Ch6 树和二叉树

- 有关树的概念

树的定义

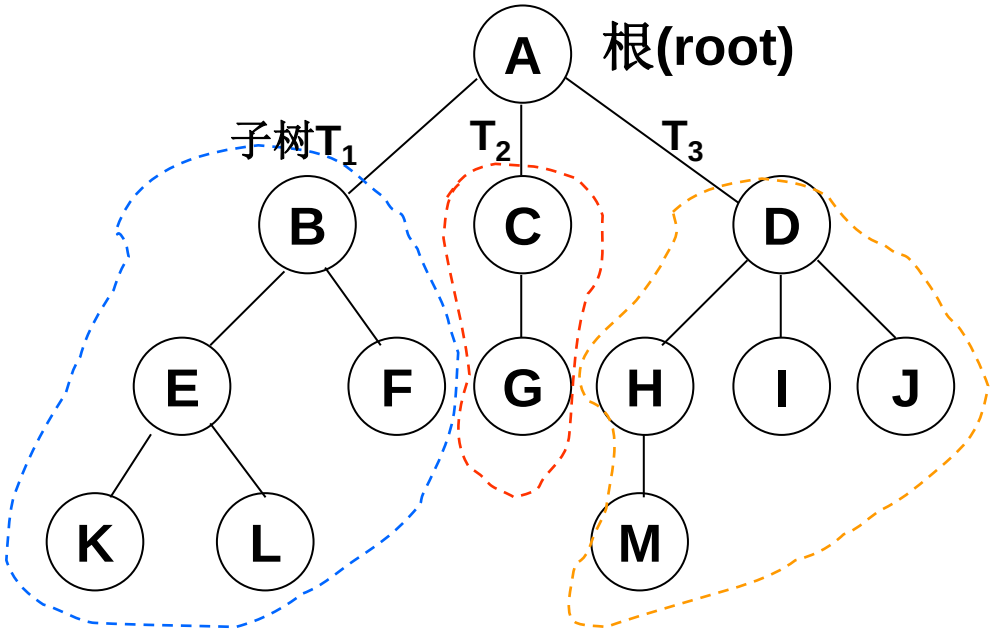
树(tree)是n(n≥0)个结点的有限集T, 对任一非空树:

- **有且仅有一个特定的结点, 称为树的根(root);**
- **当n>1时, 其余结点可分为m(m>0)个互不相交的有限集 T_1、T_2、…、T_m, 其中每一个集合本身又是一棵树, 称为根的子树(subtree).**

Φ

空树

A 只有根结点的树



有子树的树

- 有关树的概念

树的常用术语

- 结点(node) 表示树中的一个元素
- 结点的度(degree) 结点拥有的非空子树(分支)的数量
  树的度 一棵树中所有结点的度的最大值
- 叶子(leaf) 度为0的结点
- 结点的孩子(child) 一个结点的子树的根称为该结点的孩子
  孩子的双亲(parent) 该结点本身
- 兄弟(sibling) 同一双亲的孩子间的互称
  结点的子孙(descendant) 以某结点为根的子树中的任一结点
- 结点的层次(level) 从根结点算起, 根结点为第1层, 其孩子为第2层 …
  若某结点处在第L层, 则其子树的根为第L+1层
- 深度/高度(depth/height) 树中结点的最大层次数, 只有一个结点的树其高度为1
- 森林(forest) m(m≥0)棵互不相交的树的集合

- 有关树的概念

树的常用术语例示说明

结点A的度：3
结点B的度：2
结点M的度：0

树的度：3

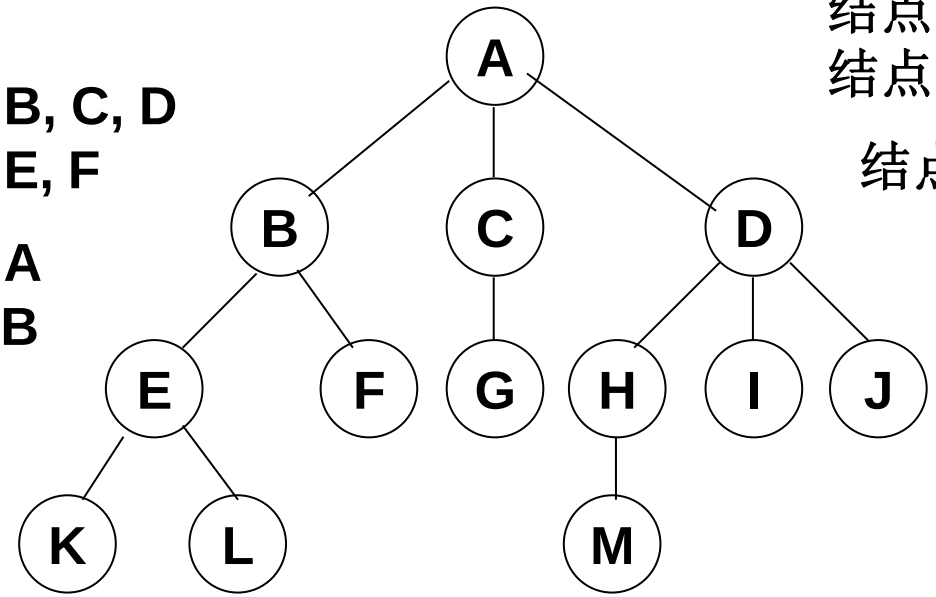
叶子：K, L, F, G, M, I, J

结点B, C, D为兄弟
结点K, L为兄弟

结点A的孩子：B, C, D
结点B的孩子：E, F

结点D的子孙：H, I, J, M

结点C的双亲：A
结点E的双亲：B



树的深度：4

结点A的层次：1
结点M的层次：4

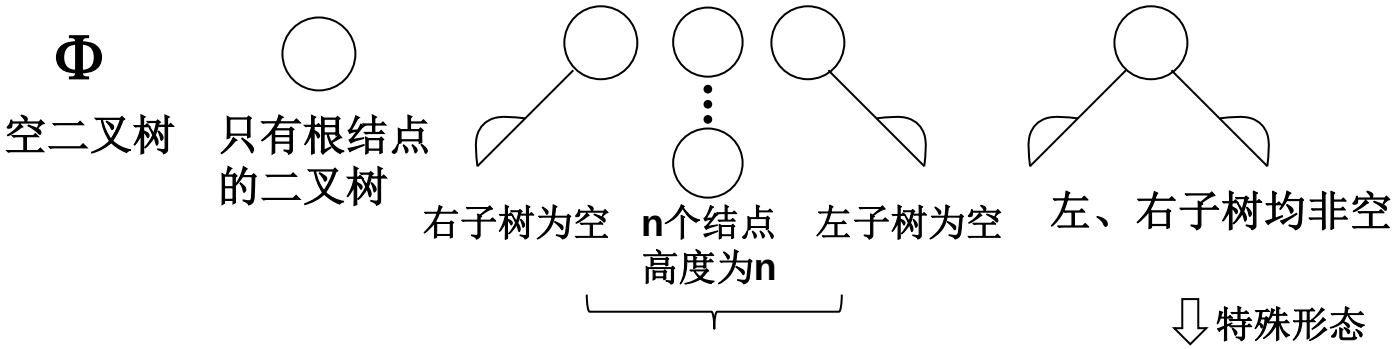
结点F, G为堂兄弟
结点A是结点F, G的祖先

- 二叉树(binary tree)的定义及性质

二叉树的定义　是n(n≥0)个结点的有限集, 它或为空树(n=0), 对非空树有:

- 有一个称之为根的结点
- 除根结点外的其余结点分为两棵互不相交的分别称为左子树(left)和右子树(right)的二叉树构成

二叉树的形态(types of binary trees) /
二叉树的计数(enumeration of binary trees)



A degenerate / pathological tree

- **Full Binary Tree**
- **Complete Binary Tree**
- **Balanced Binary Tree**

- 二叉树(binary tree)的定义及性质

两种特殊形态的二叉树 -- Full Binary Tree、Complete Binary Tree

满二叉树的含义及特点[注: 一些文献又称为完美二叉树 (Perfect Binary Tree)]

定义 深度为k, 且含有$2^k - 1$个结点的二叉树

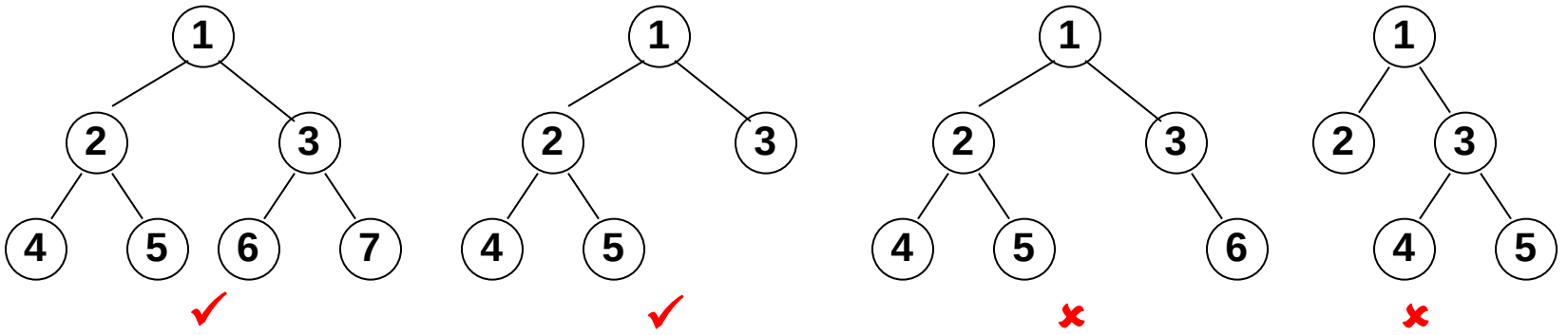
$$(= 2^0 + 2^1 + \cdots + 2^{k-1} = \sum_{i=1}^{k}(\text{第}i\text{层上的最大结点数}) = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1)$$

特点 每一层上的结点数都是最大值2^{i-1}, 满二叉树中没有度=1的结点, 叶子结点在同一层

完全二叉树的含义及特点

定义 深度为k有n个结点的二叉树, 当且仅当其每一个结点都与深度为k的满二叉树中编号从1至n的结点一一对应 ($n \le 2^k - 1$).

特点
1. 满二叉树是完全二叉树, 但完全二叉树不一定是满二叉树
2. 完全二叉树中度=1的结点个数或为0个或为1个.
3. 叶结点只可能在层次最大的两层上出现, 即至多只有最下面两层上结点的度可以小于2, 且最下层上的结点均集中在该层最左边的若干位置上
4. 对任一结点, 若其右分支下子孙的最大层次为L, 则其左分支下子孙的最大层次必为L或L+1



- 二叉树**(binary tree)**的定义及性质

二叉树的性质

- 二叉树的第**i(i≥1)**层上最多有2^{i-1}个结点**[**用归纳法证明**]**
  The maximum number of nodes at level 'i' of a binary tree is 2^{i-1}
  (第**1**层上有$2^{1-1}=1$个**,** 第**i-1**层上至多有2^{i-2}个**,** 则第**i**层**2** × $2^{i-2} = 2^{i-1}$ **)**

- 深度为**h (h≥1)**的二叉树最多有$2^h - 1$个结点**[~** 满二叉树**]**
  Maximum number of nodes in a binary tree of height 'h' is $2^h - 1$

  ($= 2^0 + 2^1 + \cdots + 2^{h-1} = \sum_{i=1}^{h}$(第**i**层上的最大结点数) $= \sum_{i=1}^{h} 2^{i-1} = 2^h - 1$ **)**

- 在非空二叉树中**,** 若叶结点**(**即度为**0)**数为n_0**,** 度为**1**结点数为n_1**,** 度为**2**结点数为n_2**,** 则有$n_0 = n_2+1$
  In Binary tree, number of leaf nodes is one more than nodes with two children.

  证明**:**
  设二叉树的总结点数为**n,** 则 $n = n_0+ n_1+ n_2$ ①
  又因为含 **n** 个结点的二叉树中除根结点外**,** 其余 **n-1** 个结点
  均各有一条分支指向它**,** 即含 **n** 个结点的二叉树有 **n-1**条分支**:**
  $n-1 = n_0\times0 + n_1\times1+ n_2\times2$ ②
  由①②得**:** $n_0 = n_2+1$

- 二叉树(binary tree)的定义及性质

二叉树的性质

• 含有n个结点的完全二叉树的深度是$\lfloor \log_2 n \rfloor + 1$ [$= \lceil \log_2 n \rceil$]

当$\log_2 n$为非整数时,

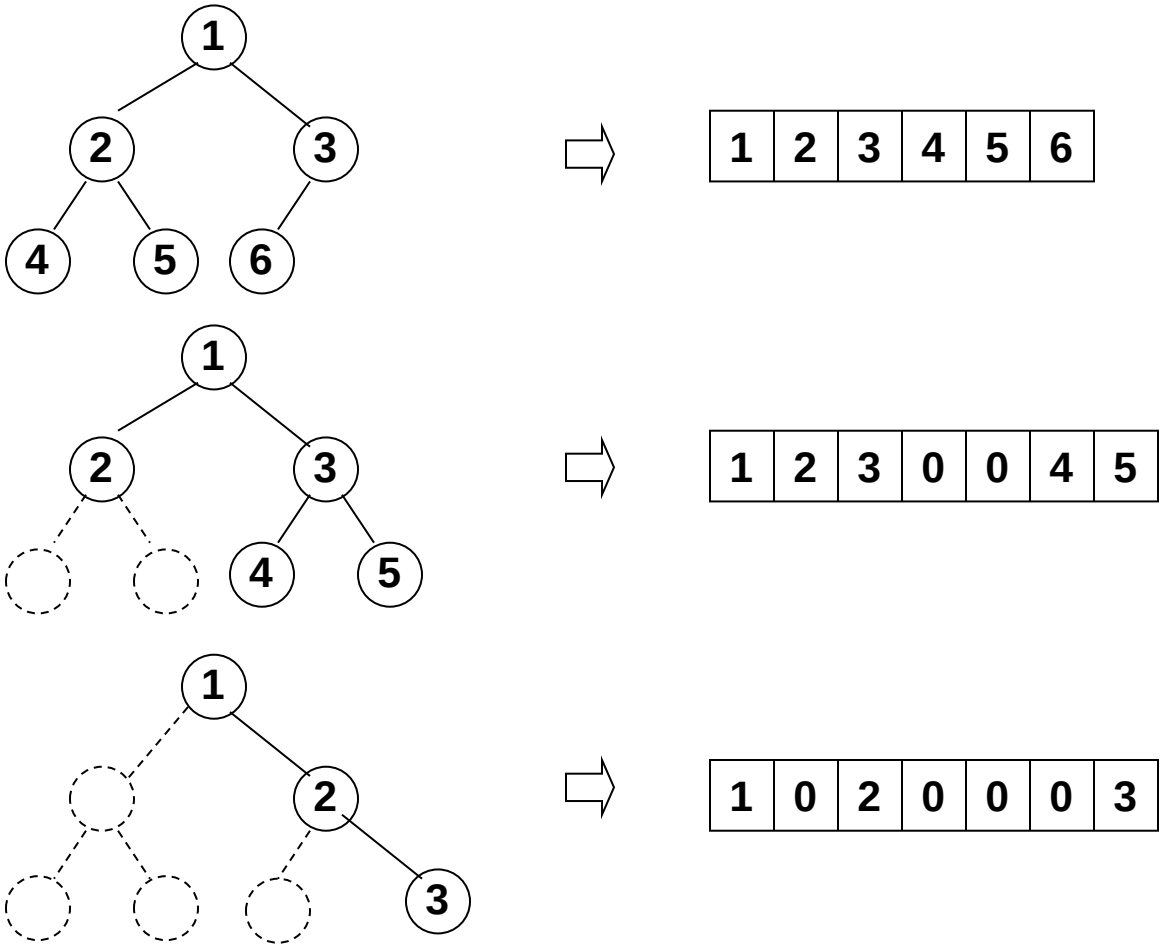
(设该完全二叉树的深度为k, 则根据完全二叉树的定义可知,
其前 k–1 层是深度为 k–1 的满二叉树, 共有 $2^{k-1} - 1$ 个结点;
又由性质2有 $n \le 2^k - 1$, 综合得: $2^{k-1} - 1 < n \le 2^k - 1$
➜ $2^{k-1} < n + 1 \le 2^k$ ➜ $2^{k-1} \le n < 2^k$ ➜ $k - 1 \le \log_2 n < k$
因为 k – 1 和 k 是两个相邻的整数, 故有 $k - 1 = \lfloor \log_2 n \rfloor$
由此得: $k = \lfloor \log_2 n \rfloor + 1$)

• 若对一棵有n个结点的完全二叉树的结点按层序编号, 则对任一编号为i(1≤i≤n)的结点, 有:

1) 若i=1, 则结点i是二叉树的根, 无双亲; 若i>1, 则其双亲是$\lfloor i/2 \rfloor$;
2) 若n<2i, 则结点i无左孩子; 若n≥2i, 则其左孩子是2i;
3)若n<2i+1, 则结点i无右孩子; 若n≥2i+1, 则其右孩子是2i+1.

- 二叉树的存储结构

顺序结构　用一维数组依次自上而下、从左到右存储完全/满二叉树上的结点.



结论: 深度为k的二叉树, 若k个结点, 则该树退化为一条"单链", 却仍然需要 2^k-1个存储单元. 因此, 顺序存储结构只适合于完全/满二叉树.

- 二叉树的存储结构

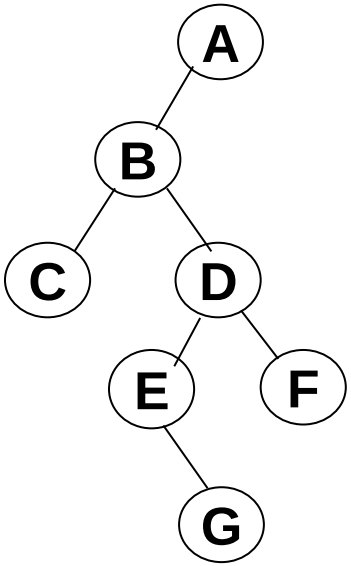
链式结构 --- 二叉链表(binary linked tree)、三叉链表

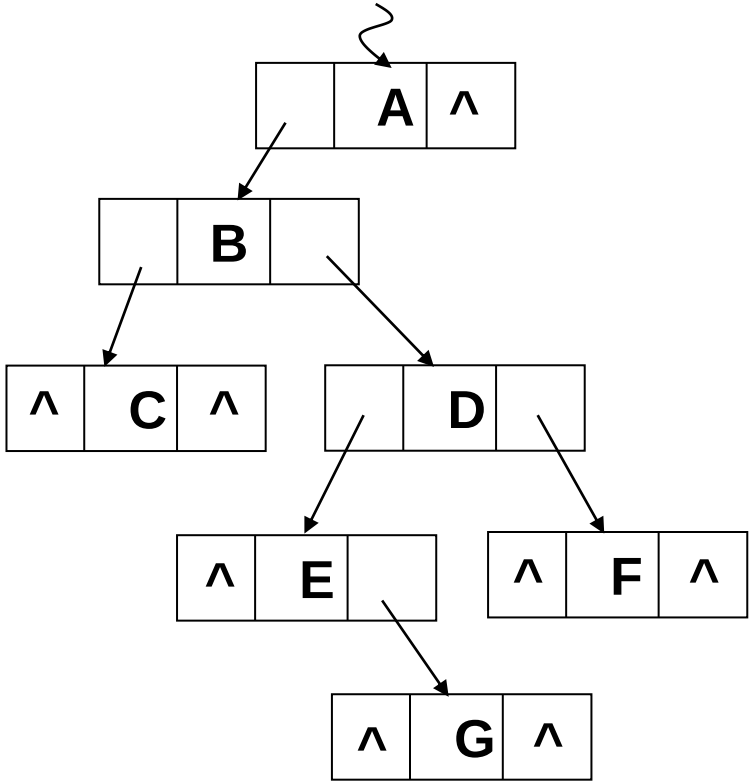
二叉链表法的每个结点:

- 两个指针域: 分别指向其左、右孩子结点
- 一个数据域: 存放数据元素本身信息

lchild	data	rchild

```
typedef struct btnode{
    BTElemType data;
    BTNode *lchild, *rchild;
} BTNode, *BTree;
```





- 二叉树的存储结构

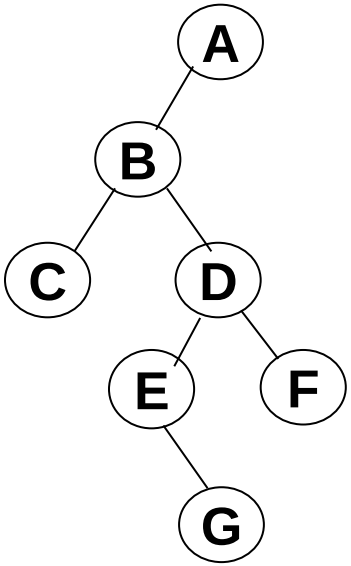
链式结构 --- 二叉链表(binary linked tree) 、三叉链表

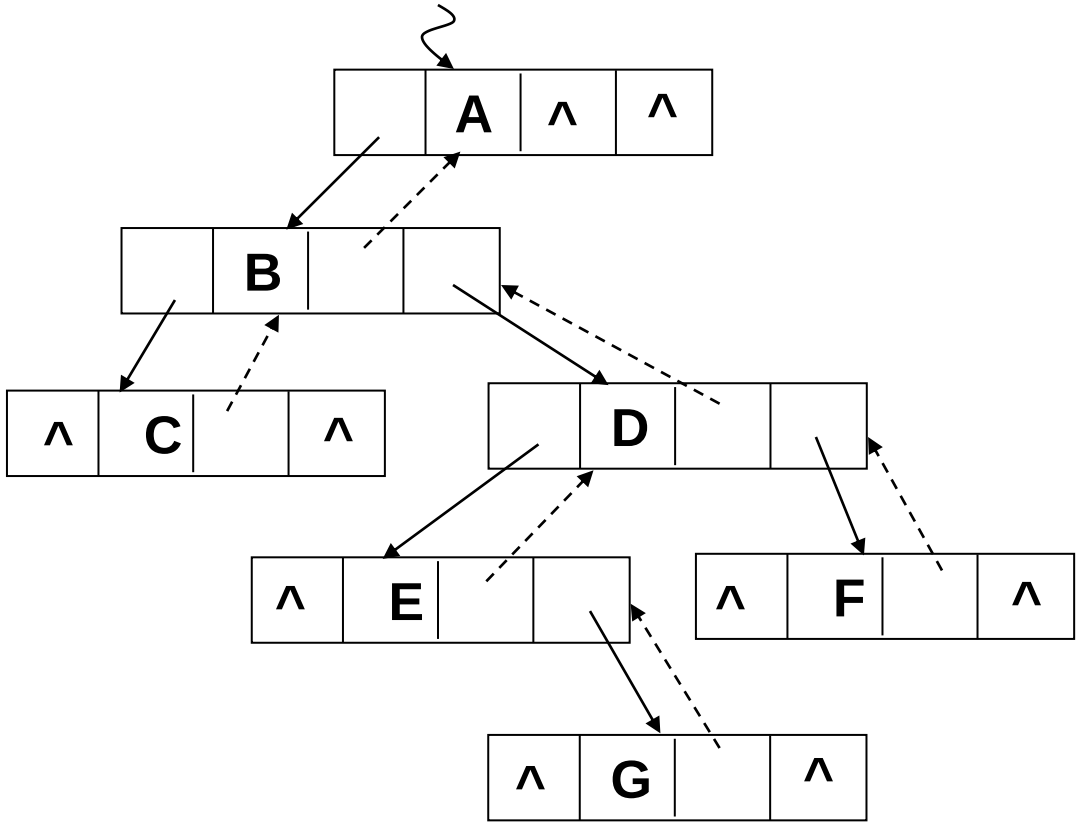
三叉链表法的每个结点:

- 三个指针域: 有一个指针指向其双亲结点
- 一个数据域: 存放数据元素本身信息

lchild	data	parent	rchild

```
typedef struct btnode{
    BTElemType data;
    BTNode *lchild,*rchild,*parent;
} BTNode, *BTree;
```





- 二叉树的遍历(Traversal of Binary Trees)

遍历 按一定*次序* *访问*二叉树中的所有结点, 且每个结点仅被访问一次.

→ 其具体操作依赖于应用，如输出、更新、验证等. **P129/A.6.1: Visit()**

对线性结构, 则只有一种逻辑上的次序: 顺序
对非线性结构, 则有两种逻辑上的次序: 深度优先 和 广度优先

设 **V – visiting the root**
L – traversing the left subtree
R – traversing the right subtree

6 ways

VLR (preorder traversal)
LVR (inorder traversal)
LRV (postorder traversal)
VRL RVL RLV

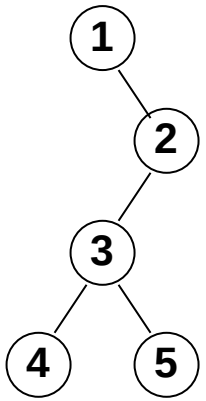
深度优先

VLR(先序/先根遍历) 结果: 1 2 3 4 5
LVR(中序/中根遍历) 结果: 1 4 3 5 2
LRV(后序/后根遍历) 结果: 4 5 3 2 1

深度优先 ⟺

层序遍历结果: 1 2 3 4 5

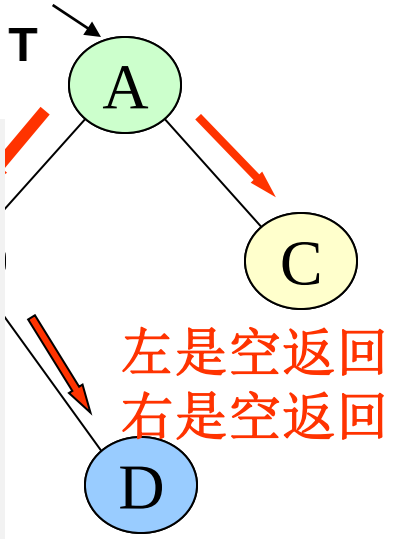
广度优先 ⟺



遍历的算法描述(递归 或 迭代) ⇨

- 二叉树的遍历(Traversal of Binary Trees)

```
void Postorder(BTNode *root, void (*Visit)(BTElemType x))
{  if (root) {
      Postorder(root->lchild);
      Postorder(root->rchild);
      Visit(root->data);
   }
}
```



主程序

pre(T)

后序序列：D B C A

先序序列：A B D C

中序序列：B D A C

T → A
printf(A);
pre(T→L);
pre(T→R);

T → B
printf(B);
pre(T→L);
pre(T→R);

T → C
printf(C);
pre(T→L);
pre(T→R);

T → D
printf(D);
pre(T→L);
pre(T→R);

T → ∧
返回

T → ∧
返回

T → ∧
返回

T → ∧
返回

左是空返回
右是空返回

- 二叉树的遍历(Traversal of Binary Trees)

二叉树遍历的时间复杂度T(n) = ? (以递归描述算法为例)

该时间复杂度可定义为: $T(n) = T(k) + T(n - k - 1) + c$
其中: k和n-k-1分别为二叉树根结点左右子树结点的个数;
c(或为1或为O(1))为递归调用之外所需要的时间开销.

1. 最坏情况: 一个子树为空(k=0)，而另一个子树为非空

$T(n) = T(0) + T(n-1) + c = T(0) + (T(0) + T(n-2) + c) + c = 2T(0) + T(n-2) + 2c$
$= 3T(0) + T(n-3) + 3c = 4T(0) + T(n-4) + 4c = \ldots = (n-1)T(0) + T(1) + (n-1)c = nT(0) + (n)c$
这里T(0)可看作是一个常量值(即遍历一个空树的时间开销).
➔ $T(n) = n(c+d) = O(n)$

2. 假设左右子树中含有相同的结点个数

$T(n) = 2T(n/2) + c = 2[2T(n/4) + c] + c = 4T(n/4) + 3c = 4[2T(n/8) + c] + 3c = 8T(n/8) + 7c =$
$16T(n/16) + 15c = \ldots = 2^k T(n/2^k) + (2^k -1) c$
$\because n = 2^k$ or $n/2^k = 1$ and $T(1) = d$
$\therefore T(n) = 2^k T(n/2^k) + (2^k -1) c = n T(1) + (n-1) c = n d + (n-1) c = O(n)$

二叉树遍历的辅助空间复杂度S(n) = ? (以递归描述算法为例)

辅助空间: 假设不考虑递归算法所需的隐式栈的空间占用, 则为O(1). 否则:
最坏情况下所需的额外辅助存储空间为O(n);
一般情况下为O(h), h为二叉树的最大高度.

- 二叉树的遍历(Traversal of Binary Trees)

中序遍历的基本思想(non-recursive version)

1. 设置一个栈s, 存放指向*根结点*的指针;
2. 第一次遇到根结点时并不访问, 而是入栈, 然后中序遍历其左子树;
3. 左子树遍历结束后, 第二次遇到根结点, 则将根结点(指针)退栈并访问该结点, 然后中序遍历其右子树;
4. 退栈时, 若栈空则结束.

其算法描述P130/A. 6.2 or P131/A. 6.3

```
//Iterative version: P131/A. 6.3
void Inorder(BTNode *root, void (*Visit)(BTElemType x))
{  p = root;
   while ( p || !StackEmpty(s) ) {
       if (p) { Push(s, p); p = p->lchild;  } //根指针入栈, 遍历左子树
       else { p = Pop(s);  //根指针退栈
              Visit(p->data); //访问根结点
              p = p->rchild;  //继续遍历其右子树
        } //end of else
    } //end of while
 }
```

- 二叉树的遍历(Traversal of Binary Trees)

后序遍历的基本思想(non-recursive version):

和中序遍历相比, 前两次经过根结点时均不访问它, 只有在第三次遇到根结点时才出栈且访问它. 因此需要再另设一辅助栈s1记录经过根结点的次数.

```
void PostOrder(BTNode *root, void (*Visit)(BTElemType x))
{  p = root;
   while (p || !StackEmpty(s)) {
      if (p) { Push(s, p); Push(s1, 1);   // 第一次经过, 入栈, 继续遍历其左子树
               p = p->lchild; }
      else if (Pop(s1) == 1) { Push(s1, 2);  // 第二次经过
                               p = GetTop(s);  // 且不退栈, 继续遍历其右子树
                               p = p->rchild; }
           else { q = Pop(s);  // 第三次经过, 退栈并访问
                  Visit(q->data);  Pop(s1); }
   }
}
```

- 二叉树的遍历(Traversal of Binary Trees)

按层遍历(Traversal of binary trees by levels)

-- 也称二叉树的广度优先遍历(Breadth first traversal for the binary tree)

Method Idea: Use a queue to keep track of the children of a node until it is time to visit them.

```
void LevelOrder(BTNode *root, void (*Visit)(BTElemType x))
{  BTNode *q[MAX];  //设辅助变量: 队列—指针数组, 树中结点数≤MAX
   front = rear  = 0;  p = root;
   if (p) { rear++; q[rear] = p;
       while (front != rear) {
           front++;  p = q[front]; //队首元素出列
           Visit(p->data);
           if (p->lchild) {rear++; q[rear] = p->lchild;}
           if (p->rchild) {rear++; q[rear] = p->rchild;}
       }
    }
} //Time Complexity: O(n) where n is number of nodes in the binary tree
```

- 二叉树的遍历(Traversal of Binary Trees)

遍历二叉树的其它操作

“遍历”是二叉树各种操作运算的基础.

- 统计二叉树中叶结点的个数n

```
int Count(BTNode *t)
{ if !t return 0;
  else if (!t->lchild && !t->rchild)  return 1; //为叶结点
       else return Count(t->lchild) + Count(t->rchild); } //左子树叶结点个数+右…
```

- 计算二叉树的深度depth

```
int Depth(BTNode *t)
{ if (!t) return 0;
  else { hl = Depth(t->lchild); //计算左子树的深度
         hr = Depth(t->rchild); //计算右子树的深度
         if ( hl > hr) return hl +1;  else return hr +1; } }
```

- 按需求创建一棵二叉树 P131/A.6.4

```
BTNode *CreateBTree(BTNode *t)  //空格字符表示空树
{ scanf(ch);
  if ( ch == ‘ ’ )  t = Null;
  else { t = (BTNode *)malloc(sizeof(BTNode));
         t->data = ch;
         t->lchilid = CreateBTree(t->lchild);
         t->rchild = CreateBTree(t->rchild); }
  return t;
}
```

- 二叉树的遍历(Traversal of Binary Trees)

遍历二叉树的其它操作

➢ **Polish Notation(表达式的波兰表示法)P129**

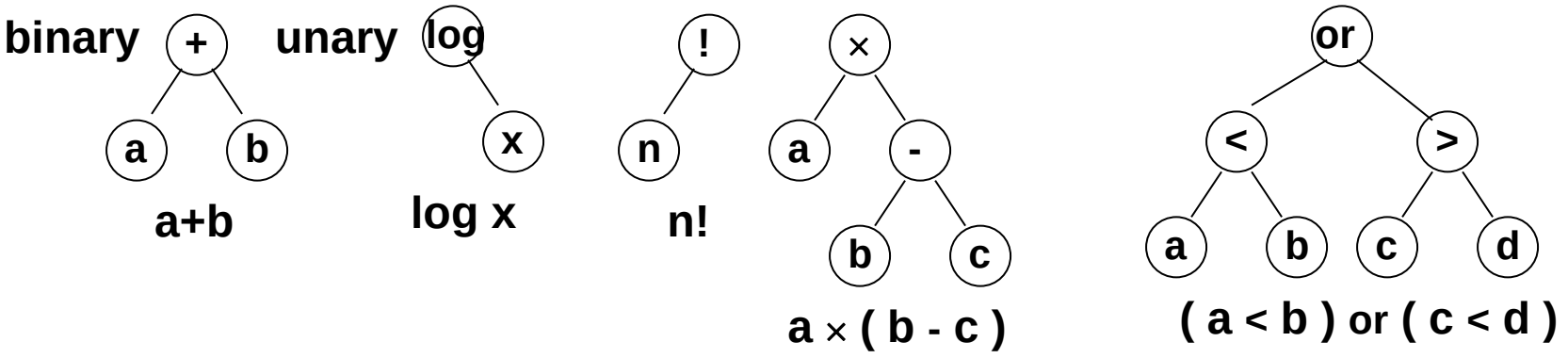
The method of writing all operators either before their operands, or after them, is called Polish Notation.

prefix form

postfix form / suffix form / reverse Polish form

When evaluating, neither parentheses nor priorities of operators need be taken into account. ← advantage

An expression tree is built up from the simple operands and operators of an expression by placing the simple operands as the leaves of a binary tree and the operators as the interior nodes.



- 二叉树的遍历(Traversal of Binary Trees)

遍历二叉树的其它操作

- **Polish Notation(表达式的波兰表示法)**

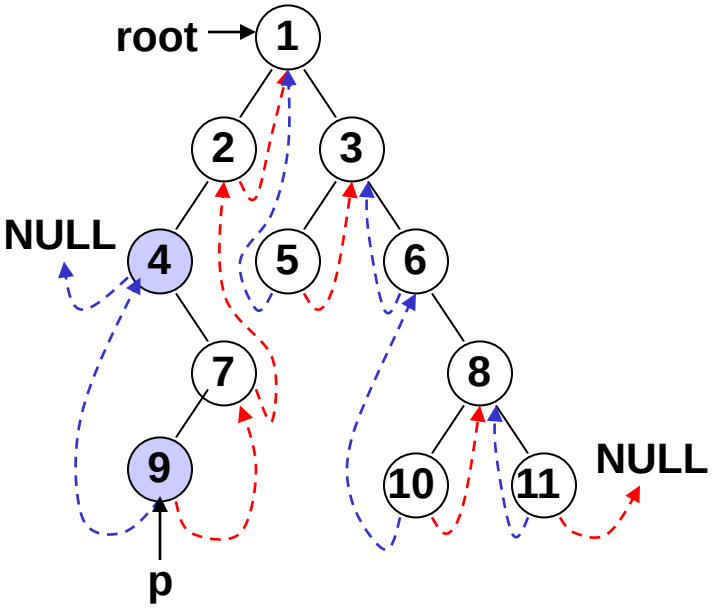
```
//Evaluate an expression in prefix form
value EvaluatePrefix(Expression expr)
{Token token;
  Value x, y;
  GetToken(token, expr);
  switch(Kind(token)) {
     case OPERAND:
        return GetValue(token);
     case UNARYOP:
        x =  EvaluatePrefix(expr);
        return DoUnary(token, x);
     case BINARYOP:
        x =  EvaluatePrefix(expr);
        y =  EvaluatePrefix(expr);
        return DoBinary(token, x, y);  }
}
```

- 线索二叉树(Threaded Binary Trees)

How to perform traversals without using either a stack or recursion?

以中序遍历为例:

```
p = root;
while (p) {
    while (p->lchild)  p = p->lchild;
    Visit(p->data);
    p = p->rchild; }
```



4 9 7 2 1 5 3 6 10 8 11

上述语句序列执行结束后有p=NULL,即最后访问过的结点其右子树为空. 这时如何去继续Visit/Locate下一个结点(即后继结点)?又怎样才能得到刚刚Visit过的结点的前面的一个结点(即前趋结点)?

Lemma: A linked binary tree with n node, n≥1, has exactly n+1 NULL links.

pointer
- a link ⇨ 指向该结点的(左/右)孩子结点(tag = 0)
- a thread(线索) ⇨ 指向该结点的前趋/后继结点(tag = 1)

通过添加tag标志域加以区别

- 线索二叉树(Threaded Binary Trees)

线索二叉树的数据结构 P132:

```
typedef struct btnode {
    BTElemType  data;
    struct btnode  *lchild, *rchild;
    int ltag, rtag;
} BTNode, *BTree;
```

其中:

若结点p
- 有左孩子时, 则p->ltag = 0 且p->lchild指向其左孩子结点
- 无左孩子时, 则p->ltag = 1 且p->lchild指向其前趋结点

若结点p
- 有右孩子时, 则p->rtag = 0 且p->rchild指向其右孩子结点
- 无右孩子时, 则p->rtag = 1 且p->rchild指向其后继结点

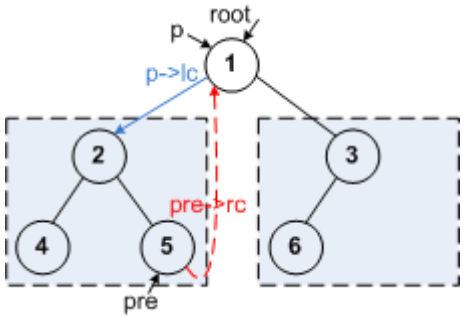
把指向前趋或后继结点的指针称作线索(single threaded or double threaded).
对二叉树以某种次序遍历并且加上线索的过程称作线索化.
经过线索化后的二叉树称作线索二叉树.

线索二叉树的逻辑表示:
- 线索要用带arrow的虚线表示
- 从结点的左/右下方出发, 左右分明, 指向正确

- 线索二叉树(Threaded Binary Trees)

二叉树线索化的算法(以中序为例): P134/A.6.6 and P135/A.6.7
为描述的简单起见, 和A.6.6相比, 不设附加头结点且对A.6.7作了些修正

```
void InOrderThreading(BTNode *t)
{ BTNode *pre = NULL;  //若p指向中序遍历时的当前结点, 则用pre指向其中序遍历的前趋结点
  InThreading(t);
  pre->rchild = NULL;  pre->rtag =1; //处理最后一个结点
 }
```




```
void InThreading(BTNode *p)
{ if (p) {
    InThreading(p->lchild);  //中序线索化左子树
    //准备处理当前结点
    if (!(p->lchild)) {p->ltag = 1; p->lchild = pre} else p->ltag = 0;
    if (pre) {
      if (!(pre->rchild)) {pre->rtag = 1; pre->rchild = p; } else pre->rtag = 0;
    }
    pre = p;  //保持pre指向p的前趋, 继续处理下一个结点
    InThreading(p->rchild);  //中序线索化右子树
  } }
```

该算法线索化过程的基本思路如图所示): 针对当前结点p和其前趋结点pre, 且只关注p的lchild、ltag和pre的rchild、rtag. 即算法中的两行斜体语句.

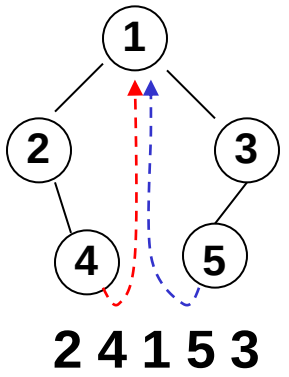
- 线索二叉树(Threaded Binary Trees)

线索二叉树的遍历(以中序为例):

至此, 遍历线索二叉树就无需递归, 也无需栈.

- 已知某结点由p指向, 找出它的后继结点

```
struct BTNode *InSuccessor(BTNode *p)
//idea: 若p->rtag = 1, 则p->rchild 就是结点p的后继结点;
//若p->rtag = 0, 表明p有右孩子, 那么p的后继结点应是中序遍历p结点的右子树时
//访问的第一个结点, 即是其右子树的最左尾结点(leftmost)
{ if (p->rtag == 1)  q = p->rchild;
  else { q = p-> rchild;
         while (!(q->ltag)) q = q->lchild; }
  return q; }   //如4的后继是1, 1的后继是5
```



- 已知某结点由p指向, 找出它的前趋结点

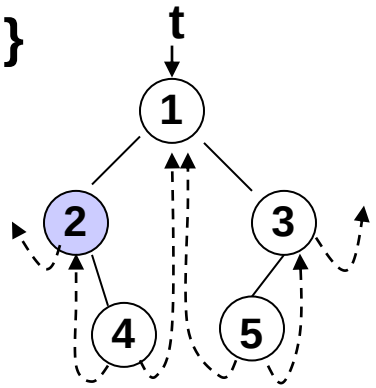
```
struct BTNode *InPredecessor(BTNode *p)
//idea: 若p->ltag = 1, 则p->lchild就是结点p的前趋结点;
//若p->ltag = 0, 表明p有左孩子, 那么p的前趋结点应是中序遍历p的左子树时
//最后访问的结点, 即是其左子树的最右尾结点(rightmost)
{ if (p->ltag == 1)  q = p->lchild;
  else { q = p-> lchild;
         while (!(q->rtag)) q = q->rchild; }
  return q; }   //如5的前趋是1, 1的前趋是4
```

- 线索二叉树(Threaded Binary Trees)

线索二叉树的遍历(以中序为例):

- 对中序线索二叉树进行遍历

```
struct BTNode *InOrderThread(BTNode *t)
{ p = t;
  if (p)  {
     while (!(p->ltag)) p = p->lchild; //查找线索二叉树的最左结点
     Visit(p->data);
     p = InSuccessor(p); //求p结点的后继结点的算法
     while (p) { Visit(p->data); p = InSuccessor(p); }
  }
} //右图中序遍历结果序列: 2 4 1 5 3
```



那么, 又如何对后序线索二叉树或先序线索二叉树进行遍历?

后序线索二叉树的基本运算[补充]:

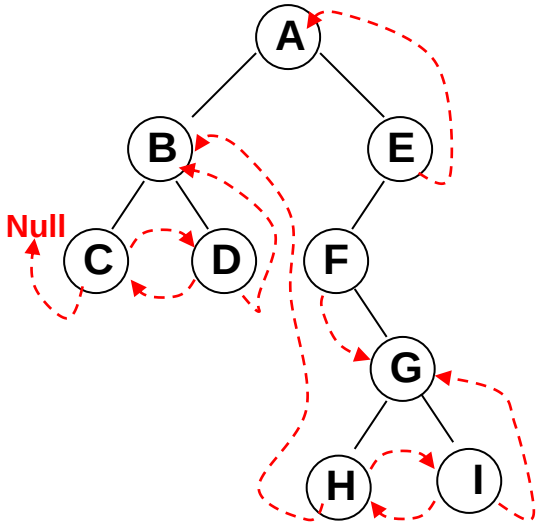
- 已知某结点由p指向, 找出它的前趋结点

  • 若 p->ltag == 1 (即*p的左子树为空), 则p->lchild是前趋线索, 指向其后序前趋结点. 如, H前趋是B, F前趋是G, C前趋是空

  • 若 p->ltag == 0 (即*p的左子树为非空), 则p->lchild不是前趋线索. 因为在后序遍历时, 根是在遍历其左右子树之后被访问的, 故*p的后序前趋必是其两棵子树中最后一个遍历到的结点. 因此,

  若p->rchild!=Null, 则*p的前趋是*p的右孩子即p->rchild. 如,A前趋是E

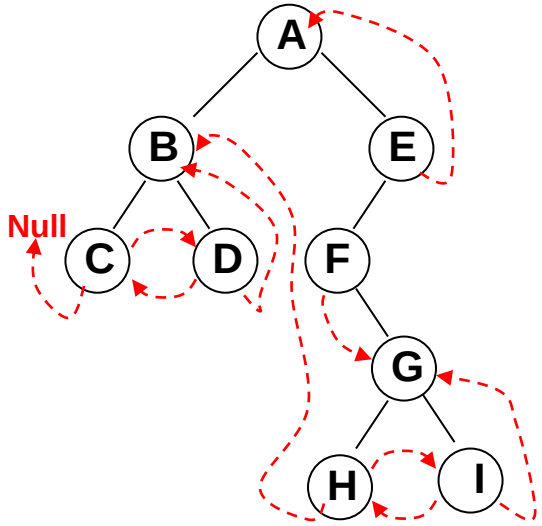
  若p->rchild==Null, 则*p的前趋是*p的左孩子即p->lchild. 如,E前趋是F



⇦ 一棵后序线索二叉树

后序线索二叉树的基本运算[补充]:

- 已知某结点由p指向, 找出它的后继结点
  - 若*p是树根, 则*p是该二叉树后序遍历最后一个访问的结点, 因此*p的后继为空. 如, A后继是空
  - 若 p->rtag == 1 (即*p的右子树为空), 则p->rchild是后继线索, 指向其后序后继结点. 否则:
    - 若*p是其双亲的右孩子, 则*p的后继就是其双亲结点. 如, G后继是F
    - 若*p是其双亲的左孩子, 需分两种情况考虑:
      - 若*p无右兄弟, 则*p的后继是其双亲结点. 如, F后继是E
      - 若*p有右兄弟, 则*p的后继是其双亲结点的右子树中第一个后序遍历到的结点, 它是该子树中的"最左下的叶结点". 如, B后继是其双亲A的右子树中的最左下的叶结点H (注: 不是F, 因为它不是叶结点)



⇧ 一棵后序线索二叉树

结论: 在后序线索二叉树中, 若已知p指针, 则从*p出发就能找到其后序前趋; 若已知p指针且p不指向根, 则仅当*p的右子树为空时, 才能直接由p->rchild得到其后继结点, 否则必须知道*p的双亲结点才能找到其后继.

➜在后序线索二叉树中, 线索对查找指定结点的后继并无多大作用.

前序线索二叉树的基本运算[补充]:

和后序线索二叉树类似, 在前序线索二叉树中, 找某一点*p的后继也非常方便, 仅从*p出发就能找到; 但找其前趋也必须知道*p的双亲结点, 而当树中结点未设双亲指针时, 同样要进行从根结点开始的前序遍历才能找到*p的前趋.

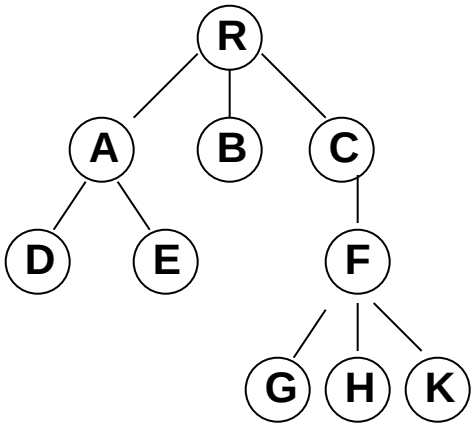
结论: ①一般只针对中序线索二叉树, 因为找某一点*p的前趋结点和后继结点均非常方便; ②对于找前趋和后继结点这二种运算而言, 线索树优于非线索树. 但在对线索树进行插人和删除操作时, 线索树比非线索树的时间开销大. 因为除了修改相应的指针外, 还要修改相应的线索.

- 树的存储结构

①双亲表示法 在树中, 每个结点有多个孩子, 而每个结点的双亲唯一. 利用这一性质, 以一组连续空间存储树的结点, 同时在每个结点中附设一个“指示” / “下标”用来指示其双亲结点在表中的位置.

```
typedef struct ptnode {
    TElemType  data;
    int parent;  //双亲位置域
} PTNode;
```

```
typedef struct {
    PTNode  node[MaxTreeSize];
    int n;  //树中结点数
} PTree;
```



数组下标	结点信息	双亲指示
1	R	0
2	A	1
3	B	1
4	C	1
5	D	2
6	E	2
7	F	4
8	G	7
9	H	7
10	K	7

设有 PTree t;
若 t.node[i].parent = j,
则 t.node[i]的双亲是t.node[j].

显然, 双亲表示法便于访问一个结点的双亲, 但找其孩子的操作较复杂, 因为这时需遍历整个表.

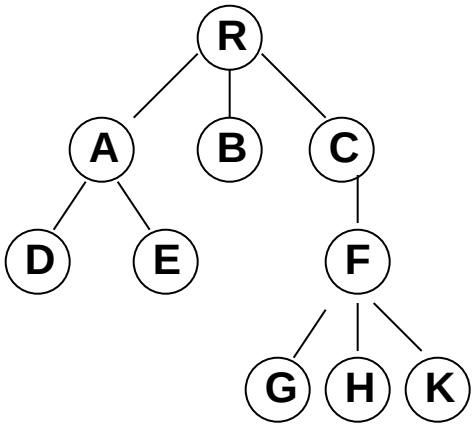
例子: 编写算法计算一棵树的高度, 假设棵的存储结构采用双亲表示法.[见下页]

- 树的存储结构

①双亲表示法 在树中, 每个结点有多个孩子, 而每个结点的双亲唯一. 利用这一性质, 以一组连续空间存储树的结点, 同时在每个结点中附设一个“指示” / “下标”用来指示其双亲结点在表中的位置.

```
typedef struct ptnode {
    TElemType  data;
    int parent;  //双亲位置域
} PTNode;
```

```
typedef struct {
    PTNode  node[MaxTreeSize];
    int n;  //树中结点数
} PTree;
```



数组下标	结点信息	双亲指示
1	R	0
2	A	1
3	B	1
4	C	1
5	D	2
6	E	2
7	F	4
8	G	7
9	H	7
10	K	7

例子: 编写算法计算一棵树的高度, 假设棵的存储结构采用双亲表示法.

```
//思路: 用双亲表示法作为树的存储结构, 便于找到结点的双亲. 因此, 计算出树中每个结点的层次, 根据树的高度的定义, 取其最大层次就是树的高度. 而对每个结点计算层次的方法是, 找其双亲、双亲的双亲、…, 直至(根)结点双亲为0为止.
int Depth(PTree t) {
  for ( i = 1; i <= t.n; i++ ) {
      temp = 0; f = i;
      while (f>0) {temp++; f = t.node[f].parent; }
            //层次加1, 并取新的双亲
       if (temp > maxdepth) maxdepth = temp; }
           //更新树的高度
   return maxdepth; }
```

- 树的存储结构

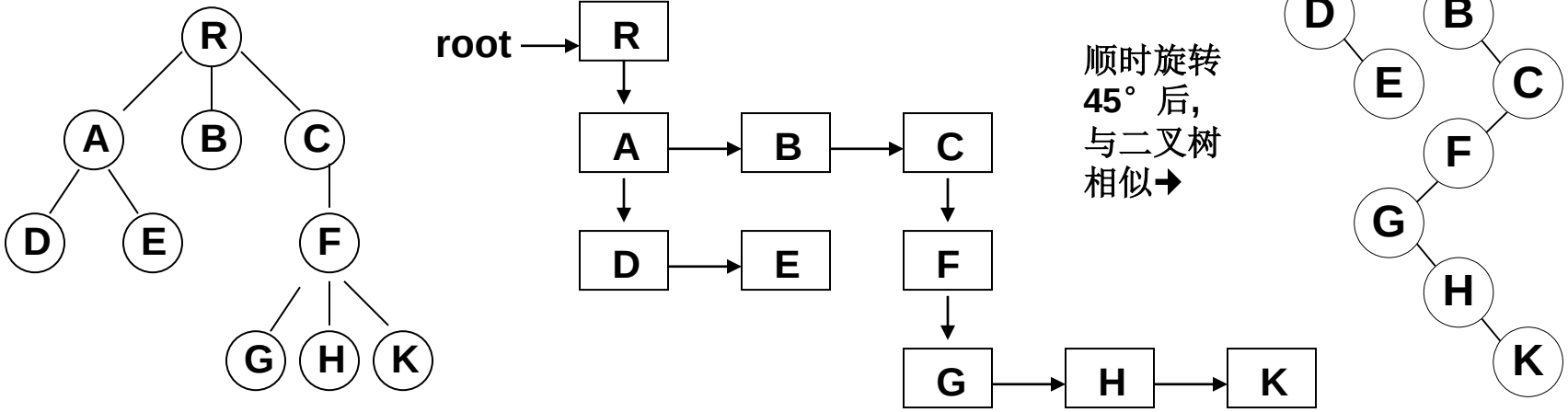
②孩子表示法 (又称多叉链表) 每个结点有多个指针域, 分别指向其子树的根

- 结点同构: 结点的指针个数相等, 为树的度D
- 结点不同构: 结点指针个数不等, 为该结点的度d

data	child1	child2	···	childD

data	degree	child1	child2	···	childd

③孩子兄弟表示法 (又称二叉树表示法)



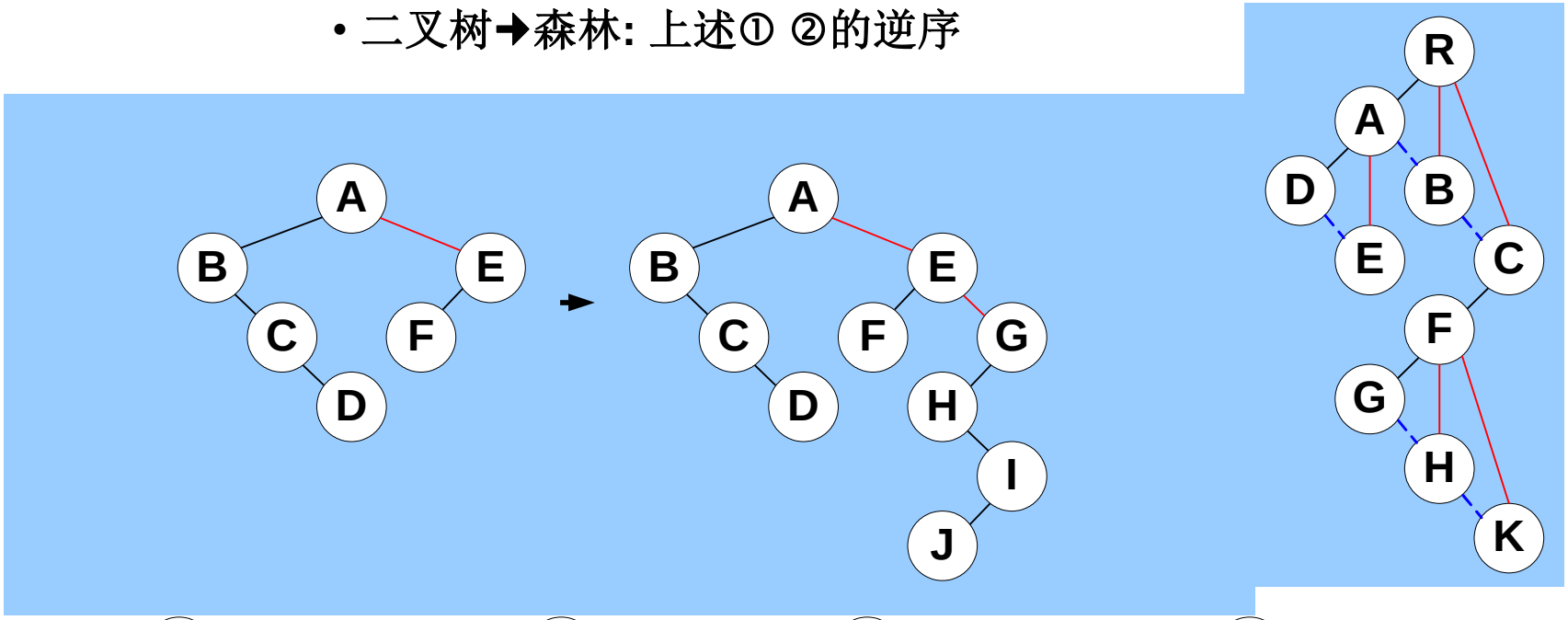

```
typedef struct tnode {
    TElemType  data;
    struct tnode  *firstchild, *nextsibling;
} TNode, *Tree;
```

- 该表示法和二叉树二叉链表表示法相同, 只是解释不同; 可借助二叉树的一些算法解决树/森林中的问题.
- 任何一棵树(或一个森林)可唯一地对应到一棵二叉树; 反之, 任何一棵二叉树也能唯一地对应到一棵树(或一个森林).

- 树、二叉树、森林之间的转换(利用孩子兄弟表示法来转换 即 是对树/森林孩子兄弟表示法的逻辑验证)
  - 树➜二叉树: ①加线　在所有兄弟结点之间加一连线
    ②去线　对每个结点, 除保留与其长子的连线外, 去掉该结点与其它孩子的连线
    ③旋转(顺时45°)整理
  - 二叉树➜树: ①还原加线②还原去线③还原(逆时45°)整理

并非任意一棵二叉树
均可转换成一棵树

  - 森林➜二叉树: ①将森林中的每棵树➜二叉树②按森林图形顺序, 依次将后一棵二叉树作为前一棵二叉树的根结点的右子树, 如此循环.
    - 二叉树➜森林: 上述① ②的逆序



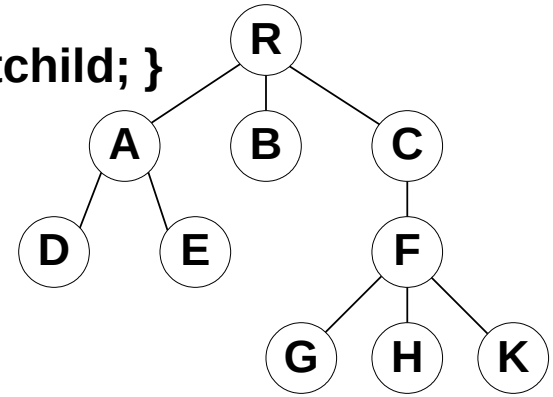
- 树的遍历

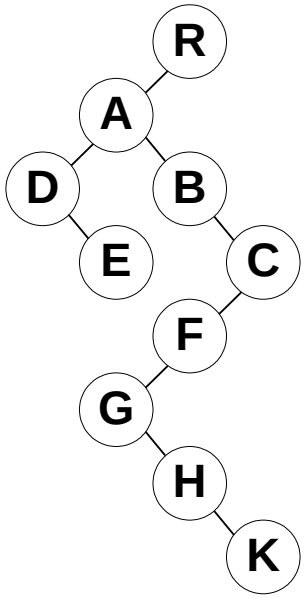
  树的遍历的定义 **P138**

  - 先序/先根遍历: 先访问树的根结点, 然后 依次 先序遍历根的每棵子树
  - 后序/后根遍历: 先 依次 后序遍历每棵子树, 然后访问根结点

```
void PreOrderTree(TNode *root, void (*Visit)())
{  p = root;
   while (p || !StackEmpty(s)) {
       while (p) { Visit(p->data);  Push(s, p);  p = p->firstchild; }
       p = Pop(s);  p = p->nextsibling; }
}  //上图所示树的先序遍历结果序列: RADEBCFGHK
```

注: 先序遍历一棵树的结果序列恰好等于
先序遍历该树对应的二叉树的结果序列.





- 森林的遍历

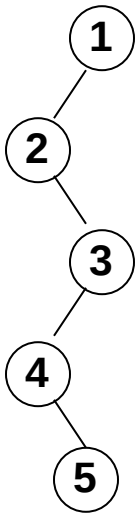
  森林遍历的定义 **P139**

  - 先序/先根遍历: 先访问森林中第一棵树的根结点, 然后先序遍历第一棵树中根结点的各子树所构成的森林, 再依次先序遍历除第一棵树外的其它树所构成的森林
  - 后序/后根遍历: 先后序遍历森林中第一棵树的根结点的各子树所构成的森林, 然后访问第一棵树的根结点, 再依次后序遍历除第一棵树外的其它树所构成的森林

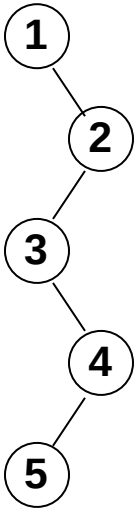
  注: 和遍历树类似, 先序遍历森林等同于先序遍历该森林对应的二叉树; 后序遍历森林等同于中序遍历该森林对应的二叉树.

Exercises (After class):

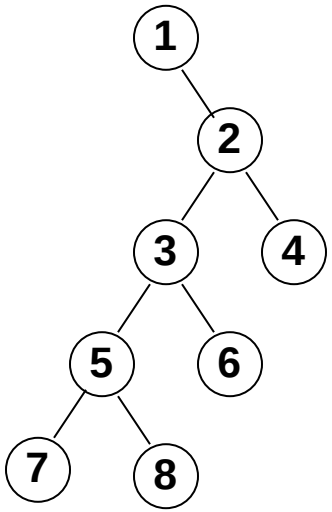
1. **Give the order of visiting the vertices of each of the following binary trees under (a)preorder (b)inorder (c) postorder traversal.**



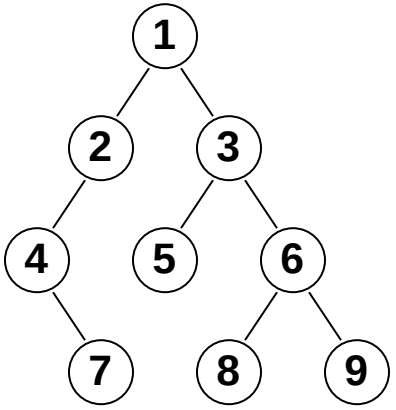
(1)



(2)



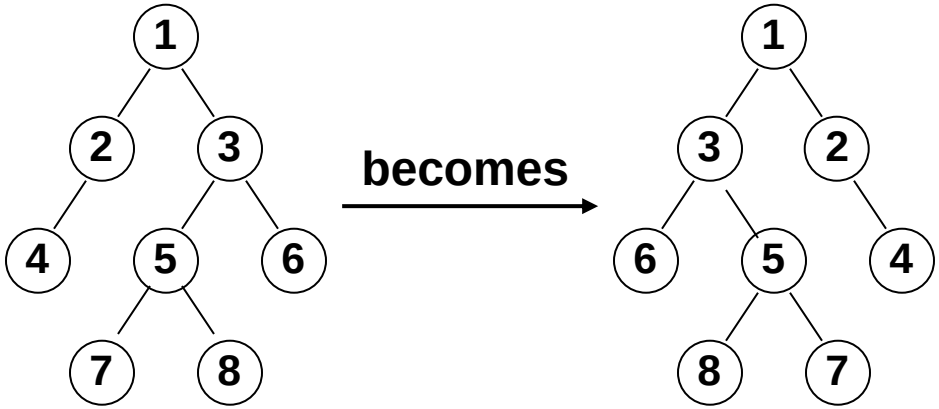
(3)



(4)

2. **Write a function void CopyTree(Tree *root, Tree *newroot) that will make a copy of a linked binary tree.**

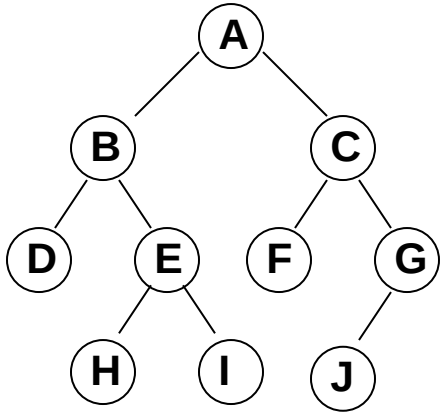
3. **Write a function that will interchange all left and right subtrees in a linked binary tree.**



4. **Write a function to perform a *double-order traversal* of a binary tree, meaning that at each node of the tree, the function first visits the node, then traverses its left subtree(in *double-order*), then visits the node again, then traverses its right subtree(in *double-order*).**

5. **Write a function that will return the width of a linked binary tree, that is, the maximum number of nodes on the same level.**

6. Draw the threaded-tree of the following binary tree in 1)preorder 2)inorder 3)postorder



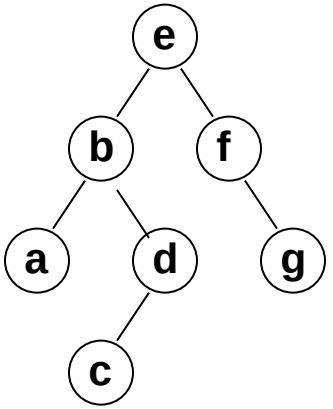
7. Write a function to insert a new node on the right of one (pointer *p* points to it) in a inorder threaded binary tree.

- 二叉排序(查找)树(Binary Sort Trees / Binary Search Trees)

定义(P227):

A binary search tree is a binary tree that is either empty or in which every node contains a key and satisfies the conditions:

1. **The key in the left child of a node(if exists) is less than the key in its parent node.**
2. **The key in the right child of a node(if exists) is greater than the key in its parent node.**
3. **The left and right subtrees of the root are again binary search trees.**



特征: 对二叉查找树进行中序遍历, 则得到一个按关键字值升序的结果序列.

例题: BST中关键字值最小(最大)的元素是哪一个结点? 并写出相应的算法.

最小/最大的关键字分别为BST中的最左/最右尾结点.

```
BTNode *minKey(BTNode *root) {
    BTNode *current = root;
    while (current->lchild != NULL) //查找BST中的最左尾结点
        current = current->lchild;
    return current; }
```

- **二叉排序(查找)树(Binary Sort Trees / Binary Search Trees)**

  运算/操作:

  1. 查找: BTNode *BSTreeSearch(BTNode *root, KeyType target)

  2. 插入/创建: BTNode *BSTInsertion(BTNode *root, BTNode *newnode)

  3. 删除: BTNode *BSTDeletion(BTNode *root, BTNode *p)

  算法分析: Chapter 9(待讲)

- 二叉排序(查找)树(Binary Sort Trees / Binary Search Trees)

Search From A Binary Search Tree

BTNode *BSTreeSearch(BTNode *root, KeyType target)

***precondtion*: The binary search tree to which root points has been created.**

***postcondition*: If an entry in the binary search tree has key equal to target, then the return value points to such a node; otherwise, the function returns NULL.**

```
//recursive version: P228 Algorithm 9.5(a)
{ if (root)
      if (LT(target, root->data))
          root = BSTreeSearch(root->lchild, target);
      else if (GT(target, root->data))
                root = BSTreeSearch(root->rchild, target);
  return root;  }
//recursion removal
{ p = root;
  while (p && NE(target, p->data))
       if (LT(target, p->data)) p = p->lchild;
       else p = p->rchild;
  return p; }
```

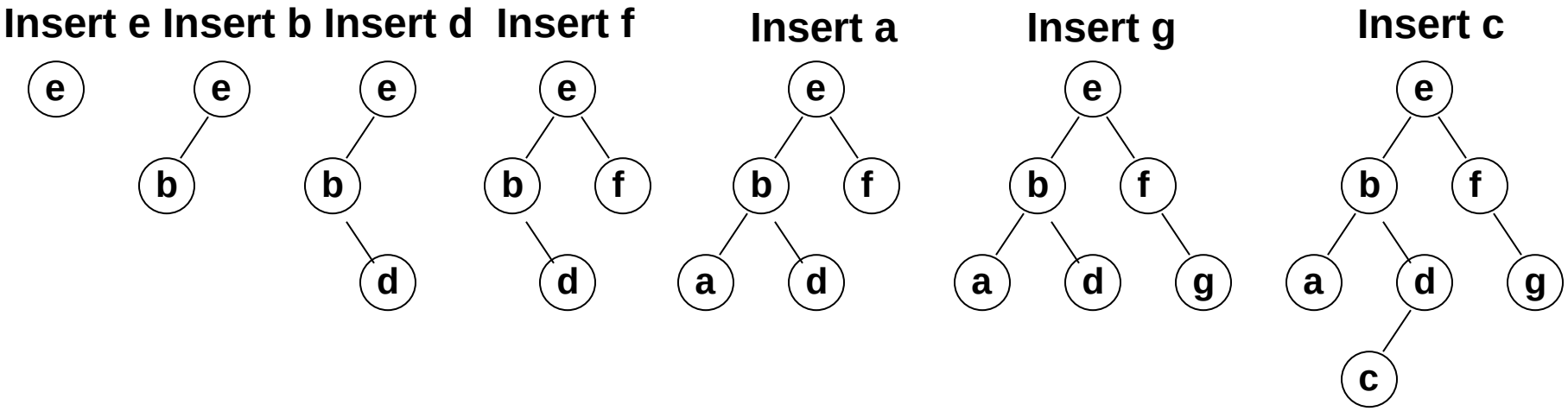
- 二叉排序(查找)树(Binary Sort Trees / Binary Search Trees)

Insertion Into A Binary Search Tree

基本思想: 建立一棵二叉排序树, 实际上就是不断地进行插入新结点的运算.
设有一组数据{k_1, k_2, … ,k_n},

1. 先让k_1做根;
2. 对于k_2, 若$k_2<k_1$, 则令k_2做k_1的左孩子, 否则令k_2做k_1的右孩子;
3. 对于k_3, 从根k_1开始比较, 若$k_3<k_1$, 则到k_1的左子树中继续比较, 否则到k_1的右子树中继续比较, 如此循环, 直至一个合适位置时进行插入操作;
4. 对于剩下数据, 重复上述第3步.

例子: 设有一组数据{e, b, d, f, a, g, c}



- 二叉排序(查找)树(Binary Sort Trees / Binary Search Trees)

Insertion Into A Binary Search Tree

算法描述:

void InsertTree(TreeNode *root, TreeNode *newnode);

***precondtion*: The binary search tree to which root points has been created. The parameter newnode points to a node that has been created and contains a key in its entry.**

***postcondition*: The node newnode has been inserted into the tree in such a way that the properties of a binary search tree are preserved.**

```
//iterative insertion : P228-229 Algorithm 9.5(b) and 9.6
{ if (!root)  { root = newnode;
                root -> lchild = root->rchild = NULL; }
  else { p = root;
         while (p) { q = p; //q记录p的双亲
                     if (LT(newnode->data, p->data))  p = p->lchild;
                     else p = p->rchild;  }
         if (LT(newnode->data, q->data))  q->lchild = newnode;
         else q->rchild = newnode; }
 }
```

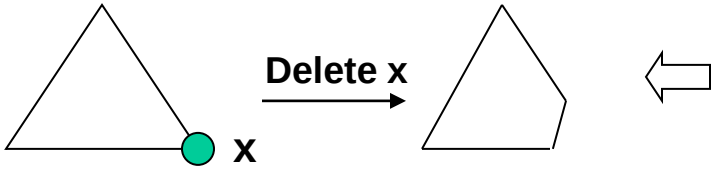
- 二叉排序(查找)树(Binary Sort Trees / Binary Search Trees)

Deletion From A Binary Search Tree

目标: 在二叉排序树中删除任意一个结点后, 仍然具有二叉排序树的特征.

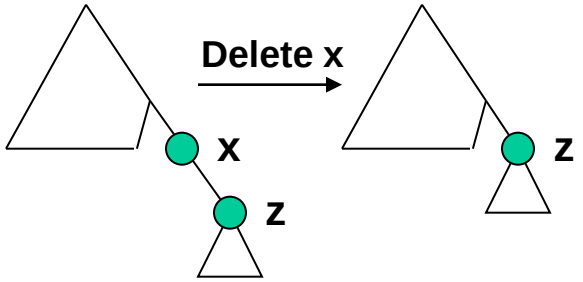
↳可为如下几种情况P229~230

Case 1: deletion of a leaf



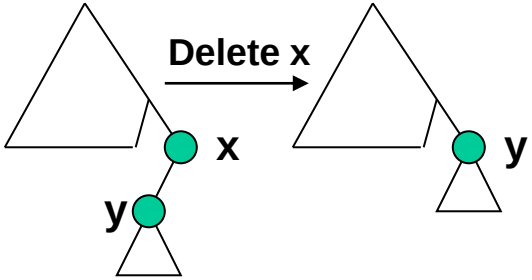
Only replace the link to the deleted node by NULL.

Case 2: the deleted node has only one subtree



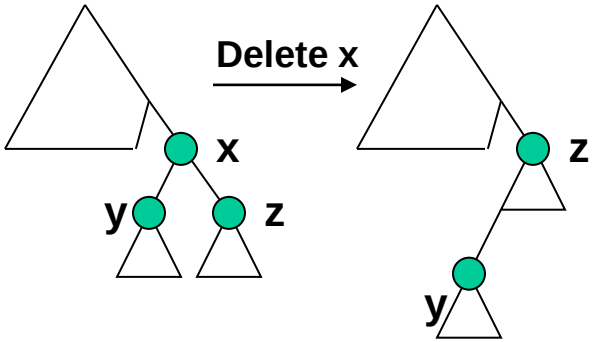
2-1: empty left subtree

To adjust the link from the parent of the deleted node to point to its subtree.

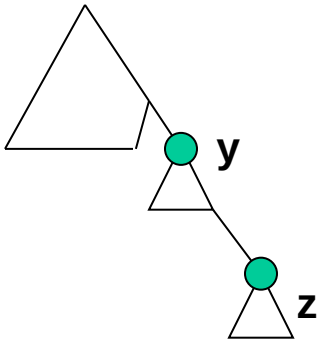


2-2: empty right subtree

Case 3: the deleted node has both left and right subtrees



To which node of the right subtree should the former left subtree be attached? It must be as far to the left as possible.



- 二叉排序(查找)树(Binary Sort Trees / Binary Search Trees)

Deletion From A Binary Search Tree

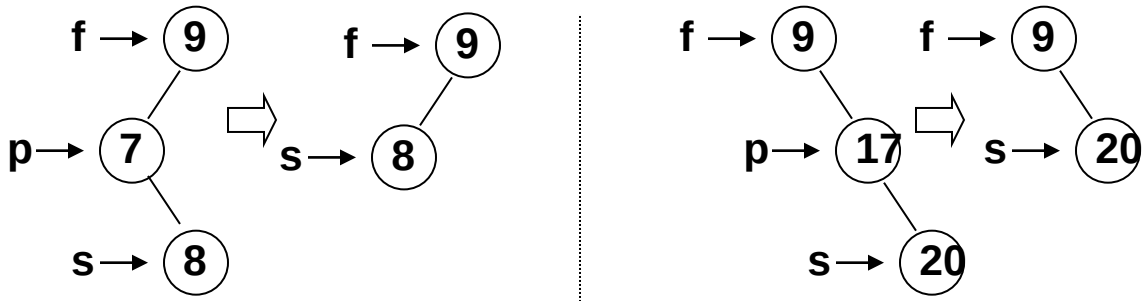
算法描述: 设p指向被删结点, f指向*p结点的双亲, s、q分别指向替代结点及其双亲. 该算法的思路是怎样找到s结点来替换p结点.

Case 1：if (!(p->lchild) && !(p->rchild)) s = NULL;

if (f->lchild == p) f->lchild =s; else f->rchild = s; free(p);

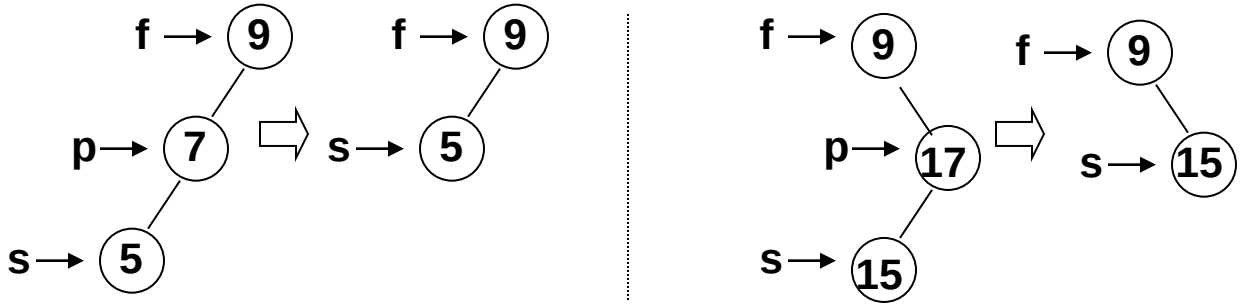
Case 2-1：if (!(p->lchild)) s = p->rchild;

if (f->lchild == p) f->lchild =s; else f->rchild = s; free(p);



Case 2-2：if (!(p->rchild)) s = p->lchild;

if (f->lchild == p) f->lchild =s; else f->rchild = s; free(p);



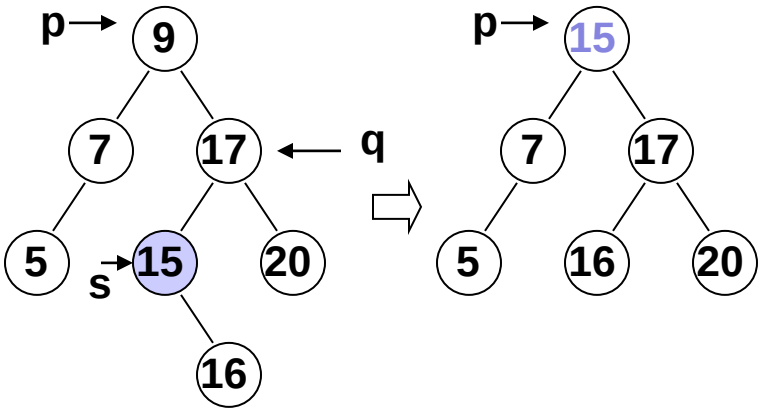
上述绿颜色语句实作为公共处理部分.

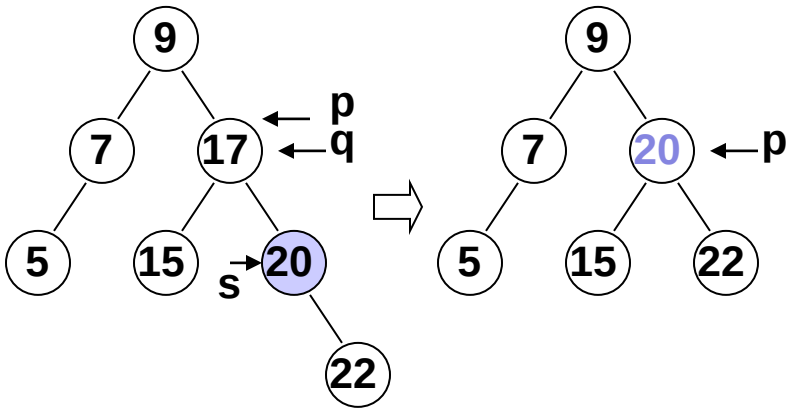
- 二叉排序(查找)树(Binary Sort Trees / Binary Search Trees)

Deletion From A Binary Search Tree

Case 3: p结点有左右孩子, 则可用其前趋或后继结点s替代p. 若用后继结点, 则s应是p的右子树的最左尾结点(因为其值最小), 这时先让p结点取s结点的值(p->data=s->data); 然后再按Case2-1情况(因为此时s肯定没有左孩子)处理删除s结点. (从二叉排序树中删除一个结点的算法P231/A.9.8)

```
if (p->lchild && p->rchild) {
   q = p;  s = p->rchild;
   while (s->lchild)  { q = s; s = s->lchild; }  //确定最左尾结点
   p->data = s->data;
   if (q == p)  //特例情况: p结点的右子树没有左孩子, 如下方右图所示
      q->rchild = s->rchild;
   else q->lchild = s->rchild;
   free(s); }
```





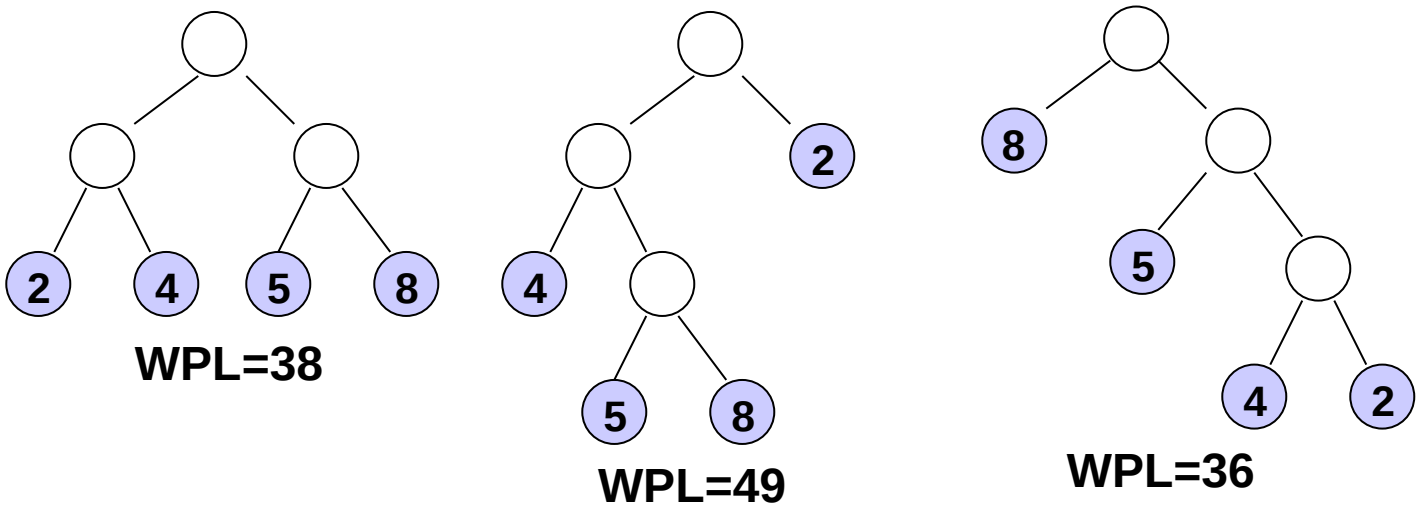
- **Huffman Trees 哈夫曼树 / 最优二叉树**

几个概念: **P144**

树的路径长度: 从根结点到每个结点的路径长度的总和.

树的带权路径长度: 从根结点到每个叶结点的路径长度与相应叶结点权值的乘积总和. 记作:

$$WPL = \sum_{k=1}^{n} L_k W_k$$



Huffman树/最优二叉树: WPL值最小的二叉树.

2-tree: as a tree in which every vertex except the leaves has exactly two children. Huffman树是2-tree.

P147: 2-tree又称为strict binary tree/正则二叉树

- **Huffman Trees 哈夫曼树 / 最优二叉树**

Huffman树的构造方法及Huffman算法: P145~146

构造**Huffman**树步骤

- 根据给定的**n**个权值**{w_1,w_2,……w_n}**, 构造**n**棵只有根结点的二叉树的森林, 即其中的每棵二叉树只有一个带权值为w_i的根结点, 且其左右子树均为空;
- 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**(**共**n-1**步**)**, 这棵树即是哈夫曼树.

一个构造**Huffman**树的例子

Huffman树结点的数据类型

```
typedef struct btnode{
    char ch;
    int freq;
    BTNode *lchild,*rchild;
} BTNode, *BTree;
```

Huffman树构造算法

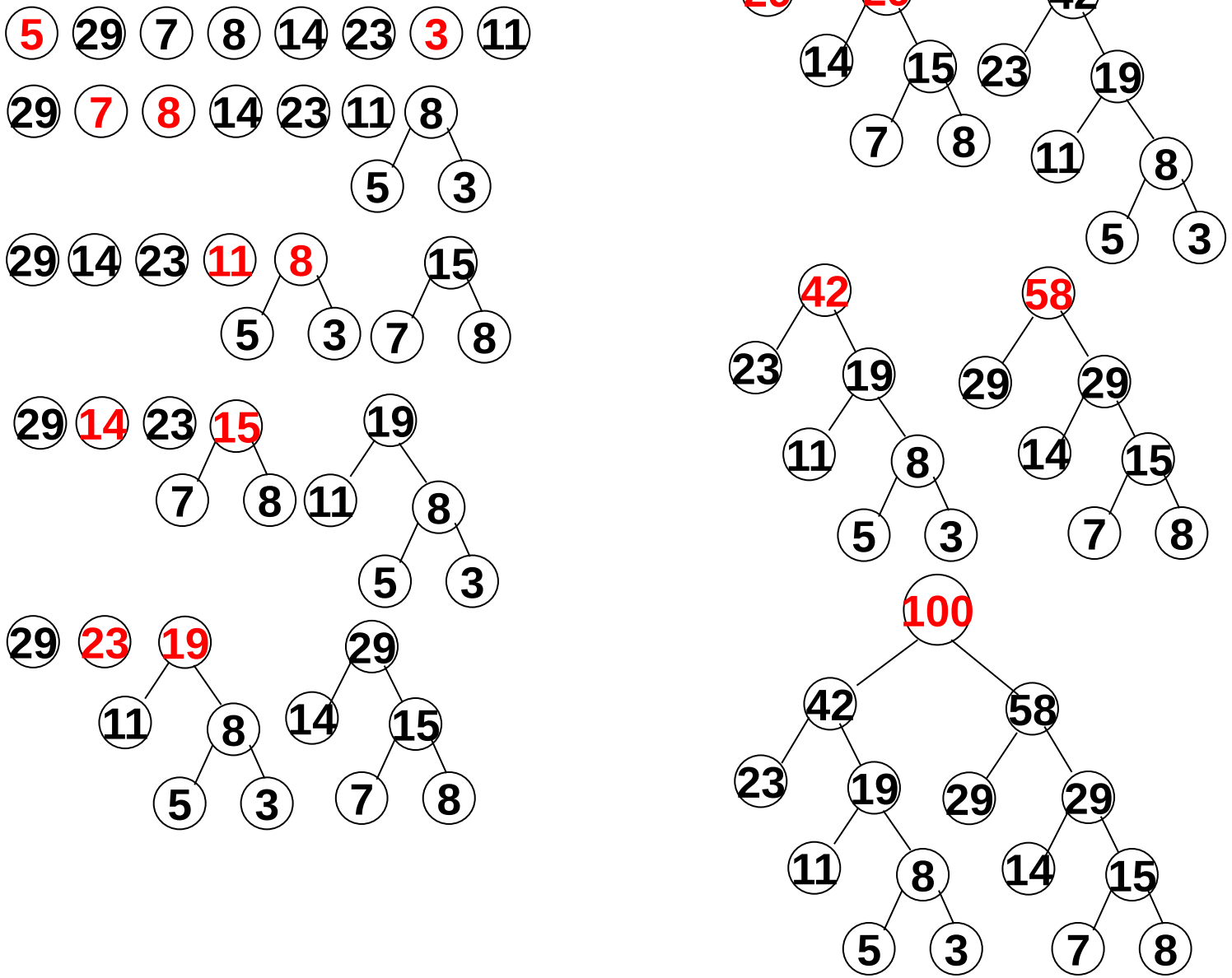
参见**P147/A.6.12**的部分描述. 注: 该算法使用的是顺序存储结构

- **Huffman Trees 哈夫曼树 / 最优二叉树**

Huffman树的构造方法及Huffman算法: P145~146

例　**W = { 5, 29, 7, 8, 14, 23, 3, 11 }**

↓

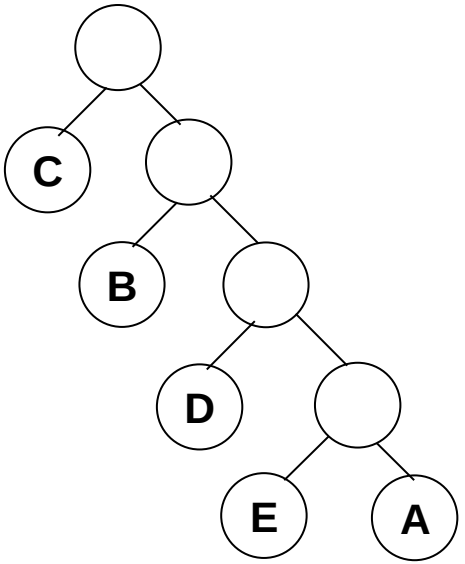


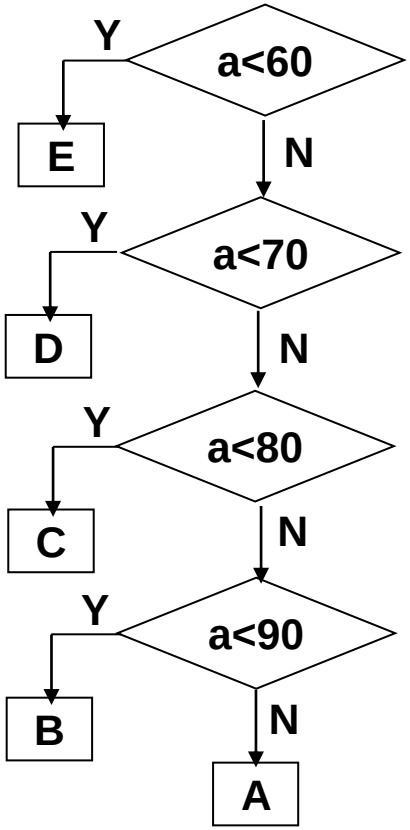
- **Huffman Trees 哈夫曼树 / 最优二叉树**

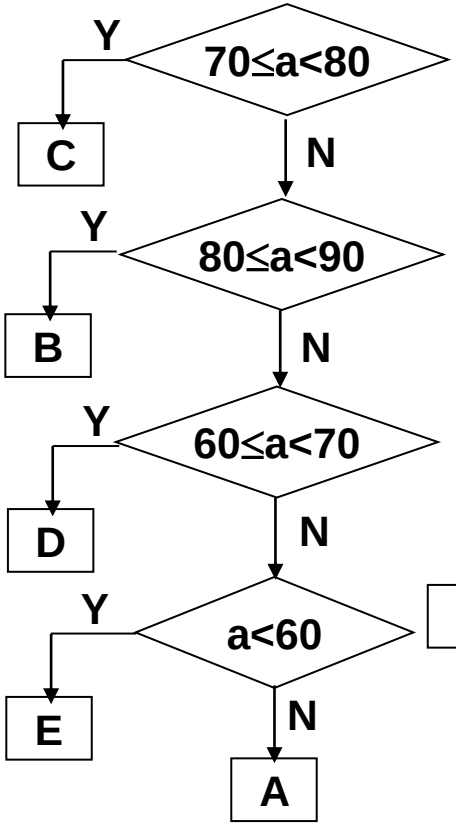
Huffman树的应用: P144~146

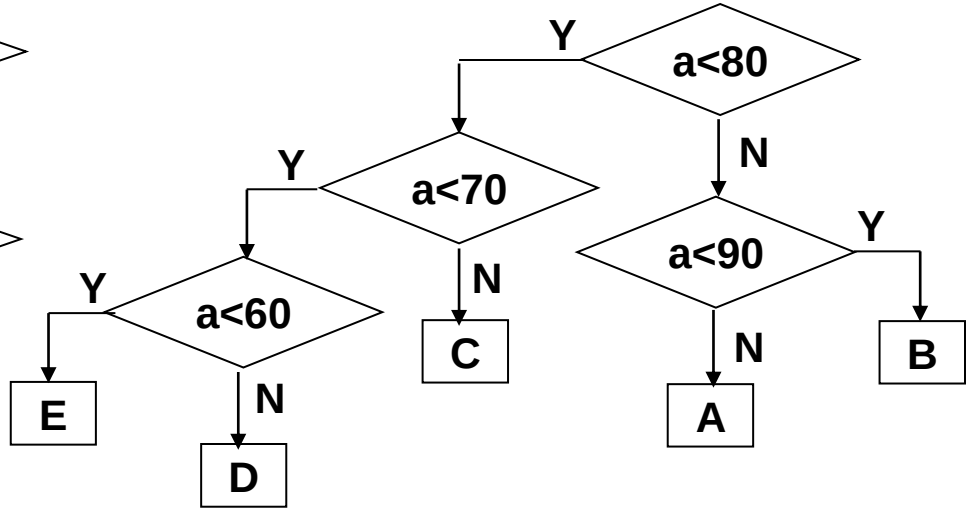
用于最佳判断过程 / 最佳判定树

等级	E	D	C	B	A
分数段	0~59	60~69	70~79	80~89	90~100
比例	0.05	0.15	0.40	0.30	0.10









- **Huffman Trees 哈夫曼树 / 最优二叉树**

Huffman树的应用: P144~146

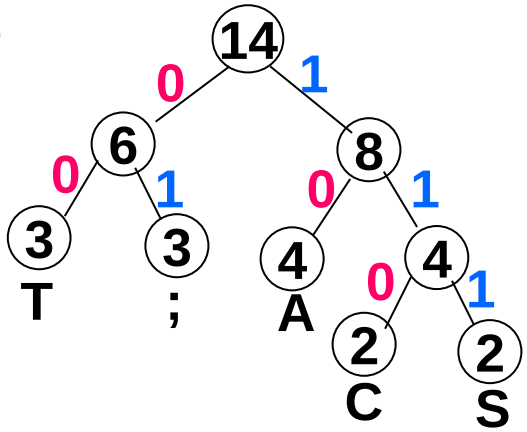
用于通信编码 / **Huffman coding**

思想　根据字符出现频率进行编码, 使电文总长最短

前缀码问题　假设C、A、T分别编码为00、01、0001, 则解码时无法确定编码串0001是CA还是T, 产生该问题的原因是C的编码与T的编码的开始部分(称为前缀)相同. 因此, 编码时要求字符集中任一字符的编码都不能是其它字符编码的前缀, 这种编码称为前缀码. Huffman编码是前缀码, 而且是变长的前缀码.

编码过程　根据字符出现频率首先构造Huffman树, 然后将树中结点指向其左孩子的分支标记为“ 0”. 指向其右孩子的分支标记为“ 1”; 每个字符的编码即为从根到每个叶结点的路径上所得到的0、1序列.

例如　要传输的字符集 D = { C, A, S, T, ; }
字符出现频率 W = { 2, 4, 2, 3, 3 }



编码表

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

若电文是{ CAS;CAT;SAT;AT }

则其编码为 “1101011101110100001111100011000”

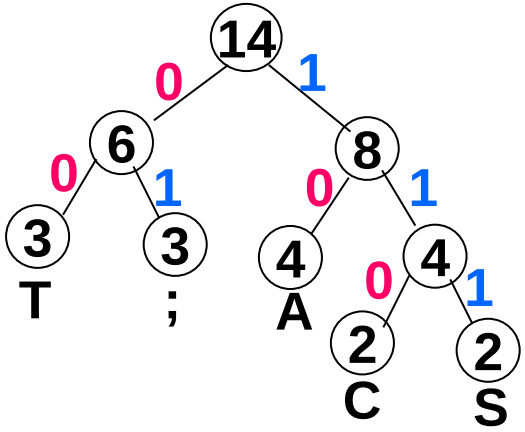
- **Huffman Trees 哈夫曼树 / 最优二叉树**

Huffman树的应用: P144~146

用于通信编码 / **Huffman编码**

译码过程　从**Huffman**树的根结点开始, 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.

例如



编码表

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

若电文为“**1101000**”, 则其译文只能是“**CAT**”

注: 若向左“**1**”、向右“**0**”，则“**0010111**”也为“**CAT**”

- **Huffman Trees 哈夫曼树 / 最优二叉树**

Huffman树的应用: P144~146

一个例题

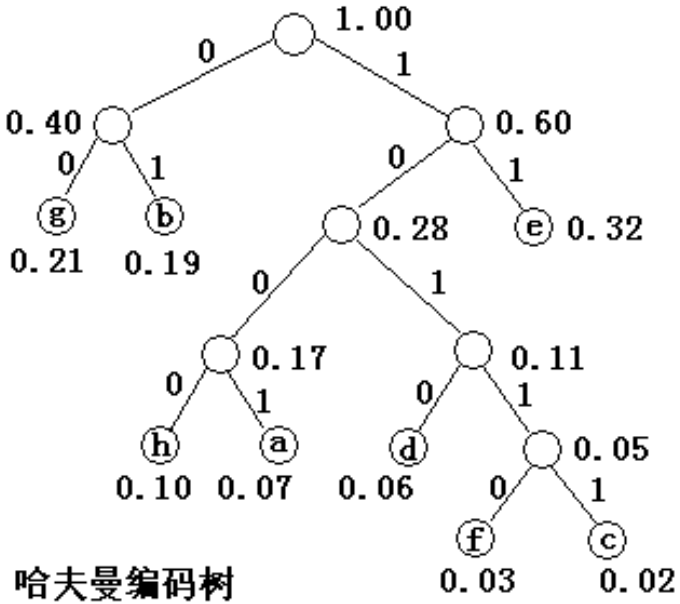
假设用于通信的电文由字符集{a,b,c,d,e,f,g,h}中的字母构成, 这8个字母在电文中出现频率分别为{0.07,0.19,0.02,0.06,0.32,0.03,0.21,0.10}.

(1)为这8个字母设计哈夫曼编码.

(2)若用3位2进制数对这8个字母进行等长编码, 则哈夫曼编码的平均码长是等长编码的百分之几? 它使电文总长平均压缩多少?

(1)根据左图可得编码表:

a: 1001
b: 01
c: 10111
d: 1010
e: 11
f: 10110
g: 00
h: 1000



哈夫曼编码树

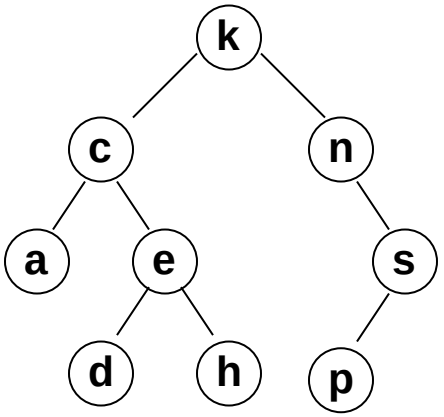
(2) 用3位2进制数进行等长编码的平均长度为3; 根据哈夫曼树编码的平均码长为:

4×0.07+2×0.19+5×0.02+4×0.06+2×0.32+5×0.03+2×0.21+4×0.10 = 2.61

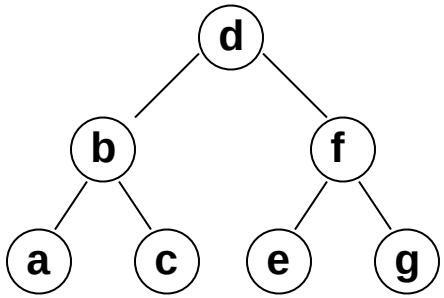
2.61/3 = 0.87 = 87%; 其平均码长是等长码的87%. 所以平均压缩率为13%.

Exercises (After class):

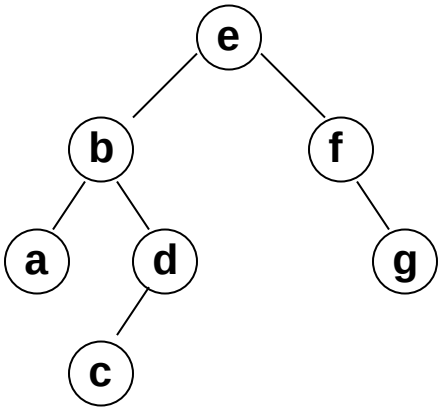
8. 设有下列二叉排序树**(binary search tree):**



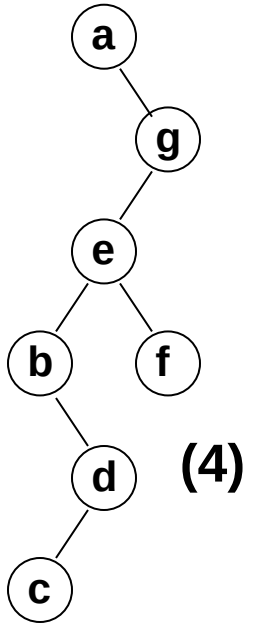
(1)



(2)



(3)



(4)

1) 将下列结点插入到上述**(1)**中**: (a) m (b) f (c) b (d) t** 要求**: Do each part independently, inserting the key into the original tree.**

2) 将下列结点从上述**(1)**中删除**: (a) a (b) p (c) n (d) s (e) e** 要求**:**
Do each part independently, deleting the key from the original tree.

3) 对上述**(2)** 给出**4**个不同的关键字顺序**,** 使得它们均能构造出该二叉排序树**;**
对上述**(3)** 给出**4**个不同的关键字顺序**,** 使得它们均能构造出该二叉排序树**;**
对上述**(4)** 给出**4**个不同的关键字顺序**,** 使得它们均能构造出该二叉排序树**;**
对此你会得出什么结论**?**

9. 将算法**9.6**改成**recursive version.**

10. 编写算法实现以下要求: **How many keys in a BST < target ?**

11. 假设通信电文使用的字符集为**{a, b, c, d, e, f}**, 各字符在电文中出现的频率分别为**{34, 5, 12, 23, 8, 18}**, 利用构造**Huffman**树写出每个字符对应编码(设**lchild<rchild**).

12. 已知某电文中共出现了**10**种不同的字母**A~J**, 它们的出现频率分别是**8**、**5**、**3**、**2**、**7**、**23**、**9**、**11**、**2**、**30**, 现对这段电文用三进制(即码字由**0**、**1**、**2**组成), 问电文编码的总长度是多少**?**并画出相应的示意图**.**