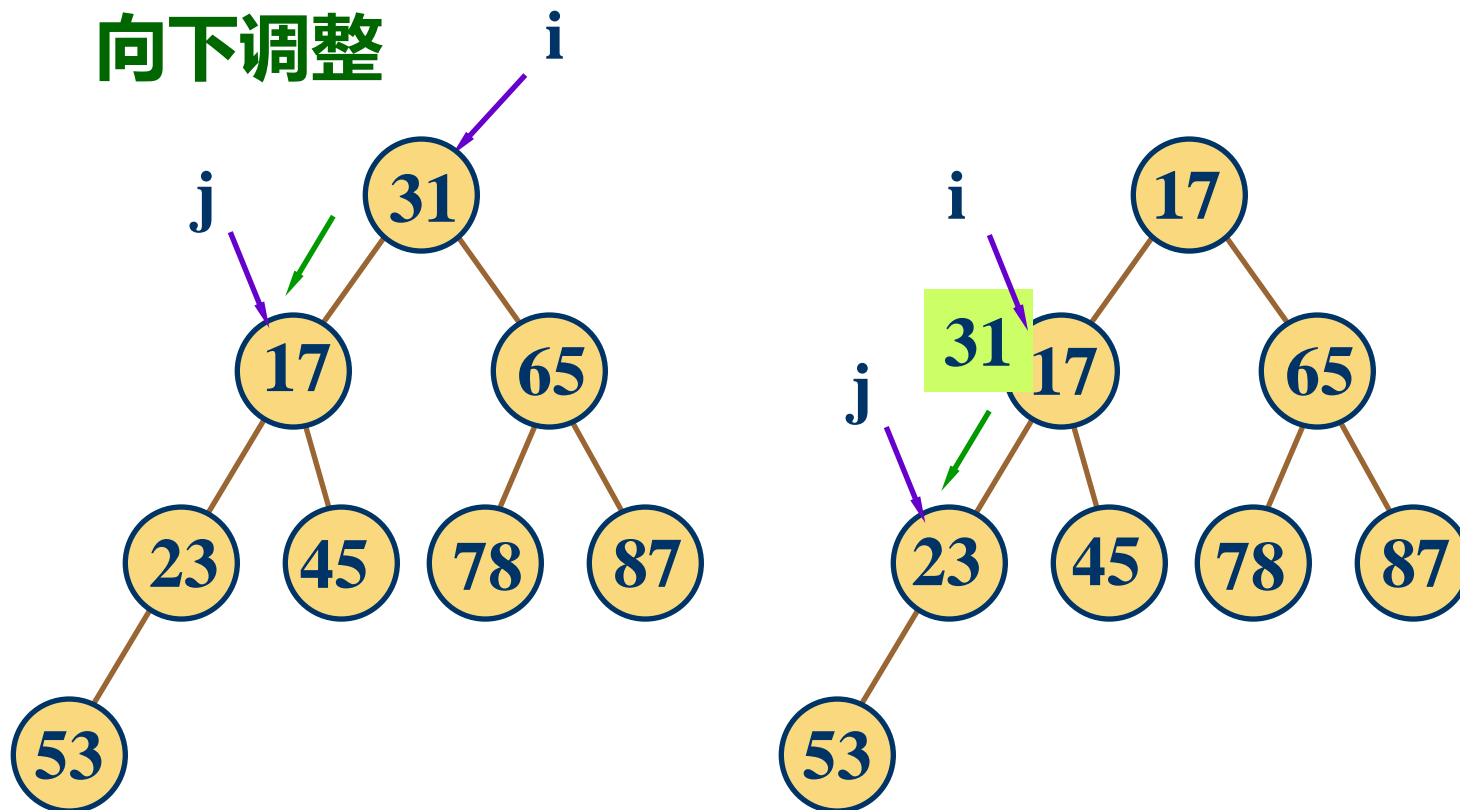


小根堆的删除和向下调整例



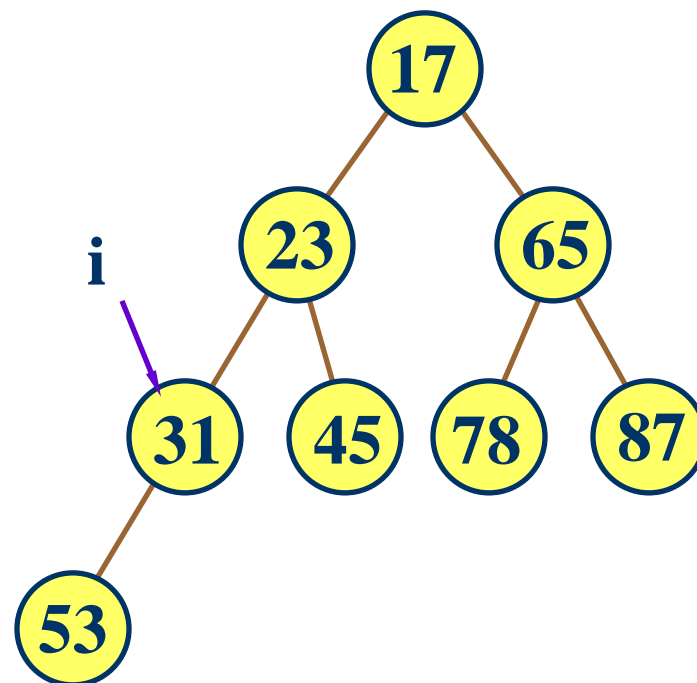
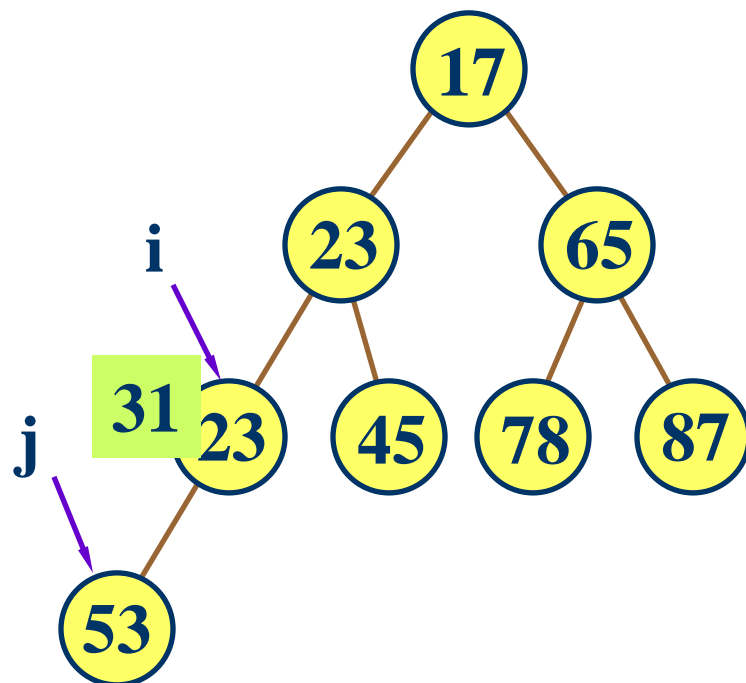
小根堆的删除和向下调整例

向下调整



小根堆的删除和向下调整例

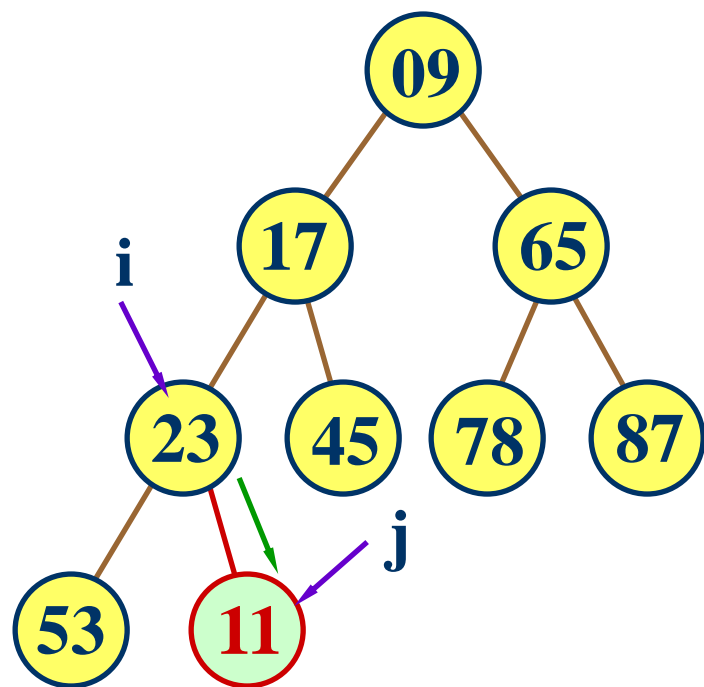
向下调整



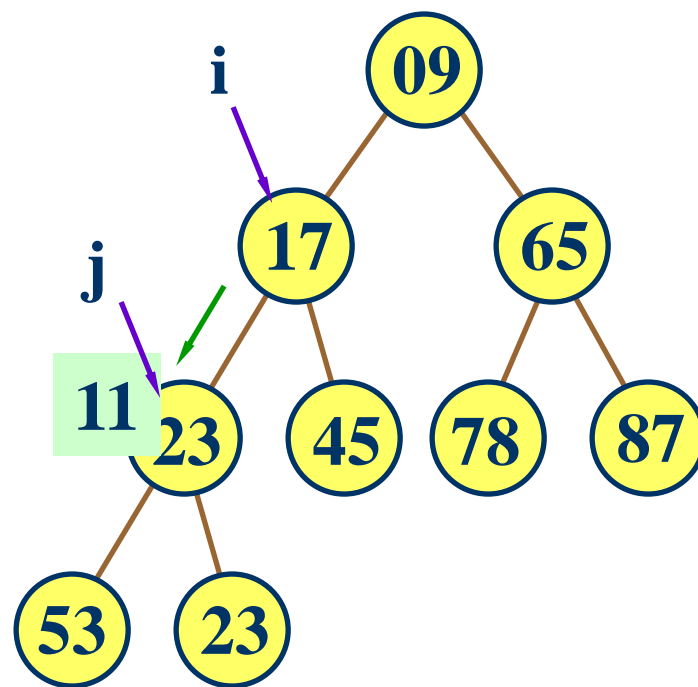
小根堆的删除算法

```
bool Remove ( minHeap& H, HElemType& x ) {  
    //从小根堆中删除堆顶元素并通过引用参数 x 返回  
    if ( H.curSize == 0 ) return false;    //堆空, 返回  
    x = H.elem[0];                        //返回最小元素  
    H.elem[0] = H.elem[H.curSize-1];  
                                           //最后元素填补到根结点  
    H.curSize--;  
    siftDown ( H, 0, H.curSize-1 );    //从新调整为堆  
    return true;  
}
```

小根堆的插入和向上调整例

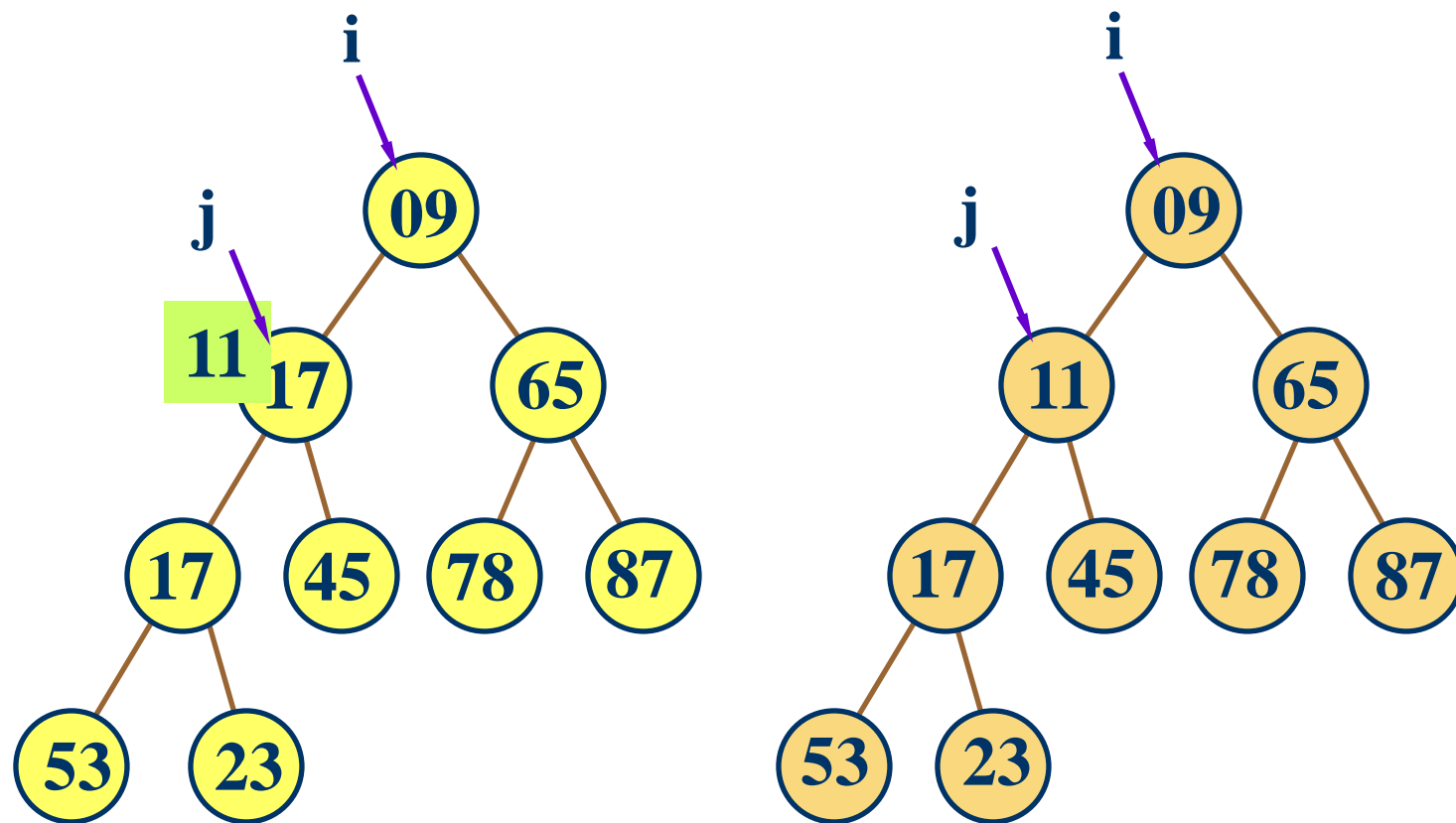


在堆中插入新元素 11



沿通向根的路径向上调整

小根堆的插入和向上调整例





小根堆的插入算法

```
bool Insert ( minHeap& H, HElemType x ) {  
    //将x插入到小根堆中并从新调整形成新的小根堆  
    if ( H.curSize == heapSize ) return false;  
        //堆满，返回插入不成功信息  
    H.elem[H.curSize] = x;           //插入到最后  
    siftUp ( H, H.curSize );         //从下向上调整  
    H.curSize++;                     //堆计数加1  
    return true;  
}
```



小根堆的向上筛选算法

```
void siftUp ( minHeap& H, int start ) {  
    //从结点start开始到结点0为止, 自下向上比较, 将集  
    //合重新调整为堆。  
    HElemType temp = H.elem[start];  
    int j = start, i = (j-1)/2;  
    while ( j > 0 ) { //沿双亲路径向上直达根  
        if ( H.elem[i] <= temp ) break; //双亲值小  
        else { H.elem[j] = H.elem[i]; j = i; i = (i-1)/2; }  
    } //双亲的值下降, j与i的位置上升  
    H.elem[j] = temp; //回送  
}
```