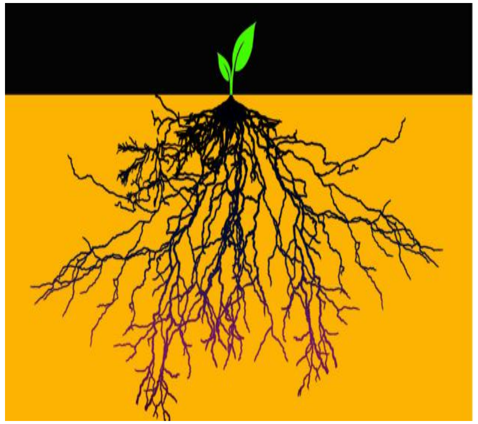
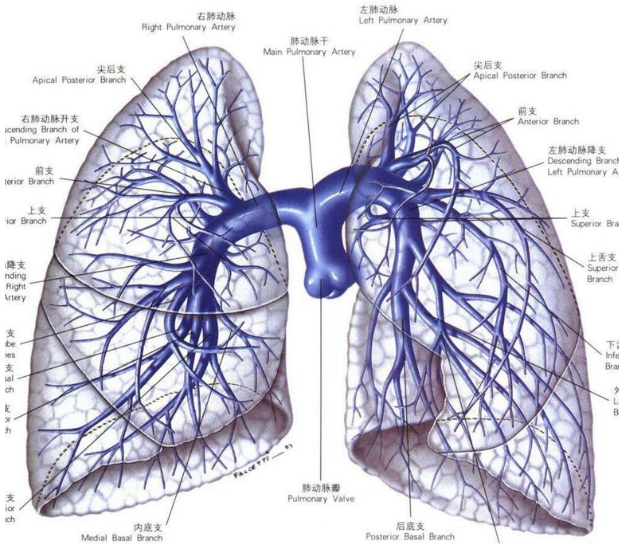
数据结构
Data Structure

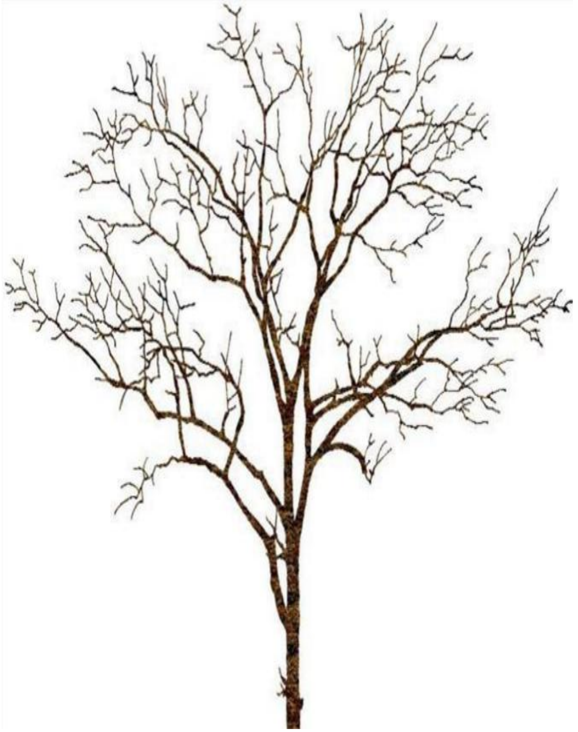
2017年秋季学期

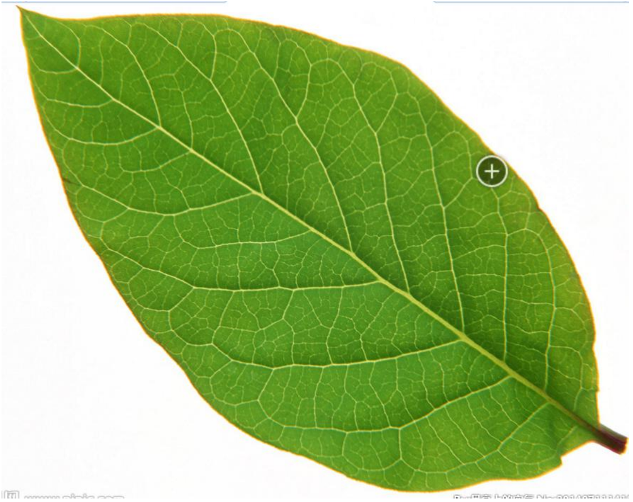
刘鹏远

树




右肺动脉
Right Pulmonary Artery
左肺动脉
Left Pulmonary Artery
肺动脉干
Main Pulmonary Artery
尖后支
Apical Posterior Branch
尖后支
Apical Posterior Branch
前支
Anterior Branch
上舌支
Superior
Branch
肺动脉瓣
Pulmonary Valve
内底支
Medial Basal Branch
后底支
Posterior Basal Branch





最佳判定树

堆

二叉搜索树/二叉排序树

huffman树应用：最佳判定树/决策树

判定树/决策树：

A decision tree is a decision support tool that uses a tree-like graph or model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility. It is one way to display an algorithm.

使用决策树进行决策的过程就是从根节点开始，测试待分类项中相应的特征属性，并按照其值选择输出分支，直到到达叶子节点，将叶子节点存放的类别作为决策结果。

介绍对象，是否见面的决策^-^

女儿：多大年纪了？

母亲：不到30。

女儿：长的帅不帅？

母亲：挺帅的。

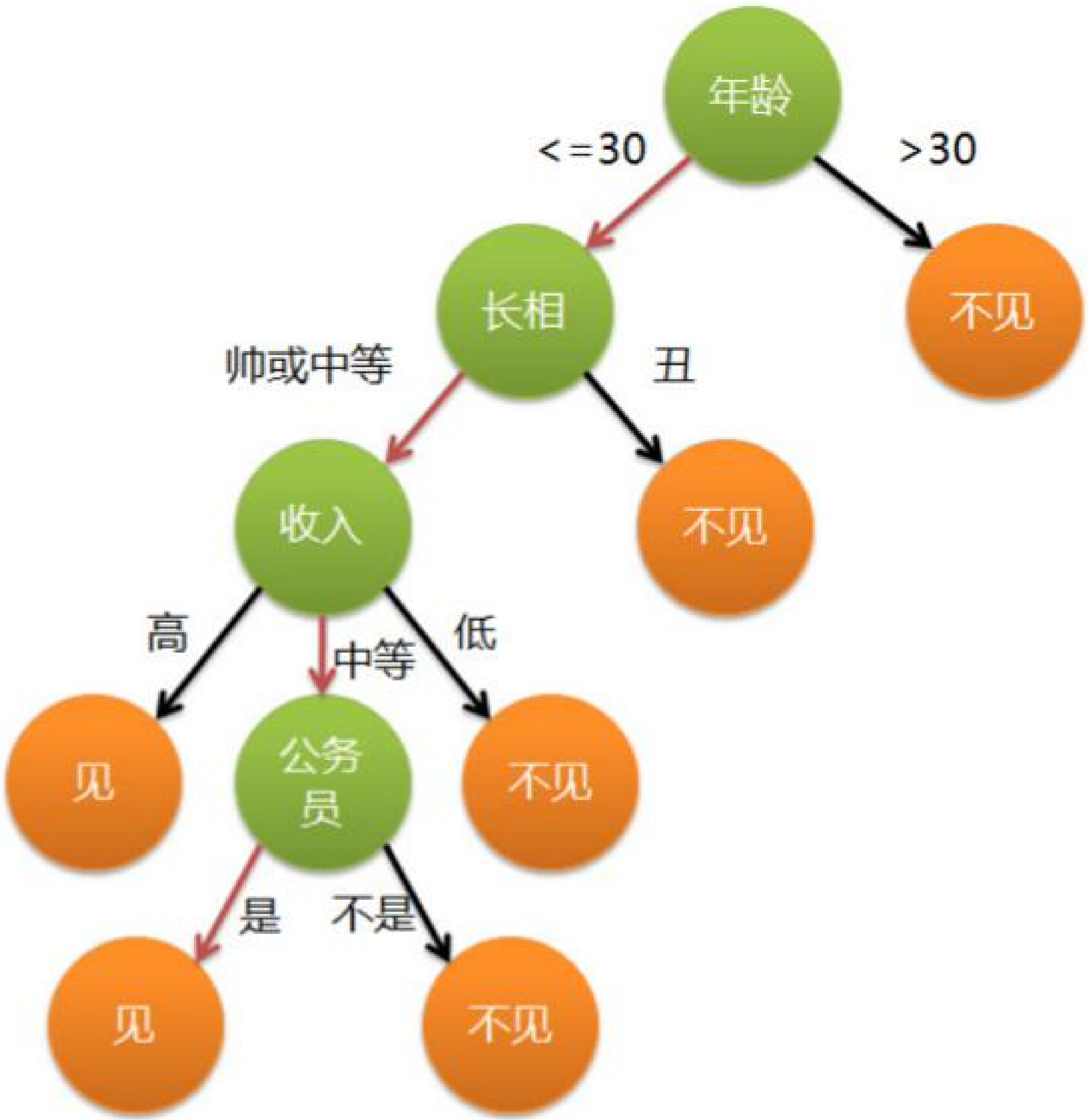
女儿：收入高不？

母亲：不算很高，中等情况。

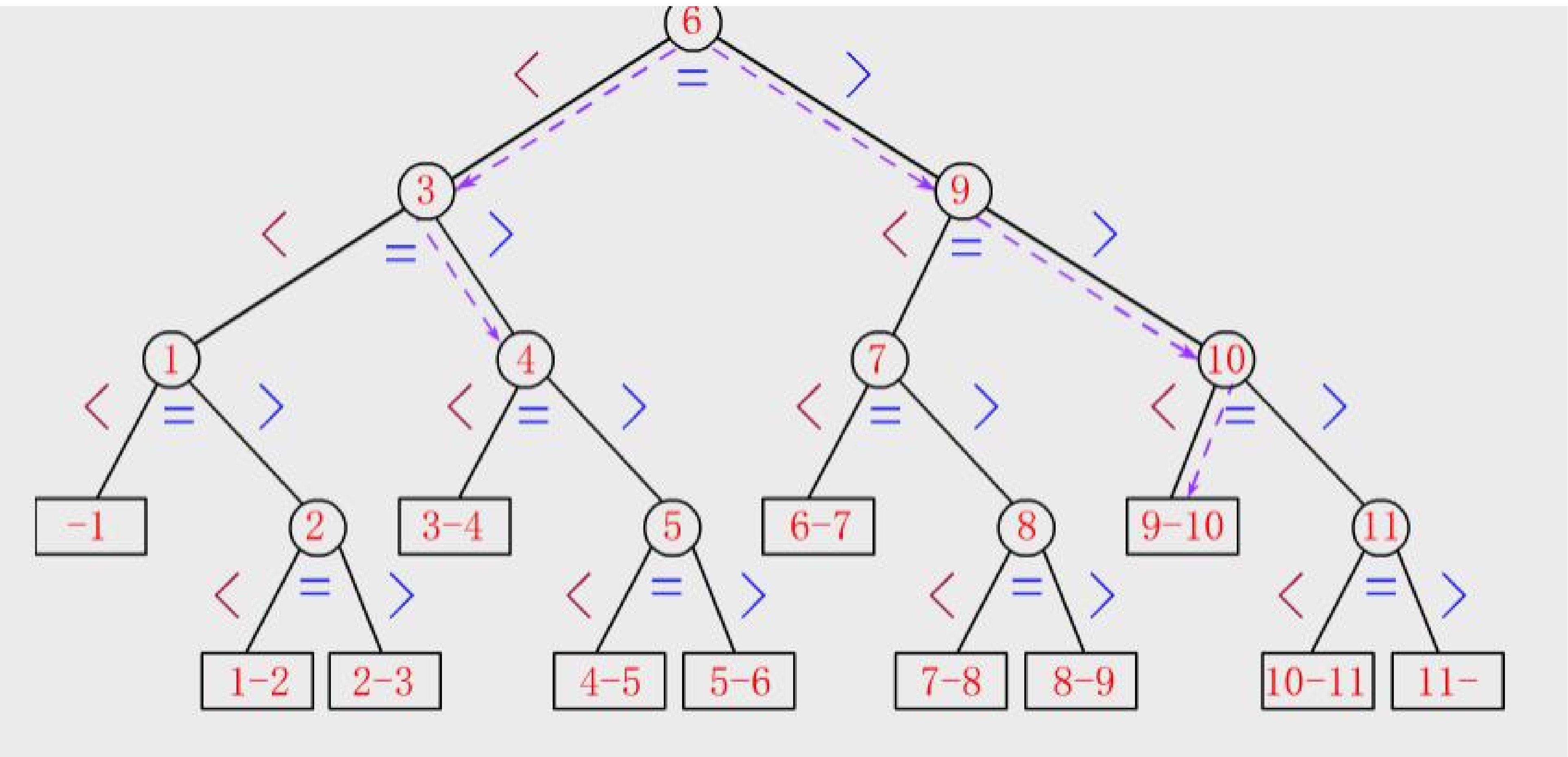
女儿：是公务员不？

母亲：是，在税务局上班呢。

女儿：那好，我去见见。



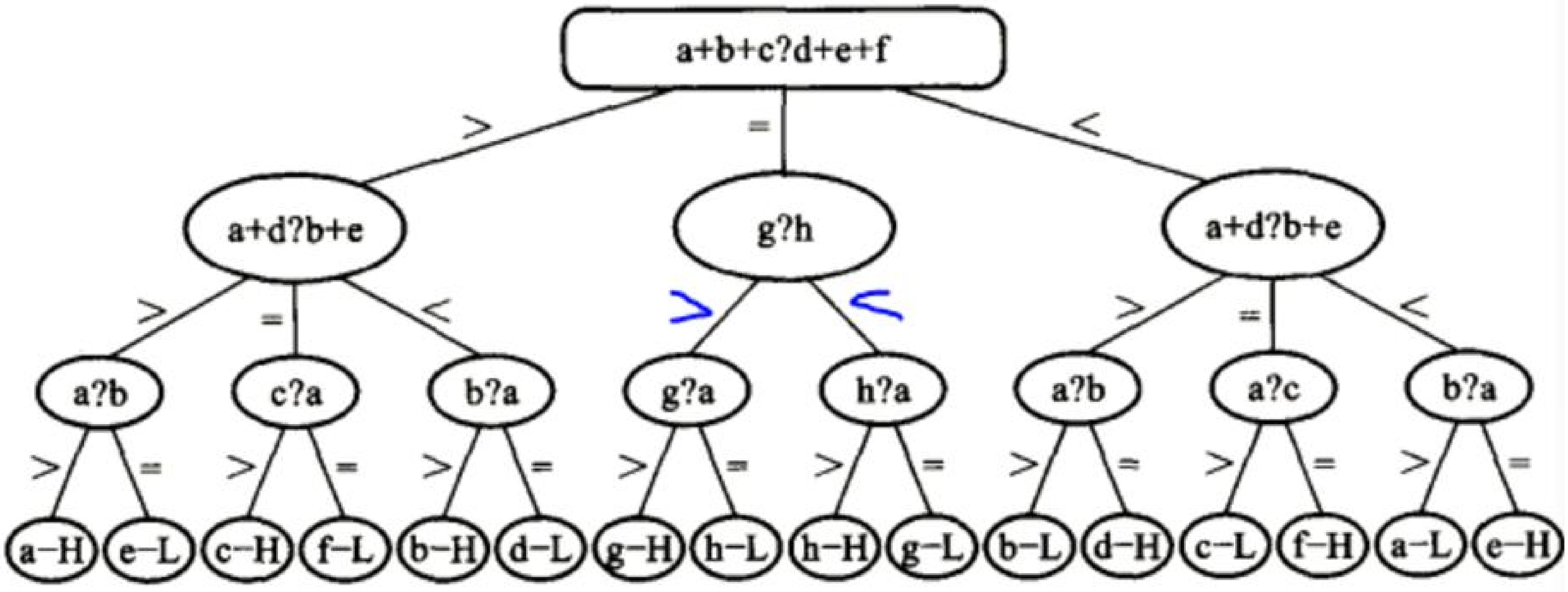
二分查找的判定树(查找时候还会详细介绍)



R[1..11]的二分查找的判定树(n=11)

硬币问题的判定树

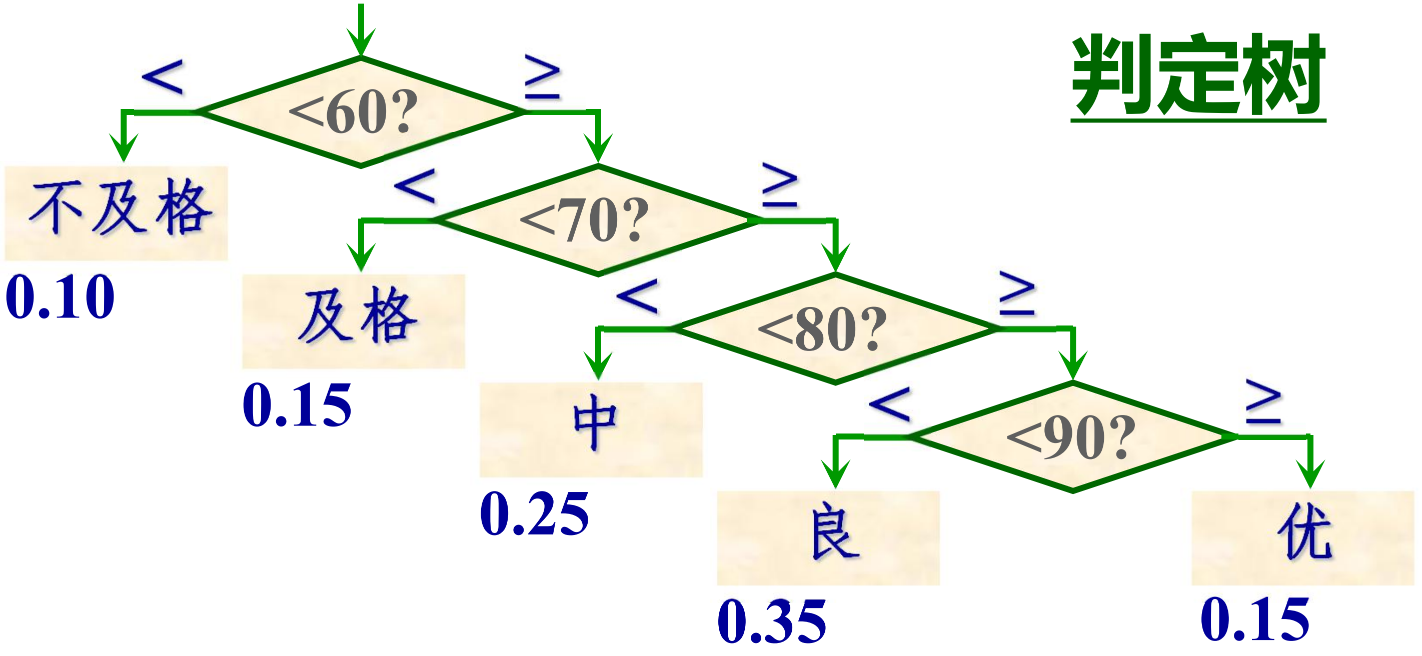
假定有8枚硬币a-h，其中一枚硬币是伪造的(更轻)



例：考试成绩分布表

如果利用下面的决策树来转换成5分制，考虑对一个100人的考试结果，需要比较的次数（还是分布已知）

[0, 60)	[60, 70)	[70, 80)	[80, 90)	[90, 100]
不及格	及格	中	良	优
0.10	0.15	0.25	0.35	0.15



该判定树的带权路径长度

$$\begin{aligned} WPL &= 0.10*1+0.15*2+0.25*3+0.35*4+0.15*4 \\ &= 3.15 \end{aligned}$$

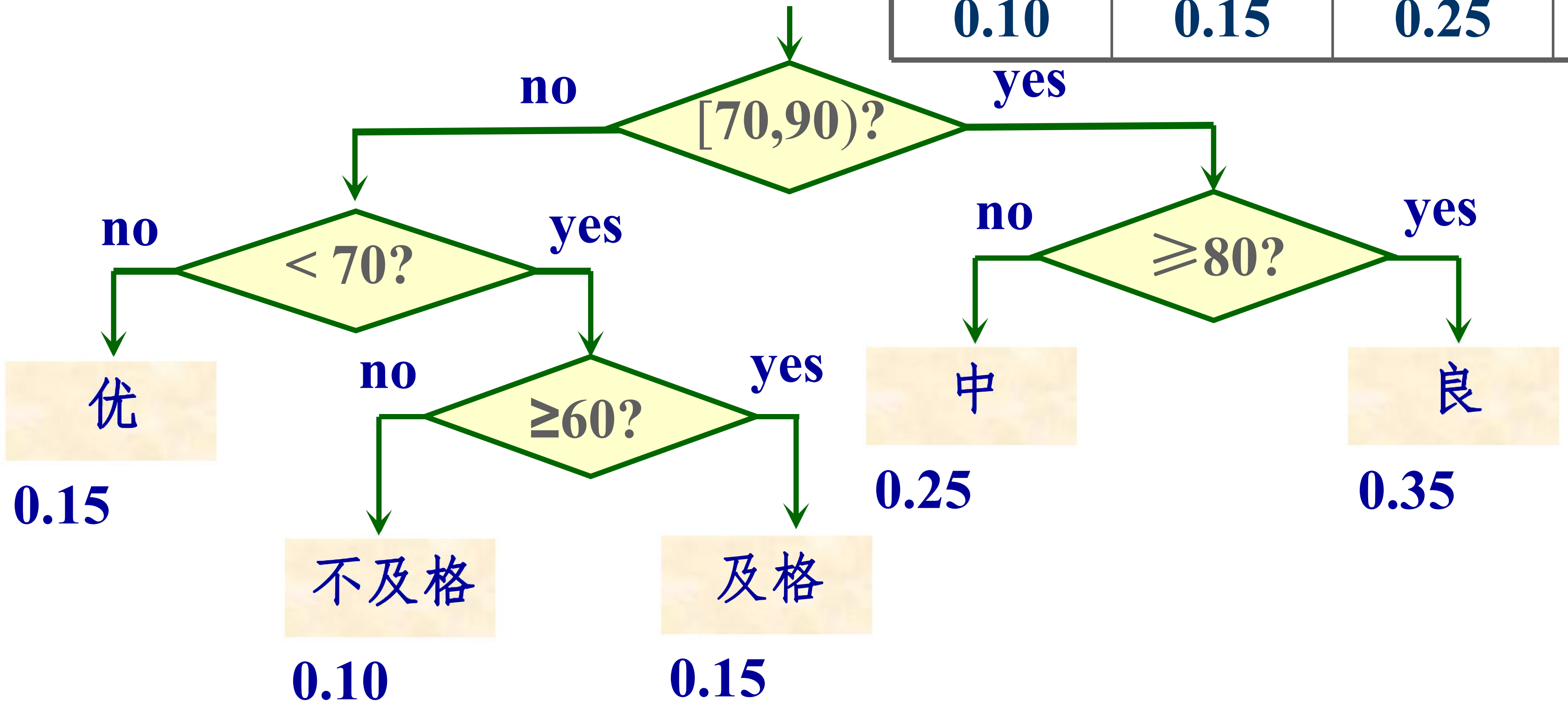
此次数越少越好。

如果按照Huffman算法的思想构造，可望得到平均比较次数更少的判定树。

下图就是按Huffman算法构造出的判定树。

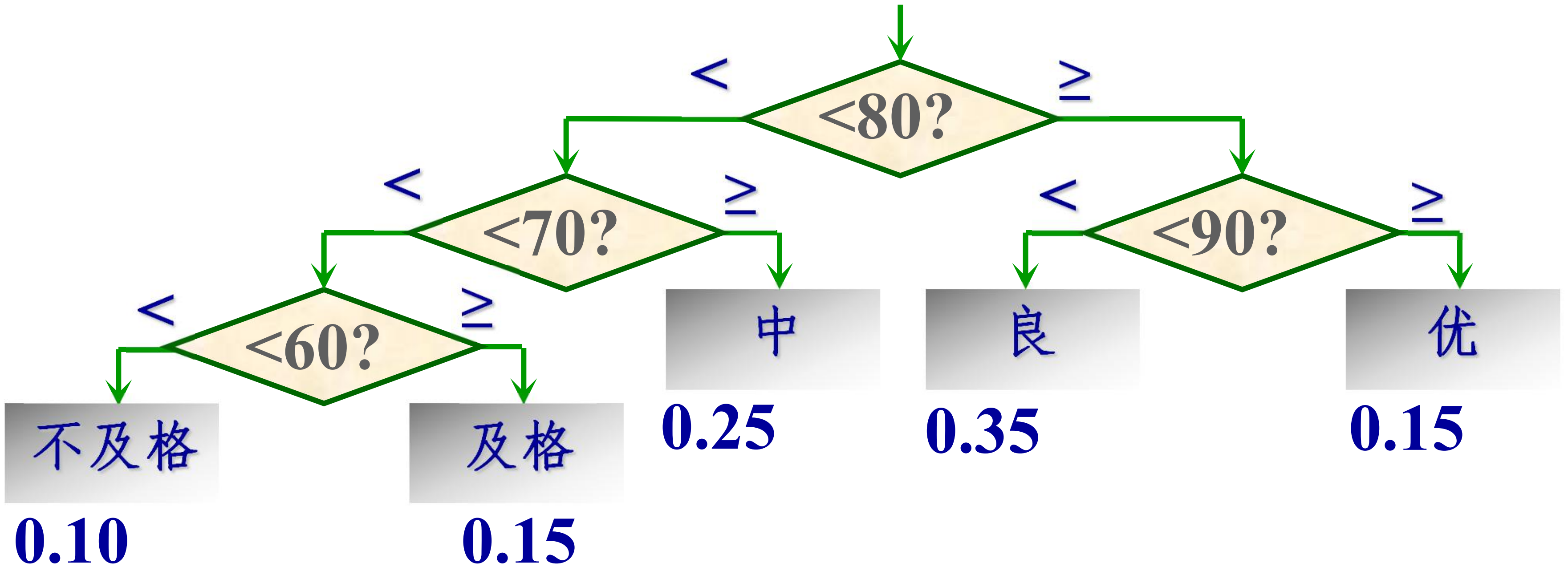
按Huffman树改造判定树

[0, 60)	[60, 70)	[70, 80)	[80, 90)	[90, 100]
不及格	及格	中	良	优
0.10	0.15	0.25	0.35	0.15



但此判定树的问题是：根结点的判定需要2次比较。为此，对它加以调整，可得到最合理的判定树。

最佳判定树



$$WPL = 0.10*3+0.15*3+0.25*2+0.35*2+0.15*2$$

$$= 0.3+0.45+0.5+0.7+0.3 = 2.25$$

最佳判定树

堆

二叉搜索树/二叉排序树

堆 (Heap)

设有一个关键字集合，按完全二叉树的顺序存储方式存放在一个一维数组中。对它们从根开始，自顶向下，同一层自左向右从 0 开始连续编号。若满足

$$K_i \le K_{2i+1} \ \&\& \ K_i \le K_{2i+2}$$

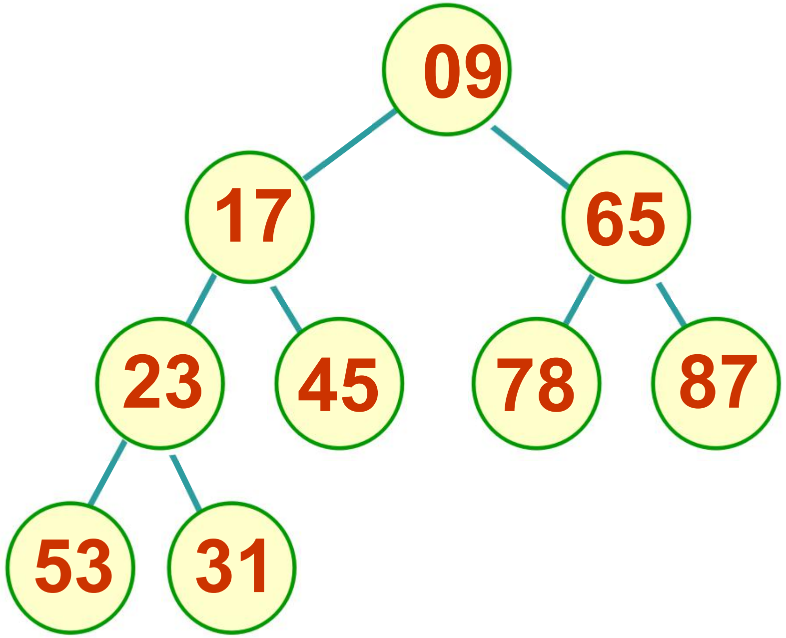
或

$$K_i \ge K_{2i+1} \ \&\& \ K_i \ge K_{2i+2},$$

则称该关键字集合构成一个堆。

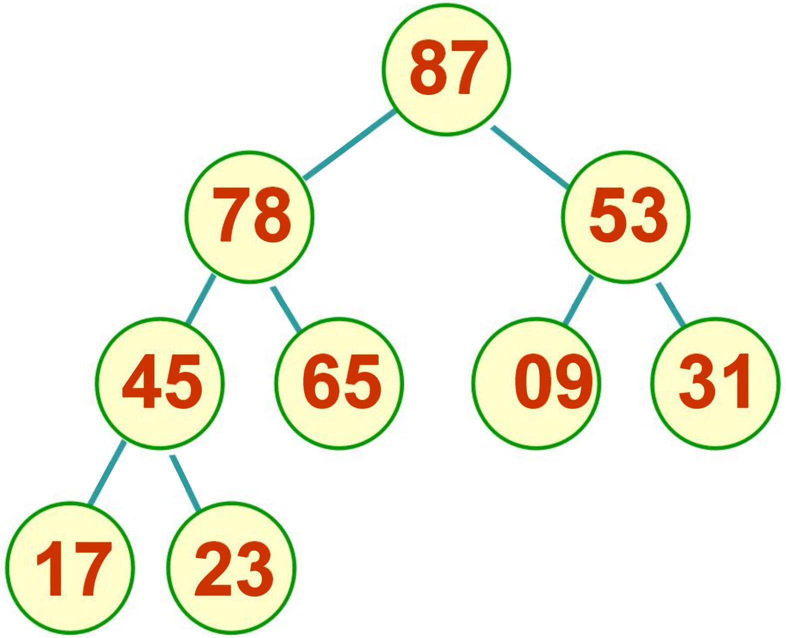
前者称为最小堆（小根堆），后者称为最大堆（大根堆）。

堆的定义



完全二叉树
顺序表示

$K_i \le K_{2i+1}$ &&
$K_i \le K_{2i+2}$



完全二叉树
顺序表示

$K_i \ge K_{2i+1}$ &&
$K_i \ge K_{2i+2}$

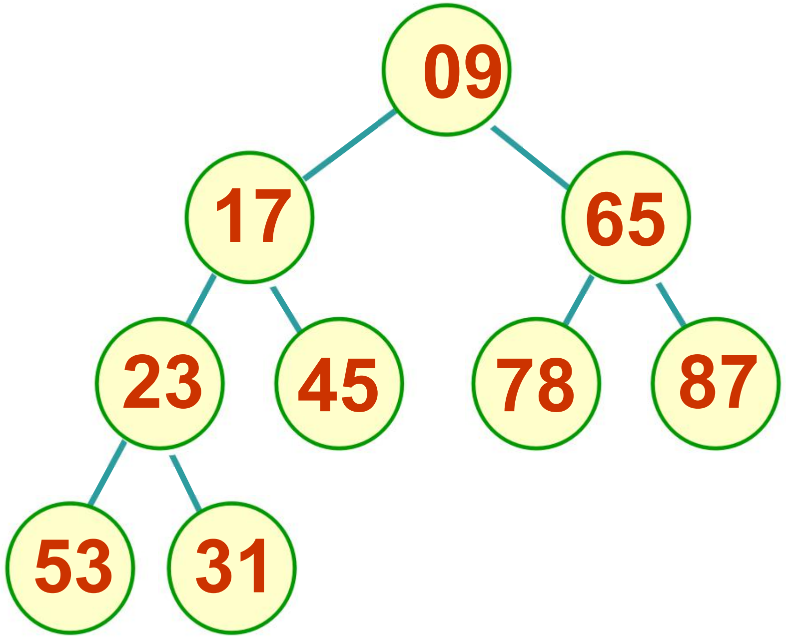
最小堆的顺序存储实现（静态）

```
#define MaxHeapSize  100;
typedef int KeyType;
typedef struct {            //堆元素的定义
    KeyType key;            //元素关键字，如huffman树节点权重
    //other data;                //元素其他数据
} HeapElem;

typedef struct {            //最小(大)堆定义
   HeapElem data[MaxHeapSize];       //存放数组
   int size;                //最小堆当前元素个数
} MinHeap;
```

将一组用数组存放的任意数据调整成堆

53 17 78 23 45 65 87 09



如何将原始的数据，变换成最小堆？

先将数据直接放入数组，此时等价于一棵完全二叉树，但是这棵树并不符合堆的定义，需要进行节点调整

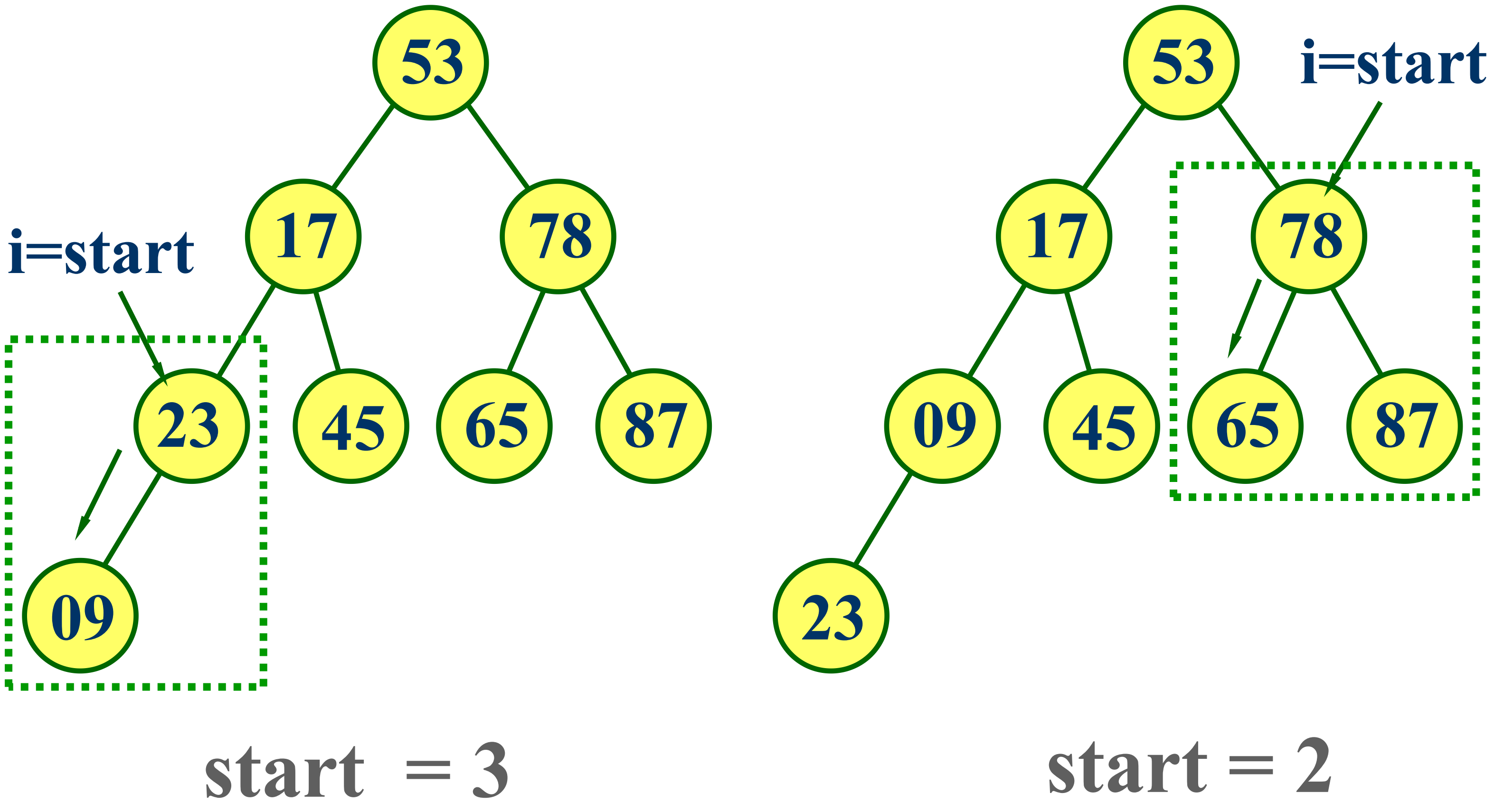
将一组用数组存放的任意数据调整成堆

53 17 78 23 45 65 87 09

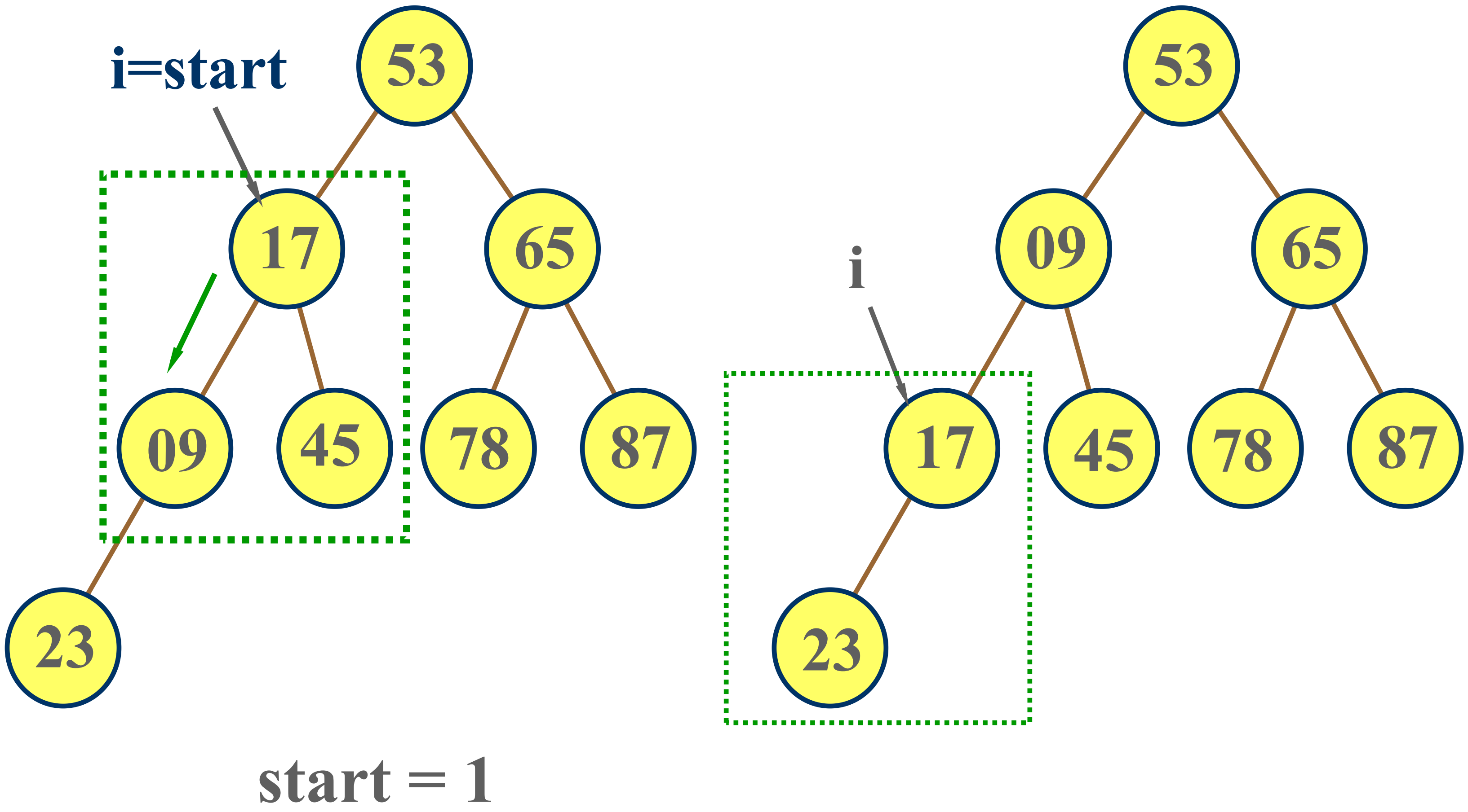
从后(最后一个分支节点)向前逐步调整为最小堆的过程

start为为每一次调整的开始点；

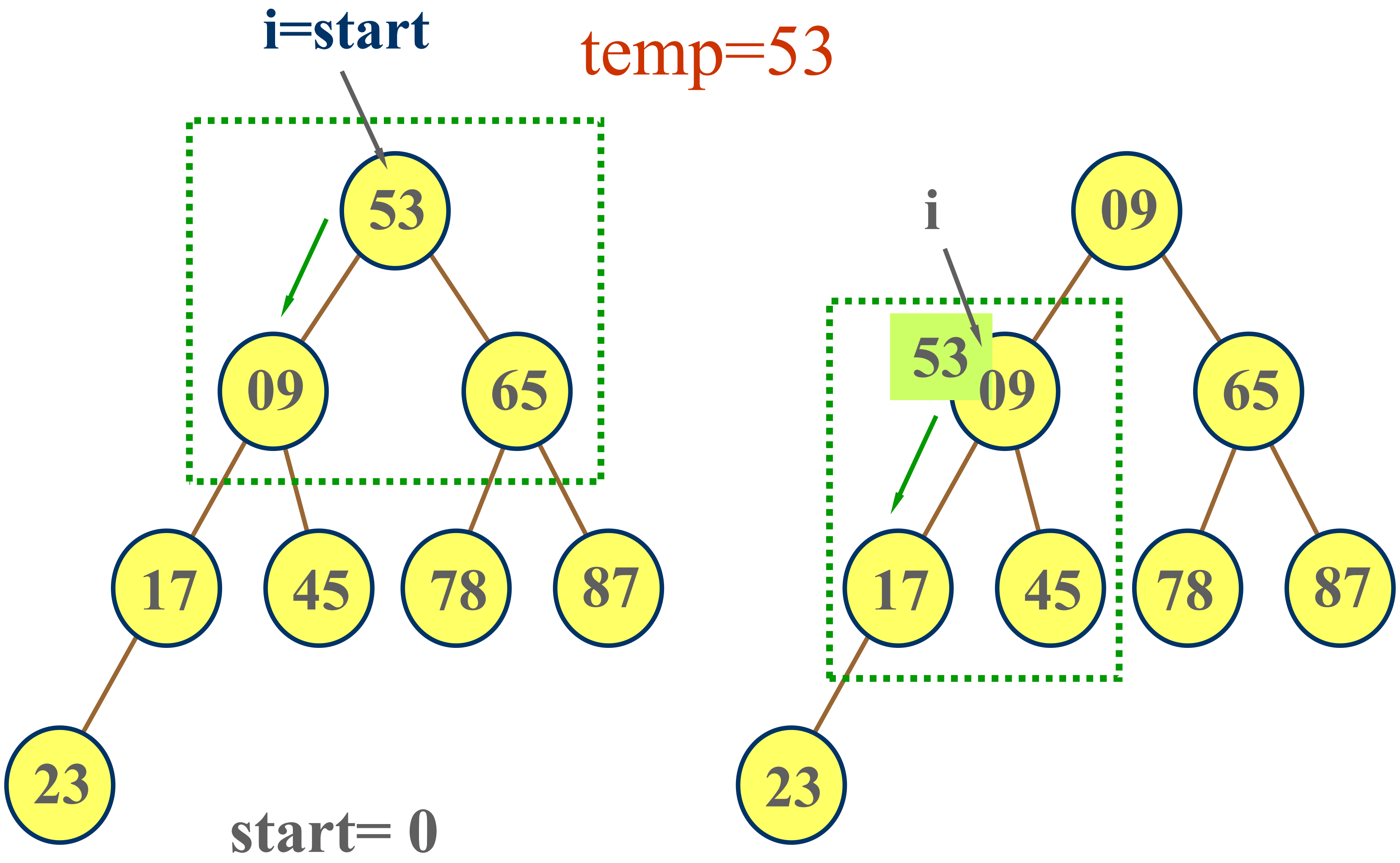
每次调整是使当前子结构中大的双亲节点元素向下挪动：sift_down

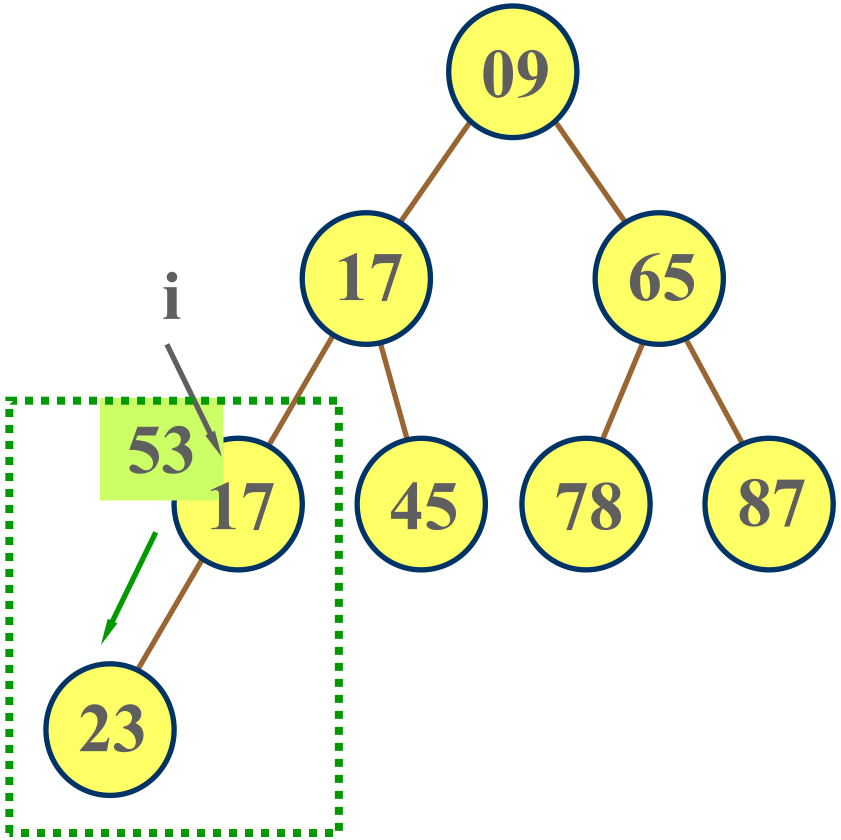


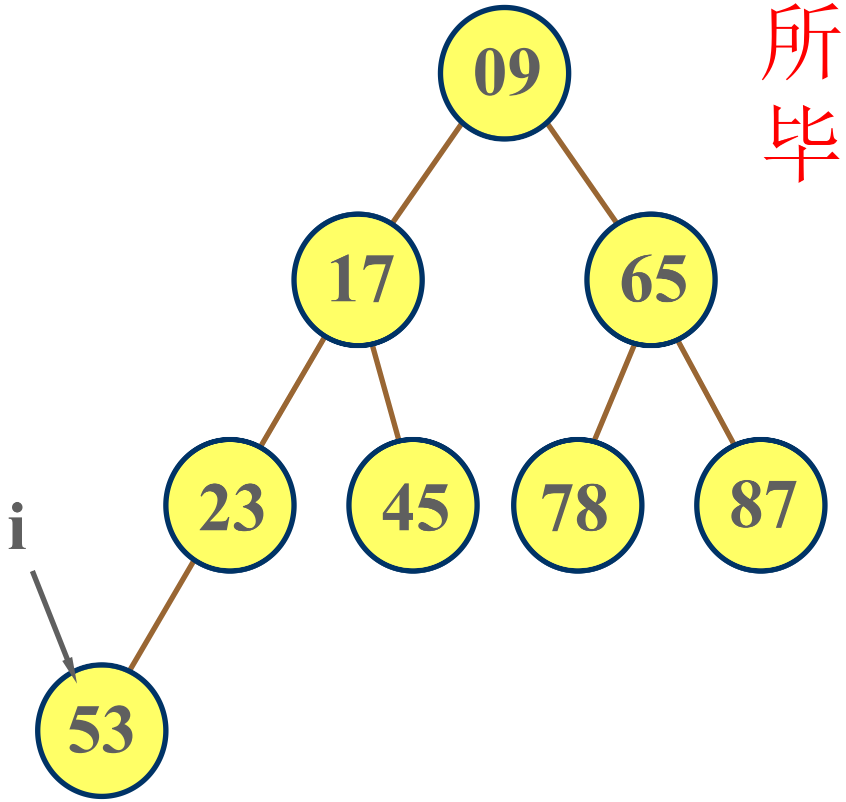
调整后，要保证仍为堆，可能需要对新调整下来的元素，找到子结构作为当前结构，继续调整，直到最后



由于可能会连续调整，因此不必要每次都进行节点交换。只要开始保存start节点，放到最后该节点就位的位置即可







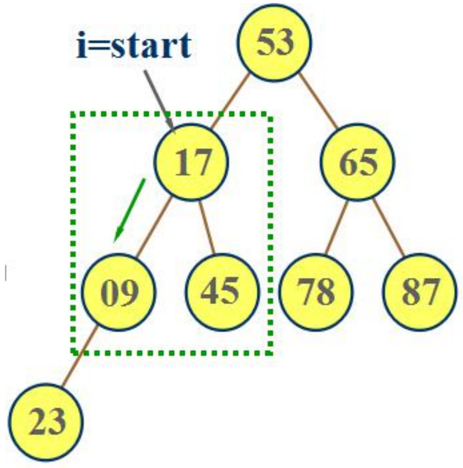
所有子结构调整完毕，因此保证是堆

当根节点编号从0开始，则有：

最后一个节点编号为n-1
最后一个分支节点编号为(n-2)/2

父节点i的左儿子=2*i+1
父节点i的右儿子=2*i+2

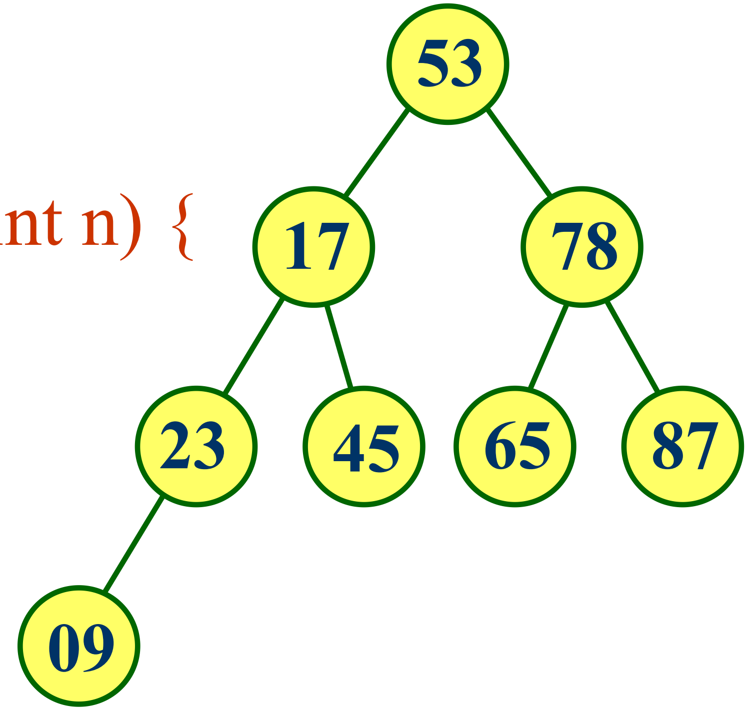
最小堆的向下筛选算法(保证下分支是堆)



start = 1

```
void SiftDown(MinHeap* H,  int start,  int EndOfHeap) {
    int i = start,   j = 2*i+1;          // j 是 i 的左子女
    HeapElem temp = H->data[i];   //保存待调整元素
    while (j <= EndOfHeap) {//EndOfHeap为最后一个节点的index
        if (j < EndOfHeap && H->data[j].key > H->data[j+1].key)  j++;
             //两子女中选小者
        if (temp.key <= H->data[j].key) break; //比最小的小，不用调
        else { H->data[i] = H->data[j]; i = j;   j = 2*j+1; }
                //比两子女小者大，则向下滑动
    } H->data[i] = temp;        //待调整元素就位
}//时间复杂度（Olog2n）
```

最小堆的建立




```
void CreatMinHeap (MinHeap* H, HeapElem arr[ ],  int n) {
//根据给定数组中的数据和节点数,建立小根堆
    for (int i = 0; i < n; i++) H->data[i] = arr[i];
    H->size = n;          //初始化完毕
    int start = (H->size−2)/2;   //从最后分支结点开始
    while (start >= 0) {           //从后向前逐步调整堆直到最小堆的根
        SiftDown(H, start, H->size−1);//index为节点数-1
        start−−;
    }
}//root节点为0
```

向最小堆插入元素

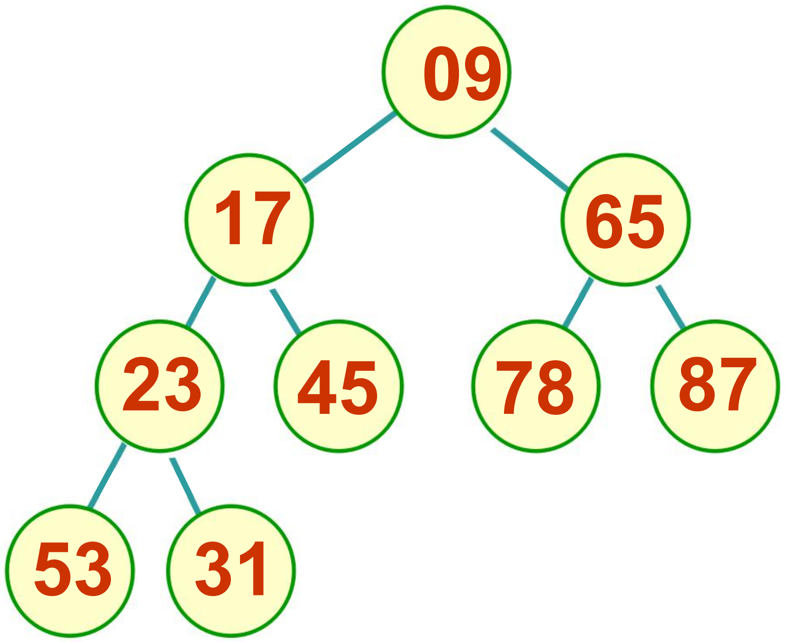
思考，如何插入？需要保持或调整仍然是最小堆。

请听题：新元素插入位置宜选择（）

A、最前；B、最后；C、中间某位置

D、随机

调整成最小堆（完全二叉树）方便



最小堆的插入和向上调整例

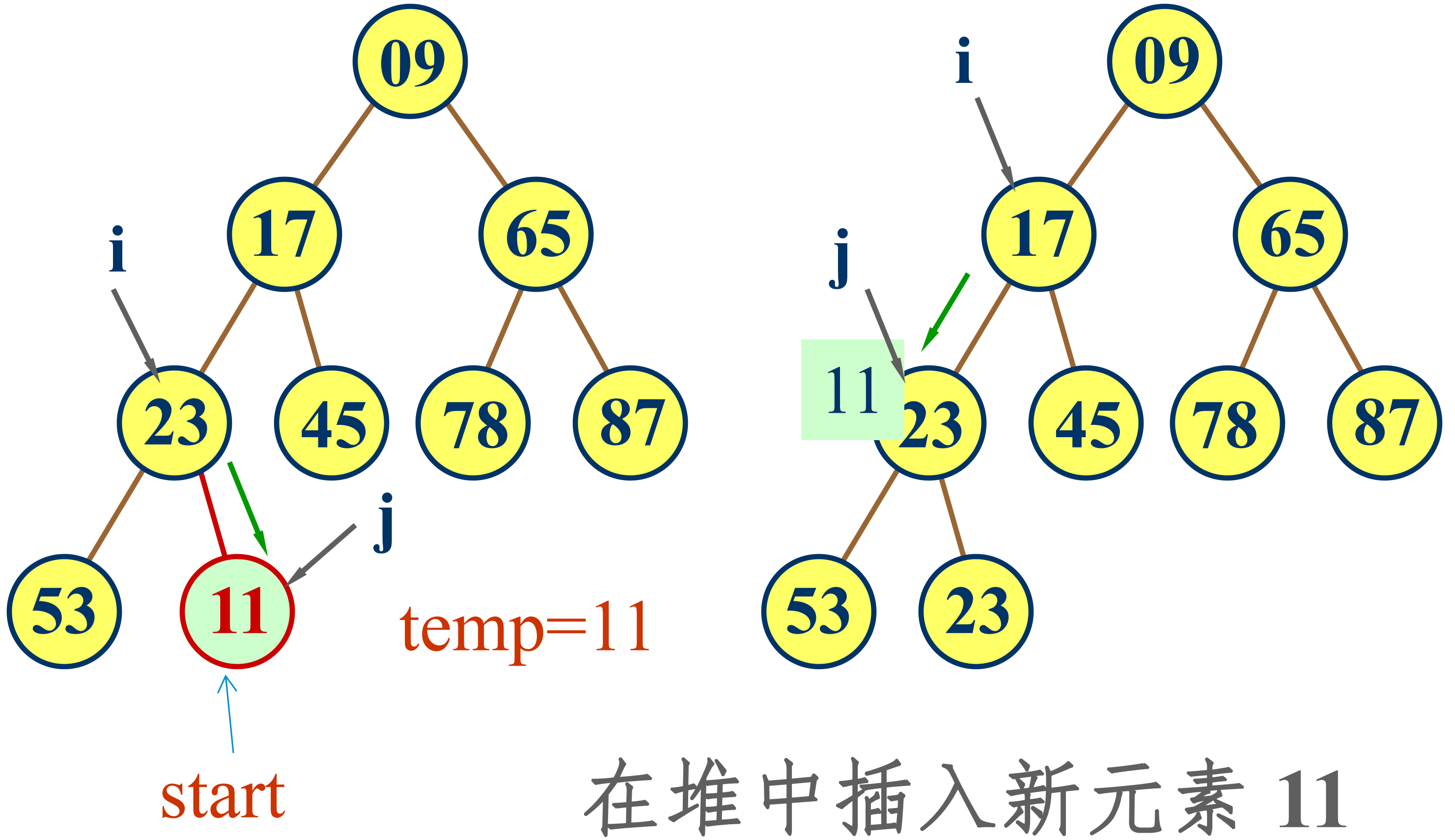
插入在堆的最后位置进行，随之找到该元素的双亲，进行调整。

调整仍然是从后向前调整，但是是将值小的节点向上调整sift_up

i指向双亲节点

j为当前节点

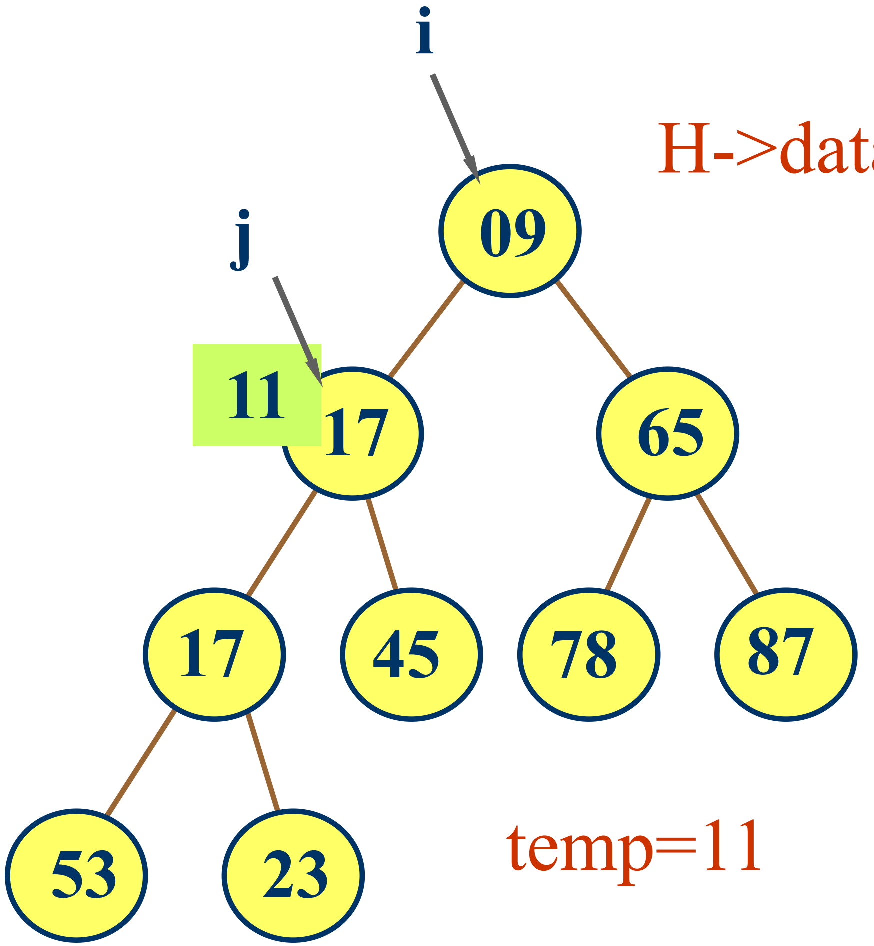
temp保存start节点



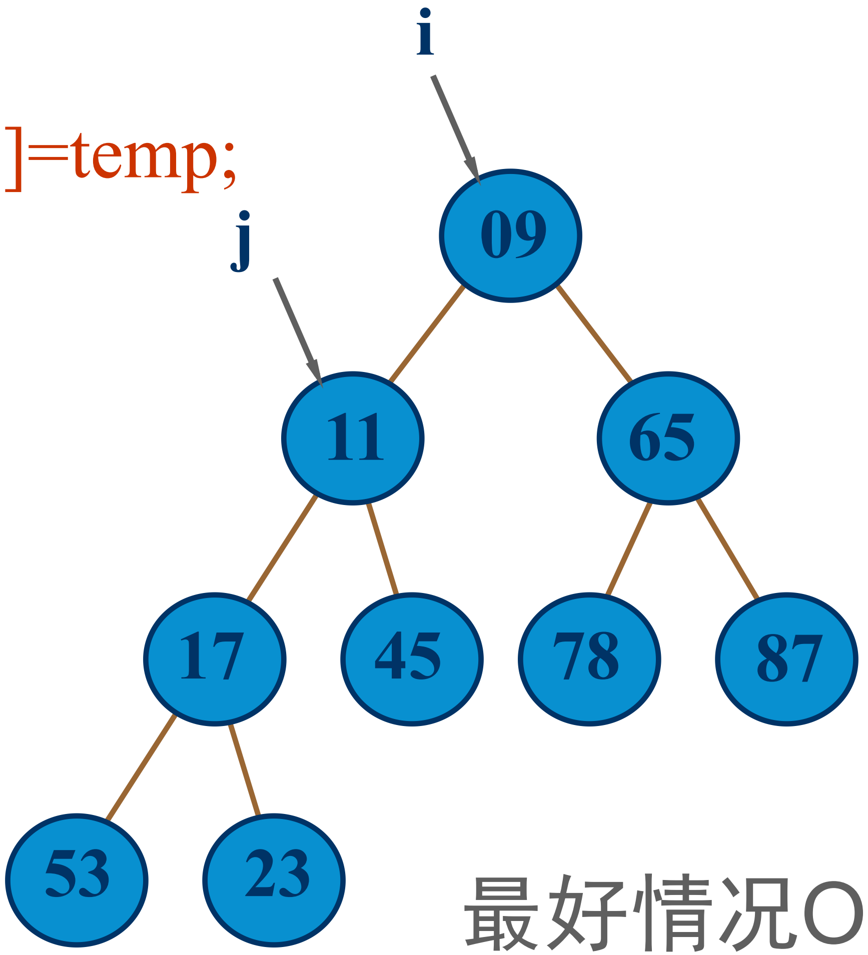
在堆中插入新元素 11

*H

H->data[j]=temp;



temp=11

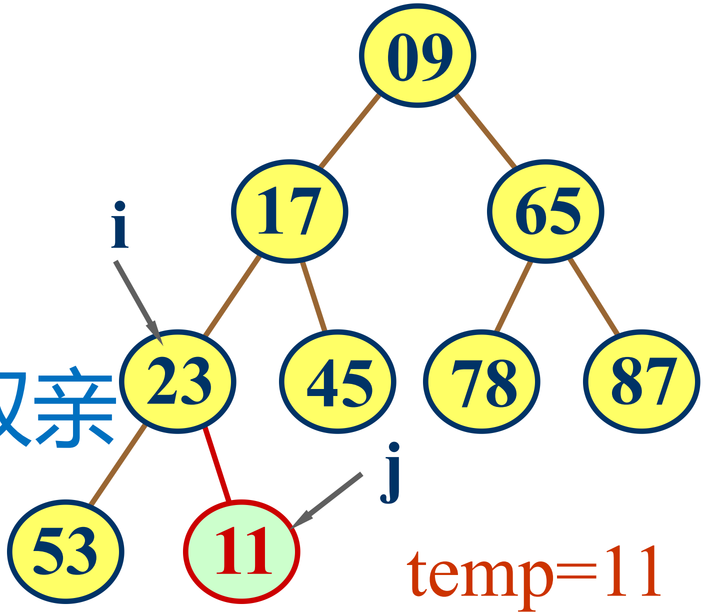


*H

最好情况O（1）

最坏情况O（log2n）

最小堆的向上筛选算法




```
void SiftUp (MinHeap* H, int start) {
//从 start 开始,向上直到0,调整堆
    int j = start,  i = (j-1)/2;            // i 是 j 的双亲
    HeapElem temp = H->data[start];
    while (j > 0) {
        if (H->data[i].key <= temp.key) break;//每次与temp比
        else {  H->data[j] = H->data[i];    // 复制一个双亲节点
                j = i;  i = (i-1) /2;  }
    }
    H->data[j] = temp;
}
```
时间复杂度（Olog2n）。思考：为何不比较左右子？

最小堆的插入算法

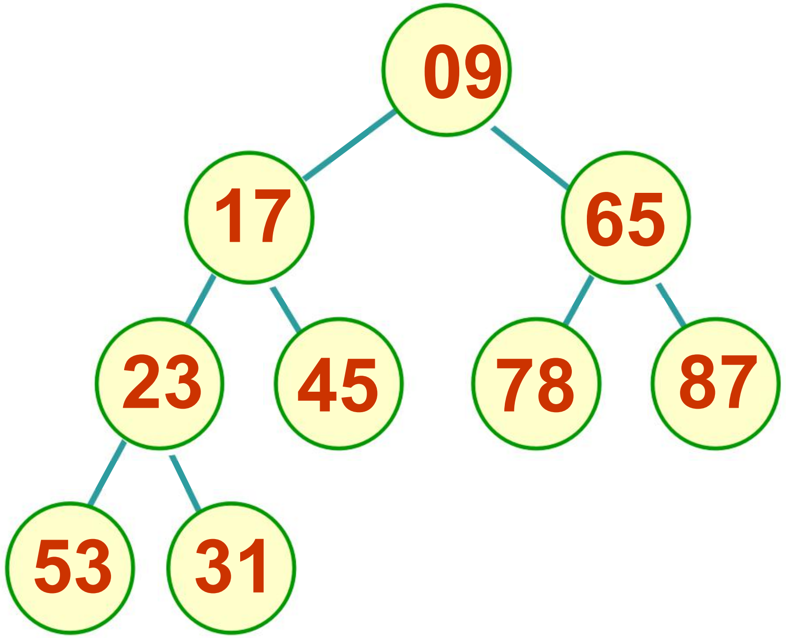
```
int insert(MinHeap* H, HeapElem x) {
//在堆中插入新元素 x
    if (H->size == MaxHeapSize)      //堆满
      { printf (“堆已满\n”);   return 0; }
    H->data[H.size] = x;                      //插在表尾
    sift_up(H, H->size);       //向上调整，元素已在，不用减一
    H->size++;                                //堆元素增一
    return 1;
}
```

从最小堆删除元素

请听题：最小堆元素删除位置宜选择（）

A、最前；B、最后；C、中间某位置

D、随机



除了动态调整成最小堆方便之外.....

须考虑什么时候或为什么要在最小堆删除一个元素？

这个元素的值是随意的嘛？

最小堆维持了一个动态的数据结构，堆顶元素永远是所有元素最小值。因为要（不断）利用最小值，所以要删除堆顶

从最小堆删除元素

OK，从堆顶将最小值删除，然后如何调整？

请选择：（）

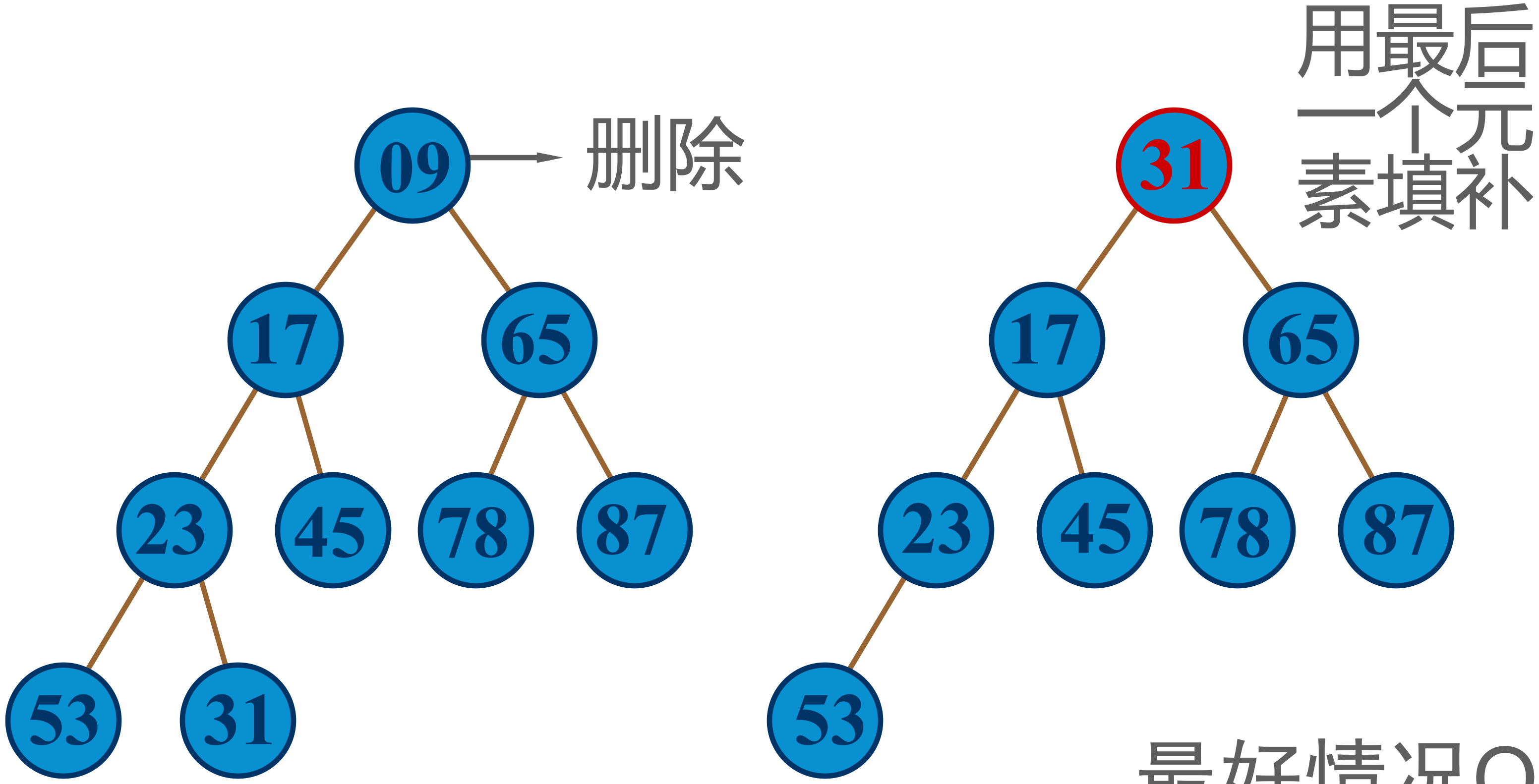
A、从左子女开始调整；B、从右子女开始调整；C、从最后一个节点开始调整

D、以上思路都不对

1、先保证是一个完全二叉树

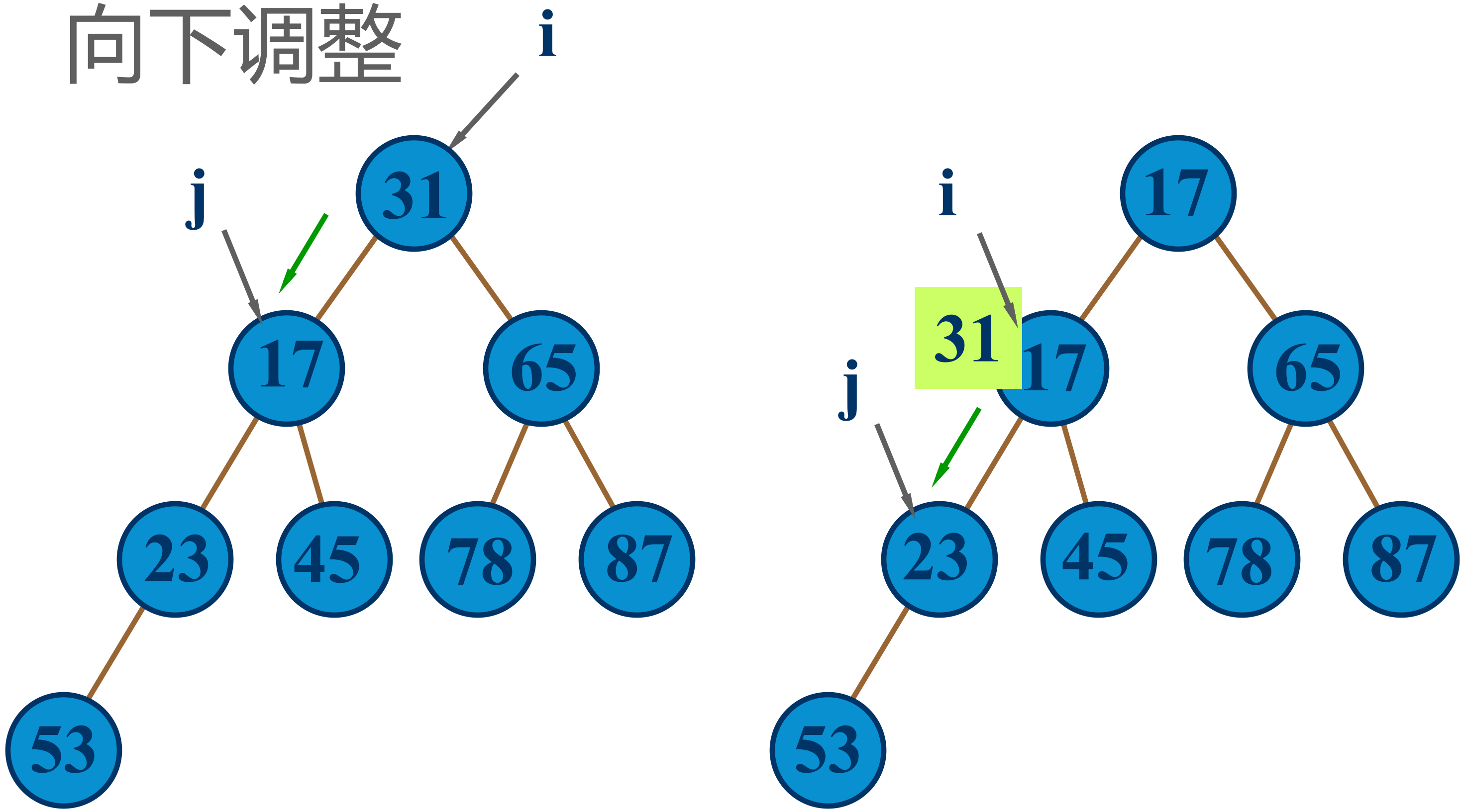
2、调整成最小堆方便

最小堆的删除和向下调整例

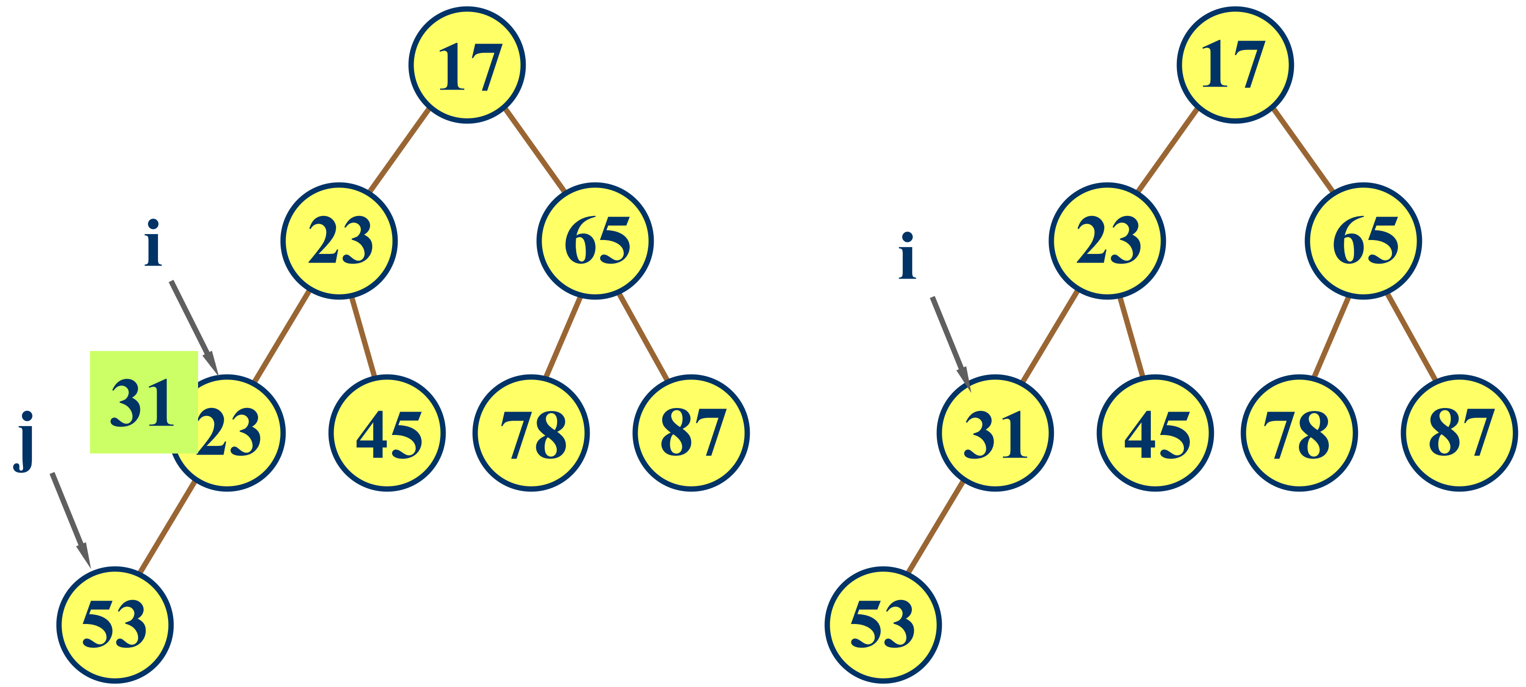


最好情况O（1）
最坏情况O（log2n）

向下调整



向下调整



最小堆的删除算法

```
int RemoveMin(MinHeap* H, HeapElem *e) {//伪码
    if (!H->size)
        { printf( " 堆已空 \n" );  return 0; }
    e = H->data[0];                       //最小元素出堆
    H->data[0] = H->data[H->size-1];//用最后元素填补
    H->size--;
    sift_down(H, 0, H->size-1);     //调整，index为节点数-1
    return 1;
}
```

堆应用

优先队列

RemoveMin方法即为DeQueue方法

Insert方法即为Enqueue方法

可补全并实现队列其他方法，就实现了优先队列

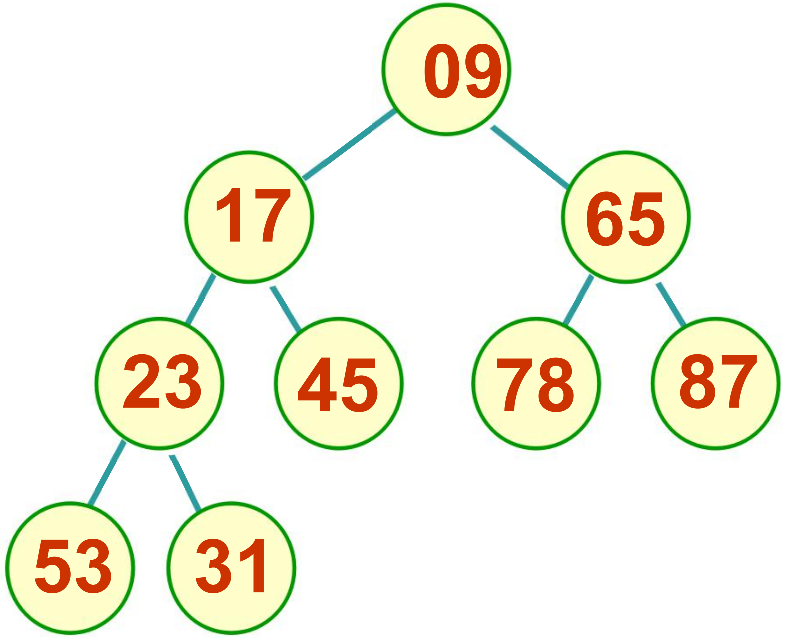
基于优先队列的Huffman树

堆排序及性能分析（排序时讲）

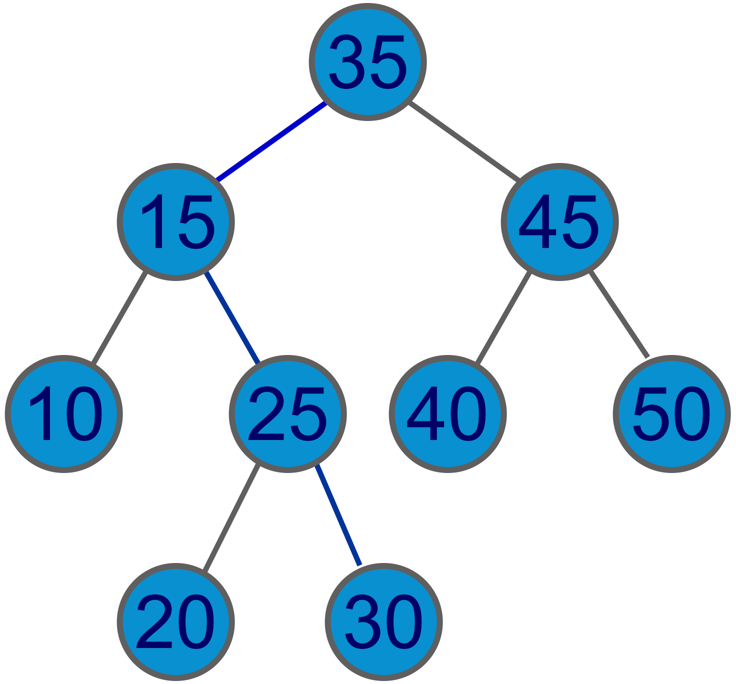
最佳判定树

堆

二叉搜索树/二叉排序树



堆



二叉搜索/排序树

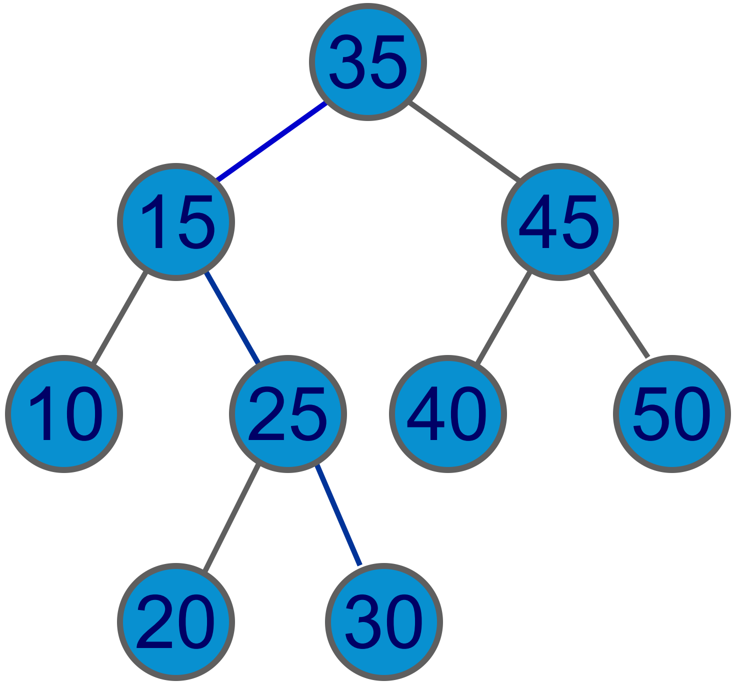
- <u>定义（递归定义）</u>

二叉排序树或者是一棵空树，或者是具有下列性质的二叉树：

- 每个结点都有一个作为查找依据的关键字(key)（暂针对所有结点的关键字互不相同。）
- 左子树（如果非空）上所有结点的关键字都小于根结点的关键字。
- 右子树（如果非空）上所有结点的关键字都大于根结点的关键字。
- 左子树和右子树也是二叉排序树。

国外教材统称为二叉搜索树。

请大家写出中序遍历结果（先猜一下）



中序遍历，可以按从小到大的顺序将各结点关键字排列起来。

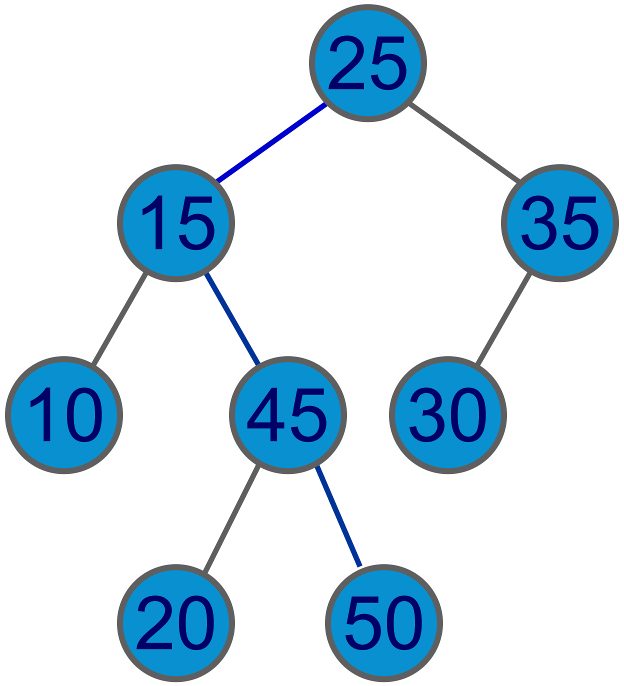
思考，为什么？　猜想是二叉排序树中文名称的由来。

那外国为啥叫二叉搜索树呢？

例题：一棵二叉树是二叉排序树的（　　）条件是树中任一结点的关键字值都大于左子女的关键字值，小于右子女的关键字值。

A. 充分但不必要　　B. 必要但不充分

C. 充分且必要　　D. 既不充分也不必要



X的必要条件是Y，意味着：X可以----->Y

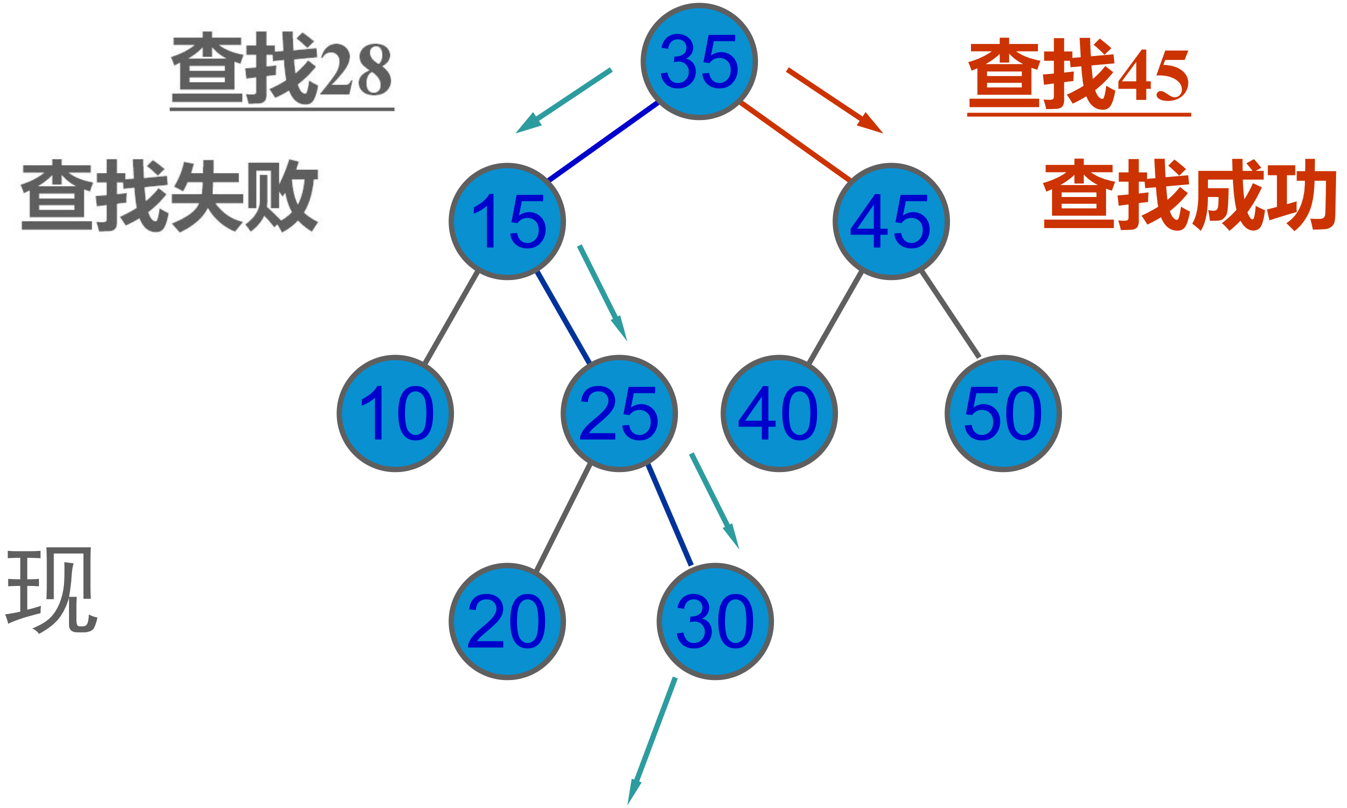
X的充分条件是Y，意味着：Y可以----->X

一般用二叉链表存储

```
typedef char ElemType;          //树结点数据类型
typedef struct node {           //二叉排序树结点
    ElemType data;
    struct node *LeftChild, *RightChild;
} BSTNode, *BST;                //二叉排序树定义
```

二叉排序树是二叉树的特殊情形，它继承了二叉树的结构，增加了自己的特性，对数据的存放增加了约束

- 在二叉排序树上进行查找，是一个从根结点开始，沿某一个分支逐层向下进行比较判等的过程。

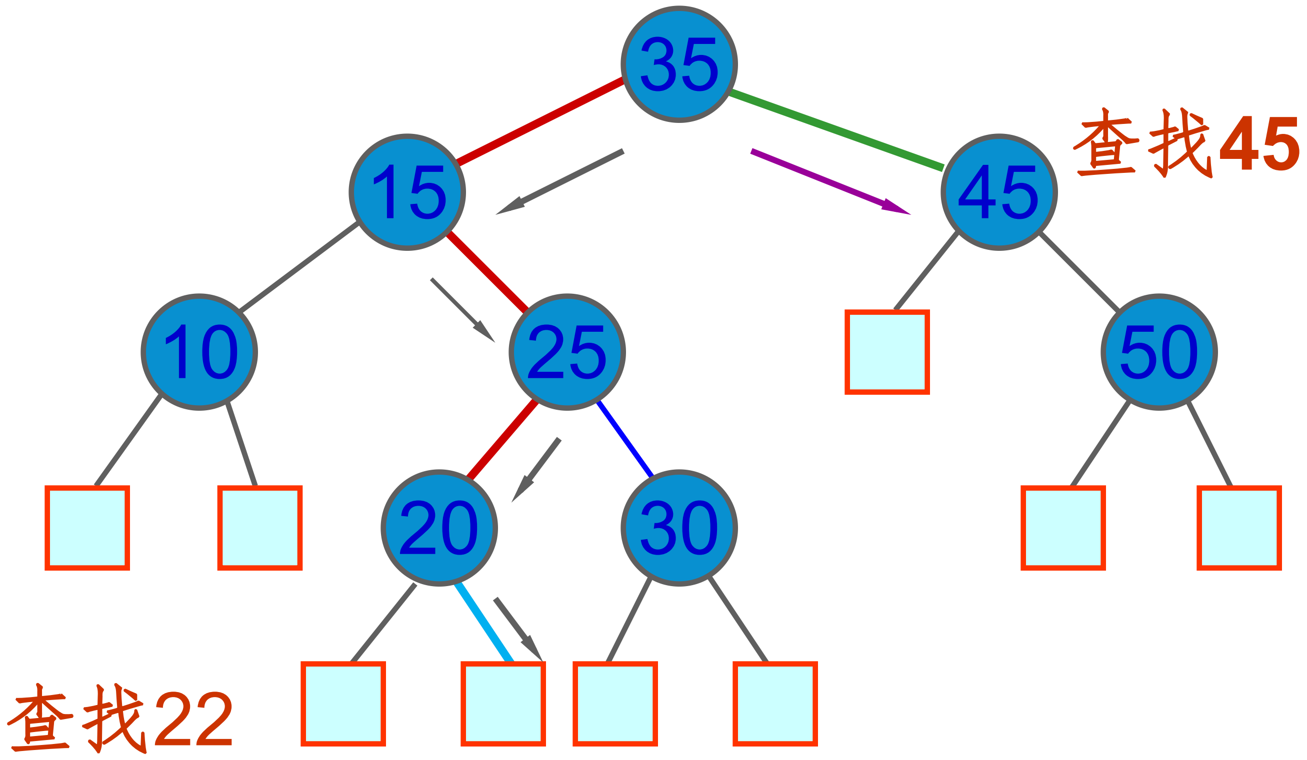


- 可递归实现

在二叉排序树中查找关键字为x的元素，查找过程从根结点开始。

- 如果根指针为NULL，则查找不成功；否则用给定值 x 与根结点的关键字进行比较：
  - 如果给定值等于根结点的关键字值，则查找成功。
  - 如果给定值小于根结点的关键字值，则继续递归查找根结点的左子树；
  - 否则。递归查找根结点的右子树。
- 查找成功时检测指针停留在树中某个结点。
- （教材228，算法9.5a）。递归实现起来比较简单。

- 可用判定树描述查找过程。内结点是树中原有结点，外结点是失败结点，代表树中没有的数据。
- 查找不成功时检测指针停留在某个失败结点。



```
void find (BST T, ElemType x, BST *p, BST *pr) {//在T中查找关键字等于 x 的结点，成功时 p 返回找到结点地址, pr 是其双亲结点(parent).//不成功时 p 为空, pr 返回最后走到的结点地址（插入位置预留）。
  if (T) {
      *p = T;  *pr = NULL;    //从根查找，根的双亲定义为空
      while (*p && *p->data != x) {
          *pr = *p;
          if (*p->data < x)  *p = *p->rightChild;
        else *p = *p->leftChild;         }    }}
```

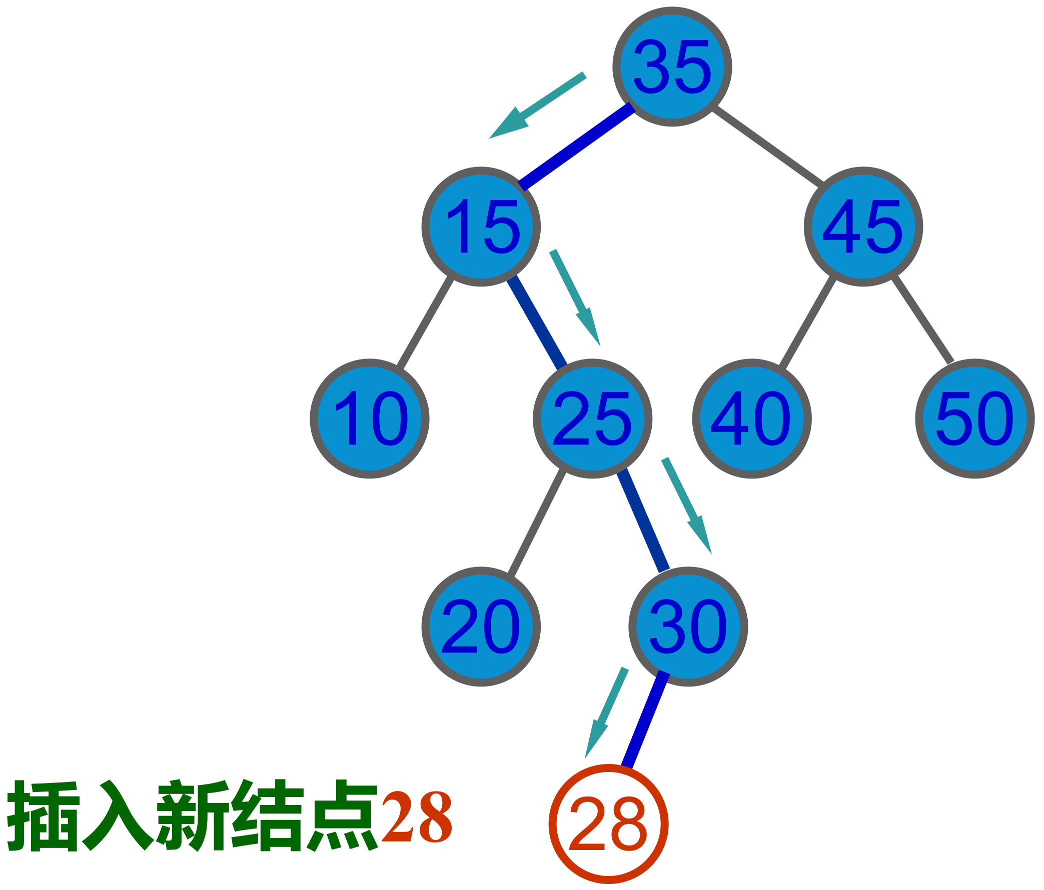
查找的关键字比较次数最多不超过树的高度O(height)。

- 每次结点的插入，都要从根结点出发查找插入位置，然后把新结点作为叶结点插入。

- 为了向二叉排序树中插入一个新元素，必须先检查这个元素是否在树中已经存在。

插入之前先使用查找算法在树中检查要插入元素有还是没有。

- 查找成功：树中已有这个元素,不再插入。
- 查找不成功：树中原来没有关键字等于给定值的结点，把新元素加到查找操作停止的地方。



```
void insert (BST * T, ElemType x) {//伪码
//将新元素 x 插到以 *t 为根的二叉排序树中
    BST pt, prt, q;
    find (*T, x, &pt, &prt);    //查找结点插入位置
    if (!pt ) {            //查找失败时可插入
        q = new BSTNode;  q->data = x;     //创建结点
        q->LeftChild = q->RightChild = NULL;
        if (!prt)  *T = q;      //父节点空，插入前为空树
        else if (x < prt->data) ptr->LeftChild = q;
            else ptr->RightChild = q;} } O(树高)
```

BST的创建过程

假设有数据 { 53, 78, 65, 17, 87, 09, 81, 15 ....}

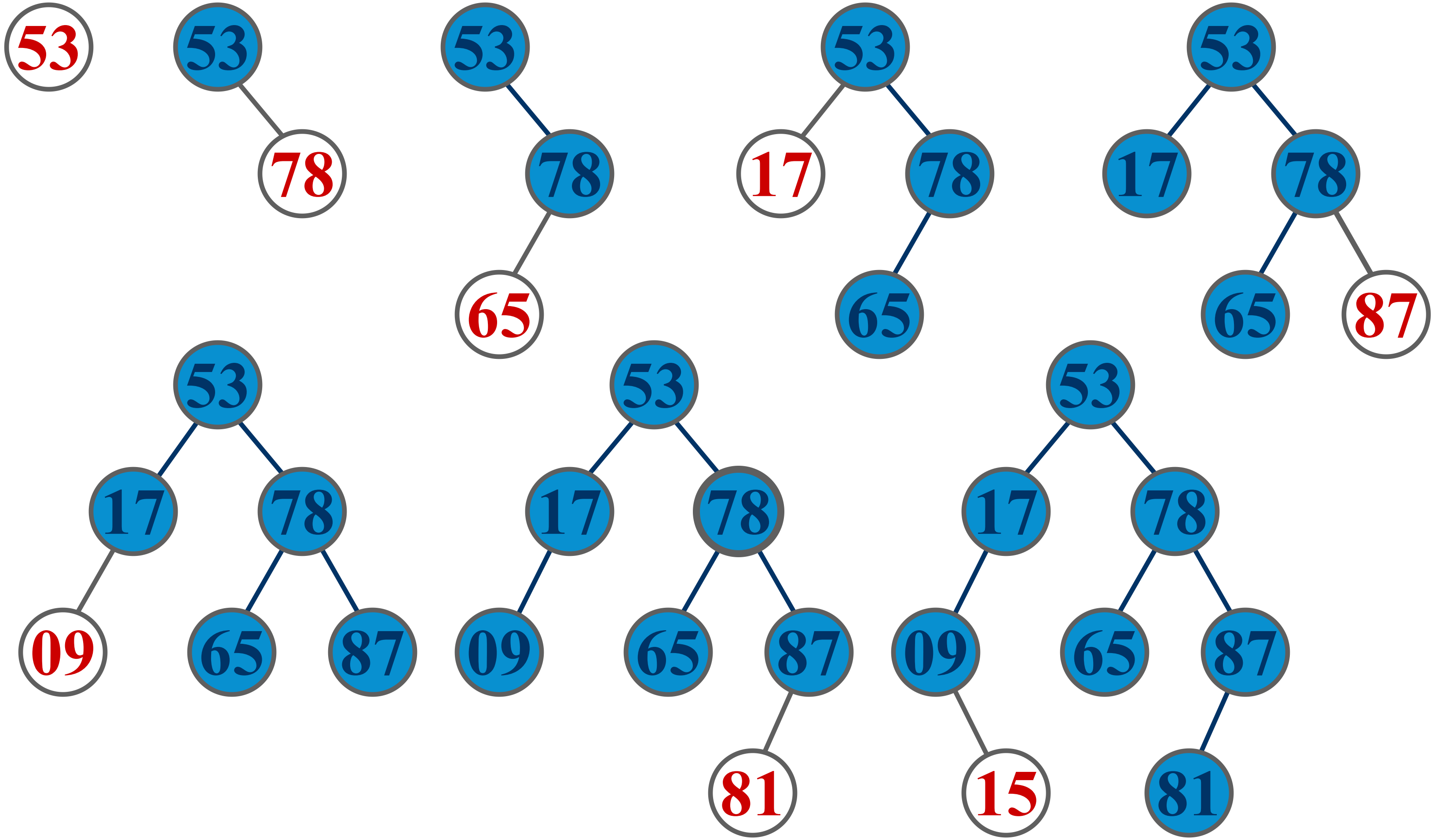
如何创建一个BST?

依次输入数据，然后插入树中即可。

且，前面insert算法已经考虑空树的情况。

BST的创建过程

- **输入数据 { 53, 78, 65, 17, 87, 09, 81, 15 }**



BST的创建算法

```
void creat(BST *T){
        ElemType e;
        while(e!=-1){
                scanf("%d",&e);
                insert(T, e);
        }
}
```

对任意一系列不重复数据，如{ 1, 2, 3 }建立的二叉排序树形态是否唯一？

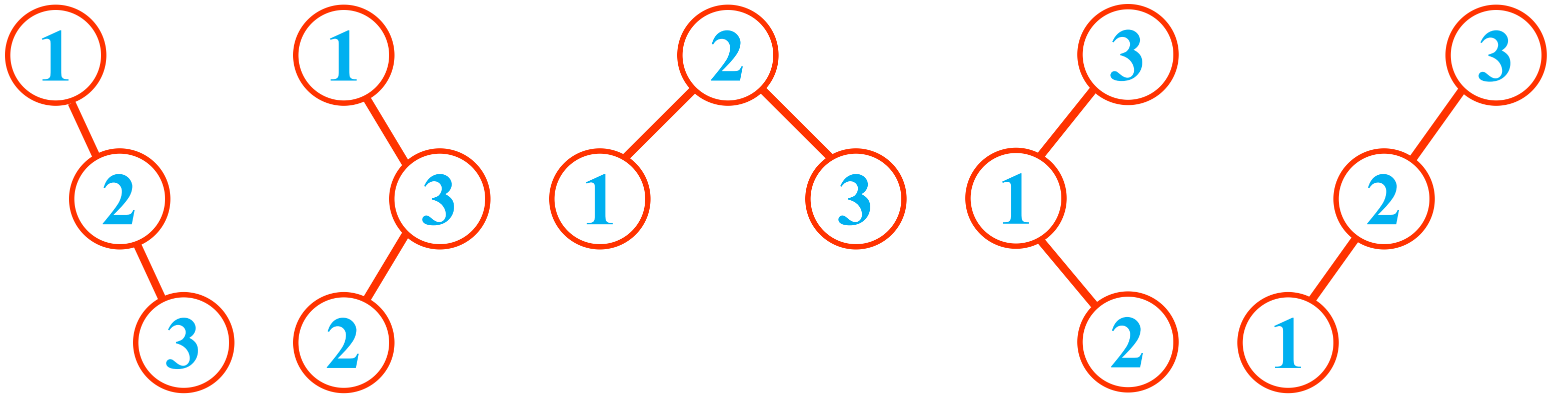
输入顺序不同，建立起来的二叉排序树的形态也不同。这直接影响到二叉排序树的查找性能。

问：

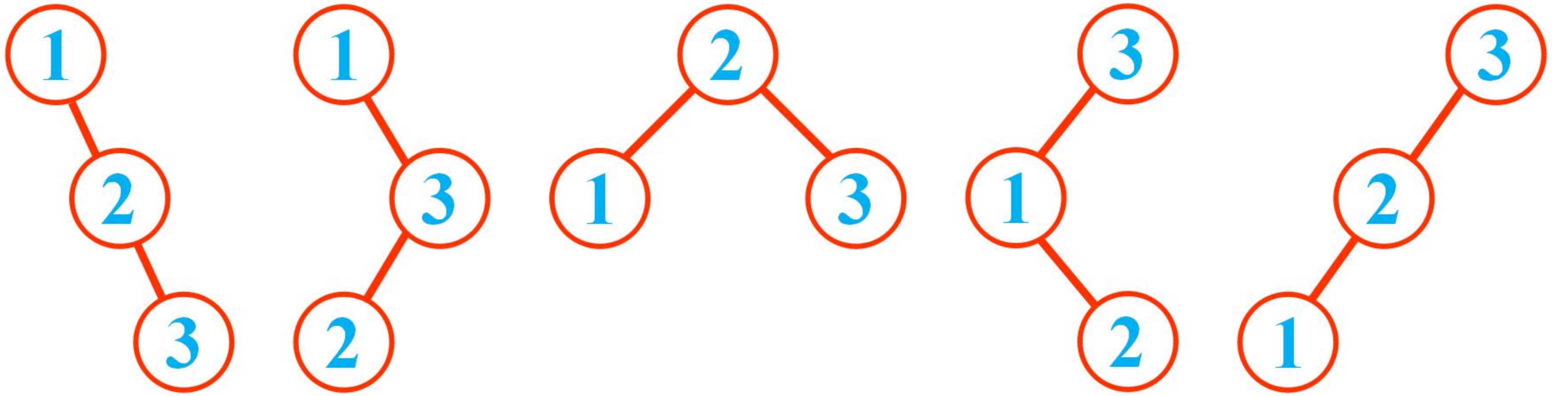
n个不同的数据（n>=3），仅输入顺序不同，则建立起来的二叉排序树形态共有：

A(n)种　　B(n-1)种　　C(n!)种　　D以上均不对

{2, 1, 3} {1, 2, 3} {1, 3, 2} {2, 3, 1} {3, 1, 2}
{3, 2, 1}



- 如果输入序列选得不好，会建立起一棵单支树，使得二叉排序树的高度达到最大。(怎么办？)



如已知输入序列全体，可使输入序列随机化，一定程度上可使树高降低（但一般是动态的树...）

- 对于有 n 个关键字的集合，其关键字有 $n!$ 种不同排列，可以证明可构成不同二叉排序树的个数是第n个Catalan数（卡特兰数）

$$当 n>0 时，c(n)=\binom{2n}{n}\frac{1}{n+1}，\quad c(0)=1$$

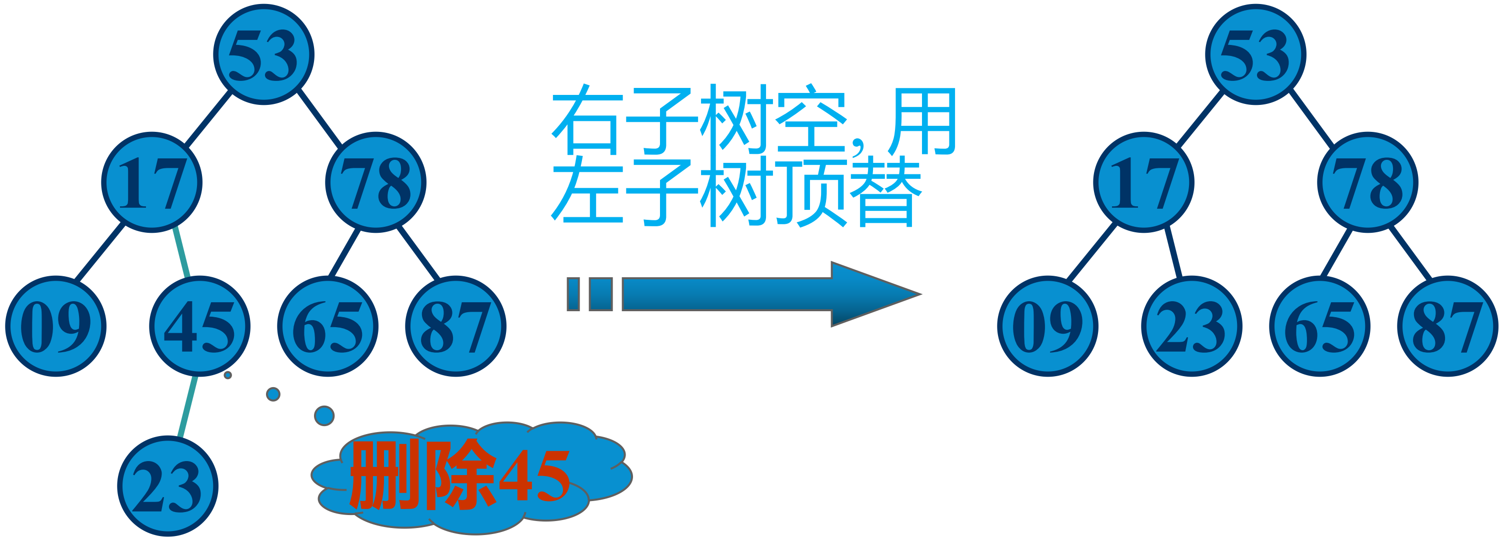
- 二叉排序树性能（O(logN)-?-O(N)）。

- 对数据随机分布的情况，可以证明，二叉排序树的平均查找长度与log(n)是等数量级的。教材P232的证明。
- 但是在一般情况下（50%左右），就不这么乐观了，性能会由于树的偏斜而下降到数量级n

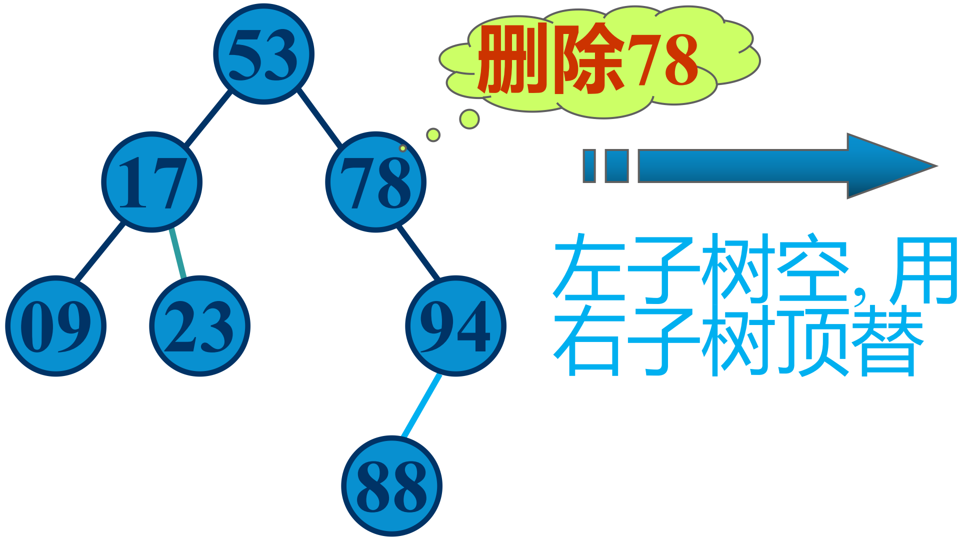
在查找过程中关键字的平均比较次数，
也称为平均查找长度ASL

- 在二叉排序树中删除一个结点时，必须将因删除结点而断开的二叉链表重新链接起来（保证是树），同时确保二叉排序树的性质不会失去。
- 为保证在删除后树的查找性能与效率不至于降低，还需要防止重新链接后树的高度增加。（有的算法看似简单，但是没有保证这一点。）

（1）被删结点的右子树为空，可以拿它的左子女结点顶替它的位置，再释放它。（为什么可以？）

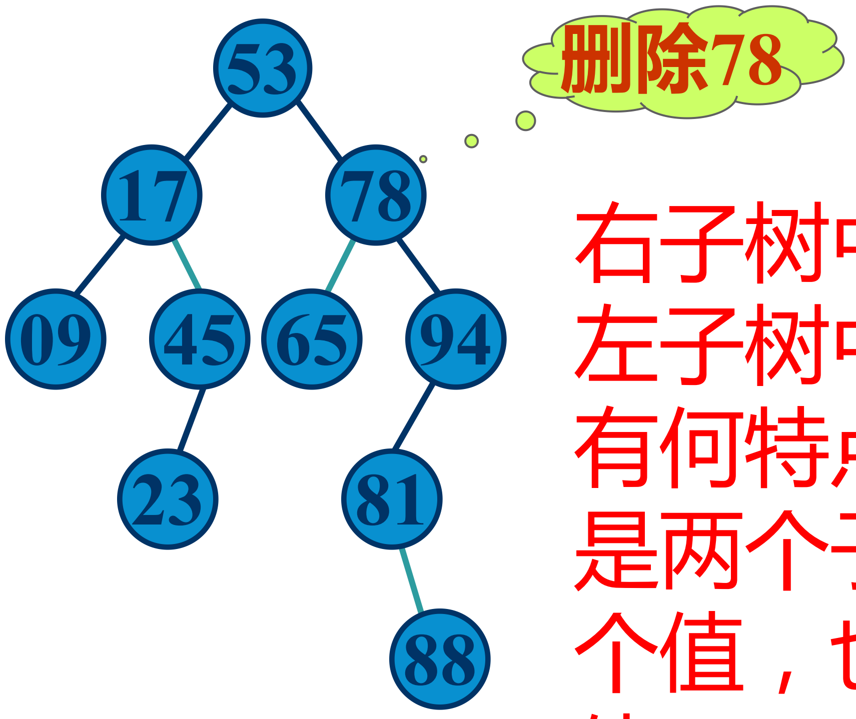


（2）被删结点的左子树为空，可以拿它的右子女结点顶替它的位置，再释放它。



（3）被删结点的左、右子树都不为空

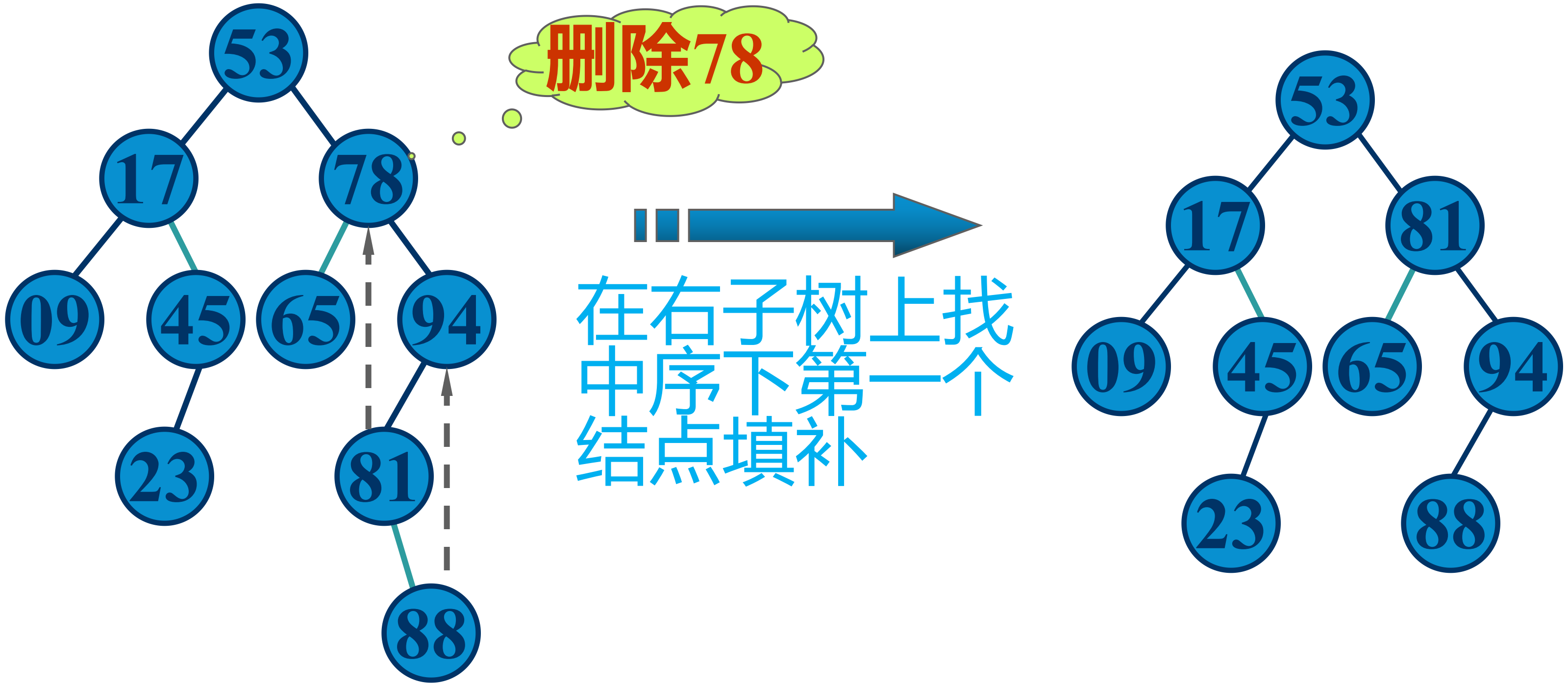
二叉排序树特点：
最小值节点在？
最大值节点在？
中序遍历第一visit
中序遍历最后visit



右子树中序遍历第一v
左子树中序遍历最后v
有何特点？
是两个子树最接近的两个值，也是最接近根的值

在其右子树中寻找中序下第一结点(所有比被删关键字大的关键字中是最小，为啥非要这样？),用它填补被删结点，再来处理这个第一v结点自身的删除。

或在其左子树中寻找中序下的最后一个结点。（看左右树高）



下午上机及作业：

最小堆的各项操作。实例用： 53 17 78 23 45 65 87 09
可将上述实例放入数组
(1)由上例建堆，打印最小堆
(2)插入80，打印最小堆
(3)删除9，打印最小堆

二叉搜索树。实例用：{ 53, 78, 65, 17, 87, 09, 81, 15 }
将上例依次输入，建BST，递归中序遍历打印之

周五进度：

图