第七章
树与二叉树的应用

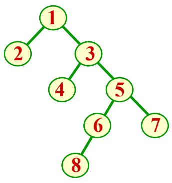
目录、搜索、调度、最优化
解决问题，前景无限

第七章 树与二叉树的应用

Huffman树

1. 路径长度 (Path Length)

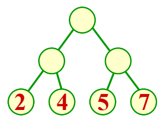
- 两个结点之间的路径长度 PL 是连接两结点的路径上的分支数。
- 右图中，结点4与结点6间的路径长度为 3。
- 树的**路径长度**是各结点到根结点的路径长度之和PL。
- 右图中，PL = 0+1+1+2+2+3+3+4 = 16。

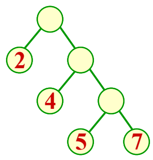


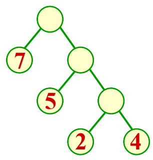
2. 带权路径长度 (Weighted Path Length, WPL)

- 二叉树的带权路径长度是各叶结点所带权值 w_i 与该结点到根的路径长度 l_i 的乘积的和。

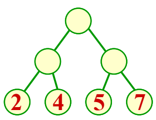
$$WPL = \sum_{i=0}^{n-1} w_i * l_i$$

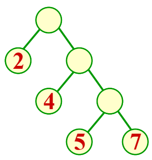


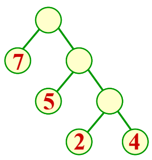




WPL = 36　　*WPL* = 46　　*WPL* = 35







WPL = 2*2+ 4*2+5*2+ 7*2 = 36	*WPL* = 2*1+ 4*2+5*3+ 7*3 = 46	*WPL* = 7*1+ 5*2+2*3+ 4*3 = 35

- 带权路径长度达到最小的二叉树即为Huffman树。
- 在Huffman树中，权值大的结点离根最近。

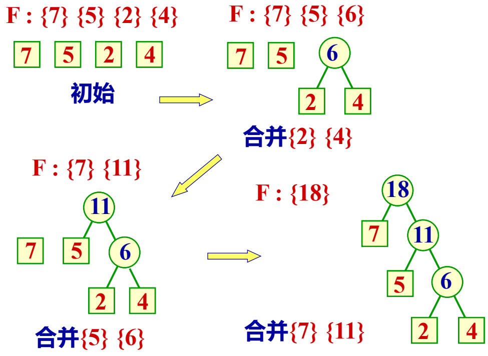
Huffman树的构造算法

- 由给定 n 个权值 $\{w_0, w_1, w_2, \ldots, w_{n-1}\}$，构造 具有 n 棵二叉树的森林 $F = \{T_0, T_1, T_2, \ldots, T_{n-1}\}$，其中每棵二叉树 T_i 只有一个带权值 w_i 的根结点, 其左、右子树均为空。
- 重复以下步骤, 直到 F 中仅剩一棵树为止:

(1) 在 F 中选取两棵根结点权值最小的二叉树， 做为左、右子树构造一棵新的二叉树。置新的二叉树的根结点的权值为其左、右子树上根结点的权值之和。

(2) 在 F 中删去这两棵二叉树。

(3) 把新的二叉树加入 F。

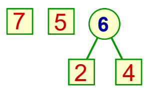


采用静态链表的Huffman树

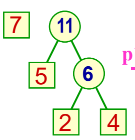
```
#define leafNumber 20            //默认权重集合大小
#define totalNumber 39           //树结点最大个数
typedef struct {
    char data;                   //结点的值
    int weight;                  //结点的权
    int parent, lchild, rchild;  //双亲、左、右子女
} HTNode;
typedef struct {
    HTNode elem[totalNumber];    //树存储数组
    int num, root;               //外结点数与根
} HFTree;
```


7 5 2 4

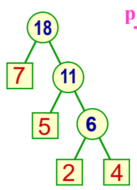
	weight	parent	lchild	rchild
0	7	-1	-1	-1
1	5	-1	-1	-1
2	2	-1	-1	-1
3	4	-1	-1	-1
4		-1	-1	-1
5		-1	-1	-1
6		-1	-1	-1



	weight	parent	lchild	rchild
0	7	-1	-1	-1
1	5	-1	-1	-1
p1→2	2	4 ~~-1~~	-1	-1
p2→3	4	4 ~~-1~~	-1	-1
i→4	6	-1	2 ~~-1~~	3 ~~-1~~
5		-1	-1	-1
6		-1	-1	-1



	weight	parent	lchild	rchild
0	7	-1	-1	-1
p1→1	5	5 ~~-1~~	-1	-1
2	2	4	-1	-1
3	4	4	-1	-1
p2→4	6	5 ~~-1~~	2	3
i→5	11	-1	1 ~~-1~~	4 ~~-1~~
6		-1	-1	-1



	weight	parent	lchild	rchild
p1→0	7	6 ~~-1~~	-1	-1
1	5	5	-1	-1
2	2	4	-1	-1
3	4	4	-1	-1
4	6	5	2	3
p2→5	11	6 ~~-1~~	1	4
i→6	18	-1	0 ~~-1~~	5 ~~-1~~

建立Huffman树的算法

```
#include "Huffman.h"
#define maxWeight 32767
void createHFTree ( HFTree& HT, char value[ ],
        int fr[ ], int n ) {
//输入数据value[n]和相应权值fr[n]，构造用三叉链
//表表示的Huffman树HT
   int i, k, s1, s2;  int min1, min2;
   for ( i = 0; i < n; i++ ) {          //外结点赋值
        HT.elem[i].data = value[i];
        HT.elem[i].weight = fr[i];
   }
```

13

```
   for ( i = 0; i < 2*n-1; i++ ) { //指针置空
     HT.elem[i].parent = -1;
     HT.elem[i].lchild =   HT.elem[i].rchild = -1;
   }
   for ( i = n; i < 2*n-1; i++ ) {          //逐步构造树
     min1 = min2 = maxWeight;
       //min1是最小权值, min2是次小权值
     s1 = s2 = 0;
       //s1是最小权值点, s2是次小权值点
     for ( k = 0; k < i; k++ )
       if ( HT.elem[k].parent == -1 )
         if ( HT.elem[k].weight < min1 ) {    //最小
```

14

```
           min2 = min1;  s2 = s1;          //原最小变次小
           min1 = HT.elem[k].weight;  s1 = k;    //新最小
         }
         else if ( HT.elem[k].weight < min2 )     //新次小
           { min2 = HT.elem[k].weight;  s2 = k; }
       HT.elem[s1].parent = HT.elem[s2].parent = i;
       HT.elem[i].lchild = s1;  HT.elem[i].rchild = s2;
       HT.elem[i].weight =
         HT.elem[s1].weight+HT.elem[s2].weight;
   }
   HT.num = n;  HT.root = 2*n-2;
}
```

15

有关Huffman树的几个要点

- Huffman树的叶结点又称为外结点，分支结点又称为内结点。外结点是结果，内结点是过程。
- Huffman树是严格二叉树。有 n 个外结点，就有 n-1 个内结点，表示需要构造 n-1 次二叉树。树中总结点数为2n-1。
- Huffman算法每次选根结点权值最小的子树来构造新二叉树时，未明确规定谁做左子树，谁做右子树，所以构造出来的Huffman树可能不惟一。
- Huffman树的最小带权路径长度是惟一的。

16

应用：Huffman编码

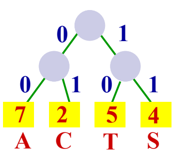
- 主要用途是实现数据压缩。设给出一段报文：

  CAST CAST SAT AT A TASA
- 字符集合是 { C, A, S, T }，各个字符出现的频度(次数) 是 W = { 2, 7, 4, 5 }。
- 若给每个字符以等长编码

  A: 00 T: 10 C: 01 S: 11

  则总编码长度为

  (2+7+4+5)*2 = 36.
- 能否减少发出的报文编码数?

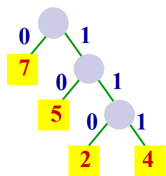


17

- 若按各个字符出现的概率不同而给予不等长的编码，可望减少总编码长度。
- 各字符出现概率为{ 2/18, 7/18, 4/18, 5/18 }，化整为 { 2, 7, 4, 5 }。以它们为各叶结点上的权值，建立Huffman树。左分支赋 0，右分支赋 1，可得Huffman编码 (变长编码)。

  A: 0 T: 10 C: 110 S: 111
- 它的总编码长度：

  7*1+5*2+(2+4)*3 = 35。
- 比等长编码的情形要短。



18

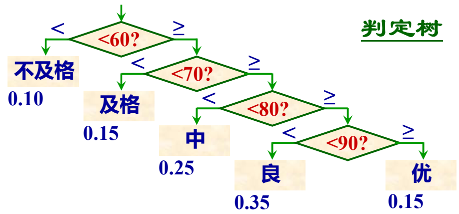
- 用Huffman编码得到的报文总编码长度正好等于Huffman树的带权路径长度WPL。
- Huffman编码是一种前缀编码，任何一个字符的编码不是其他字符编码的前缀，因此在解码时不会混淆。

应用：最佳判定树

- 利用Huffman树，可以在构造判定树（决策树）时让平均判定（比较）次数达到最小。
- 判定树是一棵扩展二叉树，外结点是比较结果，内结点是比较过程，外结点所带权值是概率。

例：考试成绩分布表

[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

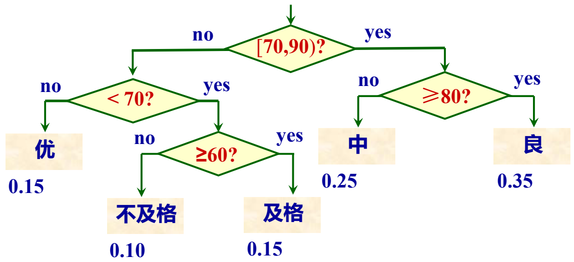
  WPL = 0.10*1+0.15*2+0.25*3+0.35*4+0.15*4

  = 3.15
- 此带权路径长度描述了在树中查找到任一外结点时的平均比较次数。此平均比较次数越少越好。
- 如果按照Huffman算法的思想构造Huffman树，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。其带权路径长度为：

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

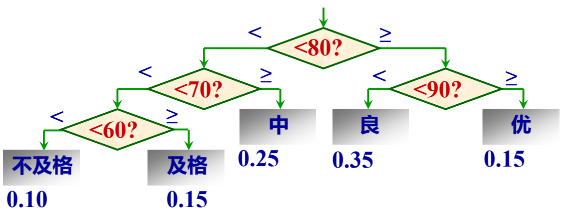
  = 0.3+0.45+0.5+0.7+0.3 = 2.25

按Huffman算法改造判定树



- 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

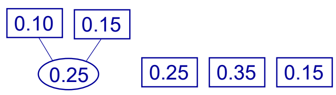
  = 0.3+0.45+0.5+0.7+0.3 = 2.25

- 其构造过程如下：

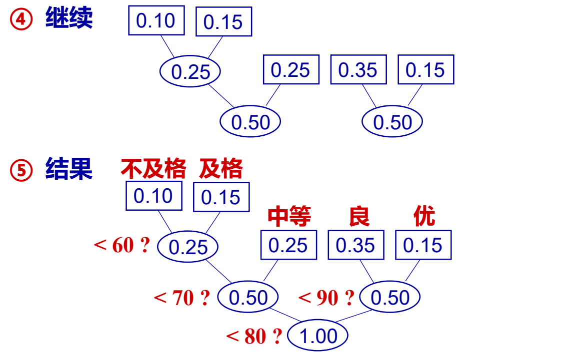
① 初始排列

0.10 0.15 0.25 0.35 0.15

② 相邻结点权值相加，取其和最小者构造二叉树。注意不要打乱原来的排列。



③ 重复上一步，直到构成一棵树为止。



堆 (Heap)

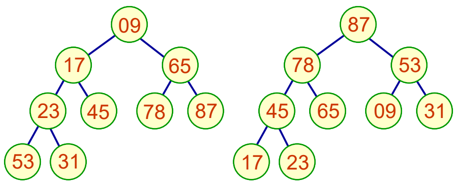
- 设有一个关键码集合，按完全二叉树的顺序存储方式存放在一个一维数组中。对它们从根开始，自顶向下，同一层自左向右从 0 开始连续编号。若满足

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

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

则称该关键码集合构成一个堆。
- 前者称为小根堆，后者称为大根堆。

堆的定义



完全二叉树
顺序表示
$K_i \le K_{2i+1}$ &&
$K_i \le K_{2i+2}$

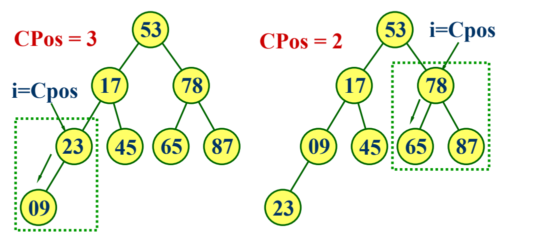
完全二叉树
顺序表示
$K_i \ge K_{2i+1}$ &&
$K_i \ge K_{2i+2}$

小根堆的定义

```
#define heapSize 40            //堆的最大元素个数
typedef int HElemType;         //堆中元素的数据类型
typedef struct {               //堆结构的定义
    HElemType elem[heapSize];  //小根堆存储数组
    int curSize;               //当前元素个数
} minHeap;
```

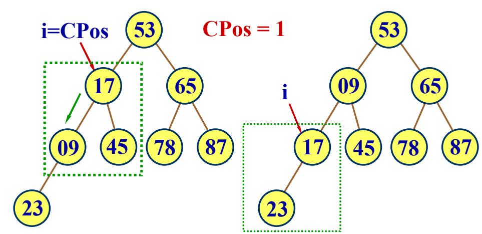
- 将存放于数组中的非小根堆序的数据调整成堆，需要从下向上，逐步进行调整。这需要使用一个称为“筛选”的函数将一棵子树调整为堆，前提是该子树的根的两棵子树都已经是堆。

- 自下向上逐步调整为小根堆的过程



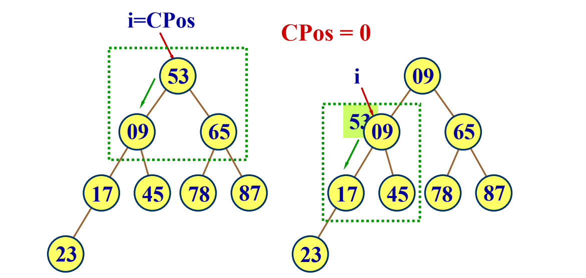
0	1	2	3	4	5	6	7
53	17	78	23	45	65	87	09

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



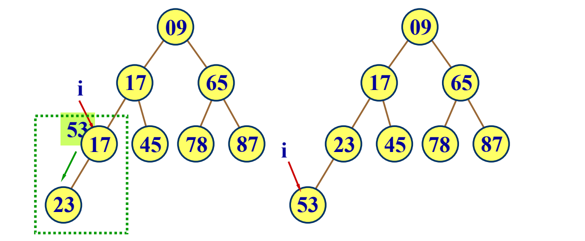
0	1	2	3	4	5	6	7
53	17	65	09	45	78	87	23

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



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

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



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

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

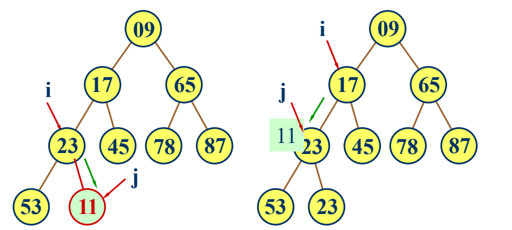
小根堆的建立

```
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
};
```

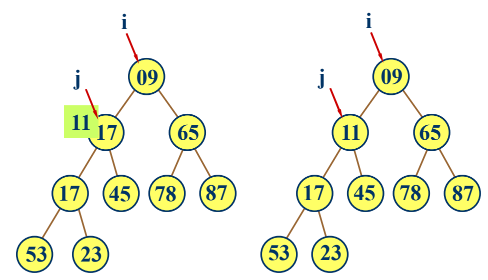

小根堆的向下筛选算法

```
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++;
      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
}
```


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



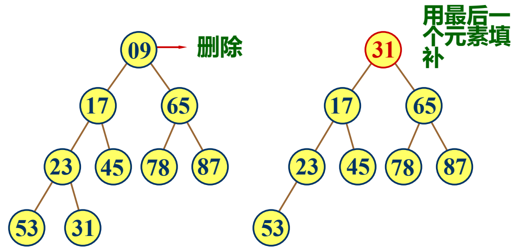
小根堆的插入算法

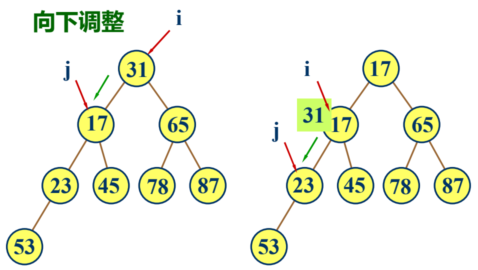
```
bool Insert ( minHeap& H, HElemType x ) {
//将x插入到小根堆中并从新调整形成新的小根堆
    if ( H.curSize == heapSize ) return false;
        //堆满，返回插入不成功信息
    H.elem[H.curSize] = x;           //插入到最后
    siftUp ( H, H.curSize );         //从下向上调整
    H.curSize++;                     //堆计数加1
    return true;
}
```

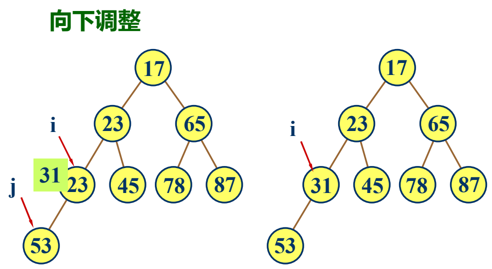

小根堆的向上筛选算法

```
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
}
```


小根堆的删除和向下调整例







小根堆的删除算法

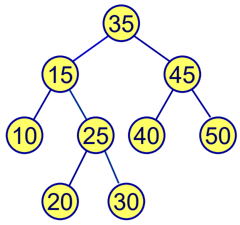
```
bool Remove ( minHeap& H, HElemType& x ) {
//从小根堆中删除堆顶元素并通过引用参数 x 返回
    if ( H.curSize == 0 ) return false;    //堆空, 返回
    x = H.elem[0];            //返回最小元素
    H.elem[0] = H.elem[H.curSize-1];
                              //最后元素填补到根结点
    H.curSize--;
    siftDown ( H, 0, H.curSize-1 );   //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

二叉查找树 (Binary Search Tree)

- 定义
  - 二叉查找树又称为二叉排序树，它或者是一棵空树，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码(key)，所有结点的关键码互不相同。
    - 左子树（如果非空）上所有结点的关键码都小于根结点的关键码。
    - 右子树（如果非空）上所有结点的关键码都大于根结点的关键码。
    - 左子树和右子树也是二叉查找树。

二叉查找树例

- 结点左子树上所有关键字小于结点关键码；
- 结点右子树上所有关键字大于结点关键码；
- 如果对一棵二叉查找树进行中序遍历，可以按从小到大的顺序将各结点关键码排列起来。



- 注意，国外教材统称为二叉搜索树或二叉查找树。

例题

- 一棵二叉树是二叉查找树的（ ）条件是树中任一结点的关键码值都大于左子女的关键码值，小于右子女的关键码值。

  A. 充分但不必要　　B. 必要但不充分

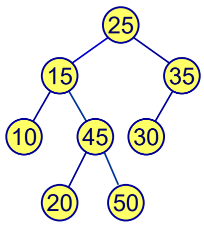
  C. 充分且必要　　D. 既不充分也不必要

【解答】 B。

通俗来讲，必要条件是指符合定义则必具有后续讲的特性。显然一棵二叉树是二叉查找树，则树中任一结点的关键码一定大于左子女的关键码，小于右子女的关键码。充分条件是指满足后续讲

的特性则定义一定成立。就是说，如果一棵二叉树任一结点的关键码都大于左子女的关键码，小于右子女的关键码，则它一定是二叉查找树。这就不一定了。

右图描述的二叉树满足树中任一结点的关键码一定大于左子女的关键码，小于右子女的关键码，但它不是二叉查找树。



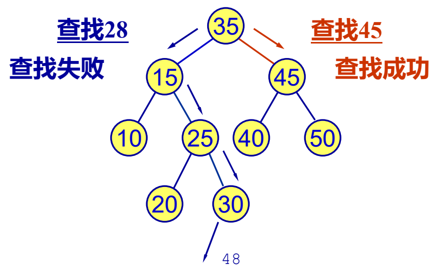
二叉查找树的结构定义

```
typedef int TElemType;     //结点关键码数据类型
typedef struct tnode {
    TElemType data;             //结点值
    struct tnode *lchild, *rchild;  //左、右子女指针
} BSTNode, *BSTree;
```

- 从面向对象观点来看，二叉查找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。

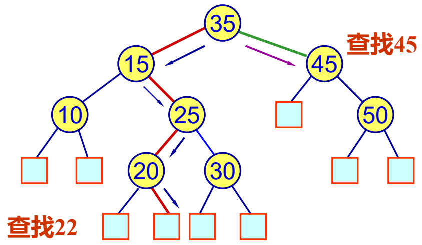
二叉查找树上的查找

- 在二叉查找树上进行查找，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。它可以是一个递归的过程。



- 假设想要在二叉查找树中查找关键码为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 与根结点的关键码进行比较：
  - ➢ 如果给定值等于根结点的关键码值，则查找成功。
  - ➢ 如果给定值小于根结点的关键码值，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。

- 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。



二叉查找树的查找算法

```
BSTNode *Search_iter ( BSTree BT, TElemType x,
  BSTNode *& pr ) {
//在二叉查找树 BT 中查找关键码等于 x 的结点，成
//功时函数返回找到结点地址，pr 是其双亲结点. 不
//成功时函数返回空，pr 返回最后走到的结点地址.
  BSTNode *p = BT;  pr = NULL;
  while ( p != NULL && p->data != x ) {
    pr = p;                    //向下层继续查找
    if ( x < p->data ) p = p->lchild;
    else p = p->rchild;
  }
```



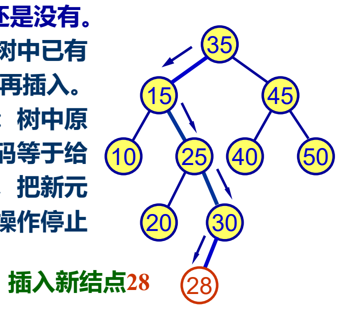
```
  return p;
}
```

- 查找的关键码比较次数最多不超过树的高度。

二叉查找树的插入

- 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。
- 为了向二叉查找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。

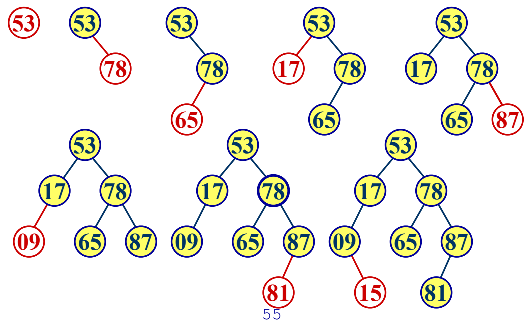
- 为此，在插入之前先使用查找算法在树中检查要插入元素有还是没有。
  - ➢ 查找成功：树中已有这个元素,不再插入。
  - ➢ 查找不成功：树中原来没有关键码等于给定值的结点，把新元素加到查找操作停止的地方。




```
bool Insert ( BSTree& t, TElemType x ) {
//向根为*t 的二叉查找树插入一个关键码为x的结点
  BSTNode *s, *p, *f;
  p = Search_iter ( t, x, f );          //寻找插入位置
  if ( p != NULL ) return false;   //查找成功，不插入
  s = ( BSTNode * ) malloc ( sizeof ( BSTNode ) );
  s->data = x;  s->lchild = s->rchild = NULL;
  if ( t == NULL ) t = s;  //原为空树，结点为根结点
  else if ( x < f->data ) f->lchild = s; //按左子女插入
  else f->rchild = s;                  //按右子女插入
  return true;
}
```

输入数据，建立二叉查找树的过程

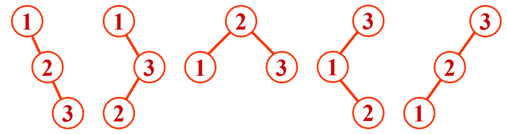
- 输入数据 { 53, 78, 65, 17, 87, 09, 81, 15 }



- 同样 3 个数据{ 1, 2, 3 }，输入顺序不同，建立起来的二叉查找树的形态也不同。这直接影响到二叉查找树的查找性能。
- 如果输入序列选得不好，会建立起一棵单支树，使得二叉查找树的高度达到最大。

{2, 1, 3}

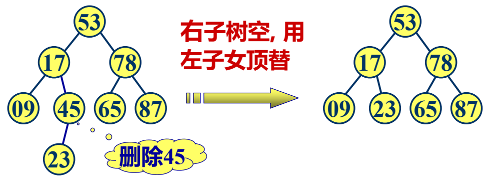
{1, 2, 3} {1, 3, 2} {2, 3, 1} {3, 1, 2} {3, 2, 1}

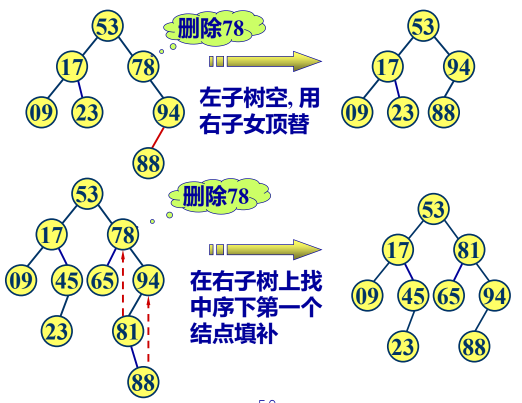


二叉查找树的删除

- 在二叉查找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，同时确保二叉查找树的性质不会失去。
- 为保证在删除后树的查找性能不至于降低，还需要防止重新链接后树的高度增加。
  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。
  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，可以在它

的右子树中寻找中序下的第一个结点(所有比被删关键码大的关键码中它的值最小),用它的值填补到被删结点中，再来处理这个结点的删除问题。当然，也可以到它的左子树中寻找中序下的最后一个结点。






```
bool Remove ( BSTree& t, TElemType x ) {
//在*t 为根的二叉查找树中删除关键码为x的结点
   BSTNode *s, *p, *f;
   if ( ( p = Search_iter ( t, x, f ) ) == NULL )
      return false;              //查找失败不删除
   if ( p->lchild != NULL && p->rchild != NULL ) {
               //被删结点*p有两个子女
      s = p->rchild;  f = p;          //找 p 的中序后继s
      while ( s->lchild != NULL )
         { f = s;  s = s->lchild; }
      p->data = s->data;  p = s;   //将*s的数据传给*p
   }
```

```
    if ( p->lchild != NULL ) s = p->lchild;
                    //单子女，记录非空子女
    else s = p->rchild;
    if ( p == t ) t = s;           //被删结点为根结点
    else if ( s != NULL && s->data < f->data )
        f->lchild = s;             //否则链接子女
    else f->rchild = s;
    free ( p );                    //释放被删结点
    return true;                   //删除成功
}
```

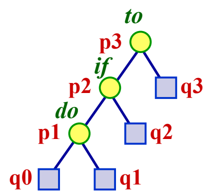

二叉查找树性能分析

- 对于有 *n* 个关键码的集合，其关键码有 *n*! 种不同排列，可构成不同二叉查找树有

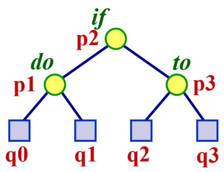
$$\frac{1}{n+1}C_{2n}^{n} \quad (棵)$$

- 用树的查找效率来评价这些二叉查找树。
- 用**判定树**来描述查找过程，在判定树中，○表示**内结点**，它包含关键码集合中的某一个关键码；□表示**外结点**，代表各关键码间隔中的不在关键码集合中的关键码。它们是查找失败时到达的结点，物理上实际不存在。

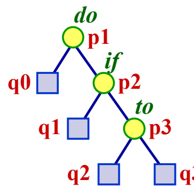
- 在每两个外部结点间必存在一个内部结点。
- 例，已知关键码集合 { a1, a2, a3 } = { do, if, to }，对应查找概率 p1, p2, p3，在各查找不成功间隔内查找概率分别为 q0, q1, q2, q3。可能的判定树如下所示。



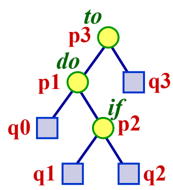
(a)



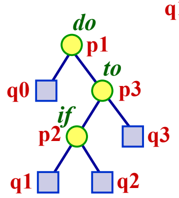
(b)



(c)



(d)



(e)

- 一棵判定树的**查找成功的平均查找长度**ASL_{succ}定义为该树所有内结点上的权值 p[i]与查找该结点时所需的关键码比较次数c[i] (= l[i])乘积之和:

$$ASL_{succ} = \sum_{i=1}^{n} p[i] * l[i].$$

- **查找不成功的平均查找长度**ASL_{unsucc}为树中所有外结点上权值q[j]与到达该外结点所需关键码比较次数c'[j] (= l'[j]−1)乘积之和:

$$ASL_{unsucc} = \sum_{j=0}^{n} q[j] * (l'[j] - 1).$$

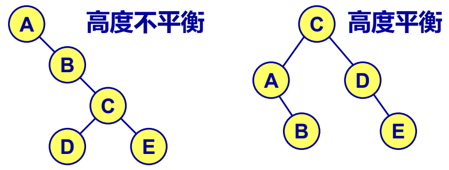
相等查找概率的情形

- 设树中所有内、外结点的查找概率都相等：
  - $p[i] = 1/3,\ 1 \leq i \leq 3$
  - $q[j] = 1/4,\ 0 \leq j \leq 3$
- 图(a): $ASL_{succ} = 1/3*(3+2+1) = 6/3 = 2$
  $ASL_{unsucc} = 1/4*(3+3+2+1) = 9/4$
- 图(b): $ASL_{succ} = 1/3*(2+1+2) = 5/3$
  $ASL_{unsucc} = 1/4*(2+2+2+2) = 8/4$
- 图(c): $ASL_{succ} = 2,\ \ ASL_{unsucc} = 9/4$
- 图(d): $ASL_{succ} = 2,\ \ ASL_{unsucc} = 9/4$
- 图(e): $ASL_{succ} = 2,\ \ ASL_{unsucc} = 9/4$

- 图(b)的情形所得的平均查找长度最小。一般把平均查找长度达到最小的判定树称作最优二叉查找树。
- 在相等查找概率的情形下，最优二叉查找树的上面 h-1（h是高度）层都是满的，只有第 h 层不满。如果结点集中在该层的左边，则它是完全二叉树；如果结点散落在该层各个地方，则有人称之为理想平衡树。

AVL树

- AVL树的定义
  - 一棵AVL树又称为高度平衡的二叉查找树，它或者是空树，或者是具有下列性质的二叉查找树：它的左子树和右子树都是AVL树，且左子树和右子树的高度之差的绝对值不超过1。



结点的平衡因子 balance factor

- 每个结点附加一个数字，给出该结点左子树的高度减去右子树的高度所得的高度差，这个数字即为结点的平衡因子 bf（balance factor）。
- AVL树任一结点平衡因子只能取 -1, 0, 1。
- 如果一个结点的平衡因子的绝对值大于 1，则这棵二叉查找树就失去了平衡，不再是AVL树。
- 如果一棵二叉查找树是高度平衡的, 且有 n 个结点，其高度可保持在$O(\log_2 n)$,平均查找长度也可保持在$O(\log_2 n)$。

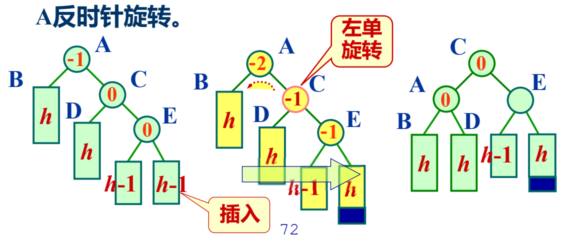
平衡化旋转

- 如果在一棵AVL树中插入一个新结点，造成了不平衡。必须调整树的结构，使之平衡化。
- 平衡化旋转有两类：
  - 单旋转 (LL旋转和RR旋转)
  - 双旋转 (LR旋转和RL旋转)
- 每插入一个新结点时，AVL树中相关结点的平衡状态会发生改变。因此，在插入一个新结点后，需要从插入位置沿通向根的路径回溯，检查各结点的平衡因子。

- 如果在某一结点发现高度不平衡，停止回溯。从发生不平衡的结点起，沿刚才回溯的路径取直接下两层的结点。
- 如果这三个结点处于一条直线上，则采用单旋转进行平衡化。单旋转可按其方向分为LL旋转和RR旋转，其中一个是另一 个的镜像，其方向与不平衡的形状相关。
- 如果这三个结点处于一条折线上，则采用双旋转进行平衡化。双旋转分为LR旋转和RL旋转两类。

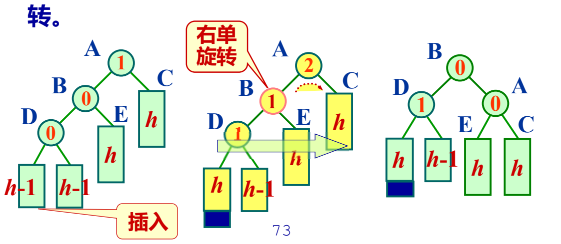
左单旋转（RR单旋）

- 在结点A的右子女C的右子树E中插入新结点，该子树高度增1导致结点A的平衡因子变成-2，出现不平衡。为使树恢复平衡，从A沿插入路径连续取3个结点A、C和E，以结点C为旋转轴，让结点A反时针旋转。



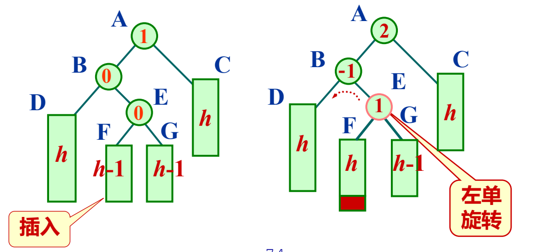
右单旋转（LL单旋）

- 在结点A的左子女的左子树D上插入新结点使其高度增1导致结点A的平衡因子增到+2，造成不平衡。为使树恢复平衡，从A沿插入路径连续取3 个结点A、B和D，以结点B为旋转轴，将结点A顺时针旋转。

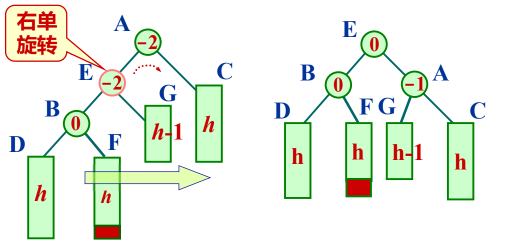


先左后右双旋转（LR双旋）

- 在结点A的左子女的右子树中插入新结点，该子树的高度增1导致结点A的平衡因子变为 2，发生了不平衡。

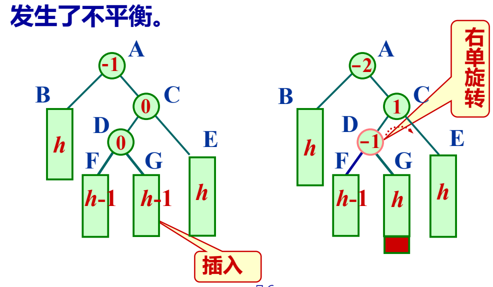


- 从结点A起沿插入路径选取3个结点A、B和E，先以结点E为旋转轴，将结点B反时针旋转，E顶替原B的位置。再以结点E为旋转轴，将结点A顺时针旋转。

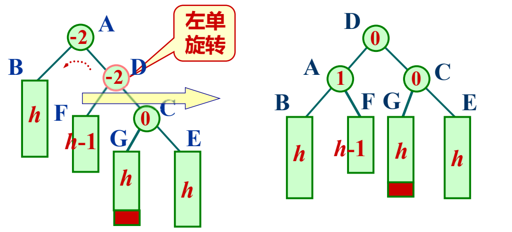


先右后左双旋转（RL双旋）

- 在根结点A的右子女的左子树中F或G上插入新结点，该子树高度增1。结点A的平衡因子变为-2，发生了不平衡。



- 从结点A起沿插入路径选取3个结点A、C和D。首以结点D为旋转轴，将结点C顺时针旋转，以D代替原来C的位置。再以D为旋转轴，将结点A反时针旋转，恢复树的平衡。



AVL树的插入

- 在向一棵本来是高度平衡的AVL树中插入一个新结点时，如果树中某个结点的平衡因子的绝对值 |bf|＞1，则出现了不平衡，需要做平衡化处理。
- 算法从一棵空树开始，通过输入一系列元素关键码，逐步建立AVL树。
- 在插入新结点后，需从插入结点沿通向根的路径向上回溯，如果发现有不平衡的结点，需从这个结点出发，使用平衡旋转方法进行平衡化处理。

- 设新结点p的平衡因子为0，其父结点为pr。插入新结点后pr的平衡因子值有三种情况：

1. 结点pr的平衡因子为0。说明刚才是在 pr 的较矮的子树上插入了新结点，此时不需做平衡化处理，返回主程序。子树的高度不变。

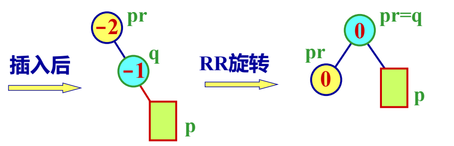
2. 结点pr的平衡因子的绝对值 |bf| = 1。说明插入前pr的平衡因子是0，插入新结点后，以pr为根的子树不需平衡化旋转。但该子树高度增加，

还需从结点pr向根方向回溯，继续考查结点pr双亲(pr = Parent(pr))的平衡状态。

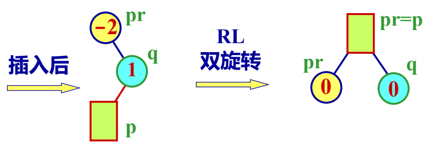
3. 结点pr的平衡因子的绝对值|bf| = 2。说明新结点在较高的子树上插入，造成了不平衡，需要做平衡化旋转。此时可进一步分2种情况讨论：

① 若结点pr的bf = -2，说明右子树高，结合其右子女q 的bf分别处理：

➢若q的bf与pr同符号(=-1)，执行RR单旋。



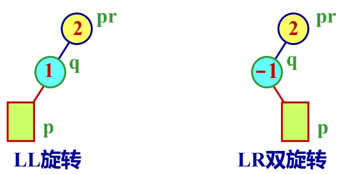
➢若q的bf与pr异符号(=1)，执行RL双旋。



② 若结点pr的bf = 2，说明左子树高，结合其左子女q 的bf分别处理：

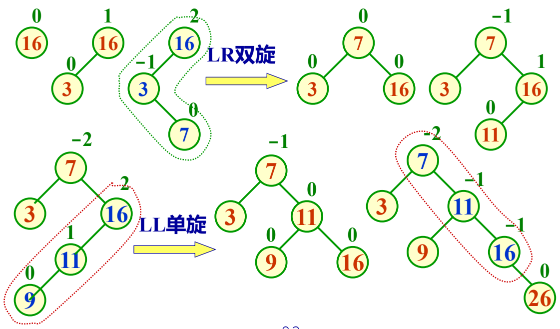
➢ 若q的bf与pr同符号(=1)，执行LL单旋；

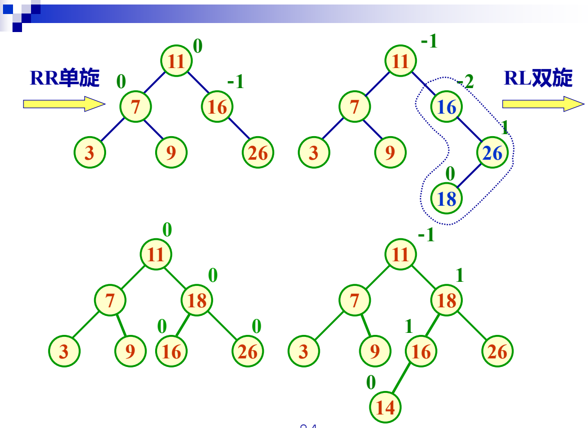
➢ 若q的bf与pr异符号(=-1)，执行LR双旋。

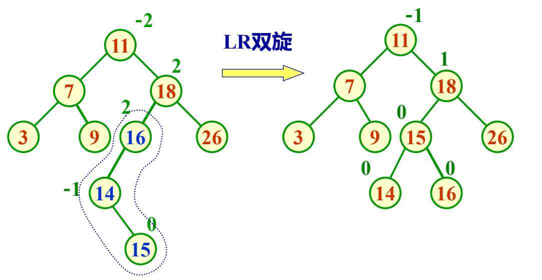


- 下面举例说明在AVL树上的插入过程。

- 例，输入关键码序列为 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }，插入和调整过程如下。







从空树开始的建树过程

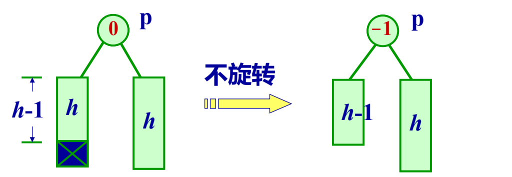
AVL树的删除

1. 如果被删结点x最多只有一个子女，那么问题比较简单:
   - 将结点x从树中删去。
   - 因为结点x最多有一个子女，可以简单地把x的双亲结点中原来指向结点x的指针改指到这个子女结点;
   - 如果结点x没有子女，x双亲结点的相应指针置为NULL。
   - 将原来以结点x为根的子树的高度减1。

2. 如果被删结点x有两个子女:
   - 查找结点x在中序次序下的直接前驱结点y (同样可以找直接后继)。
   - 把结点y的内容传送给结点x，现在问题转移到删除结点 y。把结点y当作被删结点x。
   - 因为结点y最多有一个子女，我们可以简单地用1.给出的方法进行删除。

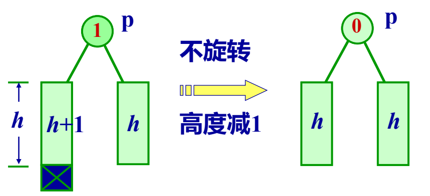
- 在执行结点x的删除之后，需要从结点x开始，沿通向根的路径反向追踪高度的变化对路径上各个结点的影响。

(1) 当前结点 p 的bf为0。如果它的左子树或右子树的高度被降低，则它的 bf 改为-1 或1。因该结点的高度未变，不必向上回溯，删除完成。



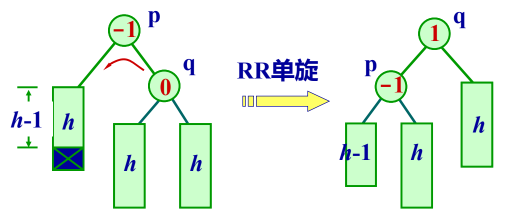
(2) 当前结点 p 的bf不为0，且较高子树的高度被降低，则 p 的bf 改为0。因该结点的高度降低，需

向上回溯，看其双亲是否失去平衡。

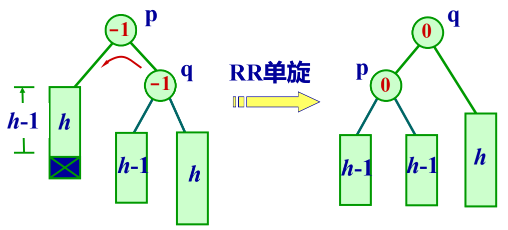


(3) 当前结点 p 的bf 不为0，且较矮子树的高度又被降低，则在结点 p 发生不平衡。需要进行平衡化旋转来恢复平衡。

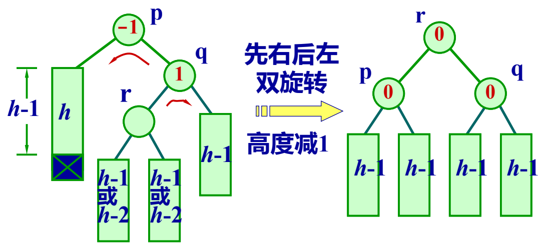
- 令 p 的较高的子树的根为 q (该子树高度未被降低)，根据 q 的 bf，有如下 3 种平衡化操作。
  ① 如果 q (较高的子树) 的 bf为 0，执行一个单旋转来恢复结点 p 的平衡，由于旋转后子树高度未降低，无需向上回溯。



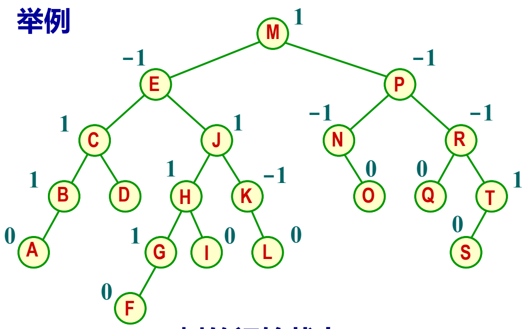
② 如果 q 的bf与 p 的bf正负号相同，则执行一个单旋转来恢复平衡, 结点 p 和 q 的bf均改为0，由于子树高度降低，需向上判断双亲结点是否失去平衡。



③ 如果 p 与 q 的bf的符号相反, 则执行一个双旋转来恢复平衡, 先围绕 q 转再围绕 p 转。新根结点的bf置为0，其它结点的bf相应处理，由于该子树高度降低，需向上回溯。

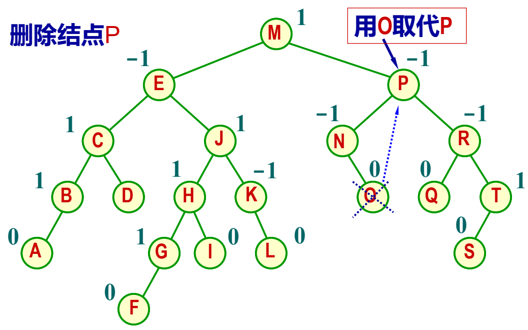


举例



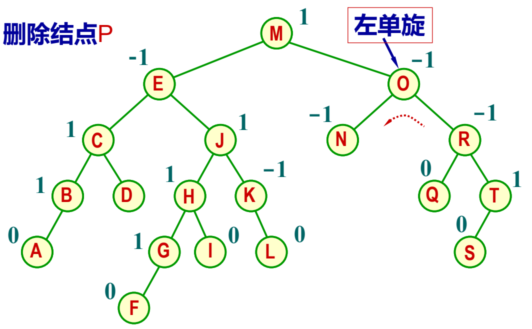
树的初始状态

删除结点P



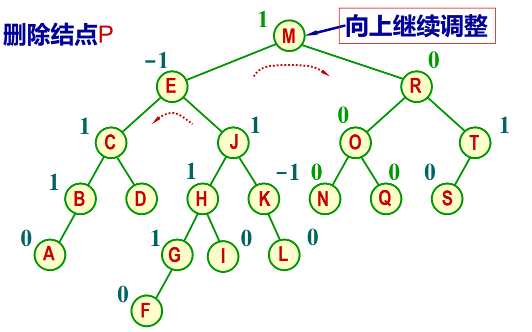
- 寻找结点P在中序下的直接前驱O, 用O顶替P, 删除O。

删除结点P



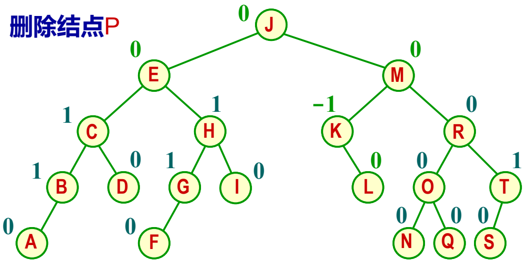
- O的较矮子树高度减1，O与R的平衡因子同号, 以R为旋转轴做左单旋

删除结点P



- M的右子树(较矮)高度减 1, M发生不平衡。M与 E的平衡因子反号, 做LR双旋转。

删除结点P



AVL树的高度

- 设在新结点插入前AVL树的高度为h，结点个数为n，则插入一个新结点，其时间代价是$O(h)$。对于AVL树来说，h多大?
- 设 N_h 是高度为 h 的AVL树的最小结点个数。根的一棵子树的高度为$h-1$，另一棵子树的高度为$h-2$，这两棵子树也是高度平衡的。因此有
  - $N_0 = 0$ (空树)
  - $N_1 = 1$ (仅有根结点)
  - $N_h = N_{h-1} + N_{h-2} + 1\ ,\ h > 1$

- 因此，有 $N_0 = 0$，$N_1 = 1$，$N_2 = N_1+N_0+1 = 2$，$N_3 = N_2+N_1+1 = 4$，$N_4 = N_3+N_2+1 = 7$，…
- 由于斐波那契数列 $F_0 = 0$，$F_1 = 1$，$F_2 = F_1+F_0 = 1$，$F_3 = F_2+F_1 = 2$，$F_4 = F_3+F_2 = 3$，$F_5 = F_4+F_3 = 5$，$F_6 = F_5+F_4 = 8$，$F_7 = F_6+F_5 = 13$，…
- 可得 $N_h = F_{h+2} - 1$。
- 如果高度 h 固定，最少结点数为N_h；最多结点数为$2^h - 1$，即满二叉树情形。
- 反之，若结点数 n 固定，最小高度$\lceil \log_2(n+1) \rceil$。最大高度不超过$1.44*\log_2(n+2)$

例题1　具有 5 层结点的AVL树至少有_(1)_个结点。若设树根结点在第 1 层，则深度最小的叶结点应在第_(2)_层。

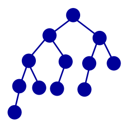
(1) A. 10　　B. 12　　C. 15　　D. 17

(2) A. 1　　B. 2　　C. 3　　D. 4

解答：(1) B (2) C

高度为 5 的AVL树的最少结点数

$N_5 = F_7 - 1 = 13 - 1 = 12$



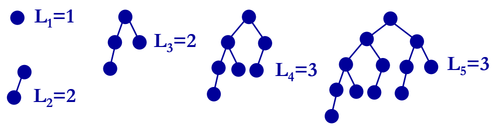
计算AVL树深度最小的叶结点所在层次的方法与推导 N_h 的方法类似：

若设高度为 h 的AVL树的深度最小的叶结点所在层次为L_h，则有

$L_1 = 1$，$L_2 = 2$，$L_h = L_{h-2}+1$，$h>2$

$\therefore L_3 = L_1+1 = 2$，$L_4 = L_2+1 = 3$，$L_5 = L_3+1 = 3$。

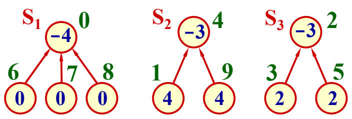
直接计算 $L_h = \lfloor h/2 \rfloor + 1$.



并查集 (Union-Find Sets)

- 并查集支持以下三种操作：
  - Union (S, Root1, Root2)　//合并操作
  - Find (S, x)　//查找操作
  - initUFSets (S)　//初始化函数
- 对于并查集来说，每个集合用一棵树表示。
- 为此，采用树的双亲表示作为集合存储表示。集合元素的编号从0到 $n-1$。其中 n 是最大元素个数。

- 在双亲表示中，第 i 个数组元素代表集合元素 i。初始时，根结点的双亲为-1，表示集合中的元素个数。
- 在同一棵树上所有结点所代表的集合元素在同一个子集合中。
- 设 S_1= {0, 6, 7, 8}, S_2= {1, 4, 9}, S_3= {2, 3, 5}



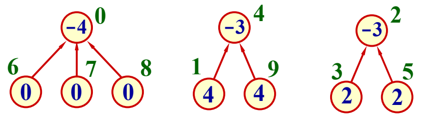
- 初始时, 用初始化函数 initUFSets(S) 构造一个森林, 每棵树只有一个结点, 表示集合中各元素自成一个子集合。

下标	0	1	2	3	4	5	6	7	8	9
parent	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

- 用 Find(S, i) 寻找集合元素 i 的根。如果有两个集合元素 i 和 j, Find(S, i) == Find(S, j), 表明这两个元素在同一个集合中，
- 如果两个集合元素 i 和 j 不在同一个集合中,可用 Union() 将它们合并到一个集合中。

下标	0	1	2	3	4	5	6	7	8	9
parent	-4	4	-3	2	-3	2	0	0	0	4

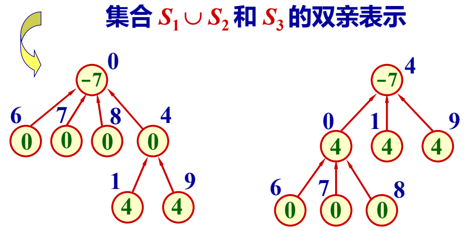
集合 S_1, S_2 和 S_3 的双亲表示



S_1 S_2 S_3

下标	0	1	2	3	4	5	6	7	8	9
parent	-7	4	-3	2	0	2	0	0	0	4

集合 $S_1 \cup S_2$ 和 S_3 的双亲表示



$S_1 \cup S_2$的可能的表示方法

用双亲表示实现并查集的结构定义

```
#define size 100
typedef struct {                    //并查集类型定义
    int parent[size];               //双亲指针数组
} UFSets;

void Initial ( UFSets& S ) {        //初始化
    for ( int i = 0; i < size; i++ )
        S.parent[i] = −1;           //每个自成单元素集合
}
```



```
int Find ( UFSets& S, int x ) {
//函数从 x 开始，沿双亲链搜索到树的根
    while (S.parent[x] >= 0)
        x = S.parent[x];        //根的parent[]值小于0
    return x;
}
```

- 合并操作Merge(UFSets& S, int R1, int R2) 要求把两个不相交集合R1与R2合并为一个集合。合并的原则是：① 把元素少的集合合并入元素多的集合；② R1和R2是两个不相交集合的根。

```
int Merge(UFSets& S, int R1, int R2) {
   int x = S.parent[R1] + S.parent[R2];  //元素总个数
   if ( S.parent[R1] <= S.parent[R2] ) {  //R1的元素多
       S.parent[R1] = x;          //R1成为合并后的根
       S.parent[R2] = R1;  return R1;
   }
   else {                 //否则R2的元素多
       S.parent[R2] = x;          //R2成为合并后的根
       S.parent[R1] = R2;  return R2;
   }
};
```

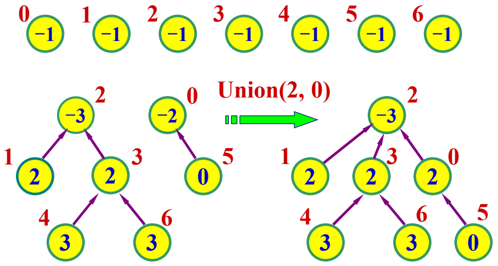

- 执行一次合并操作Merge所需时间是 $O(1)$，$n-1$ 次Merge操作所需时间是$O(n)$。
- 若再执行Find(0)，Find(1)，…，Find($n-1$)，若被搜索的元素为 i，完成 Find(i) 操作需要时间为O(i)，完成 n 次搜索需要的总时间将达到

$$O(\sum_{i=1}^{n} i) = O(n^2)$$

- 改进的方法
  - ✧ 按树的结点个数合并（已经给出）
  - ✧ 按树的高度合并
  - ✧ 压缩元素的路径长度

按高度合并

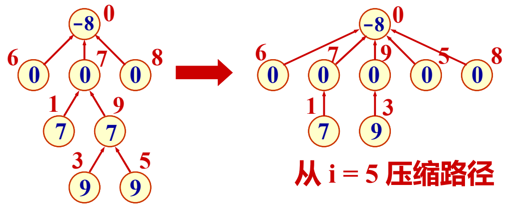
- 若把根结点parent域存放的值视为该树的高度，并查集合并操作把高度高的作为合并后的根。




```
int Merge ( UFSets& S, int R1, int R2 ) {
//先求高度，再把高度矮的接到高度高的树下面
   int x = S.parent[R1], y = S.parent[R2];
   if ( x <= y ) {               //R1高，成为合并后的根
       if ( x == y ) S.parent[R1]--;
           //高度相等，根高度加一，负数表示应减1
       S.parent[R2] = R1;  return R1;
   }
   else {                        //R2高，成为合并后的根
       S.parent[R1] = R2;  return R2;
   }
};
```


压缩元素的查找路径

- 为改进树的查找性能，可以使用如下折叠规则来"压缩路径"。即把从元素 i 到根的路径上的所有祖先结点的双亲指针都改为指向根结点。
- 结点内的数字是该结点的双亲指针。



从 i = 5 压缩路径

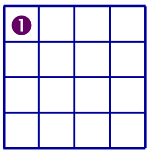

```
int CollapsingFind ( UFSets& S, int i ) {
//使用折叠规则压缩路径以提高查找效率
    for ( int j = i; parent[j] >= 0; j = parent[j] );
                    //让 j 循双亲指针走到根
    while ( parent[i] != j ) {
        int temp = parent[i];   //暂存双亲结点地址
        parent[i] = j;          //让 i 的双亲指针指向根
        i = temp;               //让 i 指向原来 i 的双亲
    }
    return j;                   //返回根
}
```

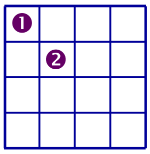
- 并查集在求解等价类问题时特别有用，虽然它很简单。下一章讨论图时，用它来判断连通分量。

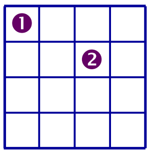
八皇后问题与树的剪枝

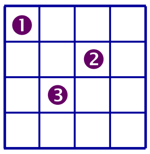
- 八皇后问题的提法是：“现有八个皇后，要放到8×8的国际象棋的棋盘上，使得它们彼此不受威胁，即没有两个皇后位于同一行、同一列或同一对角线上。问有多少种放置方法？”
- 为简化问题，我们缩小问题规模，把“八皇后问题”的规模缩小为“四皇后问题”。即在4×4的棋盘上放 4 个皇后，使得没有两个皇后在同一行或同一列或同一对角线上。
- 解题思路是采用递归和回溯的方法。

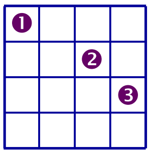
- 安放第 k 行皇后时，需要在列的方向从 1 到 4 试探 (i = 1, 2, 3, 4)
- 在第 i 列安放一个皇后，然后判断：
  - 如果在列、斜线（/）、反斜线（\）方向有其它皇后，则出现攻击，撤消第 i 列安放皇后：
    - 若 $i = 4$，回溯到上一行，继续试探；
    - 若 $i < 4$，令 i+1 继续试探下一列。
  - 如果在第 i 列没有出现攻击，在第 i 列安放的皇后不动，递归安放第 k+1行皇后。
- 如果 $k = 4$，且第 4 行已安放皇后，找到了一个布局，输出这个布局，再回溯寻找其他布局。

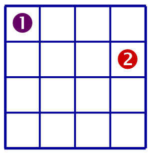

[1][1]安放

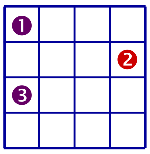

[2][1], [2][2]攻击

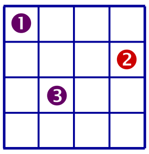

[2][3]安放

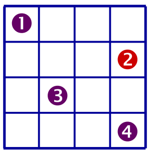

[3][1], [3][2]攻击

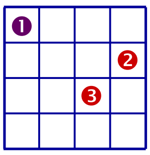

[3][3], [3][4]攻击

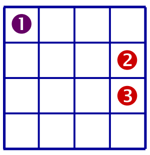

回退[2][4], 安放

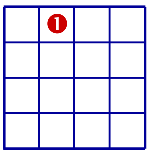

[3][1]攻击

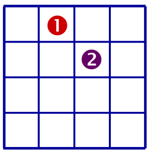

[3][2]安放

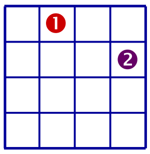

[4][1]～[4][4]攻击

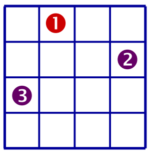

回退[3][3], 攻击

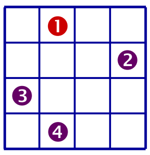

[3][4]攻击

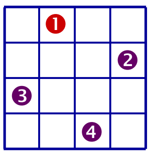

回退[2][4]. 回退[1][2]

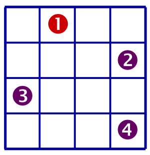

[2][1], [2][2], [2][3]攻击

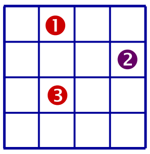

[2][4]安放

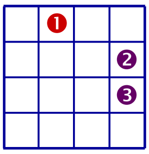

[3][1]安放

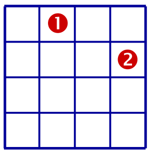

[4][1], [4][2]攻击

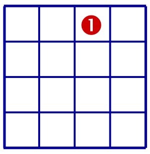

[4][3]安放, 布局

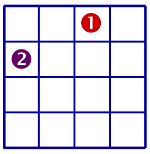

继续, [4][4]攻击

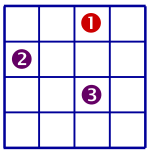

回退[3][2], 攻击

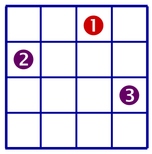

[3][3], [3][4]攻击

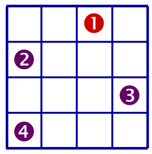

回退[2][4]

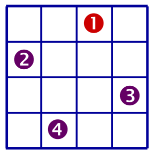

回退[1][3]

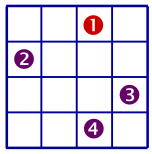

[2][1]安放

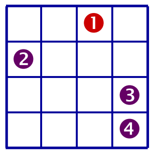

[3][1]～[3][3]攻击

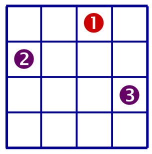

[3][4]安放

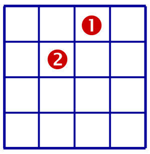

[4][1]攻击

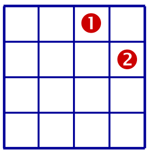

[4][2]安放, 布局

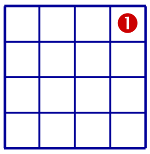

继续, [4][3]攻击

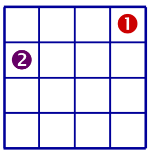

[4][4]攻击

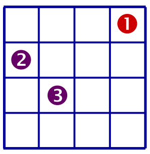

回退[3][4]

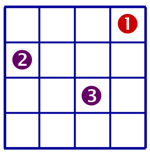

回退[2][2], 攻击

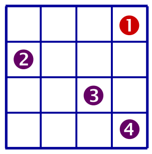

[2][3], [2][4]攻击

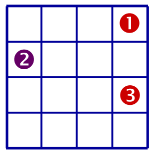

回退[1][4]

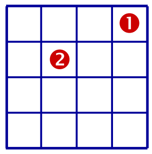

[2][1]安放

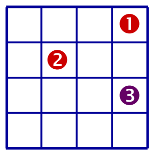

[3][1], [3][2]攻击

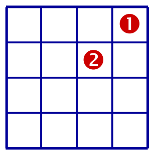

[3][3]安放

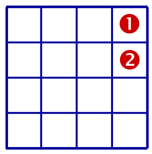

[4][1], [4][4]攻击


回退[3][4], 攻击


回退[2][2]


[3][1]～[3][4]攻击

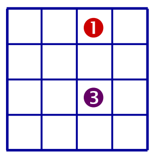

回退[2][3], 攻击

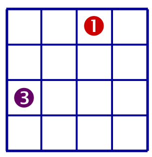

[2][4]攻击, 回退

判断攻击的方法

- 为判断列方向是否发生攻击，设置一个chess[4]数组，chess[*j*]记忆第 *j* 行皇后安放在第几列。
- 设第 1 行和第 2 行已经安放皇后，在第 3 行安放皇后时，先让chess[3] = *i* (*i* = 1, 2, 3, 4)，即假设第 3 行皇后安放在第 *i* 列，然后判断。
  - chess[*j*] = *i* ？ (*j* = 1, 2)。若 = ，则第 *i* 列已有皇后，发生攻击，否则没有攻击。
  - 例如，若chess[1] = 3，安放第 3 行皇后时，如果用第 3 列试探，有 chess[1] = *i*，则第 *i* 列已有皇后，发生攻击。

- 若 chess[3] = 1，则❶、❸在同一斜线上，chess[1] − chess[3] = −2，在横向相差 2 格，若纵向也差 2 格，即1 − 3 = −2，则在斜线方向发生攻击，判断式是*j* − *k* = chess[*j*] − chess[*k*]。
- 在反斜线方向判断攻击的式子为*k* − *j* = chess[*j*] − chess[*k*]。
- 其中 *k* 是要安放皇后的行，*j* 是前面已安放皇后的行，该行皇后记在chess[*j*]中。


列方向发生攻击


斜线方向发生攻击

求解八皇后问题的递归算法

```
#define n 8                    //皇后个数
void Queen ( int chess[ ], int k, int& sum ) {
//用递归法求解n皇后问题，chess是棋盘，k是当前
//行，sum返回求得布局数
   int i, j, u;
   if ( k <= n ) {             //递归安放皇后
      for ( i = 1; i <= n; i++ ) {   //以各列试探
         chess[k] = i;  u = 0;       //安放皇后
         for ( j = 1; j < k; j++ )   //与前k-1行皇后比对
            if ( chess[k] == chess[j] ||
                abs ( k-j ) == abs ( chess[k]-chess[j] ) )
                u = 1;                //互相攻击
```

127

```
         if ( ! u ) {
            if ( k == n ) {     //行号达到n，完成一个布局
               sum++;           //布局数加一
               for ( j = 1; j <= n; j++ )
                  printf ( "(%d, %d) ", j, chess[j] );
               printf ( "\n" );        //输出一个布局
            }
            else Queen ( chess, k+1, sum );  //递归求解
         }
      }
   }
}
```

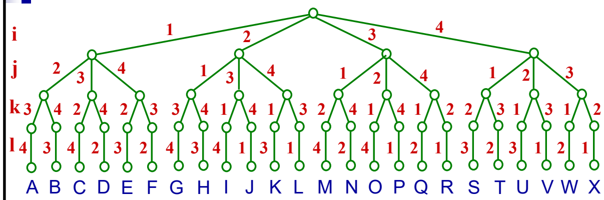
128

```
void main ( void ) {
   int layout[n+1];  int i, sum = 0;
   for ( i = 1; i <= n; i++ ) layout[i] = 0;
   Queen( layout, 1, sum );
   printf ( "%d皇后解的数目有%d\n", n, sum );
}
```

树的剪枝

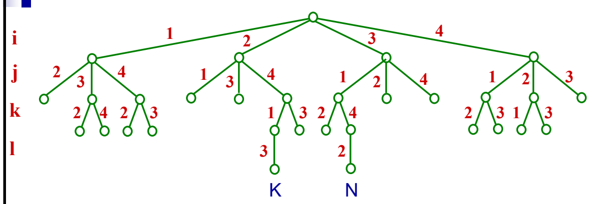
- 八皇后问题的解法从算法设计角度，属于树的剪枝策略。还是以四皇后为例，可能的状态可以用“状态树”描述。若不计攻击，可能的状态有 24 种。

129



- 根下面的四条树枝是棋盘第一行的四个落点；对于每一个落点，再分别有三条树枝，对应第二行除去同一列的三个落点，……。
- 如果考虑攻击，很多状态不可能达到，采取“剪枝”的手段，把发生攻击的中间状态下面的树枝剪去，可以省掉很多判断。

130



- 状态树主要用于描述有回溯的问题。对它进行先序遍历，在叶结点处回溯，可以了解问题所有可能的状态。

131