



第八章 二叉树的应用

哈夫曼树(Huffman)

二叉排序树



哈夫曼树(Huffman)

- 带权路径长度最短的树

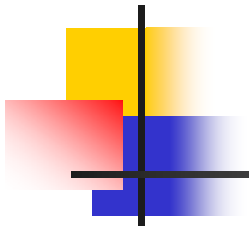


哈夫曼树(Huffman)

- - 带权路径长度最短的树

定义

- **路径**：从树中一个结点到另一个结点之间的分支构成这两个结点间的~
- **路径长度**：路径上的分支数
- **树的路径长度**：从树根到每一个结点的路径长度之和
在许多应用中，常常将树中结点赋予一个有某种意义的实数，称为该结点的**权**。
- **结点的带权路径长度**：是该结点到树根之间的路径长度与结点上权的乘积。
- **树的带权路径长度**：树中所有叶子结点(k)的带权路径长度 $\sum_k l_k$ 之和，

A decorative graphic on the left side of the slide, consisting of overlapping yellow, red, and blue squares with a black crosshair.

记作：

$$wpl = \sum_{k=1}^n w_k l_k$$

其中：

w_k 为权值

l_k 为结点到根的路径长度

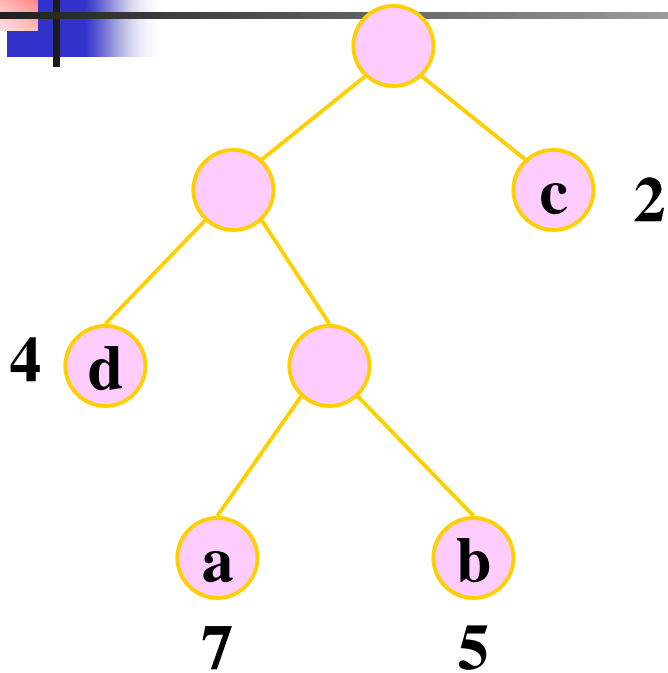
n 为叶子结点数

- **Huffman树**——设有 n 个权值 $\{w_1, w_2, \dots, w_n\}$ ，构造一棵有 n 个叶子结点的二叉树，每个叶子的权值为 w_i ，则 wpl 最小的二叉树叫~

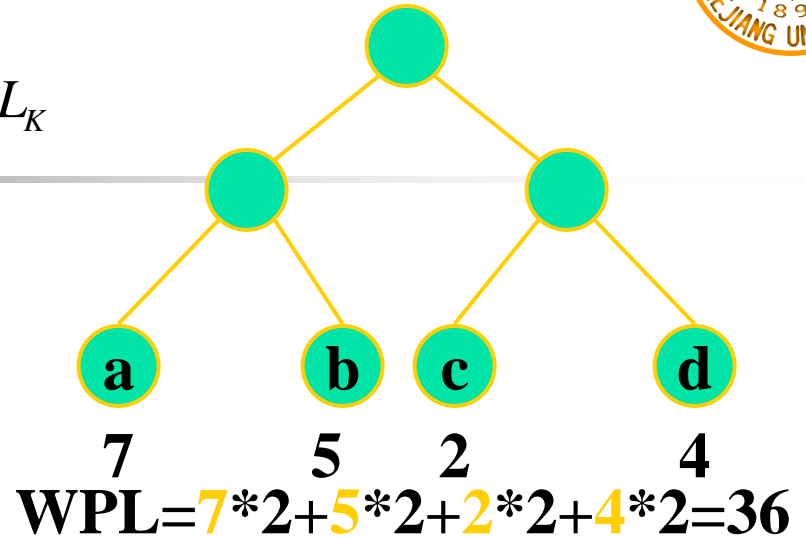


例 有4个结点，权值分别为7，5，2，4，构造有4个叶子结点的二叉树

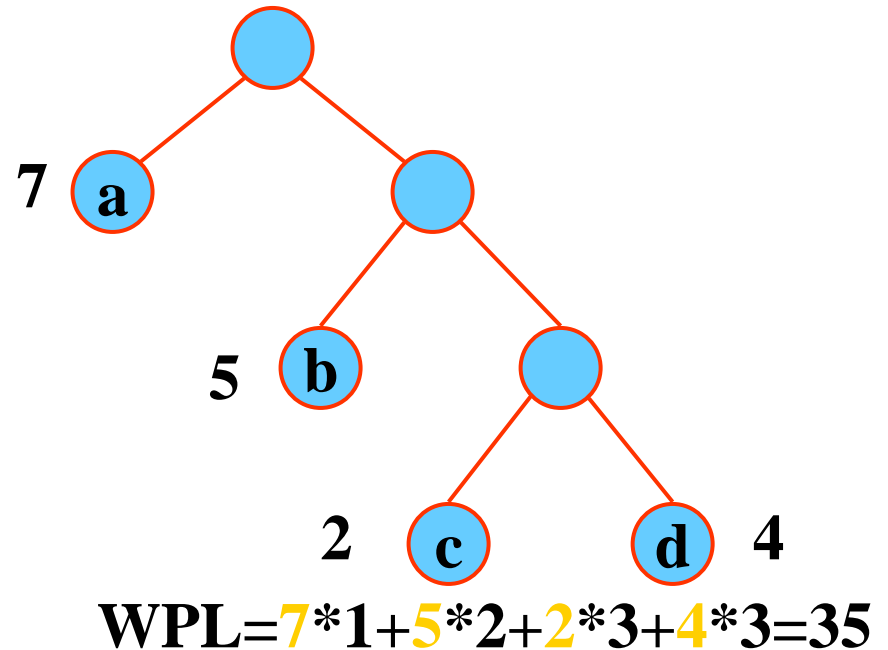
$$WPL = \sum_{k=1}^n W_K L_K$$



$$WPL = 7 * 3 + 5 * 3 + 2 * 1 + 4 * 2 = 46$$



$$WPL = 7 * 2 + 5 * 2 + 2 * 2 + 4 * 2 = 36$$



$$WPL = 7 * 1 + 5 * 2 + 2 * 3 + 4 * 3 = 35$$

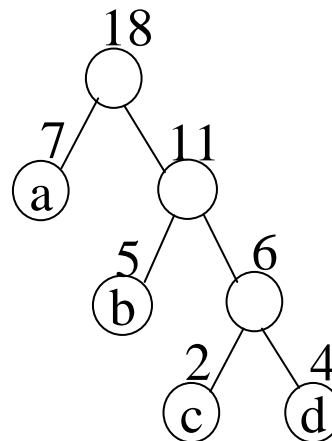
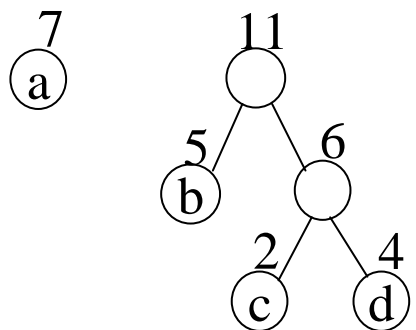
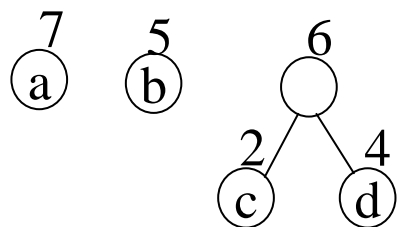
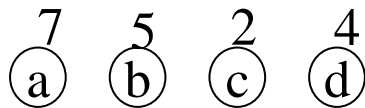


■ 构造Huffman树的方法——Huffman算法

■ 构造Huffman树步骤

- 根据给定的 n 个权值 $\{w_1, w_2, \dots, w_n\}$ ，构造 n 棵只有根结点的二叉树，令其权值为 w_j
- 在森林中选取两棵根结点权值最小的树作左右子树，构造一棵新的二叉树，置新二叉树根结点权值为其左右子树根结点权值之和
- 在森林中删除这两棵树，同时将新得到的二叉树加入森林中
- 重复上述两步，直到只含一棵树为止，这棵树即哈夫曼树

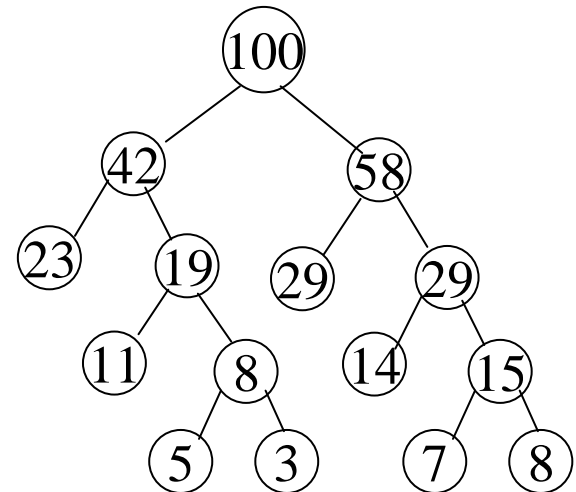
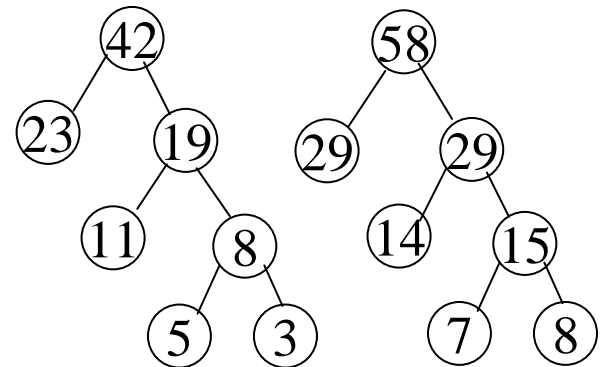
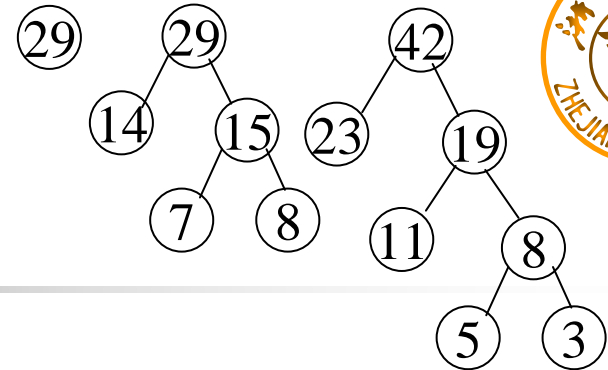
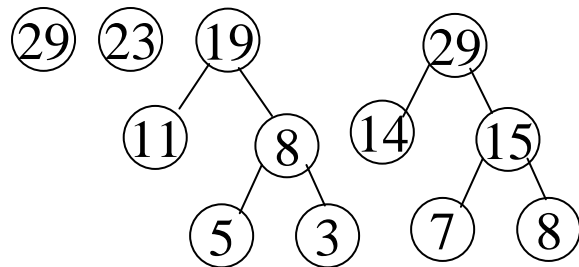
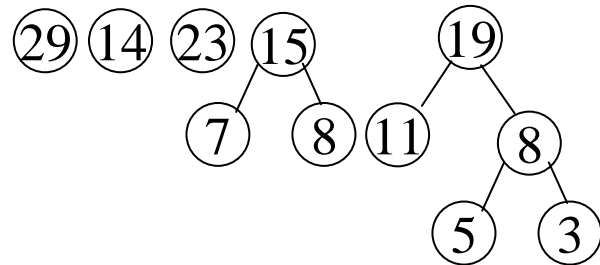
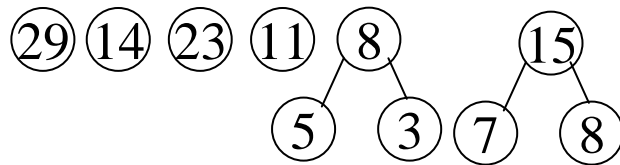
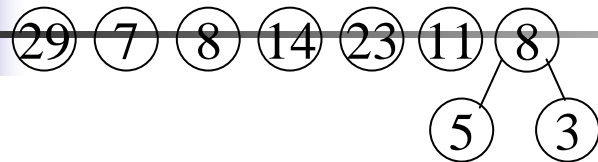
例





例 $w=\{5, 29, 7, 8, 14, 23, 3, 11\}$

5 29 7 8 14 23 3 11



赫夫曼树结点的存储结构



lchild	data 结点值	weight 权值	rchild	parent
--------	-------------	--------------	--------	--------

```
#define n                /*叶子数目*/  
#define m 2*n-1         /*结点总数*/  
typedef char datatype;  
typedef struct  
{float weight;  
  datatype data;  
  int lchild,rchild,parent;  
}hufmtree;  
hufmtree tree[m];
```



构造赫夫曼树的算法：

HUFFMAN(hufmtree tree[])

```
{ int i,j,p;
```

```
char ch;
```

```
float small1,small2,f;
```

```
for(i=0;i<m;i++)    /*初始化*/
```


```
{ tree[i].parent=0;
```

```
  tree[i].lchild=0;
```

```
  tree[i].rchild=0;
```

```
  tree[i].weight=0.0;
```

```
  tree[i].data='0';}
```

A decorative graphic in the top-left corner consisting of overlapping yellow, red, and blue squares with a black crosshair.

```
for(i=0;i<n;i++)          /*输入前n个结点的权值*/
{ scanf("%f",&f);
  tree[i].weight=f;
  scanf("%c",&ch);
  tree[i].data=ch;}
for(i=n;i<m;i++) /* 进行n-1次合并 , 产生n-1个新结点*/
{ p1=p2=0;
  small1=small2=Maxval; /* Maxval是float类型的最大值*/
```



```
for(j=0;j<=i-1;j++)
```

```
if(tree[j].parent==0)
```

```
if(tree[j].weight<small1)
```

/*改变最小权、

次小权及对应位置*/

```
{ small2=small1;
```

```
small1=tree[j].weight;
```

```
p2=p1;
```

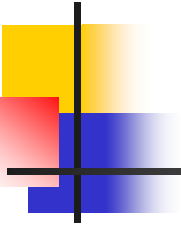
```
p1=j;}
```

```
else
```

```
if(tree[j].weight<small2) /*改变次小权及位置*/
```

```
{ small2=tree[j].weight;
```

```
p2=j;}
```



```
tree[p1].parent=i; /*给合并的两个结点的双亲域赋值*/  
tree[p2].parent=i;  
tree[i].lchild=p1;  
tree[i].rchild=p2;  
tree[i].weight= tree[p1].weight+ tree[p2].weight;  
}  
}
```

	lc	data	rc	pa
1	0	7	0	0
2	0	5	0	0
3	0	2	0	0
4	0	4	0	0
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0

(1)

	lc	data	rc	pa
1	0	7	0	0
2	0	5	0	0
3	0	2	0	5
4	0	4	0	5
i → 5	3	6	4	0
6	0	0	0	0
7	0	0	0	0

(2)

p1=3, p2=4
sm1=2, sm2=4

	lc	data	rc	pa
1	0	7	0	0
2	0	5	0	6
3	0	2	0	5
4	0	4	0	5
5	3	6	4	6
i → 6	2	11	5	0
7	0	0	0	0

(3)

p1=2, p2=5
sm1=5, sm2=6

	lc	data	rc	pa
1	0	7	0	7
2	0	5	0	6
3	0	2	0	5
4	0	4	0	5
5	3	6	4	6
6	2	11	5	7
i → 7	1	18	6	0

(4)

p1=1, p2=6
sm1=7, sm2=11



	lc	data	rc	pa
1	0	7	0	0
2	0	5	0	0
3	0	2	0	0
4	0	4	0	0
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0

(1)

i →

	lc	data	rc	pa
1	0	7	0	0
2	0	5	0	0
3	0	2	0	5
4	0	4	0	5
5	3	6	4	0
6	0	0	0	0
7	0	0	0	0

(2)

	lc	data	rc	pa
1	0	7	0	0
2	0	5	0	6
3	0	2	0	5
4	0	4	0	5
5	3	6	4	6
6	2	11	5	0
7	0	0	0	0

(3)

i →

	lc	data	rc	pa
1	0	7	0	7
2	0	5	0	6
3	0	2	0	5
4	0	4	0	5
5	3	6	4	6
6	2	11	5	7
7	1	18	6	0

(4)

i →



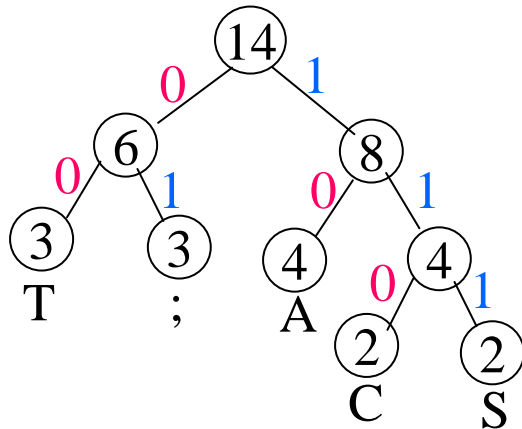
■ Huffman树应用

哈夫曼树中没有度为1的结点，称为**严格的二叉树**。

Huffman编码：数据通信用的二进制编码

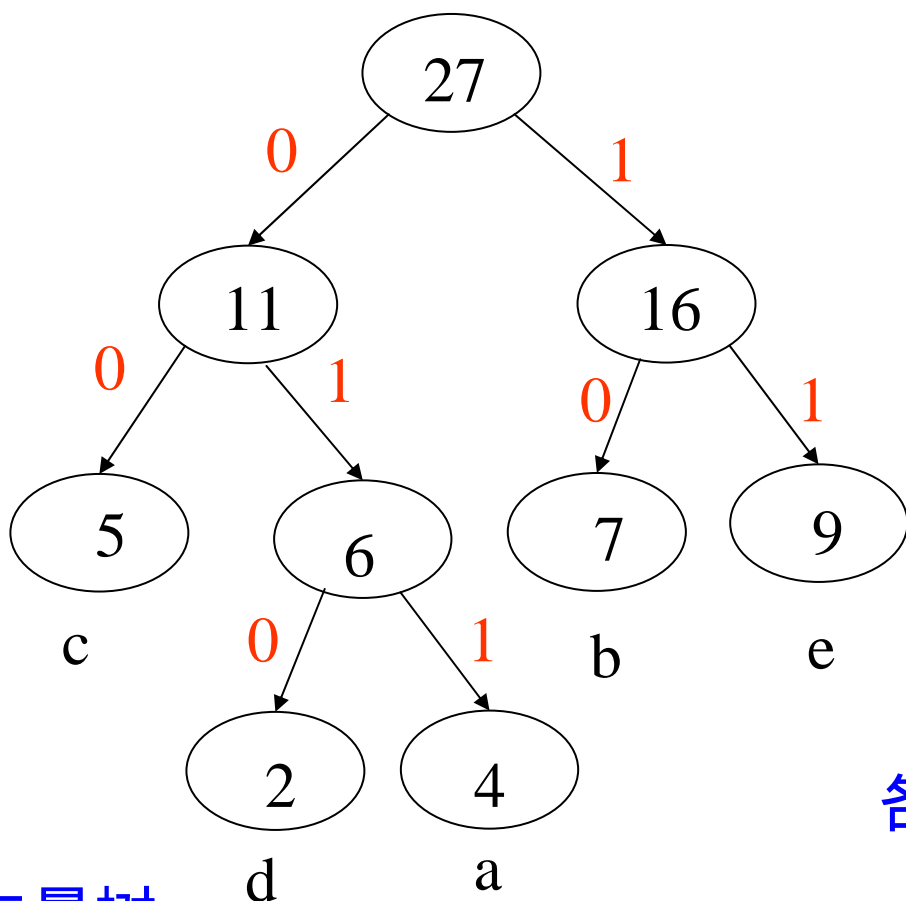
- 思想：根据字符出现频率编码，使电文总长最短
- 编码：根据字符出现频率构造Huffman树，然后将树中结点引向其左孩子的分支标“0”，引向其右孩子的分支标“1”；每个字符的编码即为从根到每个叶子的路径上得到的0、1序列

例 要传输的字符集 $D=\{C,A,S,T,;\}$
字符出现频率 $w=\{2,4,2,3,3\}$



T : 00
; : 01
A : 10
C : 110
S : 111

例：有一份电文中共使用5个字符：a、b、c、d、e，它们的出现频率依次为4、7、5、2、9，试画出对应的哈夫曼树(请按左子树根结点的权小于等于右子树根结点的权的次序构造)，并求出每个字符的哈夫曼编码。



a: 011

b: 10

c: 00

d: 010

e: 11

各字符对应的哈夫曼编码

哈夫曼树



具体的编码数组结构描述如下：

```
typedef char datatype;  
typedef struct  
{char bits[n];  
int start;  
datatype data;  
} codetype;  
codetype code[n];
```



编码算法的基本思想：

从叶子 $tree[i]$ 出发，利用双亲地址找到双亲结点 $tree[p]$ ，再利用 $tree[p]$ 的 $lchild$ 和 $rchild$ 指针域判断 $tree[i]$ 是 $tree[p]$ 的左孩子还是右孩子，然后决定分配代码是“0”还是“1”，然后以 $tree[p]$ 为出发点继续向上回溯，直到根结点为止。



哈夫曼编码算法：

```
HUFFMANCODE(codetype code[], hufmtree tree[])
```

```
/*code存放求出的哈夫曼编码的数组，
```

```
tree为已知的哈夫曼树*/
```

```
{ int i, c, p;
```

```
codetype cd;
```

```
/*cd为缓冲变量*/
```

```
for(i=0; i<n; i++)
```

```
/*n为叶子结点数目*/
```

```
{ cd.start=n;
```

```
c=i;
```

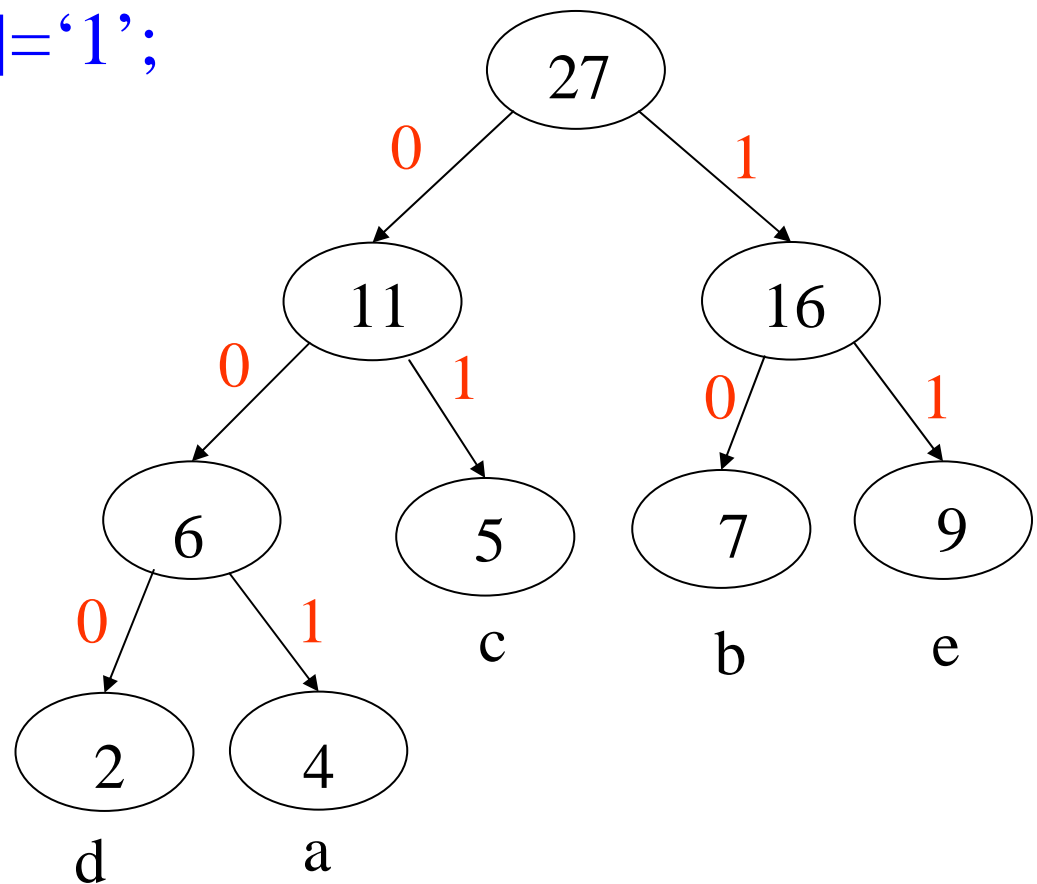
```
/*从叶子结点出发向上回溯*/
```

```
p=tree[c].parent;
```

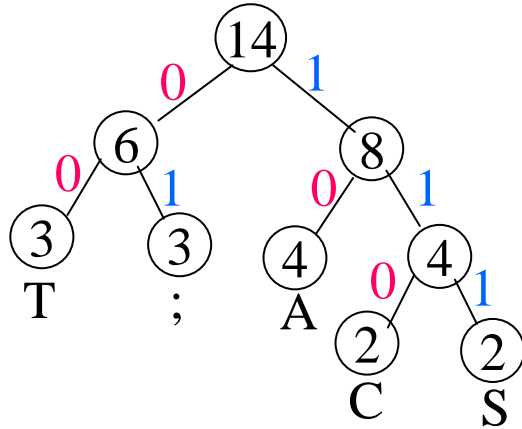
```
cd.data=tree[c].data;
```



```
while(p!=0)
{ cd.start --;
  if(tree[p].lchild==c) cd.bits[cd.start]='0';
  else cd.bits[cd.start]='1';
  c=p;
  p=tree[c].parent;
}
code[i]=cd;
}
}
```



- 译码：从Huffman树根开始，从待译码电文中逐位取码。若编码是“0”，则向左走；若编码是“1”，则向右走，一旦到达叶子结点，则译出一个字符；再重新从根出发，直到电文结束



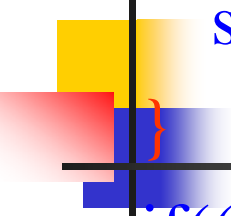
T : 00
; : 01
A : 10
C : 110
S : 111

例 电文是{CAS;CAT;SAT;AT}
其编码 “11010111011101000011111000011000”
电文为“1101000”
译文只能是 “CAT”

HUFFMANDECODE(codetype code[], hufmtree tree[])



```
{int i, c, p, b;
int endflag = -1;      /*电文结束标志取-1*/
i=m-1;                 /*从根结点开始向下收缩*/
scanf("%d", &b);       /*读入一个二进制代码*/
while(b!=endflag)
{ if(b==0) i=tree[i].lchild; /*走向左孩子*/
  else i=tree[i].rchild;     /*走向右孩子*/
  if(tree[i].lchild==0)      /*判断tree[i]是否是叶子结点
                             (因为是严格的二叉树) */
  { putchar(code[i].data);
    i=m-1; }                /*回到根结点*/
```

```
scanf("%d",&b);
```

```
/*读入下一个二进制代码*/
```

```
if((tree[i].lchild!=0)&&(i!=m-1))
```

```
/*电文读完尚未到叶子结点*/
```

```
printf("\n ERROR\n");
```

```
}
```



二叉排序树

- 定义：二叉排序树或是一棵空树，或是具有下列性质的二叉树：
 - 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值
 - 若它的右子树不空，则右子树上所有结点的值均大于或等于它的根结点的值
 - 它的左、右子树也分别为二叉排序树
- 二叉排序树的插入
 - 插入原则：若二叉排序树为空，则插入结点应为新的根结点；否则，继续在其左、右子树上查找，直至某个叶子结点的左子树或右子树为空为止，则插入结点应为该叶子结点的左孩子或右孩子
 - 二叉排序树生成：从空树出发，经过一系列的查找、插入操作之后，可生成一棵二叉排序树

A decorative graphic in the top-left corner consisting of overlapping yellow, red, and blue squares with a black crosshair.

在二叉排序树中**插入**新结点，只要保证插入后仍符合二叉排序树的定义即可：

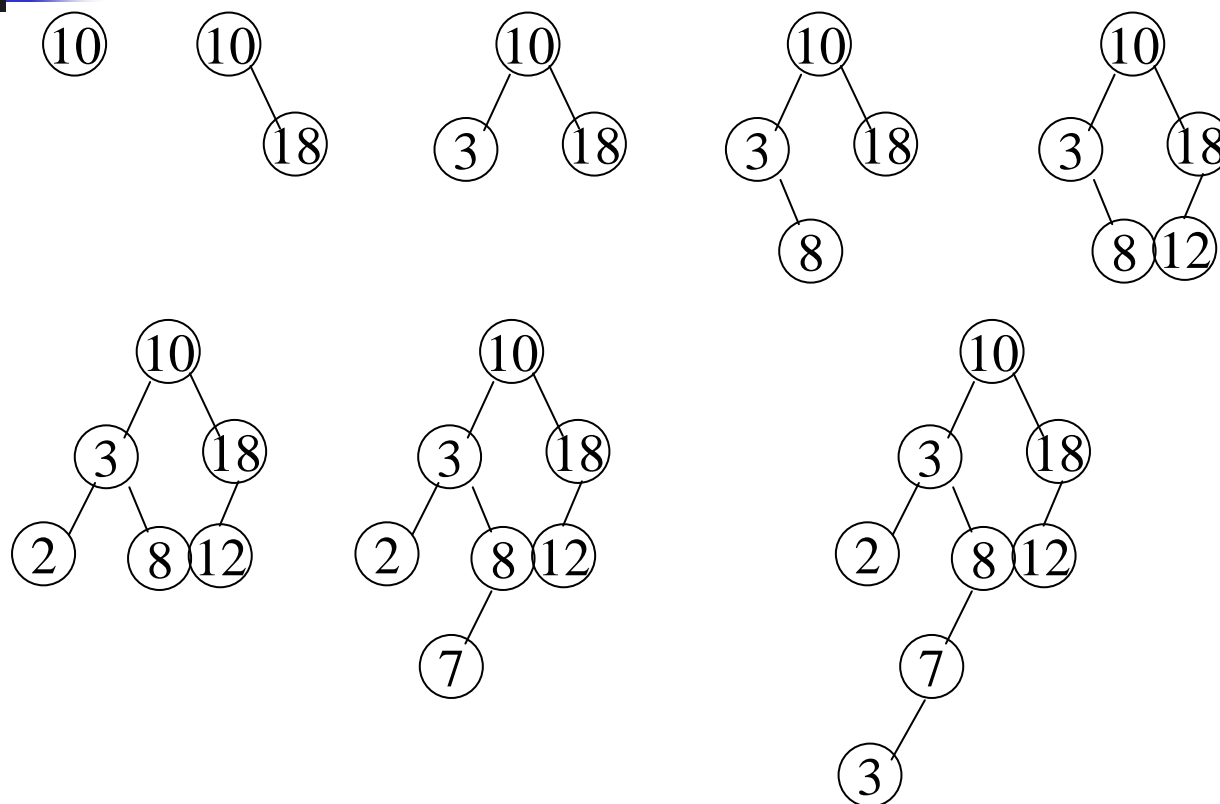
(1)若二叉排序树为空，则待插入结点*s作为根结点插入到空树中；

(2)当二叉排序树非空时，将待插结点的关键字 $s \rightarrow \text{key}$ 和树根的关键字 $t \rightarrow \text{key}$ 比较，若 $s \rightarrow \text{key} < t \rightarrow \text{key}$ ，则将待插结点*s插入到根的左子树中，否则将*s插入到根的右子树中；

(3)子树的插入过程相同。

■ 插入算法

例 {10, 18, 3, 8, 12, 2, 7, 3}



中序遍历二叉排序树可得到一个关键字的有序序列



使用二叉链表作为二叉排序树的存储结构：

```
typedef struct node
```

```
{ keytype key;    /* 关键字项 */
```

```
    datatype other; /* 其它数据项 */
```

```
    struct node *lchild, *rchild; /* 左、右指针 */
```

```
} bstnode
```



将新结点 *s 插入到 t 所指的二叉排序树中：

`bstnode *INSERTBST(t,s)` /*t为二叉排序树的根指针，
s为输入的结点指针*/

`bstnode *s, *t;`

`{bstnode *f, *p;`

`p=t;`

`while(p!=NULL)`

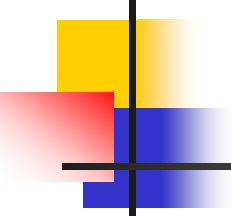
`{ f=p; /* 查找过程中，f 指向 *p 的双亲*/`

`if(s->key==p->key) return t; /* 树中已有结点*s，无须插入*/`

`if(s->key<p->key) p=p->lchild; /* 在左子树中查找插入位置*/`

`else p=p->rchild; /* 在右子树中查找插入位置*/`

`}`

A decorative graphic in the top-left corner consisting of overlapping yellow, red, and blue squares with a black crosshair.

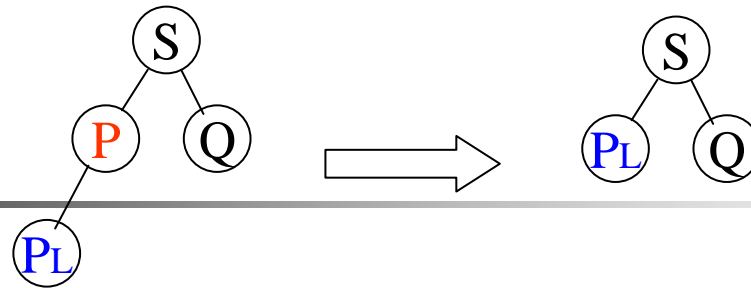
```
if(t==NULL) return s; /*原树为空，返回 s 作为根指针*/  
if(s->key<f->key) f->lchild=s; /* 将*s插入为*f的左孩子 */  
else f->rchild=s;      /* 将*s插入为*f的右孩子 */  
return t;  
}
```



二叉排序树的删除

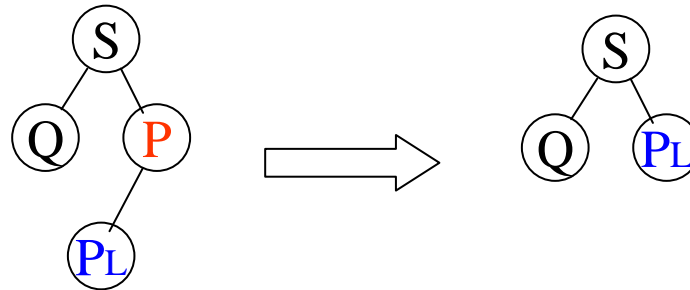
要删除二叉排序树中的p结点，分三种情况：

- p为叶子结点，只需修改p双亲f的指针f->lchild=NULL
f->rchild=NULL
- p只有左子树或右子树
 - p只有左子树，用p的左孩子代替p (1)(2)
 - p只有右子树，用p的右孩子代替p (3)(4)
- p左、右子树均非空
 - 沿p左子树的根C的右子树分支找到S，S的右子树为空，将S的左子树成为S的双亲Q的右子树，用S取代p (5)
 - 令p的左子树为f(f为p的双亲)的左子树，而p的右子树为S的右子树 (6)



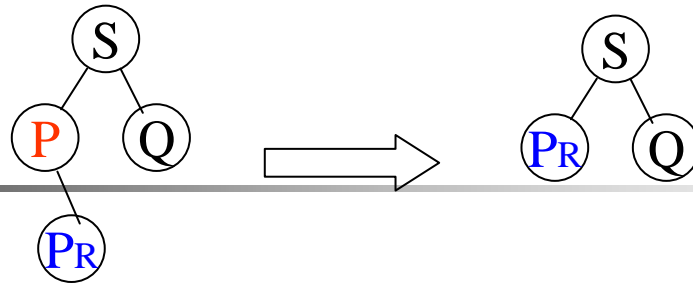
中序遍历 : P_L P S Q 中序遍历 : P_L S Q

(1)

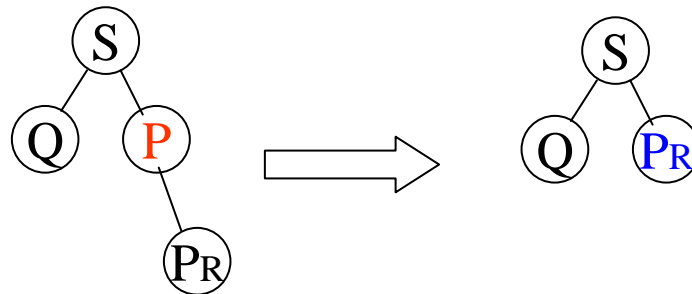


中序遍历 : Q S P_L P 中序遍历 : Q S P_L

(2)

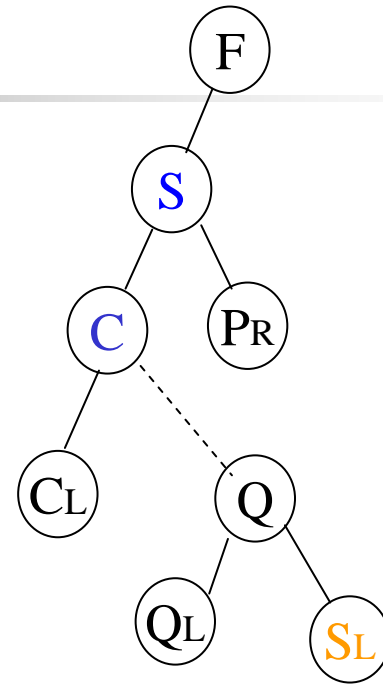
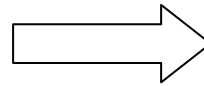
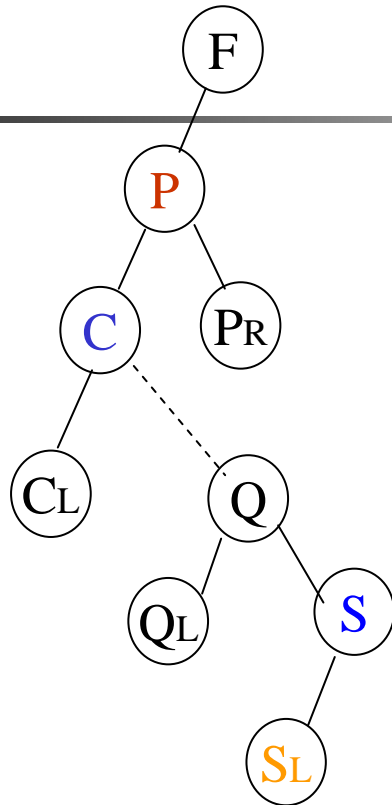


中序遍历 : P P_R S Q 中序遍历 : P_R S Q
(3)



中序遍历 : Q S P P_R 中序遍历 : Q S P_R
(4)

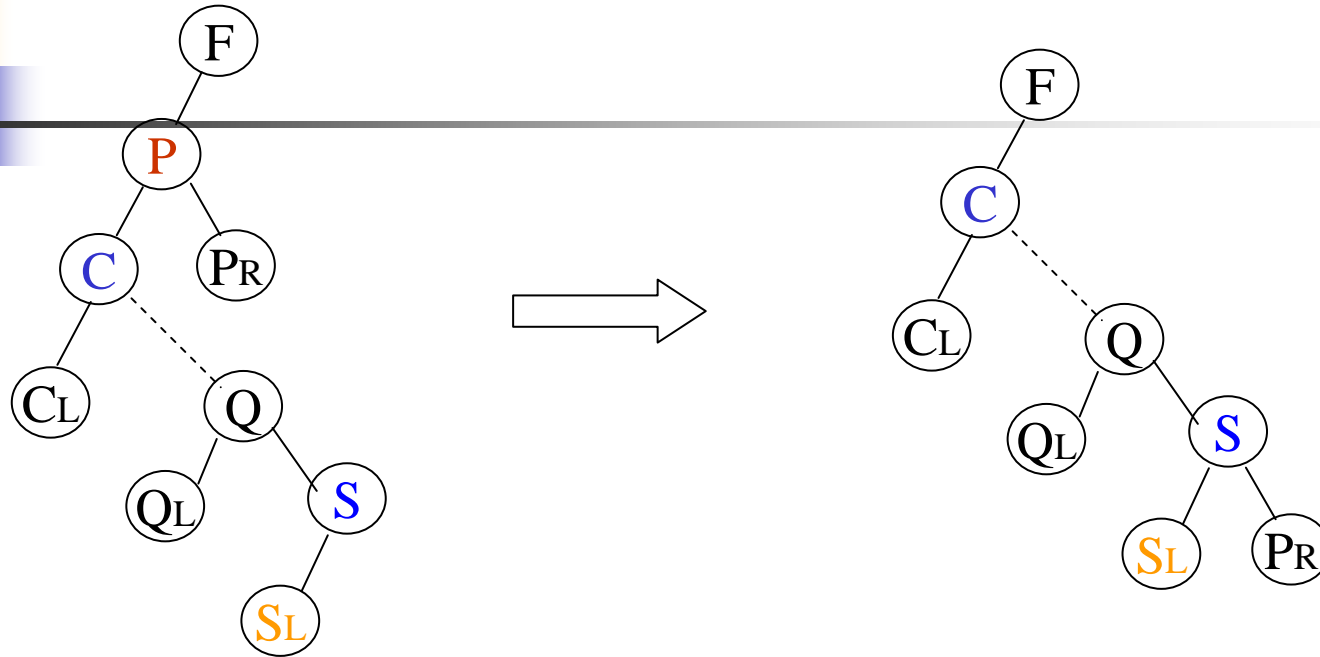
中序遍历： $C_L \ C \ \dots \ Q_L \ Q \ S_L \ S \ P \ P_R \ F$



(5)

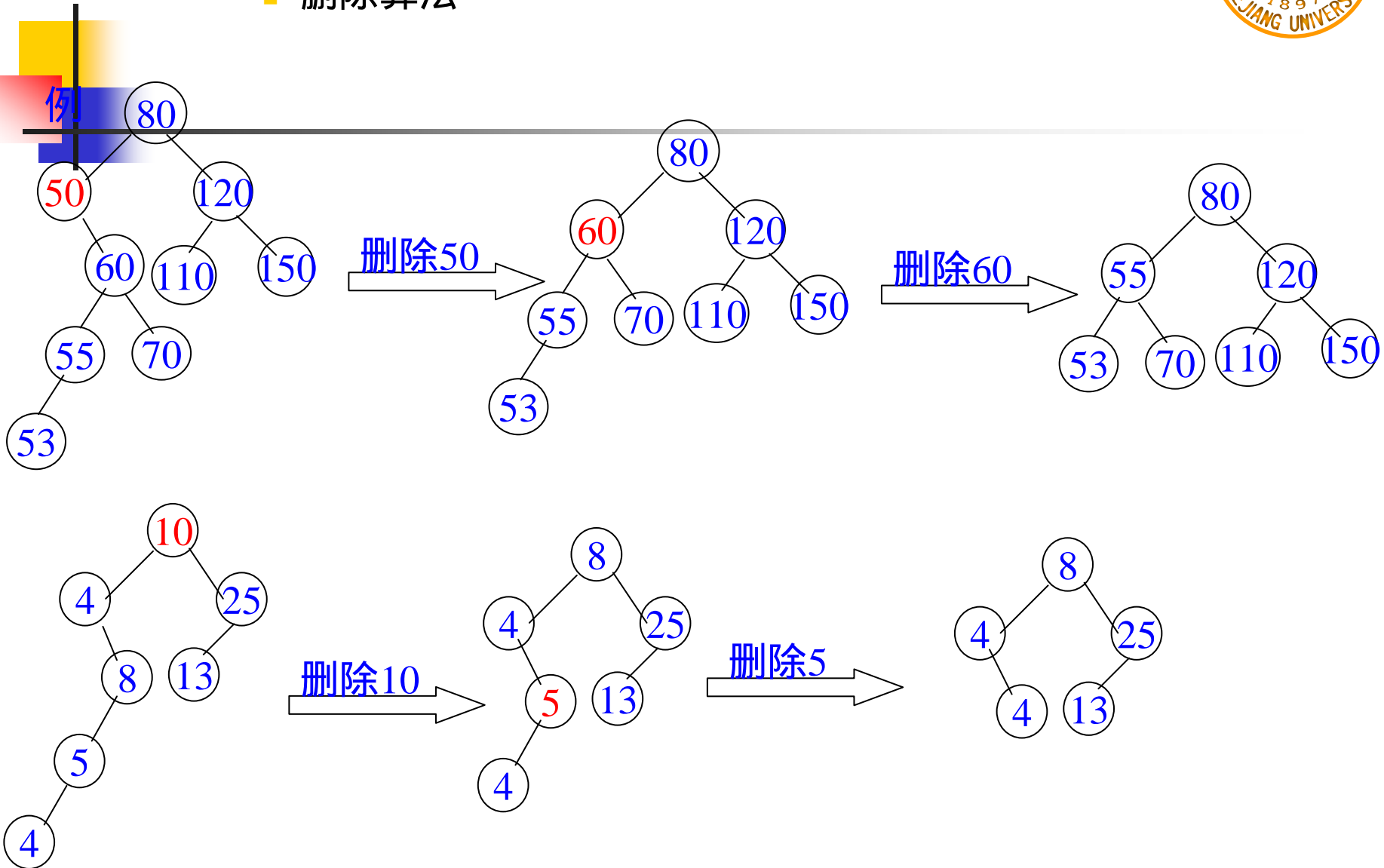
中序遍历： $C_L \ C \ \dots \ Q_L \ Q \ S_L \ S \ P_R \ F$

中序遍历 : $C_L \ C \ \dots \ Q_L \ Q \ S_L \ S \ P \ P_R \ F$



(6) 中序遍历 : $C_L \ C \ \dots \ Q_L \ Q \ S_L \ S \ P_R \ F$

■ 删除算法





Q&A
