数据结构

主讲：项若曦　助教：申智铭、黄毅
rxxiang@blcu.edu.cn
主楼南329

回顾

- 二叉树的先序、中序、后序的非递归遍历
- 层次遍历
- 双序列递归创建二叉树(★需要掌握)
- 表达式树
- (*)线索化、普通树和森林的存储

回顾

- **二叉树的应用**
  - 二叉排序树
  - Huffman树
  - 平衡二叉树（*）
  - 堆（排序）
  - 并查集（*）

回顾

- **二叉排序树**
  - 概念、性质、存储
  - 操作：遍历、查找、插入、创建、删除（遗留问题）
  - 性能分析：平均查找长度~（遗留问题）

本节内容

- 二叉排序树
  - 概念、性质、存储
  - (继续)操作：遍历、查找、插入、创建、删除
  - 性能分析：平均查找长度~
- Huffman树

二叉排序树

- 概念、存储
- 操作：查找、插入、创建、删除
- 性能分析

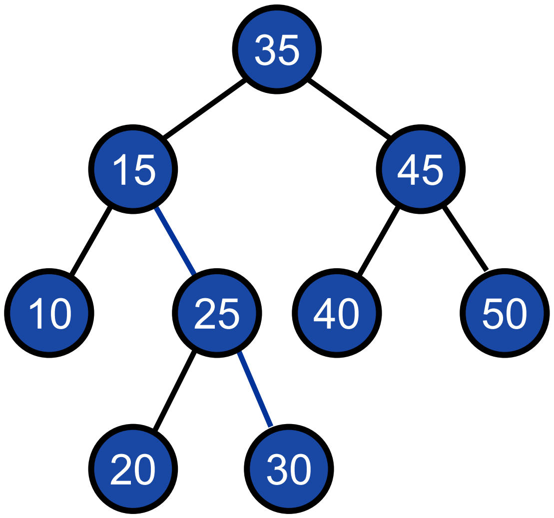
二叉排序树 (Binary Search Tree)

- **定义**
  - 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)，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。

二叉排序树

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

注意，国外教材统称为二叉搜索树。



二叉排序树的结构定义

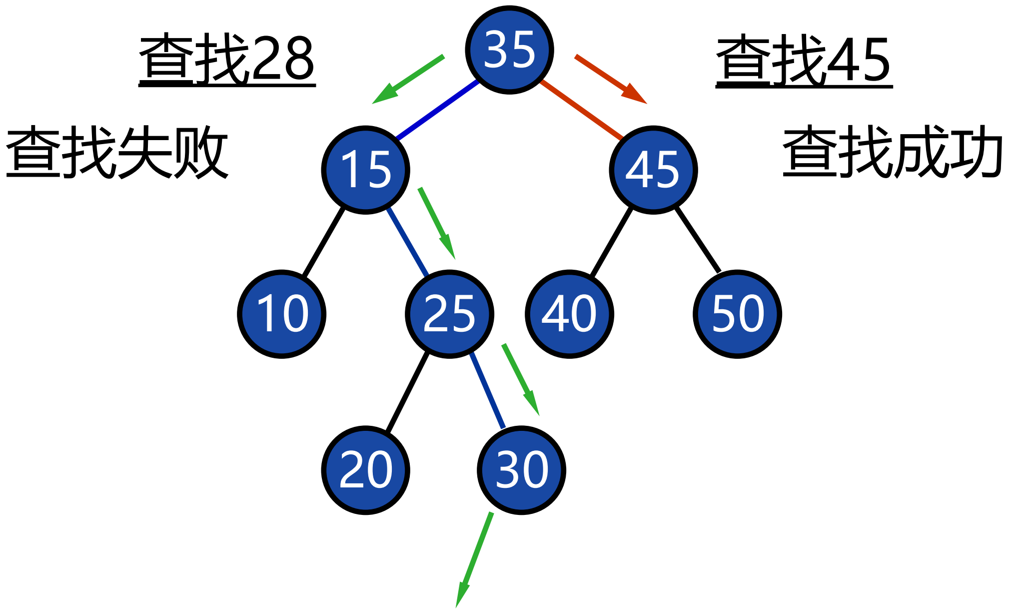
```
typedef char ElemType;          //树结点数据类型
typedef struct node {           //二叉排序树结点
    ElemType data;
    struct node *lchild, *rchild;
} BstNode, *BST;                //二叉排序树定义
```

```
typedef struct BiTNode{
    ElemType data;
    struct BiTNode *lchild, *rchild;
    // 左右孩子指针
}BiTNode, *BiTree;
```

- 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。

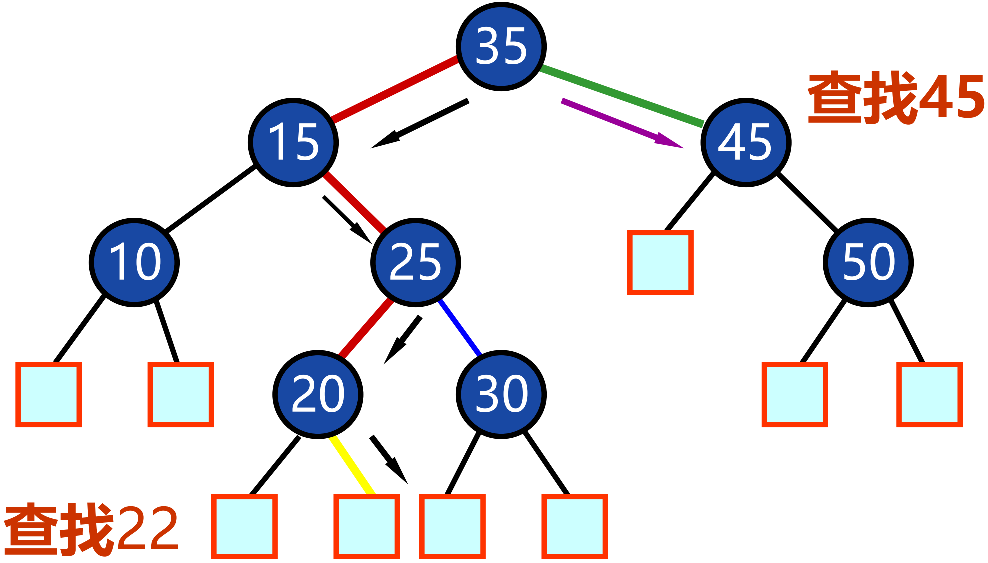
二叉排序树上的查找

- 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。它可以是一个递归的过程。



- 假设想要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。

- 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。



二叉排序树的查找算法(递归)

- 参见详见P199，算法7.4！

BiTree SearchBST(BiTree T, ElemType x)
void SearchBST(BiTree T, ElemType x, BiTree &p, BiTree &pr,);
观察递归有什么特点?

二叉排序树的查找算法版本1(递归)

T->data.key <==>T->data

```
BSTree SearchBST(BSTree T, ElemType key) {
//课本p199算法7.4 数据结构稍微和本节使用不一样。
    if ((!T) || key == T->data.key) return T;
    else if (key < T->data.key)  return SearchBST(T->lchild, key);    //在左子树中继续查找
    else return SearchBST(T->rchild, key);                //在右子树中继续查找
} //注意：查找成功返回什么？查找不成功返回什么?
```

二叉排序树的查找算法版本2(递归)

```
void Find(BST t, ElemType x, BST& p, BST &pr) {
    //在二叉排序树 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.
    //不成功时 p 为空, pr 返回最后走到的结点地址.
    if (t == NULL) { p = NULL; }
    else if (t->data == x) p = t;
    else if (t->data > x) {
        pr = t;
        Find(t->lchild, x, p, pr);
    }
    else {
        pr = t;
        Find(t->rchild, x, p, pr);
    }
}
```

二叉排序树的查找算法版本3（迭代）

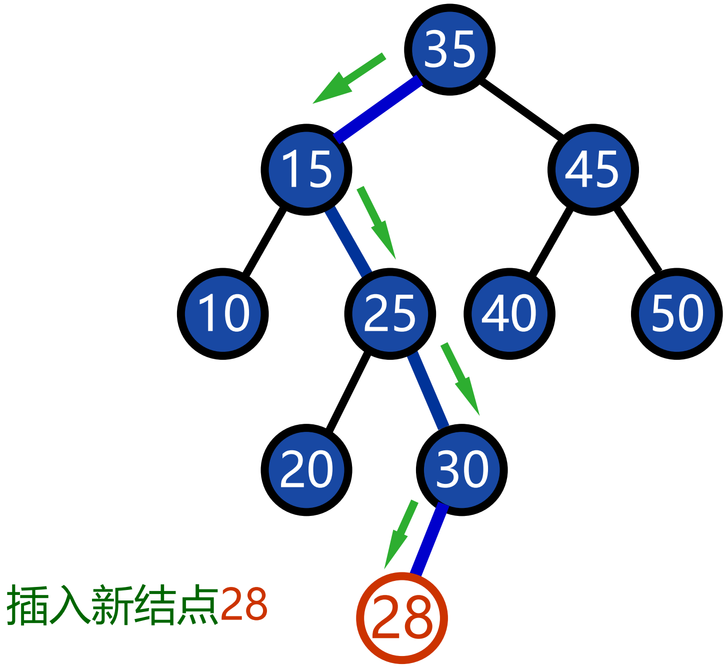
北京语言大学 BEIJING LANGUAGE AND CULTURE UNIVERSITY

```
void Find(BST t, ElemType x, BST& p, BST& pr) {
    //在二叉排序树 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.
    //不成功时 p 为空, pr 返回最后走到的结点地址.
    if (t != NULL) {
        p = t;  pr = NULL;          //从根查找
        while (p != NULL && p->data != x) {
            pr = p;
            if (p->data < x) p = p->rchild;
            else p = p->lchild;
        }
    }
}
```

查找的关键字比较次数最多不超过树的高度。

二叉排序树的插入

- 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。
- 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：树中原来没有关键字等于给定值的结点，把新元素加到查找操作停止的地方。



二叉排序树的插入版本1（非递归）

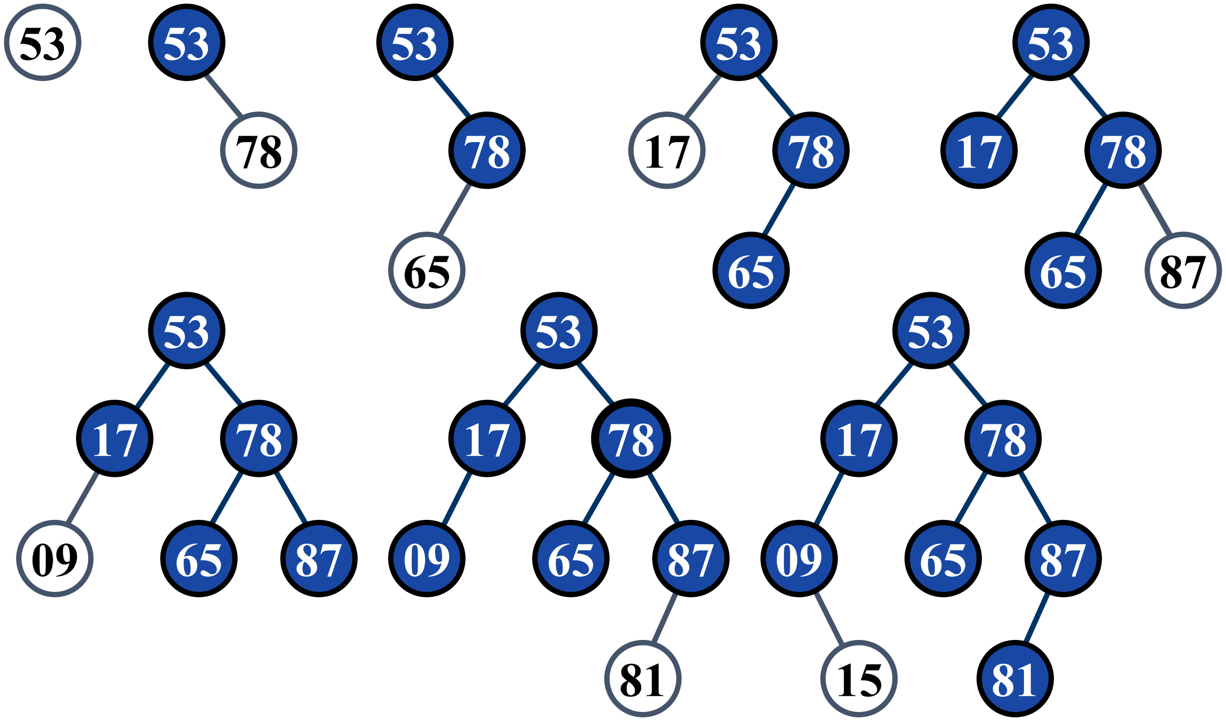

```
void Insert(BST& t, ElemType x) {
    //将新元素 x 插到以 *t 为根的二叉排序树中
    BstNode* pt, * prt=NULL, * q;
    Find(t, x, pt, prt);                 //查找结点插入位置
    if (pt == NULL) {                    //查找失败时可插入
        q = new BstNode;  q->data = x;   //创建结点
        q->lchild = q->rchild = NULL;
        if (prt == NULL) t = q;          //空树
        else if (x < prt->data) prt->lchild = q;
        else prt->rchild = q;
    }
}
```

二叉排序树的插入版本2（递归，P201算法7.5）

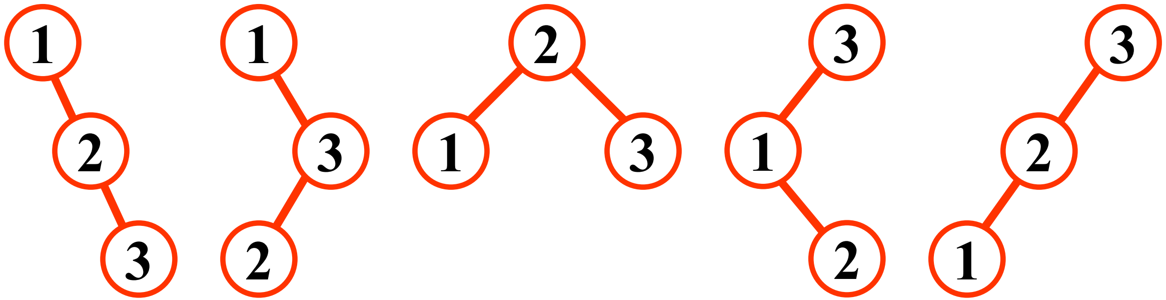

```
void InsertBST(BSTree& T, ElemType e) {
//当二叉排序树 T中不存在关键字等千e.key的数据元素时， 则插入该元素
    if (!T) {                              //找到插入位置， 递归结束
        S = new BSTNode;                   //生成新结点*S
        S->data = e;                       //新结点*S的数据域置为e
        S->lchild = S->rchild = NULL;      //新结点*S作为叶子结点
        T = S;                             //把新结点*S链接到已找到的插入位置
    }
    else if (e.key < T->data)
        InsertBST(T->lchild, e);           //将*S插入左子树
    else if (e.key > T->data)
        InsertBST(T->rchild, e);           //将*S插入右子树
}
```

输入数据，建立二叉排序树的过程

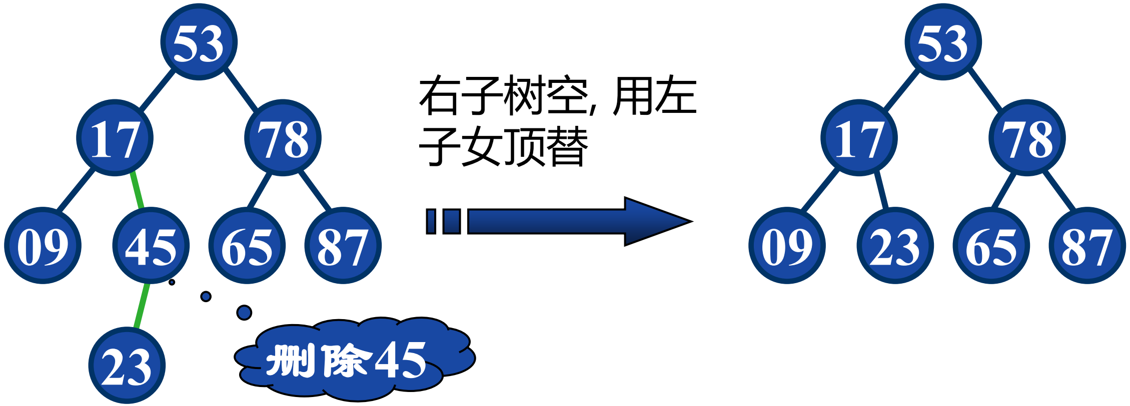
- 输入数据 { 53, 78, 65, 17, 87, 09, 81, 15 }

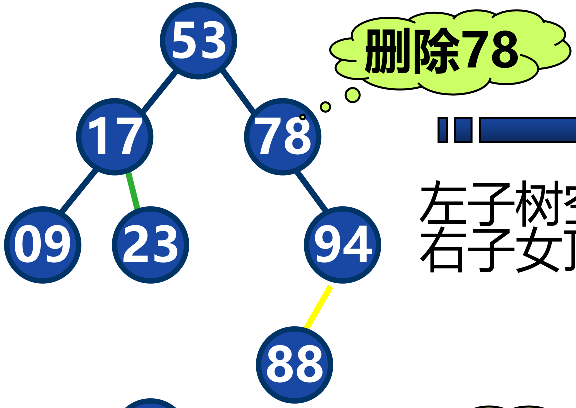


- 同样 3 个数据{ 1, 2, 3 }，输入顺序不同，建立起来的二叉排序树的形态也不同。这直接影响到二叉排序树的查找性能。
- 如果输入序列选得不好，会建立起一棵单支树，使得二叉排序树的高度达到最大。
- {2, 1, 3}
- {1, 2, 3} {1, 3, 2} {2, 3, 1} {3, 1, 2} {3, 2, 1}

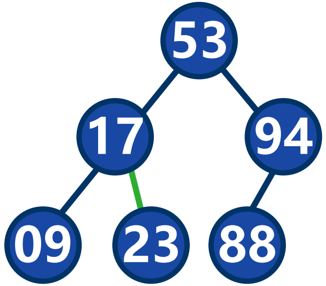


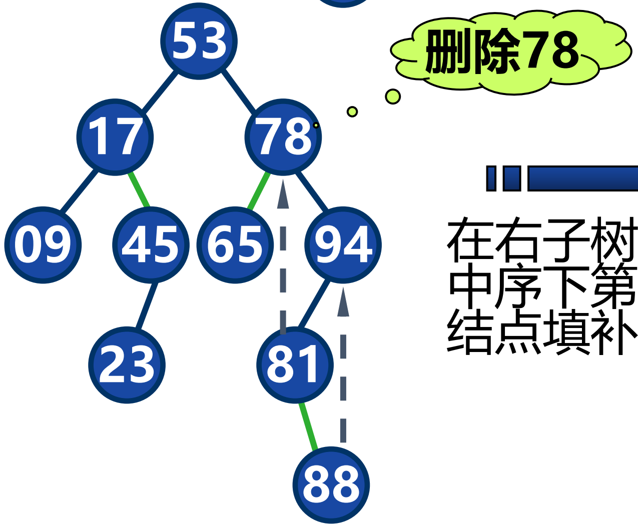
二叉排序树的删除



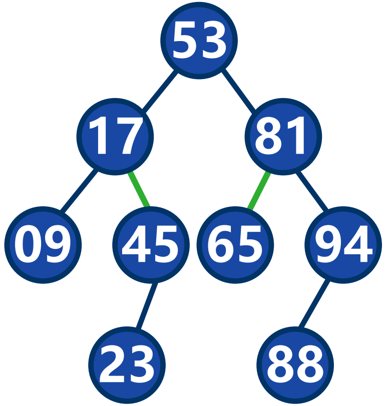


左子树空,用右子女顶替





在右子树上找中序下第一个结点填补



二叉排序树的删除

- **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，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不至于降低，还需要防止重新链接后树的高度增加。**
  - 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。
  - 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。
  - 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(所有比被删关键字大的关键字中它的值最小),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。

- **二叉排序树**
  - 性能分析

二叉排序树性能分析

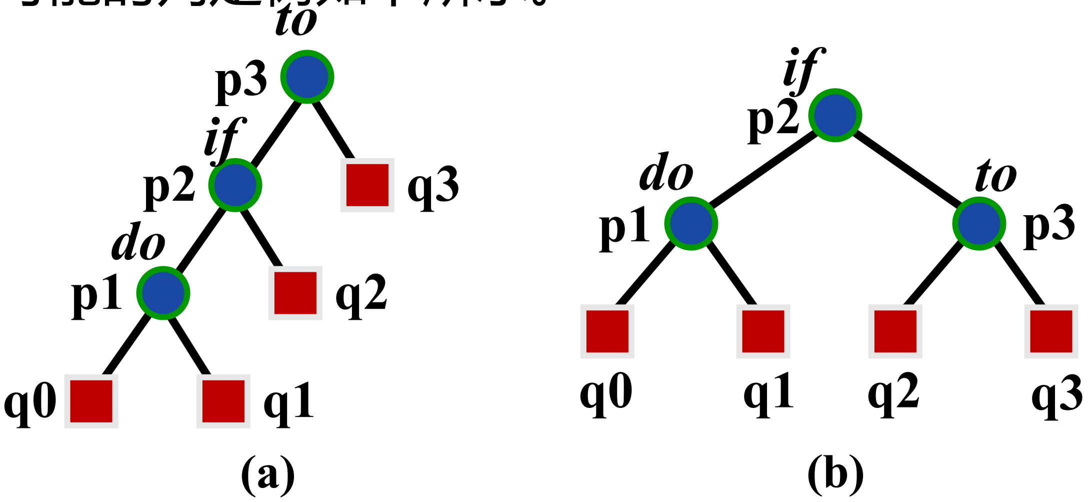
- 对于有 n 个关键字的集合，其关键字有 n! 种不同排列，可构成不同二叉排序树有

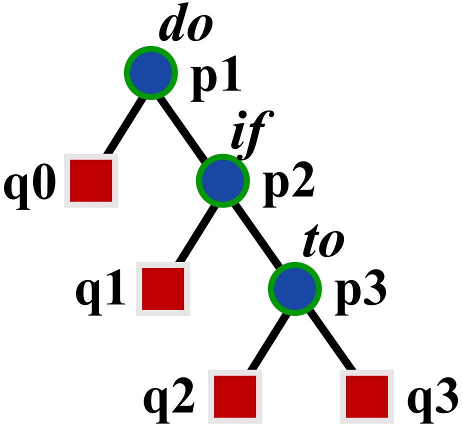
$$\frac{1}{n+1}C_{2n}^{n}$$

(棵)

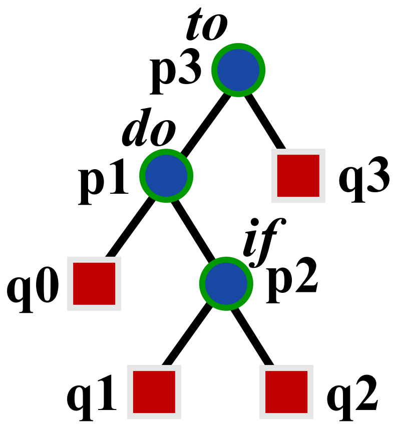
- 用树的查找效率来评价这些二叉排序树。
- 用判定树来描述查找过程，在判定树中，○表示内结点，它包含关键字集合中的某一个关键字；□表示外结点，代表各关键字间隔中的不在关键字集合中的关键字。它们是查找失败时到达的结点，物理上实际不存在。

- 在每两个外部结点间必存在一个内部结点。
- 例，已知关键字集合 { 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。

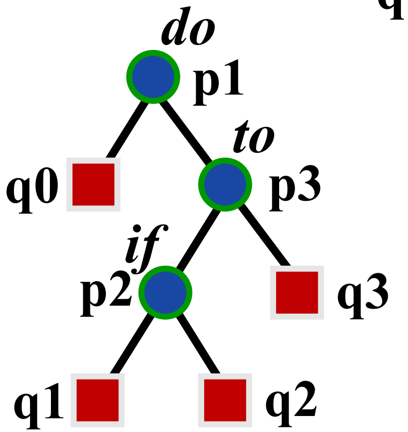



do
p1
q0
if
p2
q1
to
p3
q2
q3

(c)


to
p3
do
p1
q3
q0
if
p2
q1
q2

(d)


do
p1
q0
to
p3
if
p2
q3
q1
q2

(e)

- 一棵判定树的查找成功的平均查找长度ASLsucc 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字比较次数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ASLunsucc为树中所有外结点上权值q[j]与到达该外结点所需关键字比较次数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：

$p[i] = 1/3,\ 1 \le i \le 3$

$q[j] = 1/4,\ 0 \le j \le 3$

图(a): ASLsucc = 1/3*(3 + 2 +1) = 6/3 = 2

ASLunsucc = 1/4*(3 + 3 +2 + 1) = 9/4

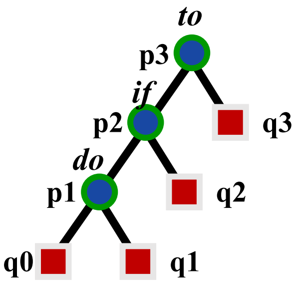
图(b): ASLsucc = 1/3*(2 + 1 + 2) = 5/3

ASLunsucc = 1/4*(2 + 2 + 2 + 2) = 8/4

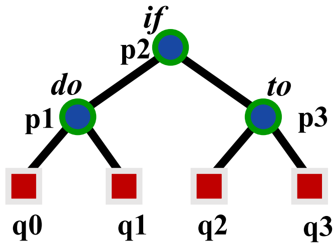
图(c): ASLsucc = 2, ASLunsucc = 9/4

图(d): ASLsucc = 2, ASLunsucc = 9/4

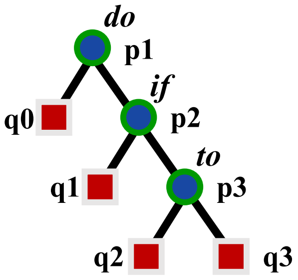
图(e): ASLsucc = 2, ASLunsucc = 9/4



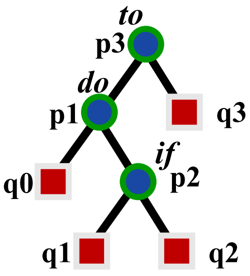
(a)



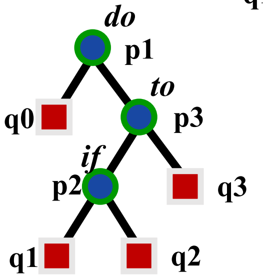
(b)



(c)



(d)



(e)

- 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排序树。
- 在相等查找概率的情形下，最优二叉排序树的上面 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。

本节内容

- **Huffman树**
  - 概念
  - 创建
  - 编码
  - 解码

本节内容

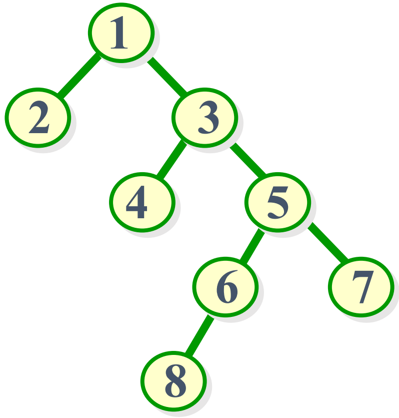
- Huffman树
  - 概念

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。

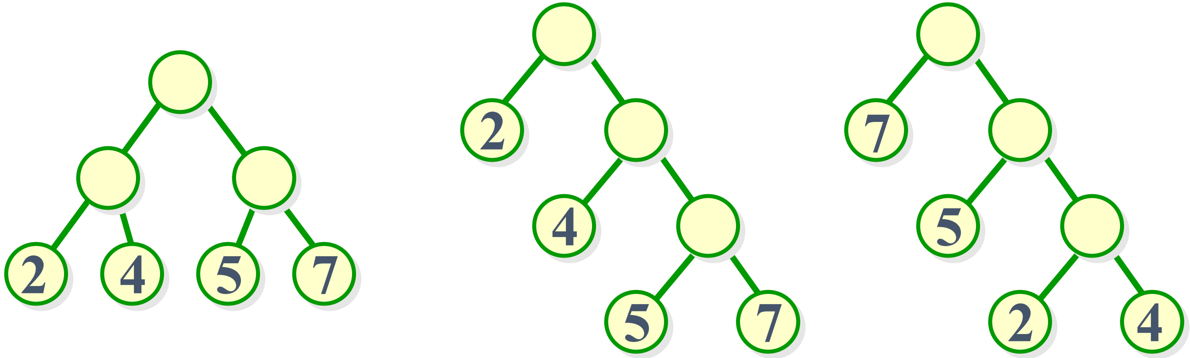
  右图中，结点4与结点6间的路径长度为 3。
- 树的路径长度是各结点到根结点的路径长度之和PL。

  右图中，PL = 0+1+1+2+2+3+3+4 = 16。

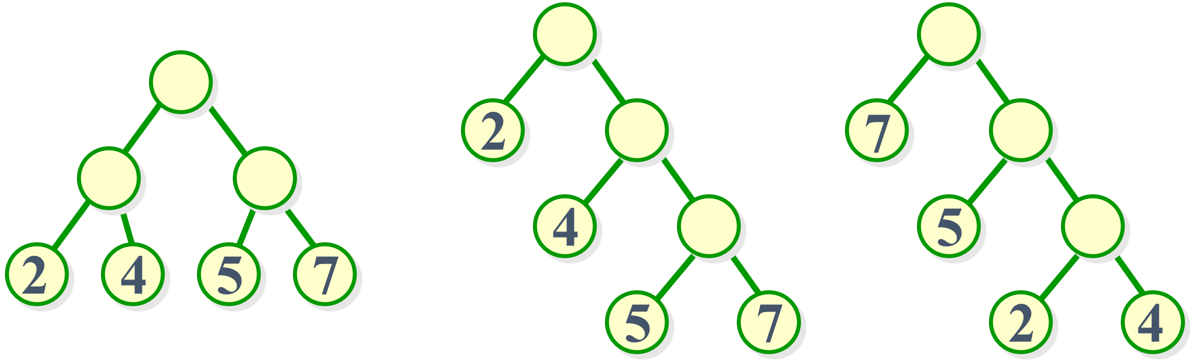


- **带权路径长度 (Weighted Path Length, WPL)**
  - 二叉树的带权路径长度是各叶结点所带权值 wi 与该结点到根的路径长度 的乘积的和。

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



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Huffman树。
- 在Huffman树中，权值大的结点离根最近。

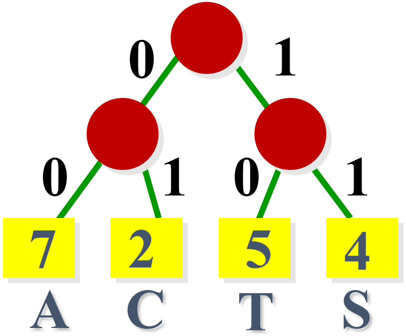
应用：Huffman编码——前缀编码

- 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



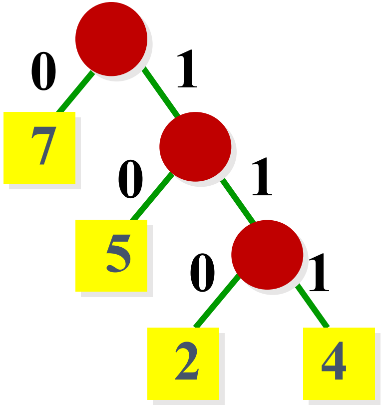
则总编码长度为

(2+7+4+5)*2 = 36.

- 能否减少发出的报文编码数?

- 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。

前缀编码——二元前缀码为例

- 二元前缀码

  用0-1字符串作为代码表示字符，要求任何字符的代码都不能作为其它字符代码的前缀

Huffman编码是前缀编码

- 非前缀码的例子

  a: 001, b: 00, c: 010, d: 01

- 解码的歧义，例如字符串 0100001

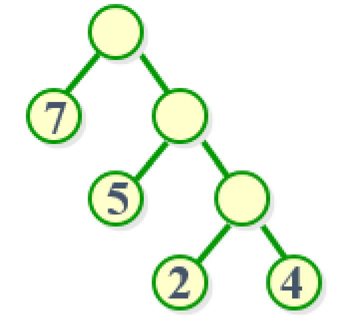
  解码1: 01, 00, 001 d, b, a

  解码2: 010, 00, 01 c, b, d

本节内容

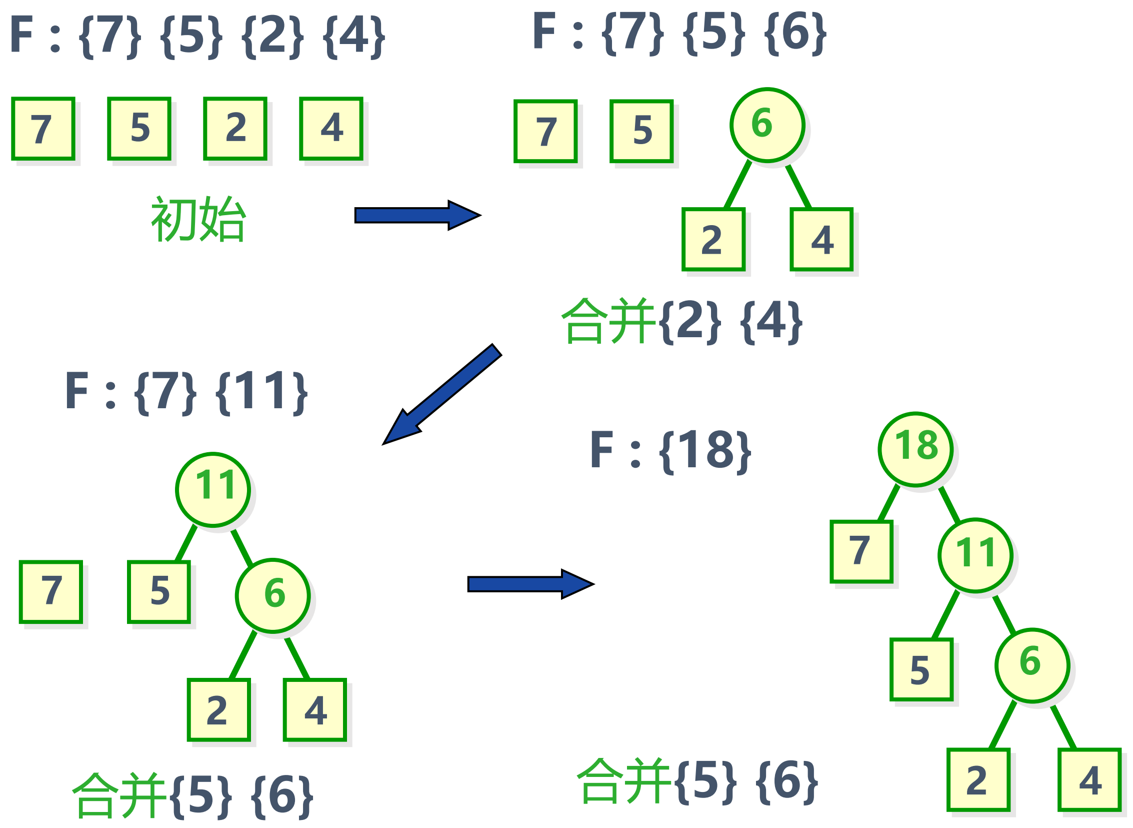
- **Huffman树**
  - 创建

- **Huffman树怎么创建？？**
  - 四个权值，在树的什么位置?
  - 二叉树一共有几个结点?
  - 构造顺序是怎么样的?



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 Ti 只有一个带权值 wi 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：
  - 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。
  - 在 F 中删去这两棵二叉树。
  - 把新的二叉树加入 F。


F : {7} {5} {2} {4}
7 5 2 4
初始
F : {7} {5} {6}
7 5 6 2 4
合并{2} {4}
F : {7} {11}
11 7 5 6 2 4
合并{5} {6}
F : {18}
18 7 11 5 6 2 4
合并{5} {6}

- 采用静态链表方式存储Huffman树的结构定义：

```
const  int n = 20;
typedef struct {
     float weight;
     int parent, lchild, rchild;
} HTNode;

typedef Struct {
     HTNode F[2n-1];
   int root;
} HuffmanTree;
```

采用静态链表的Huffman树*

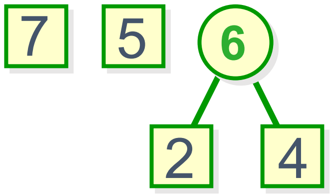
- 采用静态链表方式存储Huffman树的结构定义（课本定义）：

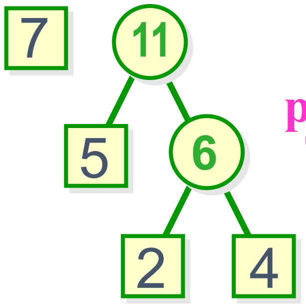
```
const  int n = 20;
typedef struct {
    float weight;
    int parent, lchild, rchild;
} HTNode, * HuffmanTree;
```

7 5 2 4

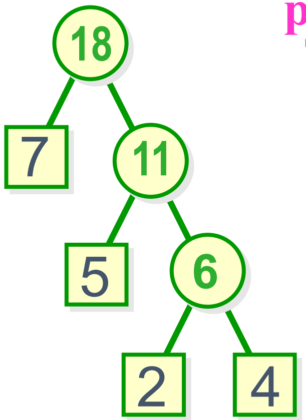
	weight	parent	lchild	rchild
0	7	-1	-1	-1
1	5	-1	-1	-1
2	2	-1	-1	-1
3	4	-1	-1	-1
4		-1	-1	-1
5		-1	-1	-1
6		-1	-1	-1

	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



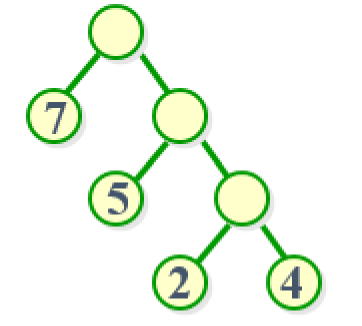
		weight	parent	lchild	rchild
p1 →	0	7	6 ~~-1~~	-1	-1
	1	5	5	-1	-1
	2	2	4	-1	-1
	3	4	4	-1	-1
	4	6	5	2	3
p2 →	5	11	6 ~~-1~~	1	4
i →	6	18	-1	0 ~~-1~~	5 ~~-1~~

建立Huffman树的算法

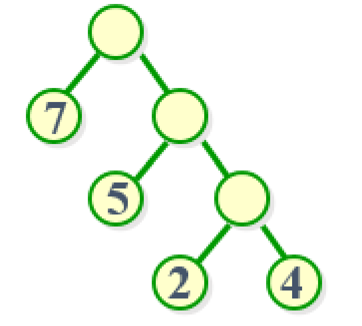
```
void CrtHuffmanTree (HuffmanTree &T, float fr[ ]) {
    int i, j, p1, p2, min1, min2;
    for (i = 0; i < n; i++) T.F[i].weight = fr[i];//
    for (i = 0; i < 2*n-1; i++) {
        T.F[i].parent = -1;
        T.F[i].lchild = -1;
        T.F[i].rchild = -1;
    }
    for (i = n; i < 2*n-1; i++) {
        min1 = min2 = maxValue;
        for (j = 0; j < i; j++)
```

建立Huffman树的算法

```
            if (T.F[j].parent == -1)
                if (T.F[j].weight < min1) {
                    p2 = p1;  min2 = min1;
                    p1 = j;  min1 = T.F[j].weight;
                }
                else if ( T.HF[j].weight < min2 )
                    { p2 = j;  min2 = T.F[j].weight; }
            T.F[i].lchild = p1;   T.F[i].rchild = p2;
            T.F[i].weight = T.F[p1].weight+T.F[p2].weight;
            T.F[p1].parent = T.F[p2].parent = i;
        }
}
```



- **Huffman树概念和相关应用：**
  - Huffman树、Huffman编码、带权路径长度WPL
  - 文件**编码**和**解码**



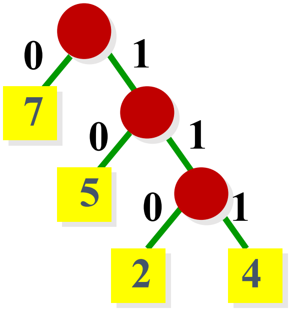
- **Huffman树概念和相关应用：**
  - 带权路径长度WPL

带权路径长度WPL的计算

给定一棵huffman树，如何得到该树的带权路径长度WPL?

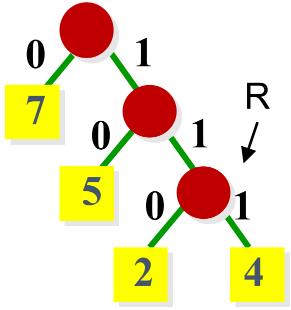
1、找到每个叶节点到根的路径长度

2、该叶节点权值*路径长度

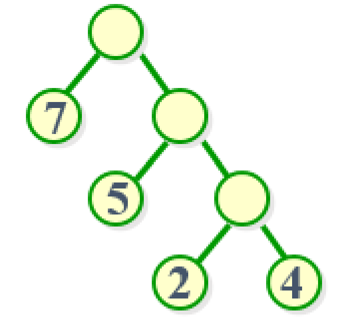


带权路径长度WPL的计算（参考代码）

```
float GetHTWPL(HuffmanTree& HT) {
    int i, n = (1 + HT.size) / 2, R;    float WPL = 0;
    for (i = 0; i < n; i++) {//对每个节点循环
        R = HT.elem[i].parent;    int pl = 0;
        while (1) {
            pl++; R = HT.elem[R].parent;
            if (R == -1) break;
        }
        WPL += pl * HT.elem[i].weight;
    }
    return WPL;
}
```

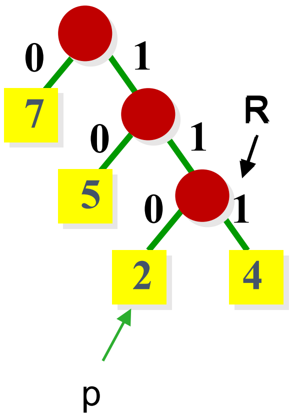


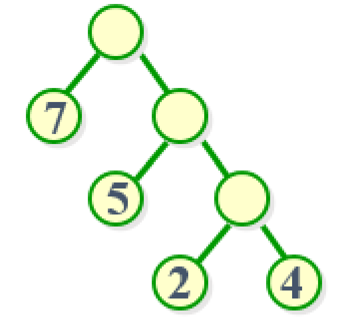
- 得到了哈夫曼树后，如何得到哈夫曼编码？
  - 向左0，向右1



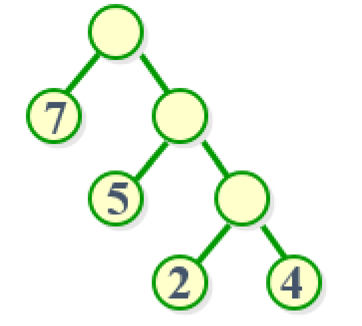
Huffman编码的打印


```
void EncodeHT(HuffmanTree& HT) {//伪码
    int i, n = (1 + HT.size) / 2, p, R;
    LinkStack S;     Init(S);
    for (i = 0; i < n; i++) {//对n个节点循环
        p = i; R = HT.elem[p].parent;
        while (1) {
            if (HT.elem[R].lchild == p) push(S, 0);
            else push(S, 1);
            p = R;   R = HT.elem[p].parent;
            if (R == -1) break;
        }
        while (!IsEmpty(S)) {
            pop(S, p);
            cout << p;
        }
        cout << endl;
    }
}
```





- **哈夫曼编码如何存储？参见课本P142**
  - 向左0，向右1
  - 编码如何存储？
  - typedef char **HuffmanCode;

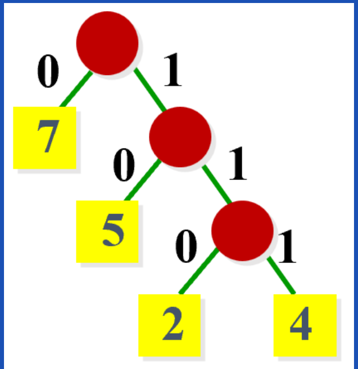


- **Huffman树概念和相关应用：**
  - 文件编码、解码

本节内容

- **Huffman树**
  - 文件编码

在建好哈夫曼树的基础之上：



编码：对给定文件字符序列/文字，给出其Huffman01编码序列（报文）；

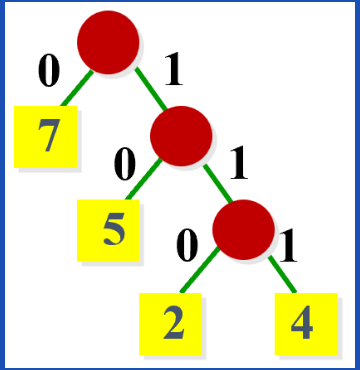
CASTCCCAASTT

->

110011110110110110001111010

本节内容

- **Huffman树**
  - 解码



解码：把得到的Huffman编码，逆转换成相应的字符序列。

11001111011011011000111101
->
CASTCCCAASTT

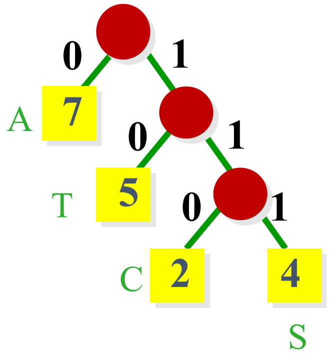
以文件解码为例

给定一棵huffman树（一维数组中），给定传输过来的01数据，如何解码得到传输前的原报文？

0011110110111

就是根据报文，找叶节点

报文控制着从根节点到叶节点的路径



思路：

1、依据报文从霍夫曼树的根节点行进

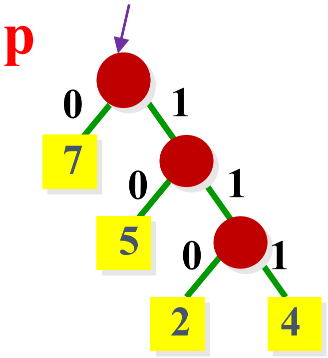
如果遇到叶节点，则输出叶节点的字符。回到根节点。

2、重复1直至报文结束

解码思路

框架：给定HT、seq：0101101111

0、字符串指针i=0；树工作节点p指向root



1、字符串每个字符循环：

1.1 if 字符为'0'：

p <--- p的左子；

else： p <--- p的右子;

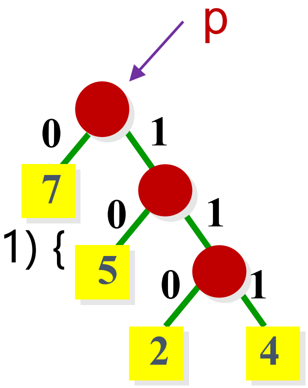
1.2 if p是叶节点：

打印p->data;

p指向R;

文件解码参考代码


```
void DecodeHT(HuffmanTree& HT, char* seq) {
    int i = 0, p = T.size - 1;            //i为字符串工作指针
    while (seq[i] != '\0') {              //p为HT树工作指针
        if (seq[i] == '0')  p = HT.elem[p].lchild;
        else p = HT.elem[p].rchild;
        if (HT.elem[p].lchild == -1 && HT.elem[p].rchild == -1) {
            cout << HT.elem[p].data);  p = HT.size - 1;
        }//p回根
        i++;
    }
}//改造HTNode，使之有权值以及data字段
```



i

0101101111

总结

- **Huffman树（构造、编码、文件编码和解码）**
  - Huffman树：带权最短路径树，最短传输位。
  - Huffman树输入：频度、概率；可以用文本来统计出来。
  - 基于Huffman树的文件编码和解码。

作业预告

- 二叉排序树（构造、查找、增删等）
- Huffman树（构造、编码、文件编码和解码）

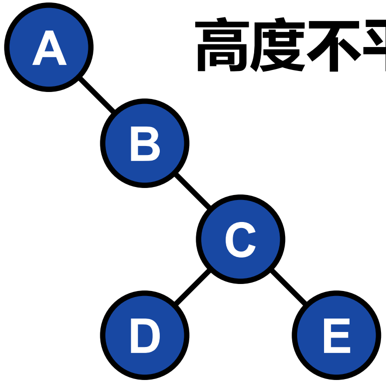
本节内容

- 平衡二叉树（选修）
- 并查集（选修）

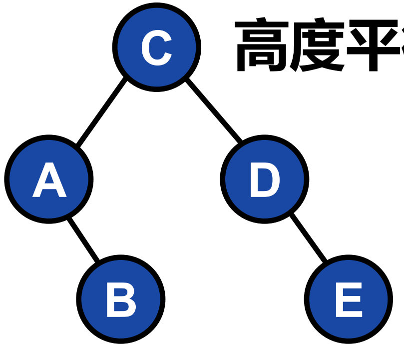
平衡二叉树

平衡二叉树的定义

- 一棵AVL树或者是空树，或者是具有下列性质的二叉排序树：它的左子树和右子树都是平衡二叉树，且左子树和右子树的高度之差的绝对值不超过1。



高度不平衡



高度平衡

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）。
- 平衡二叉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，则这棵二叉排序树就失去了平衡，不再是平衡二叉树。
- 如果一棵二叉排序树是高度平衡的, 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$O(\log_2 n)$。

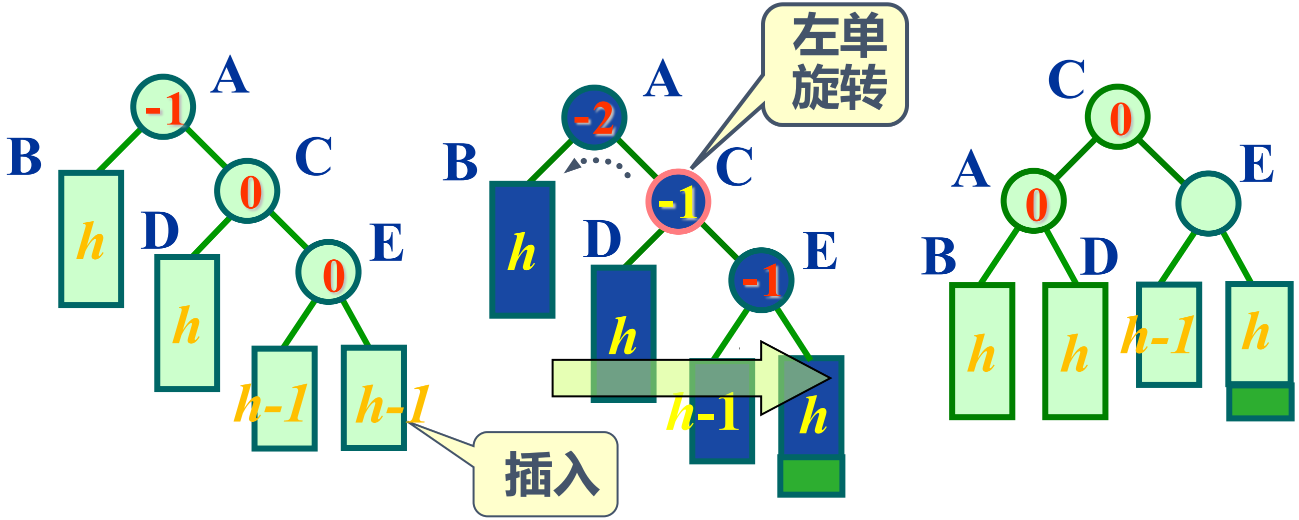
平衡化旋转

- 如果在一棵平衡二叉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，平衡二叉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。

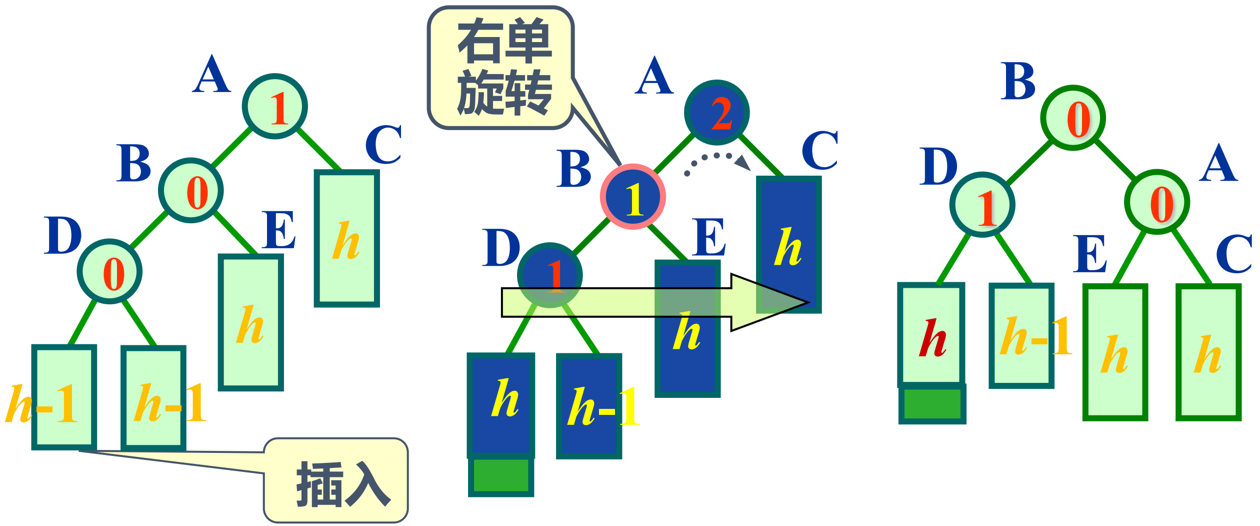
左单旋转（RR单旋）

- 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。



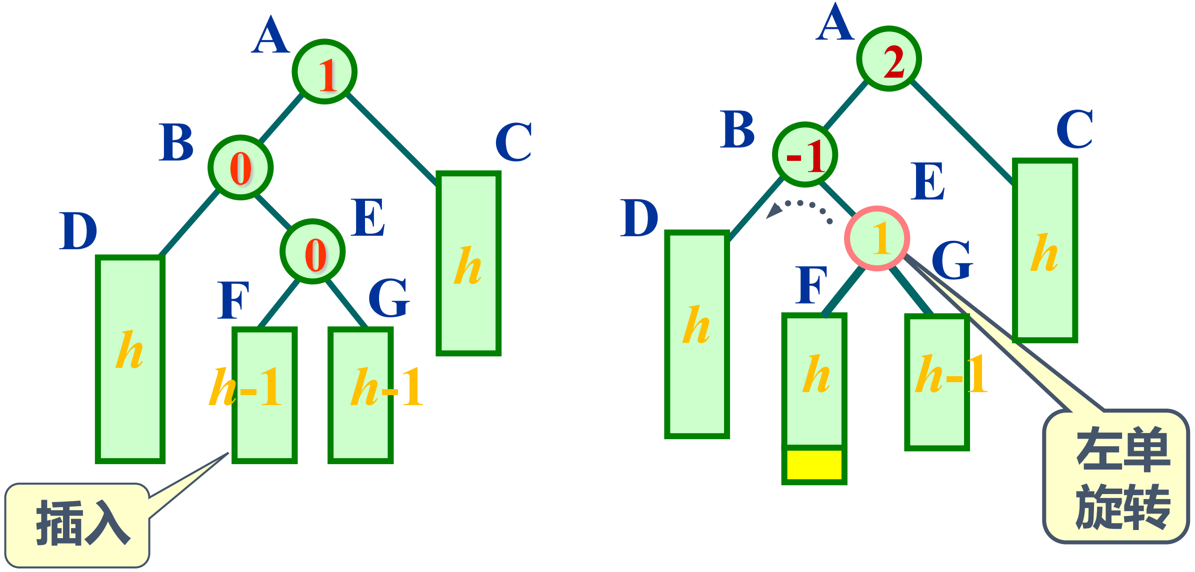
右单旋转（LL单旋）

- 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。

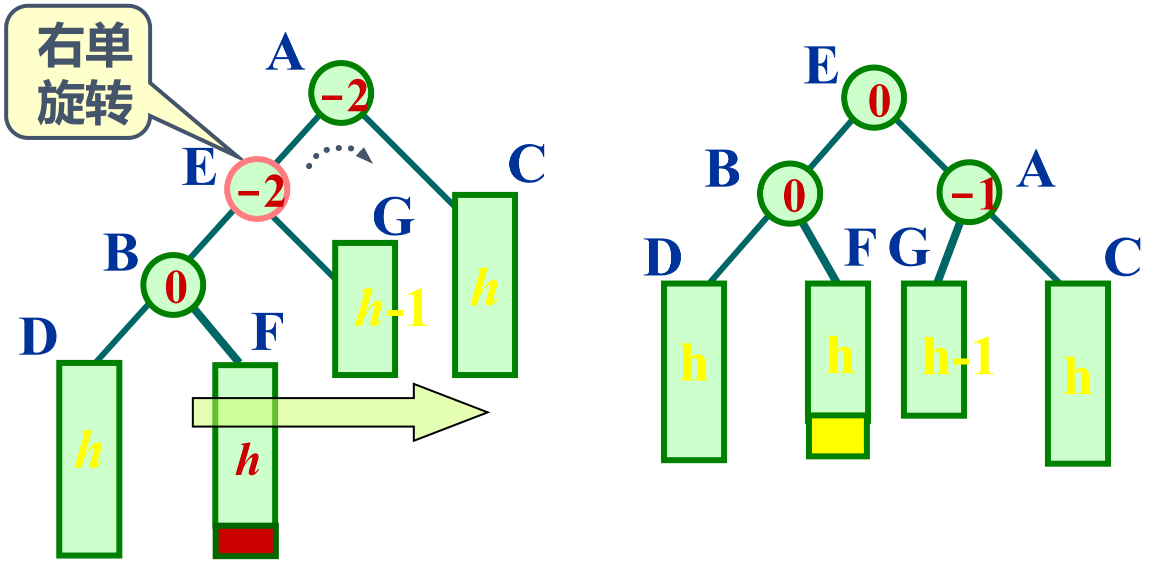


先左后右双旋转（LR双旋）

- 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。

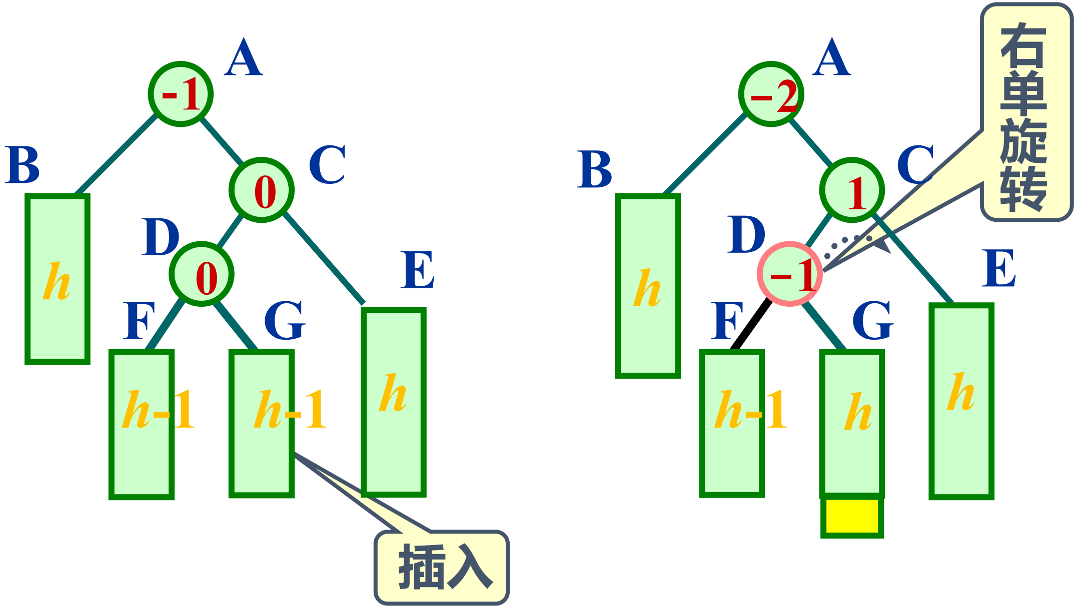


- 从结点A起沿插入路径选取3个结点A、B和E，先以结点E为旋转轴，将结点B反时针旋转，E顶替原B的位置。再以结点E为旋转轴，将结点A顺时针旋转。

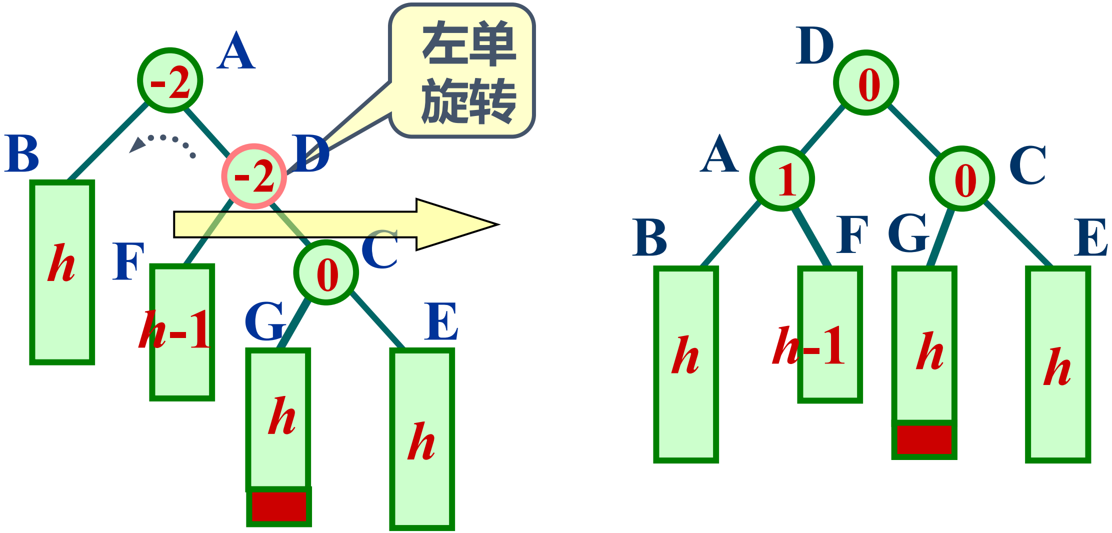


先右后左双旋转（RL双旋）

- 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。



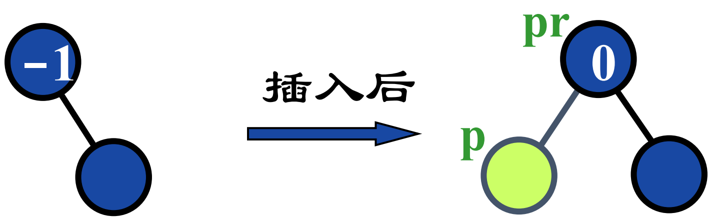
- 从结点A起沿插入路径选取3个结点A、C和D。首以结点D为旋转轴，将结点C顺时针旋转，以D代替原来C的位置。再以D为旋转轴，将结点A反时针旋转，恢复树的平衡。



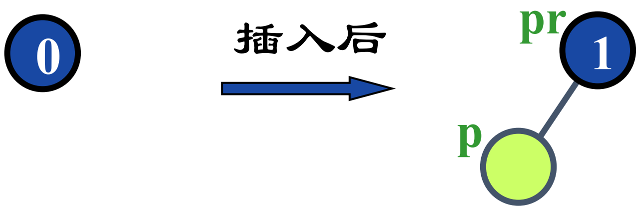
平衡二叉树的插入

- 在向一棵本来是高度平衡的平衡二叉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字，逐步建立平衡二叉树。
- 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：
- 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。

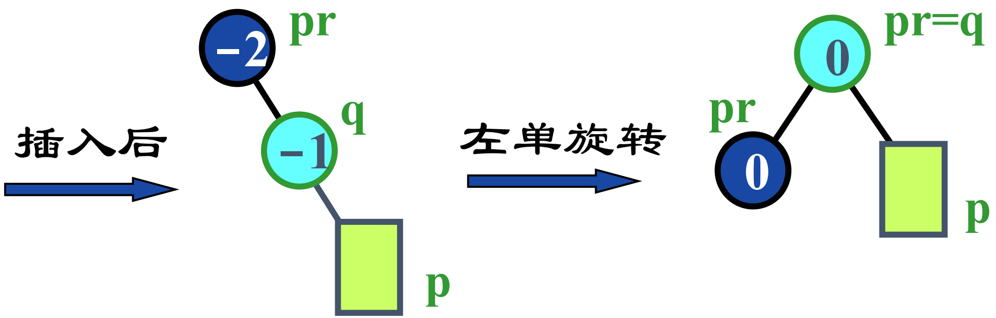


- 结点pr的平衡因子的绝对值 |bf| = 1。说明插入前pr的平衡因子是0，插入新结点后，以pr为根的子树不需平衡化旋转。但该子树高度增加，还需从结点pr向根方向回溯，继续考查结点pr双亲(pr = Parent(pr))的平衡状态。



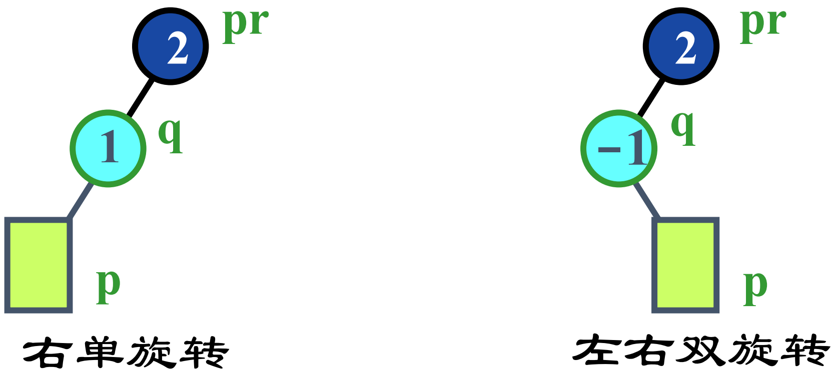
- 结点pr的平衡因子的绝对值|bf| = 2。说明新结点在较高的子树上插入，造成了不平衡，需要做平衡化旋转。此时可进一步分2种情况讨论：
  - 若结点pr的bf = -2，说明右子树高，结合其右子女q 的bf分别处理：

- 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。

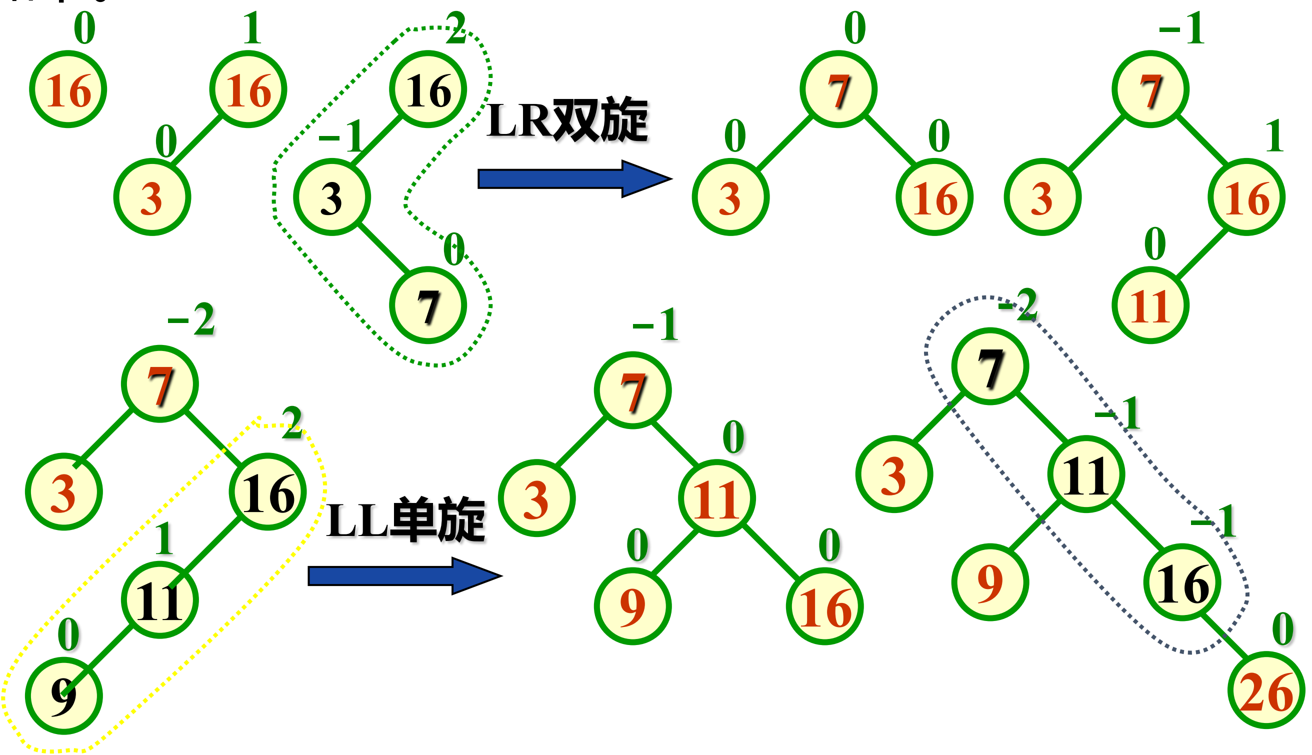
- 若结点pr的bf = 2，说明左子树高，结合其左子女q 的bf分别处理：
  - 若q的bf与pr同符号(=1)，执行LL单旋；
  - 若q的bf与pr异符号(=-1)，执行LR双旋。



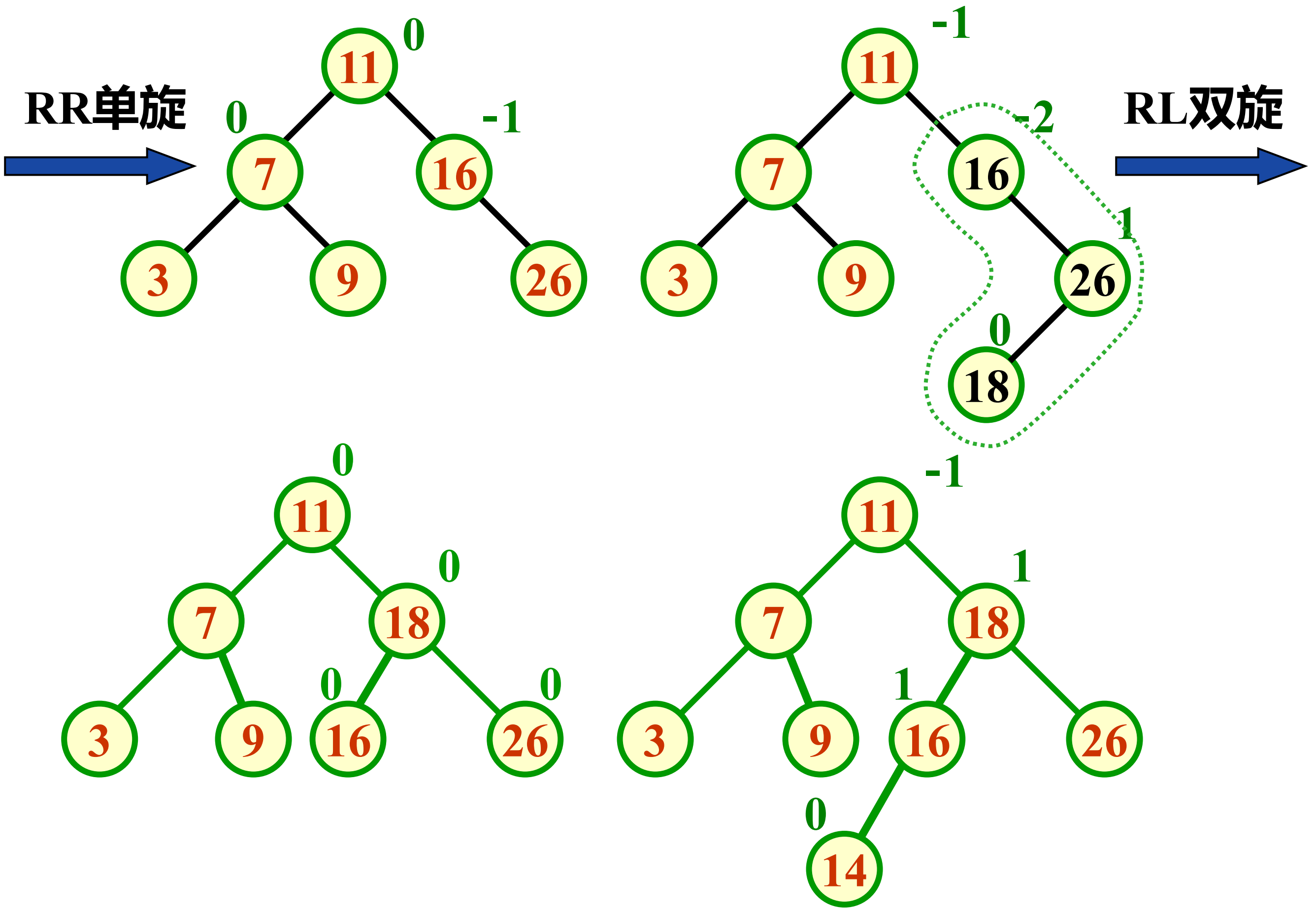
- 下面举例说明在平衡二叉树上的插入过程。

从空树开始的建树过程

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

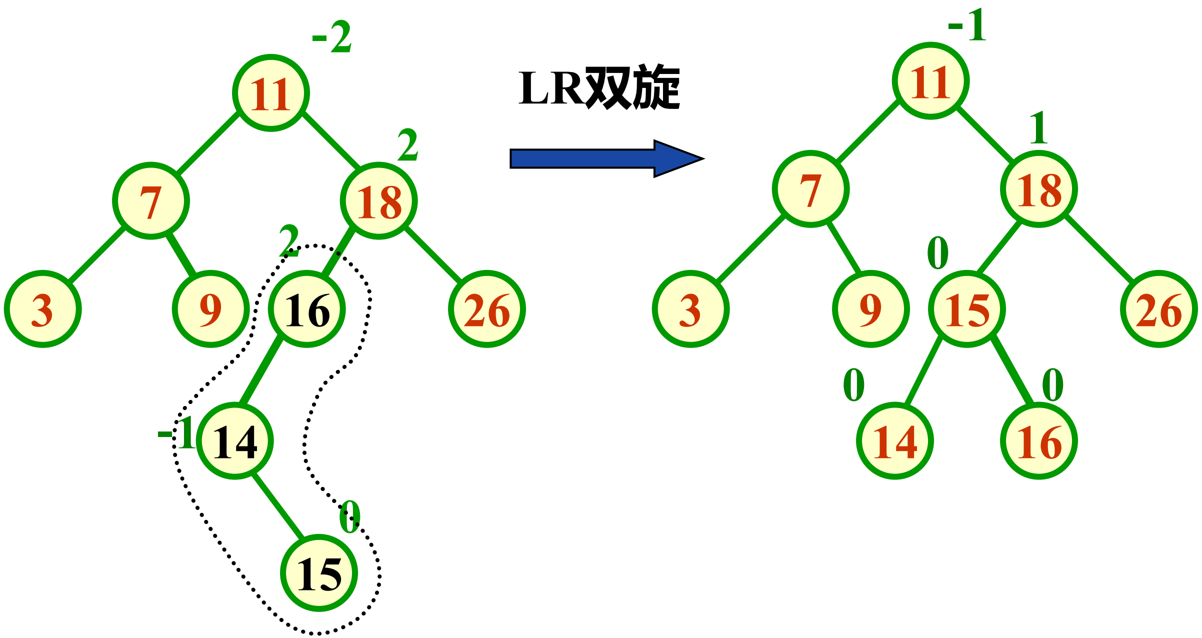


从空树开始的建树过程



RR单旋
RL双旋
0
11
0
7
-1
16
3
9
26
-1
11
7
-2
16
3
9
1
26
0
18
0
11
7
0
18
0
16
0
26
3
9
-1
11
7
1
18
3
9
1
16
26
0
14

从空树开始的建树过程

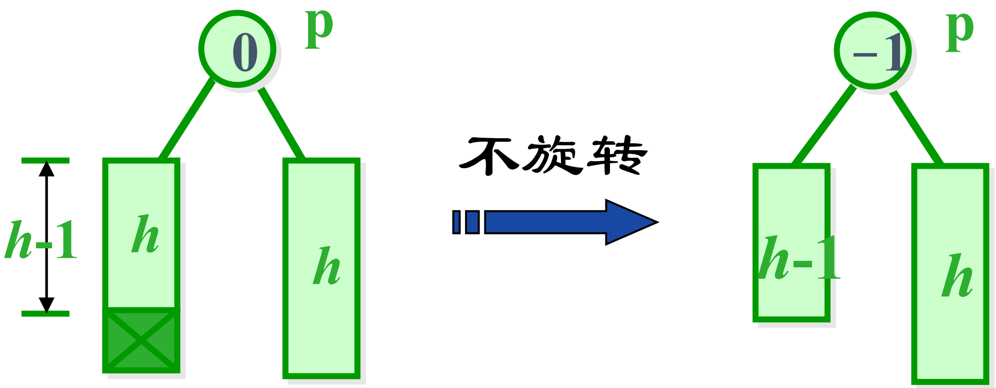


平衡二叉树的删除

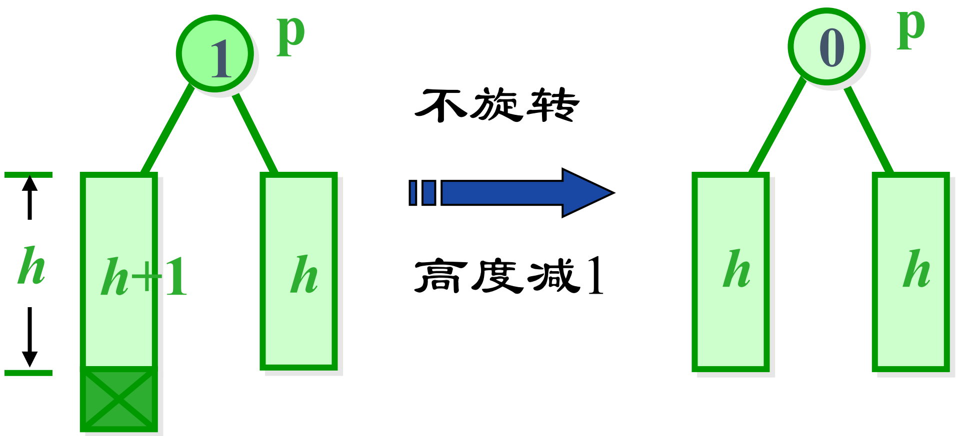
- 如果被删结点x最多只有一个子女，那么问题比较简单:
  - 将结点x从树中删去。
  - 因为结点x最多有一个子女，可以简单地把x的双亲结点中原来指向结点x的指针改指到这个子女结点；
  - 如果结点x没有子女，x双亲结点的相应指针置为NULL。
  - 将原来以结点x为根的子树的高度减1。

- 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。

- 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，插入完成。

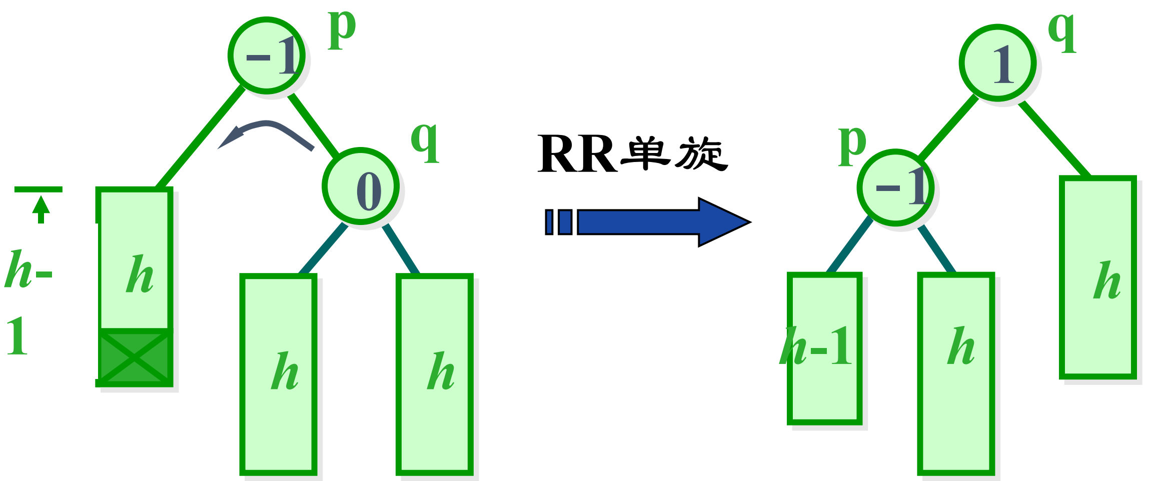


- 当前结点 p 的bf不为0，且较高子树的高度被降低，则 p 的bf 改为0。因该结点的高度降低，需向上回溯，看其双亲是否失去平衡。

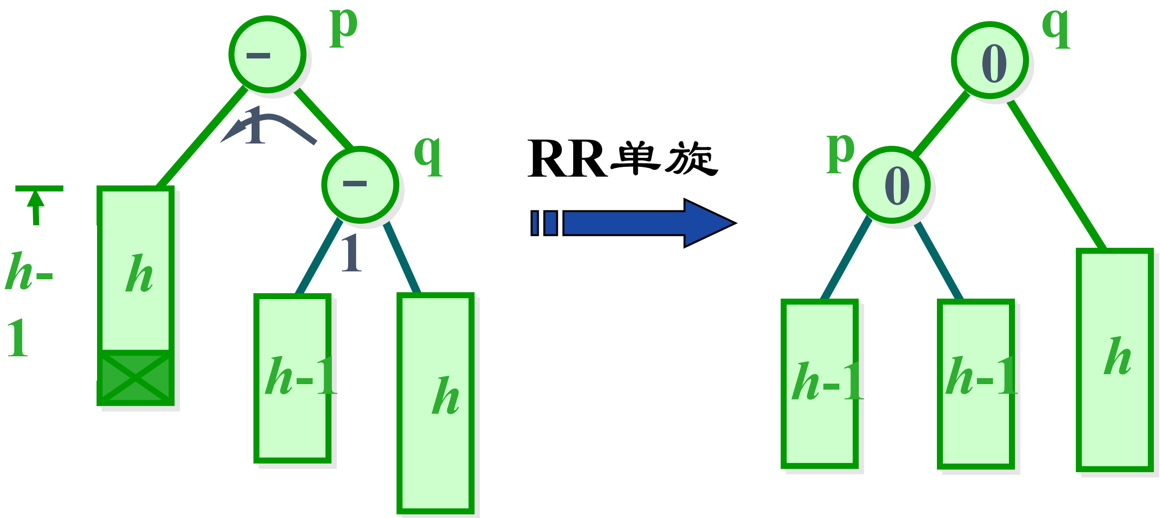


- 当前结点 p 的bf 不为0，且较矮子树的高度又被降低，则在结点 p 发生不平衡。需要进行平衡化旋转来恢复平衡。

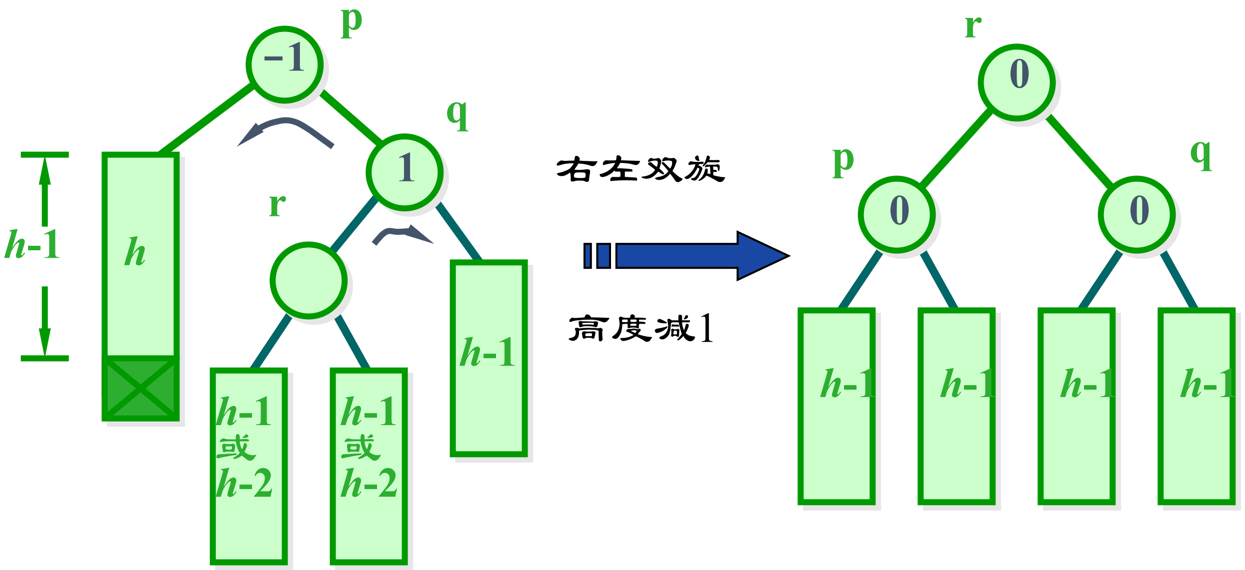
- 令 p 的较高的子树的根为 q (该子树高度未被降低)，根据 q 的 bf，有如下 3 种平衡化操作。
  - 如果 q (较高的子树) 的 bf为 0，执行一个单旋转来恢复结点 p 的平衡，由于旋转后子树高度未降低，无需向上回溯。



- 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。



- 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。



平衡二叉树的高度

- 设在新结点插入前平衡二叉树的高度为h，结点个数为n，则插入一个新结点，其时间代价是O(h)。对于平衡二叉树来说，h多大?
- 设 N_h 是高度为 h 的平衡二叉树的最小结点个数。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)
  - $N_h = N_{h-1} + N_{h-2} + 1 , h > 1$

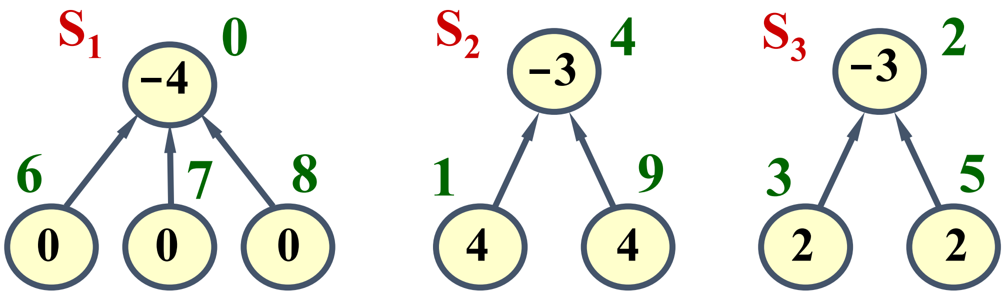
本节内容

- 并查集（选修）

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 S1= {0, 6, 7, 8}, S2= {1, 4, 9}, S3= {2, 3, 5}



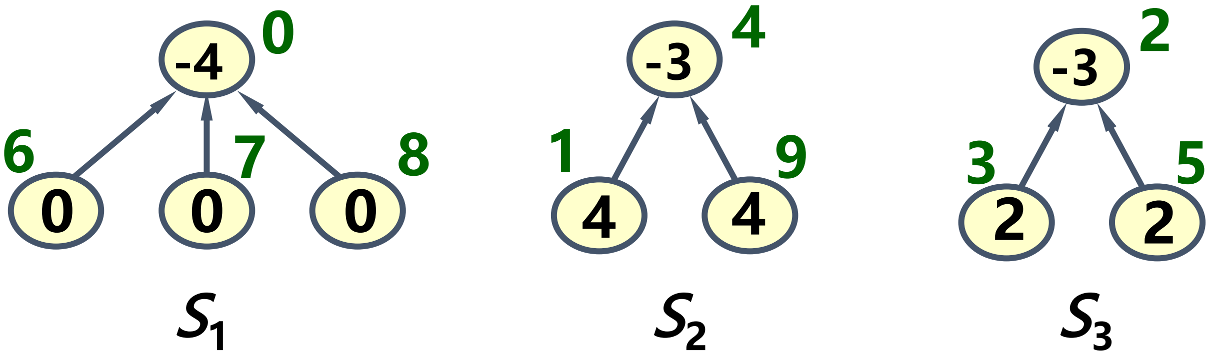
- 初始时, 用初始化函数 initUFSets(S) 构造一个森林, 每棵树只有一个结点, 表示集合中各元素自成一个子集合。

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。

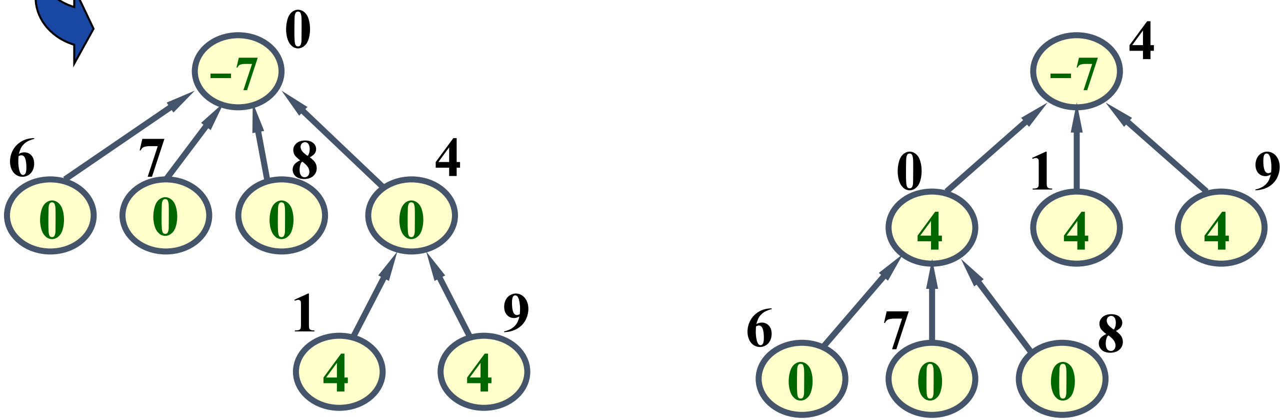
下标	0	1	2	3	4	5	6	7	8	9
parent	-4	4	-3	2	-3	2	0	0	0	4

集合 S_1, S_2 和 S_3 的双亲表示



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

```
const int DefaultSize = 10;

typedef struct UFSets {
    int *parent;      //集合元素数组(双亲表示)
    int size;         //集合元素的数目
};

void initUFSets (UFSets& S, int sz) {
    S.size = sz;                        //集合元素个数
    S.parent = new int[S.size];     //创建双亲数组
    for (int i = 0; i < S.size; i++) S.parent[i] = -1;
};
```

```
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;
};
void Union (UFSets& S, int Root1, int Root2) {
//求两个不相交集合Root1与Root2的并
     S.parent[Root1]  += S.parent[Root2];
     S.parent[Root2]  = Root1;
                    //将Root2连接到Root1下面
};
```