第六章 树和二叉树

线性结构：	线性表，栈，队列
	串，数组，广义表
非线性结构：	树和二叉树
	图，网

6.1树的定义

6.1.1 定义和术语

1.树（tree)：

树是n(n≥0)个结点的有限集T,

当n=0时，T为空树；

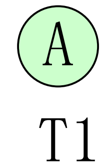
当n>0时,

(1)有且仅有一个称为T的根的结点,

(2)当n>1时,余下的结点分为m(m>0)个互不相交的有限集$T_1, T_2, ..., T_m$，每个T_i(1≤i≤m)也是一棵树,且称为根的子树。

例1. 一个结点的树

T＝{A}



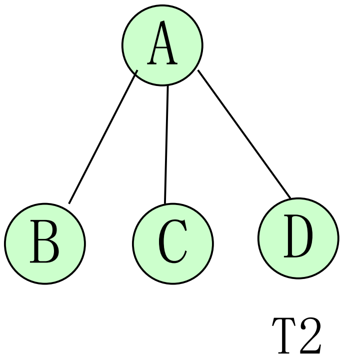
例2. 四个结点的树

T＝{A,B,C,D}

T1={B}

T2={C}

T3={D}



例3　有16个结点的树

T={A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P}

T1={B, C, D, E, F}

T11={C, D, E}

T111={D}

T112={E}

T12={F}

T2={G, H}

T21={H}

T3={I, J, K, L, M, N, O, P}

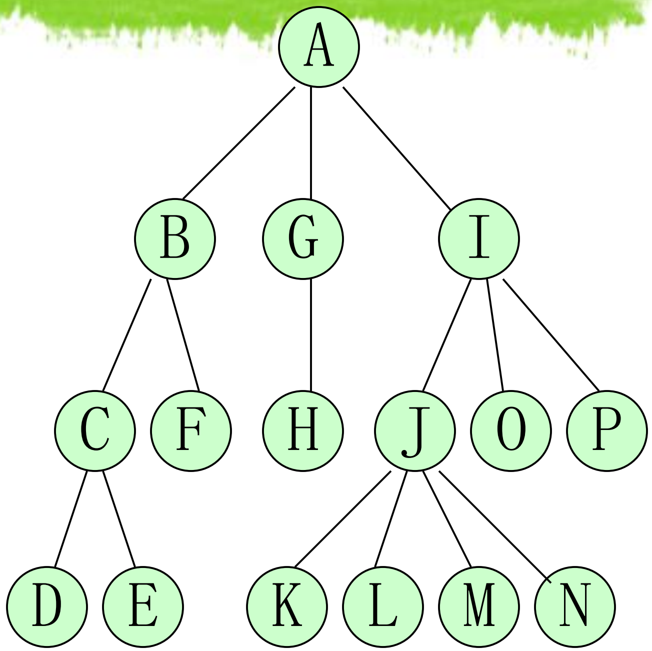
T31={J, K, L, M, N}

T32={O}

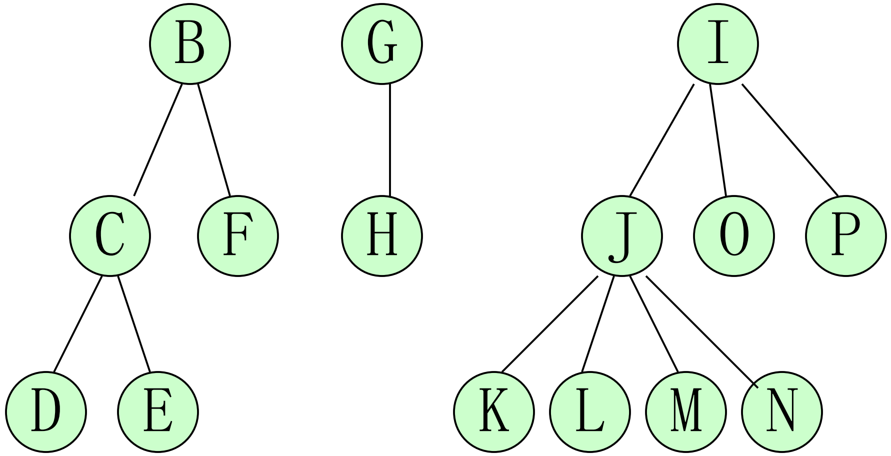
T33={P}

T311={K}　　. . .

T312={L}



树T



T1　T2　T3

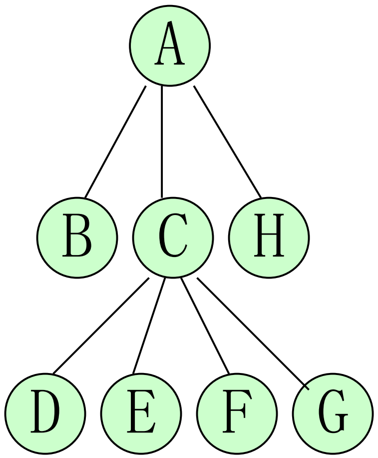
2. **结点的度(degree):**

结点的子树数目

3. **树的度:**

树中各结点的度的最大值

4. **n度树:** 度为n的树



4度树

5. **叶子(终端结点):** 度为0的结点

6. **分枝结点(非终端结点,非叶子):**

度不为0的结点

7. **双亲(父母,parent)和孩子(儿子,child) :**

若结点C是结点P的子树的根,称P是C的双亲, C是P的孩子。

8. 结点的层(level):

规定树T的根的层为1，其余任一结点的层等于其双亲的层加1。

9. 树的深度(depth, 高度):

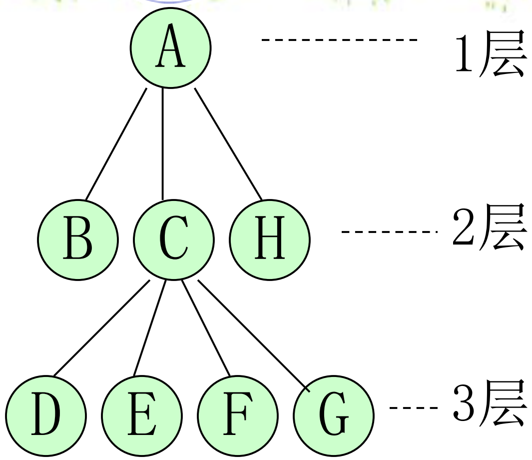
树中各结点的层的最大值。

10. 兄弟(sibling):

同一双亲的结点之间互为兄弟。

11. 堂兄弟:

双亲在同一层的结点互为堂兄弟。



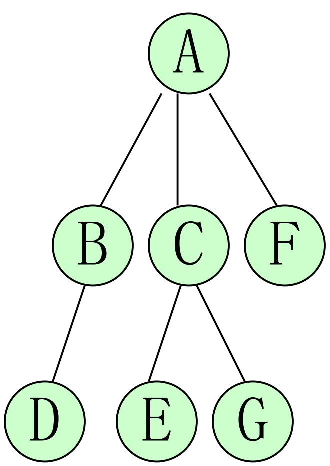
4度树

12. **祖先：** 从树的根到某结点所经分枝上的所有结点为该结点的祖先。

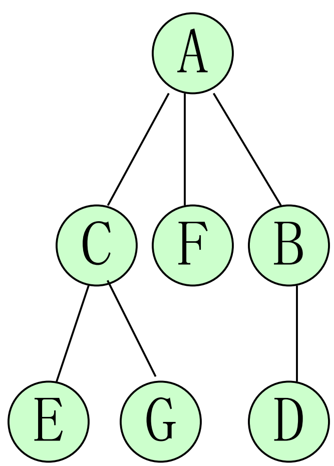
13. **子孙：** 一个结点的所有子树的结点为该结点的子孙。

14. **有序树:** 若任一结点的各棵子树，规定从左至右是有次序的, 即不能互换位置, 则称该树为有序树。

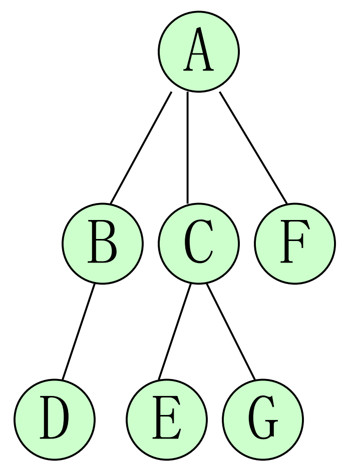
15. **无序树：** 若任一结点的各棵子树，规定从左至右是无次序的, 即能互换位置, 则称该树为无序树。



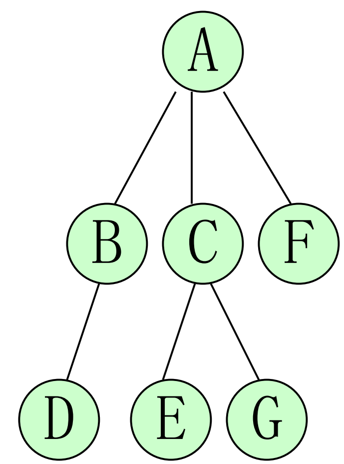
无序树T1



无序树T1



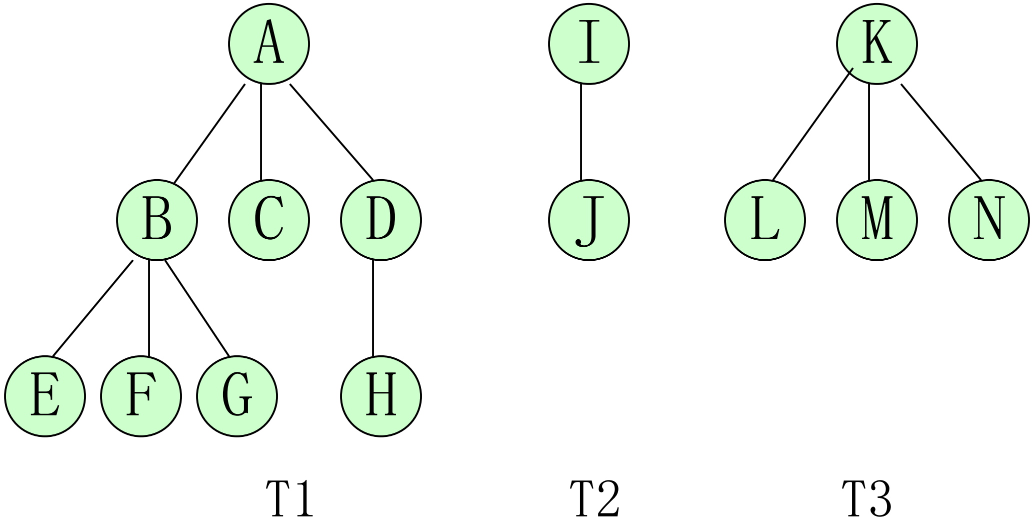
有序树T2



有序树T2

16. 森林：

m(m≥0)棵互不相交的树的集合。



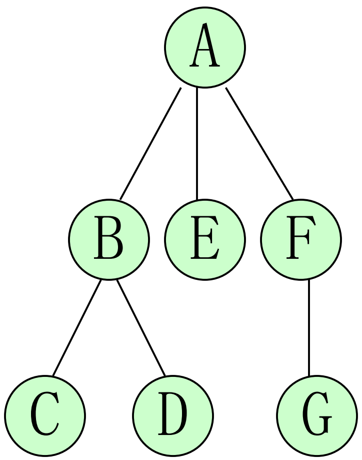
森林F={T1, T2, T3}

6.1.2. 树的其它表示形式

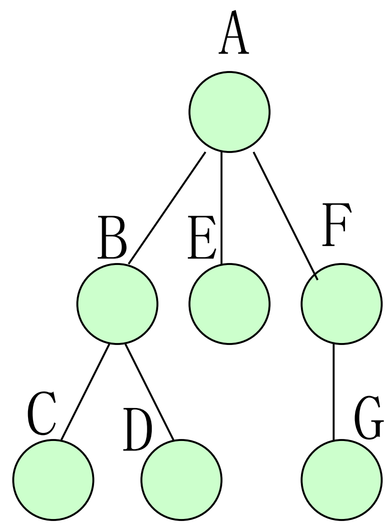
1. 广义表

树T的广义表=(T的根(T_1, T_2, . . . , T_m))

其中T_i是T的子树，也是广义表。 ($1 \leqslant i \leqslant m$)



树T

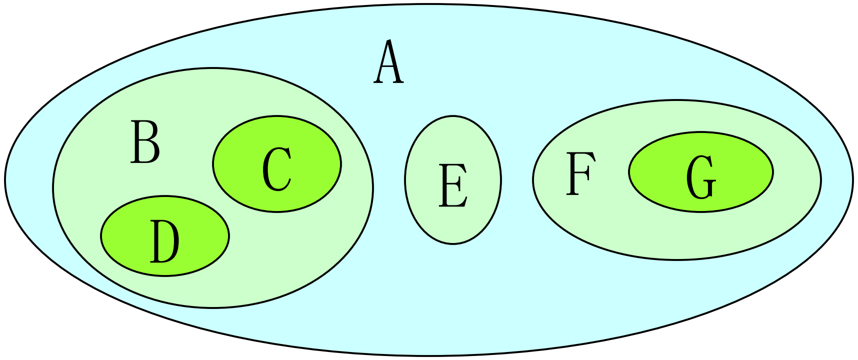


广义表A的树形表示

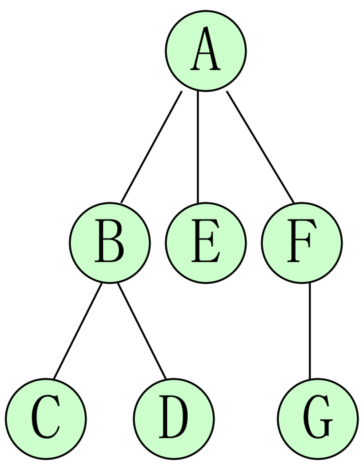
树T的广义表形式=(A(B(C,D),E,F(G)))

广义表A=(B,E,F)=((C,D),( ),(G))=((( ),( )) ,( ),(( )))

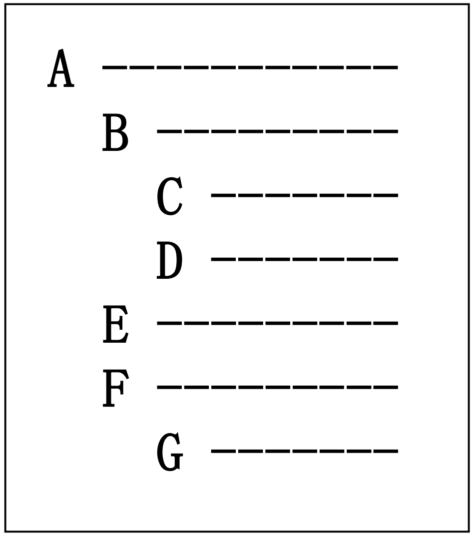
2. 嵌套集合：



3. 凹入表/书目表



树T



凹入表

6.1.3 树的基本操作

1. 置T为空树：T={ }
2. 销毁树T
3. 生成树T
4. 遍历树T：按某种规则(次序)访问树T的每一个结点一次且一次的过程。
5. 求树T的深度
6. 求树T的度
7. 插入一个结点
8. 删除一个结点
9. 求结点的层号
10. 求结点的度
11. 求树T的叶子/非叶子
12. ……………………

6.2 二叉树(binary tree)

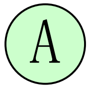
6.2.1 定义和术语

1. 二叉树的递归定义

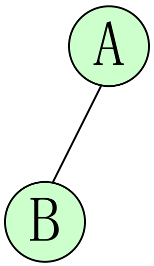
二叉树是有限个结点的集合，它或者为空集；或者是由一个根结点和两棵互不相交的，且分别称为根的左子树和右子树的二叉树所组成。

若二叉树为空集，则为空二叉树。

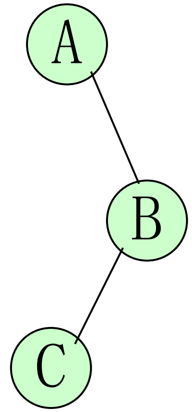
例:



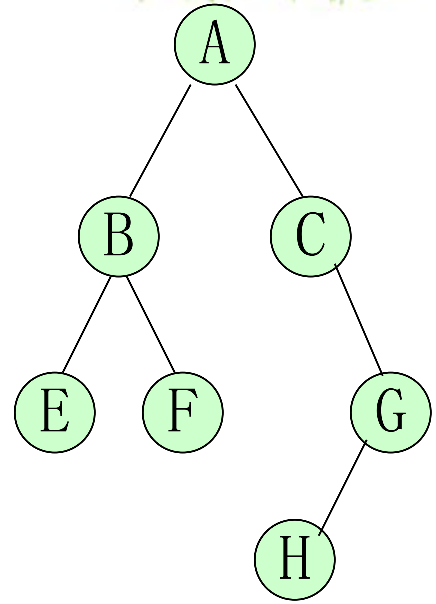
二叉树T1



二叉树T2

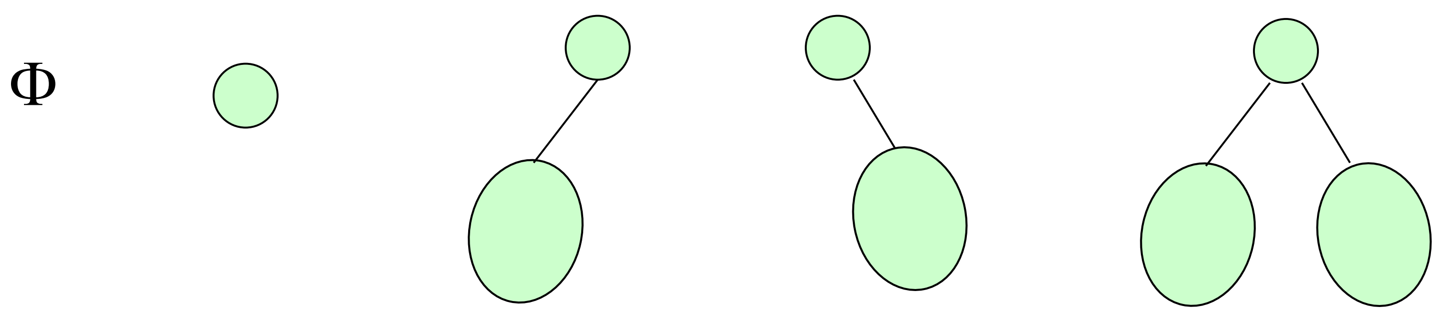


二叉树T3



二叉树T4

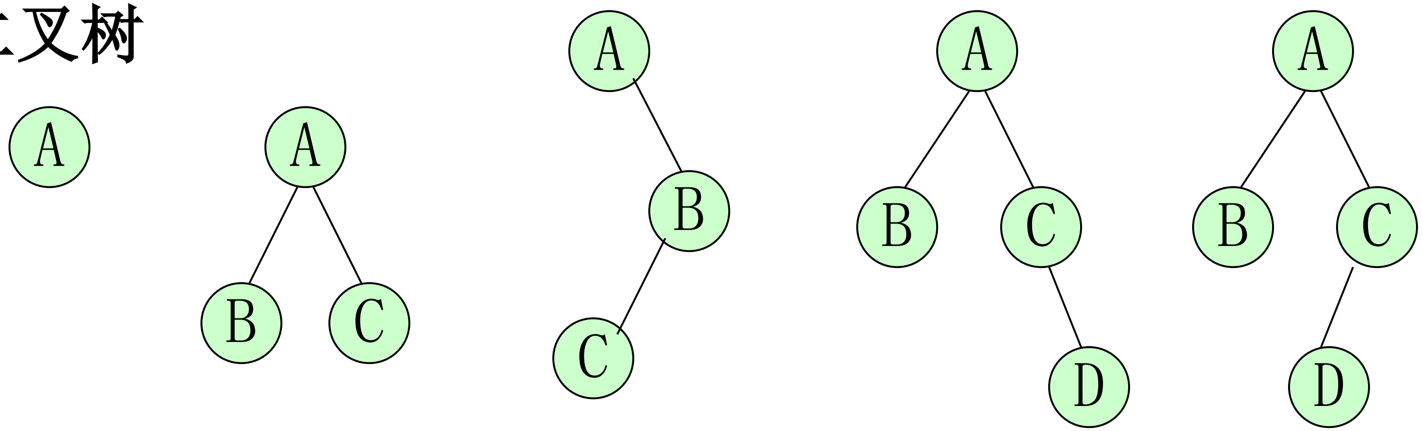
2. 二叉树的5种基本形态:



T1　T2　T3　T4　T5

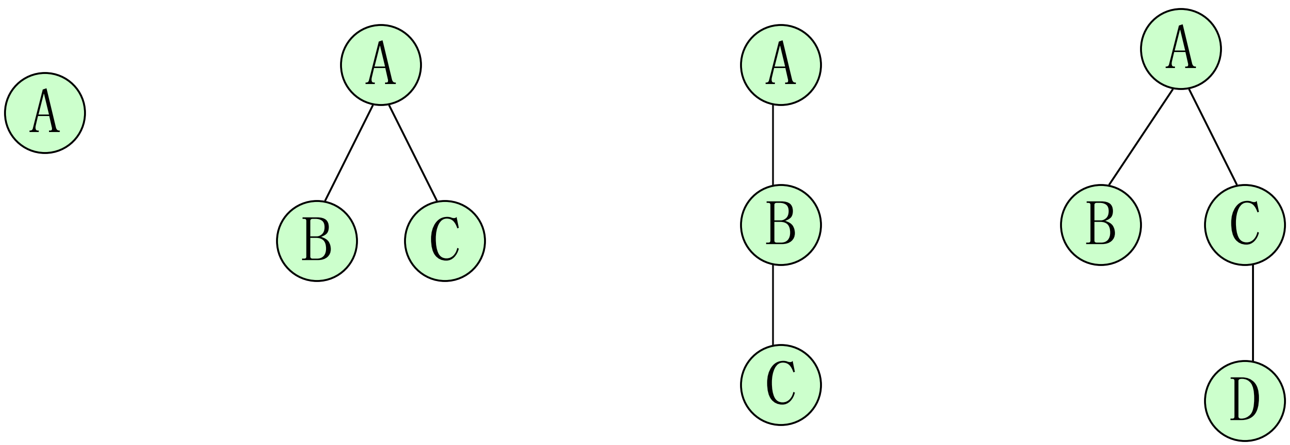
3.二叉树与2度树的区别

(1)二叉树



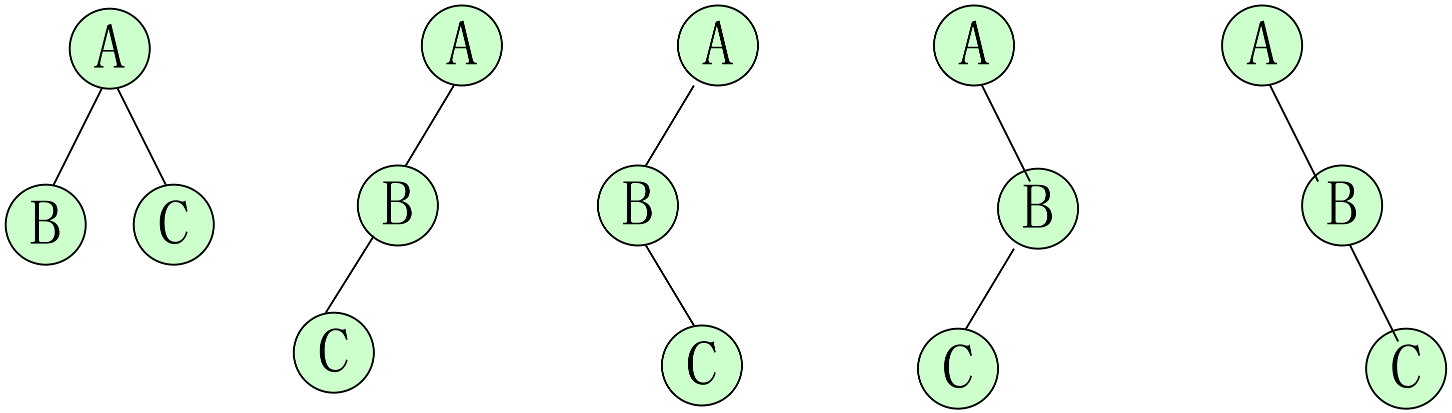
T1 T2 T3 T4 T5

(2)树



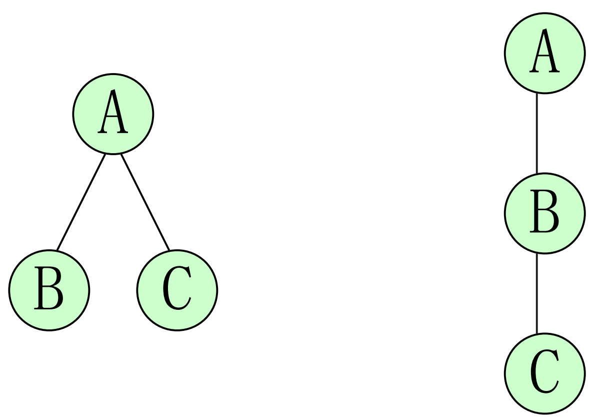
T1	T2	T3	T4
0度树	2度树	1度树	2度树

4.三个结点不同形态的二叉树(?种)



T1 T2 T3 T4 T5

5. 三个结点不同形态的树(2种)



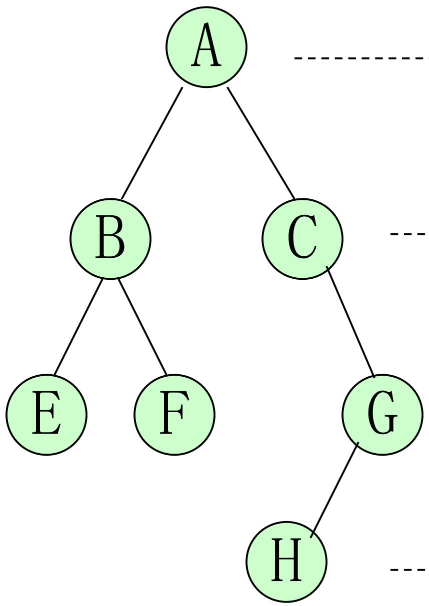
T1 T2

6.二叉树的基本操作

1.置T为空二叉树：T={ }
2.销毁二叉树T
3.生成二叉树T：生成哈夫曼树、二叉排序树、平衡二叉树、堆
4.遍历二叉树T：
　　按某种规则访问T的每一个结点一次且仅一次的过程。
5.二叉树 ←→ 树
6.二叉树 → 平衡二叉树
7.求结点的层号
8.求结点的度
9.求二叉树T的深度
10.插入一个结点
11.删除一个结点
12.求二叉树T的叶子/非叶子 …………………

6.2.2 二叉树的性质和特殊二叉树

1. 二叉树的第i(i≥1)层最多有2^{i-1}个结点



$\leq 2^0 = 1$个

$\leq 2^1 = 2$个

$\leq 2^2 = 4$个

$\leq 2^3 = 8$个

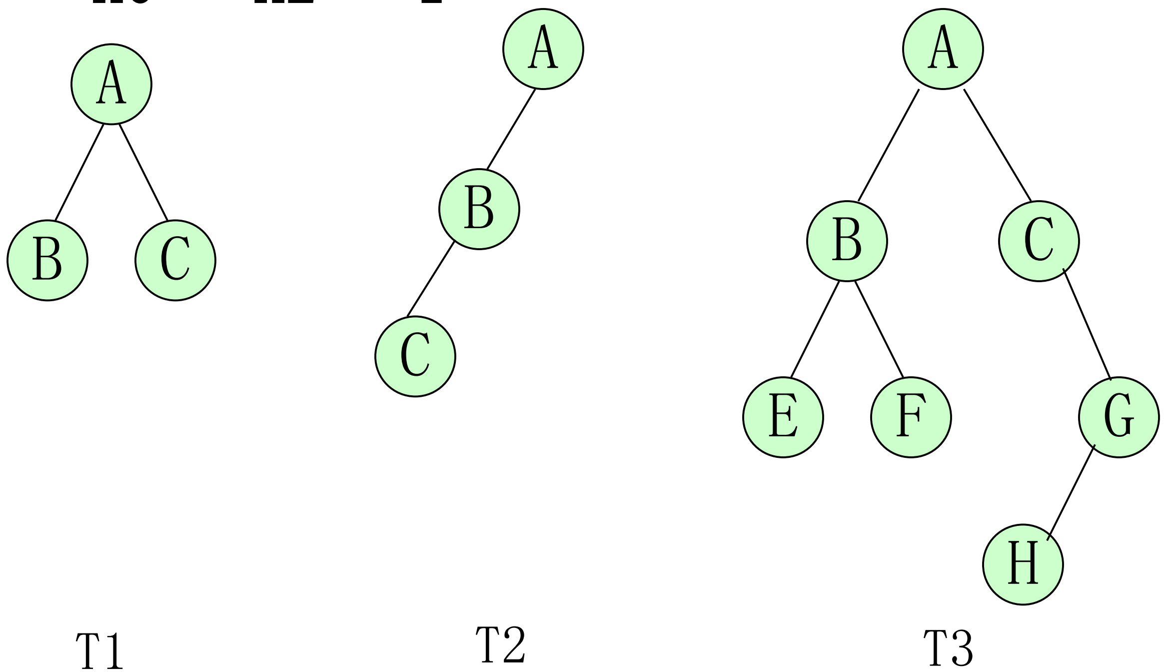
二叉树

2. 深度为k的二叉树最多有2^k-1个结点

$$2^0+2^1+\ ...\ +2^{k-1}=\frac{2^0(1-2^k)}{1-2}=2^k-1$$

3. 叶子的数目=度为2的结点数目+1

$$n0 = n2 + 1$$



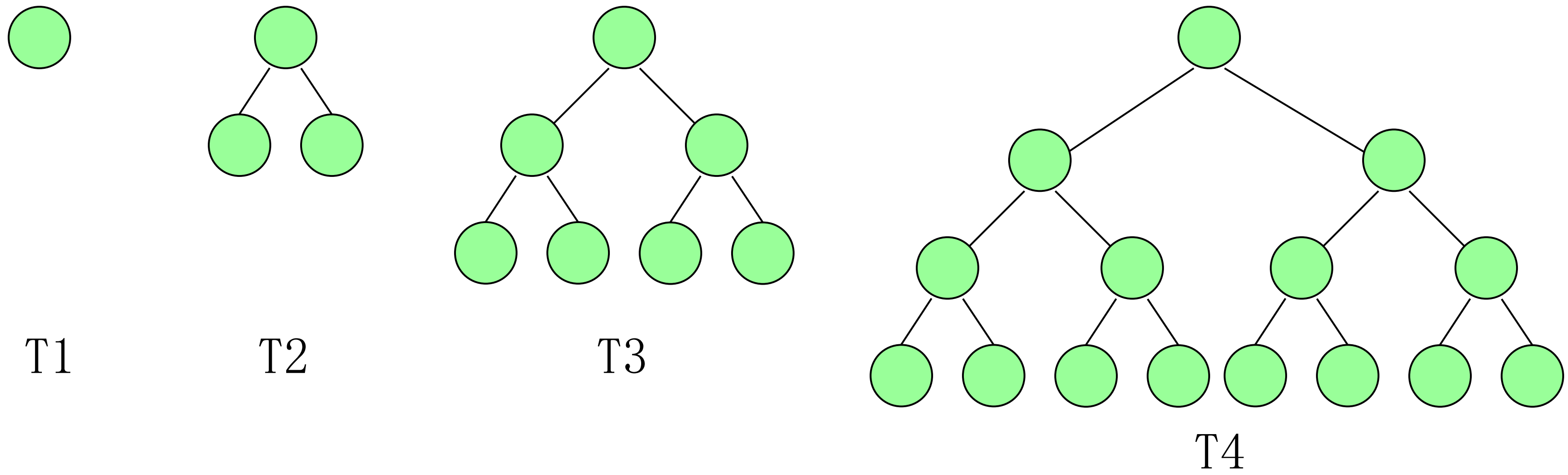
T1： n0=2, n2=1, 2=1+1

T2： n0=1, n2=0 1=0+1

T3： n0=3, n2=2 3=2+1

4.满二叉树（full binary tree)----

深度为k且有2^k-1个结点的二叉树。



（1）n个结点的满二叉树的深度=$\log_2(n+1)$

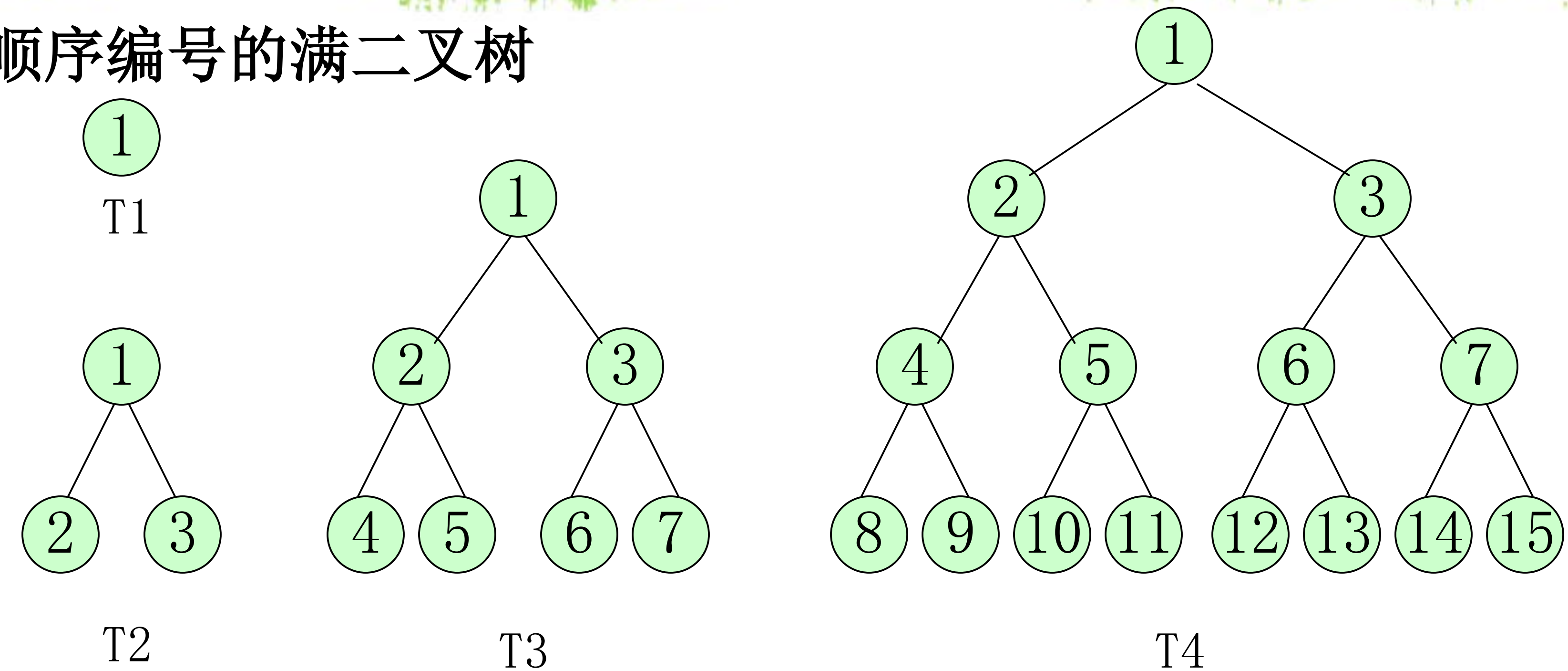
设深度为k

∵ $2^k - 1 = n$

$2^k = n + 1$

∴ $k=\log_2(n+1)$

(2)顺序编号的满二叉树



设满二叉树有n个结点，编号为1, 2, . . . , n

- 左小孩为偶数，右小孩为奇数；
- 结点i的左小孩是2i，　　$2i \leqslant n$

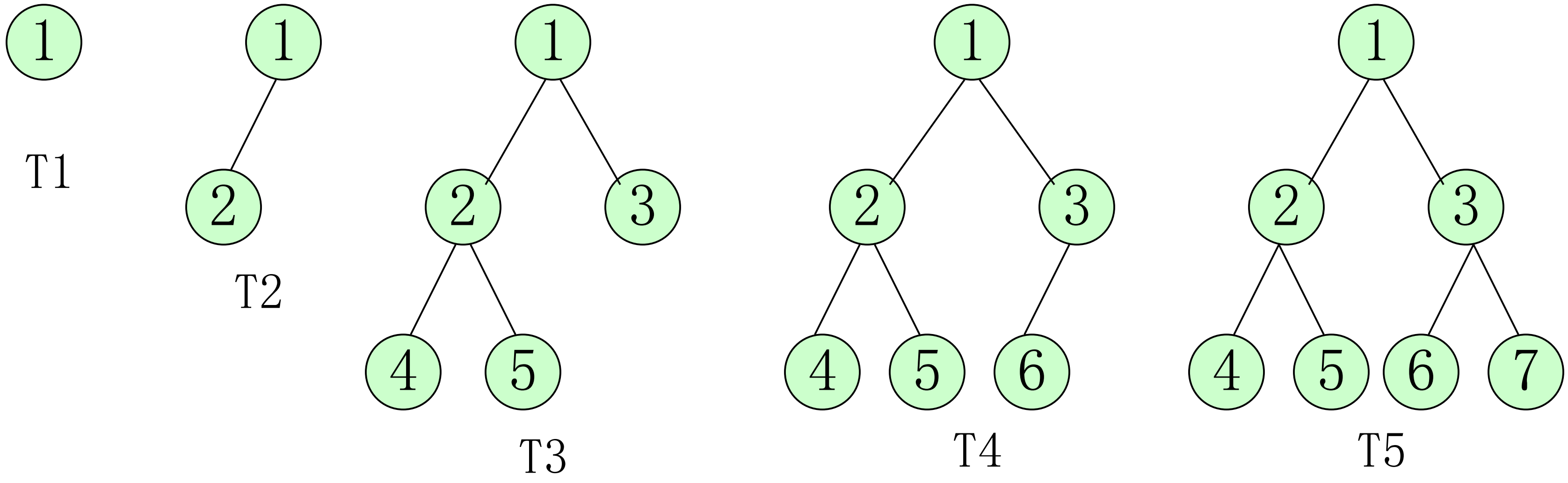
  结点i的右小孩是2i+1，　　$2i+1 \leqslant n$

  结点i的双亲是$\lfloor i/2 \rfloor$；　　$2 \leqslant i \leqslant n$
- 结点i的层号$= \lfloor \log_2 i \rfloor + 1 = \lceil \log_2(i+1) \rceil$　　$1 \leqslant i \leqslant n$

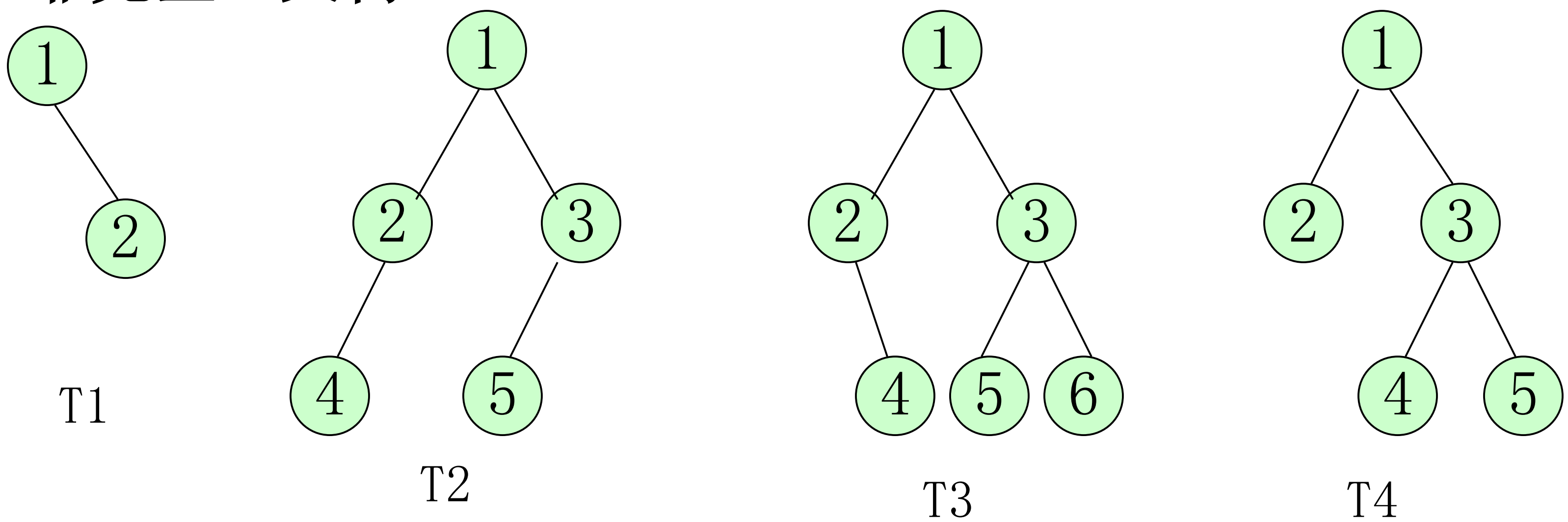
5.完全二叉树（full binary tree）:

深度为k的有n个结点的二叉树，当且仅当每一个结点都与同深度的满二叉树中编号从1至n的结点一一对应，称之为完全二叉树（其它教材称为“顺序二叉树”）。

例. 完全二叉树：



例 非完全二叉树:



n(n>0)个结点的完全二叉树的深度?

$$\lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil$$

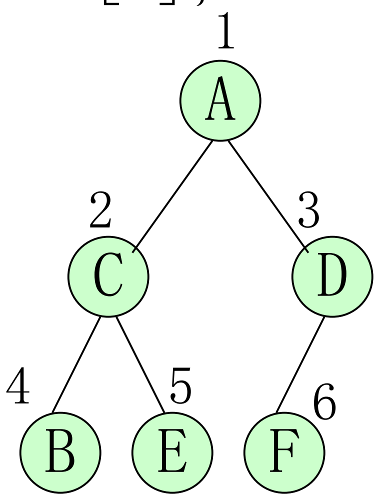
6.2.3 二叉树的存储结构

1.顺序结构

（1） n个结点的完全二叉树，使用一维数组:

例.　　ElemType tree[n+1];

　　　　char t[7];



T1

t[1..6]

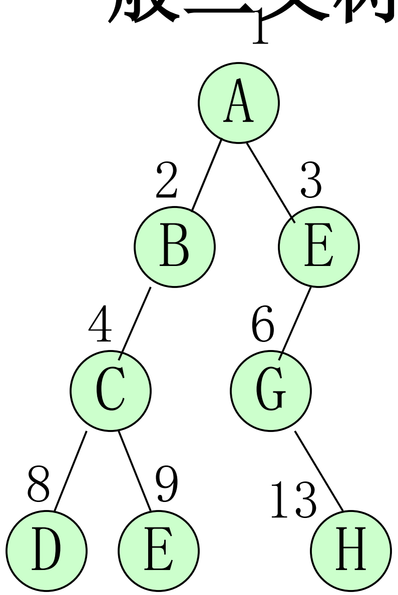
//	A	C	D	B	E	F
0	1	2	3	4	5	6

T1的顺序结构

父子关系

- 元素(结点)t[i]的双亲是t[i/2],　　　2≤i≤n
- 元素(结点)t[i]的左小孩是t[2*i],　　2i≤n
- 元素(结点)t[i]的右小孩是t[2*i+1],　2i+1≤n

(2) 一般二叉树



T2

t[1..13]

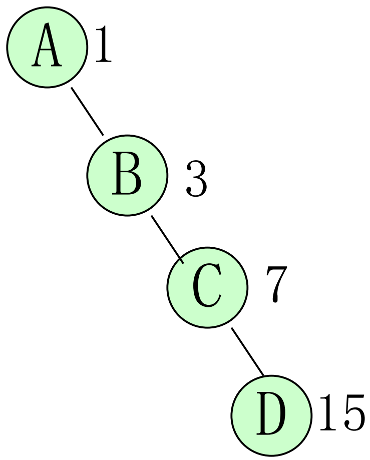
A	B	E	C		G		D	E				H
1	2	3	4	5	6	7	8	9	10	11	12	13

T2的顺序结构

父子关系

- 若t[i]存在,t[i]的双亲是t[i/2]; $2 \leqslant i \leqslant n$
- 若t[2*i]存在,t[i]的左小孩是t[2*i]; $2i \leqslant n$
- 若t[2*i+1]存在,t[i]的右小孩是t[2*i+1]。 $2i+1 \leqslant n$

(3)右单枝树



T3

t[1..15]

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A		B				C								D

T3的顺序结构

深度为k的二叉树，最多需长度为2^k-1的一维数组。
若是右单枝树，空间利用率为:

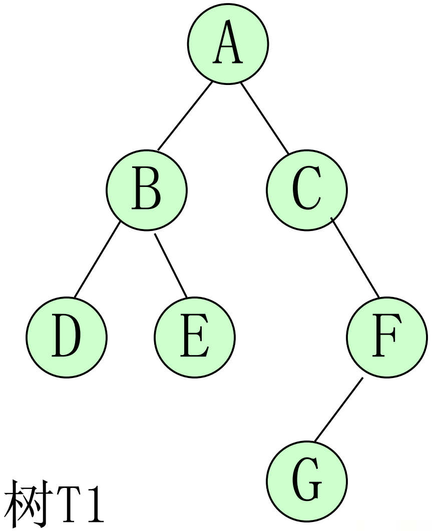
$$\alpha = \frac{k}{2^k - 1}$$

k=4,　$\alpha=4/15$
k=10,　$\alpha=10/1023$
　　　≈ 0.0098

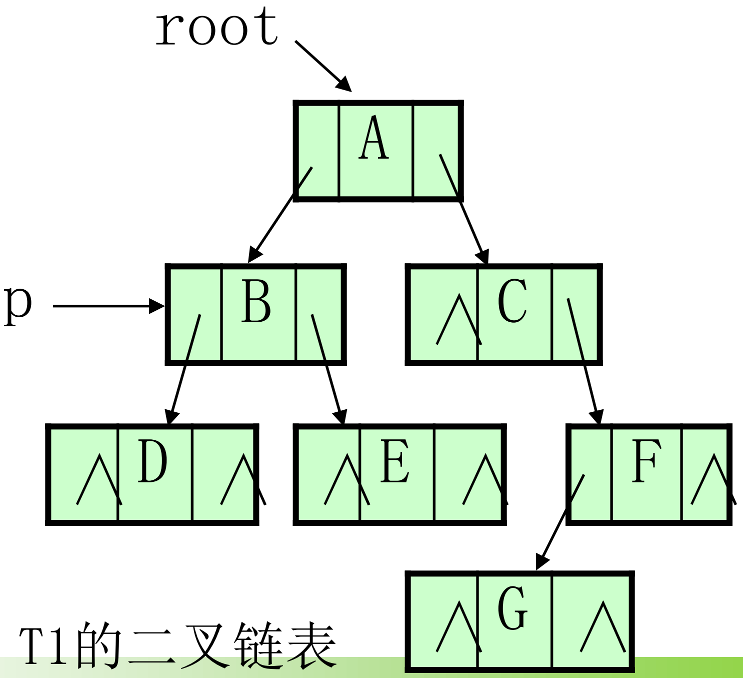
2. 链式存储结构

(1) 二叉链表

```
struct BiTNode                    //结点类型
{ struct BiTNode * lchild;        //左孩子指针
  ElemType data;                  //抽象数据元素类型
  struct BiTNode * rchild;        //右孩子指针
 } *root, *p;
```



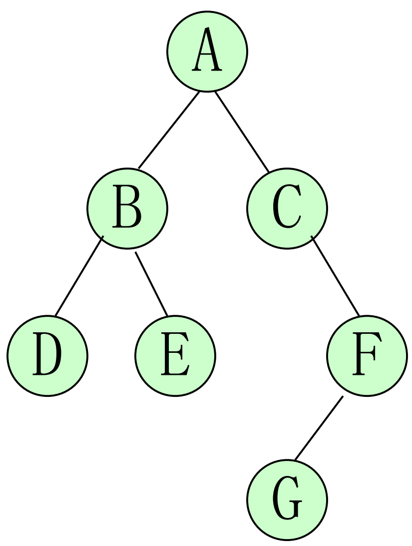
二叉树T1



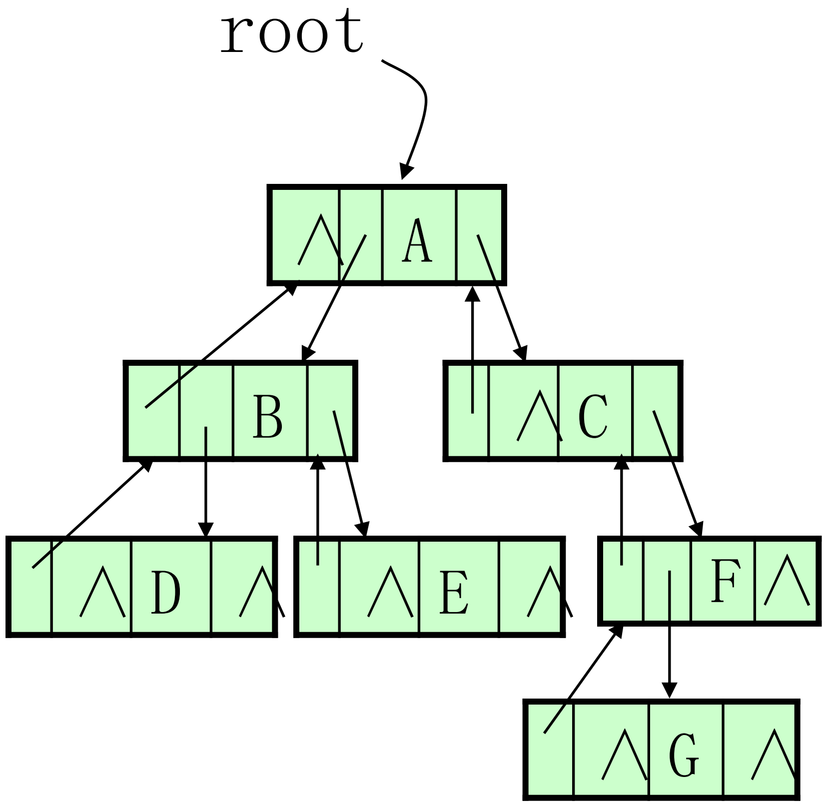
T1的二叉链表

(2)三叉链表

```
struct BiTNode
{ struct BiTNode *parent,*lchild,*rchild ;
  ElemType data;
 } *root,*p;
```



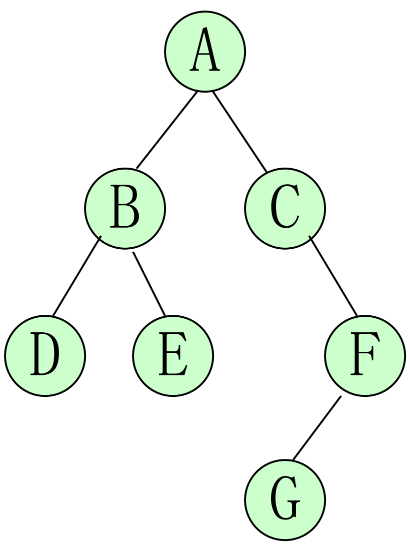
二叉树T2



T2的三叉链表

(3)静态链表

```
struct bnode2
{ ElemType data;
  int lchild,rchild;
 }t[n+1];
```



二叉树

root

	lchild	data	rchild
0	///	///	///
1	2	A	3
2	4	B	5
3	0	C	6
4	0	D	0
5	0	E	0
6	7	F	0
7	0	G	0

一维数组t[0..7]

6.3 遍历二叉树和线索二叉树

6.3.1 遍历二叉树

按某种规则访问二叉树的每一个结点一次且仅一次的过程。

设：D----访问根结点,输出根结点；

L----递归遍历左二叉树；

R----递归遍历右二叉树。

遍历规则(方案)：

DLR----前序遍历(先根,preorder)

LDR----中序遍历(中根,inorder)

LRD----后序遍历(后根,postorder)

DRL----逆前序遍历

RDL----逆中序遍历

RLD----逆后序遍历

前序遍历

前序遍历二叉树递归定义：

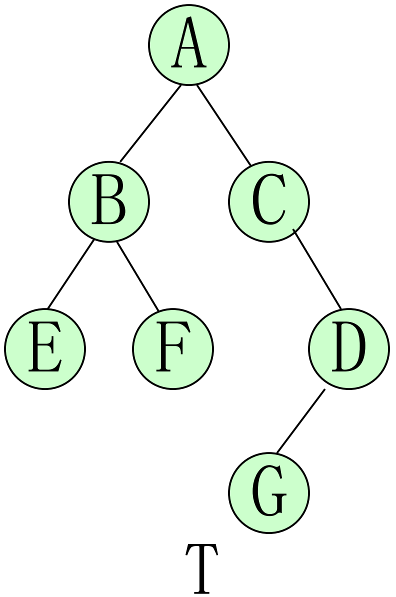
若二叉树为空，则遍历结束；

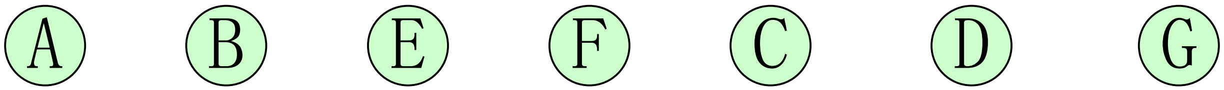
否则，执行下列步骤：

（1）访问根结点；

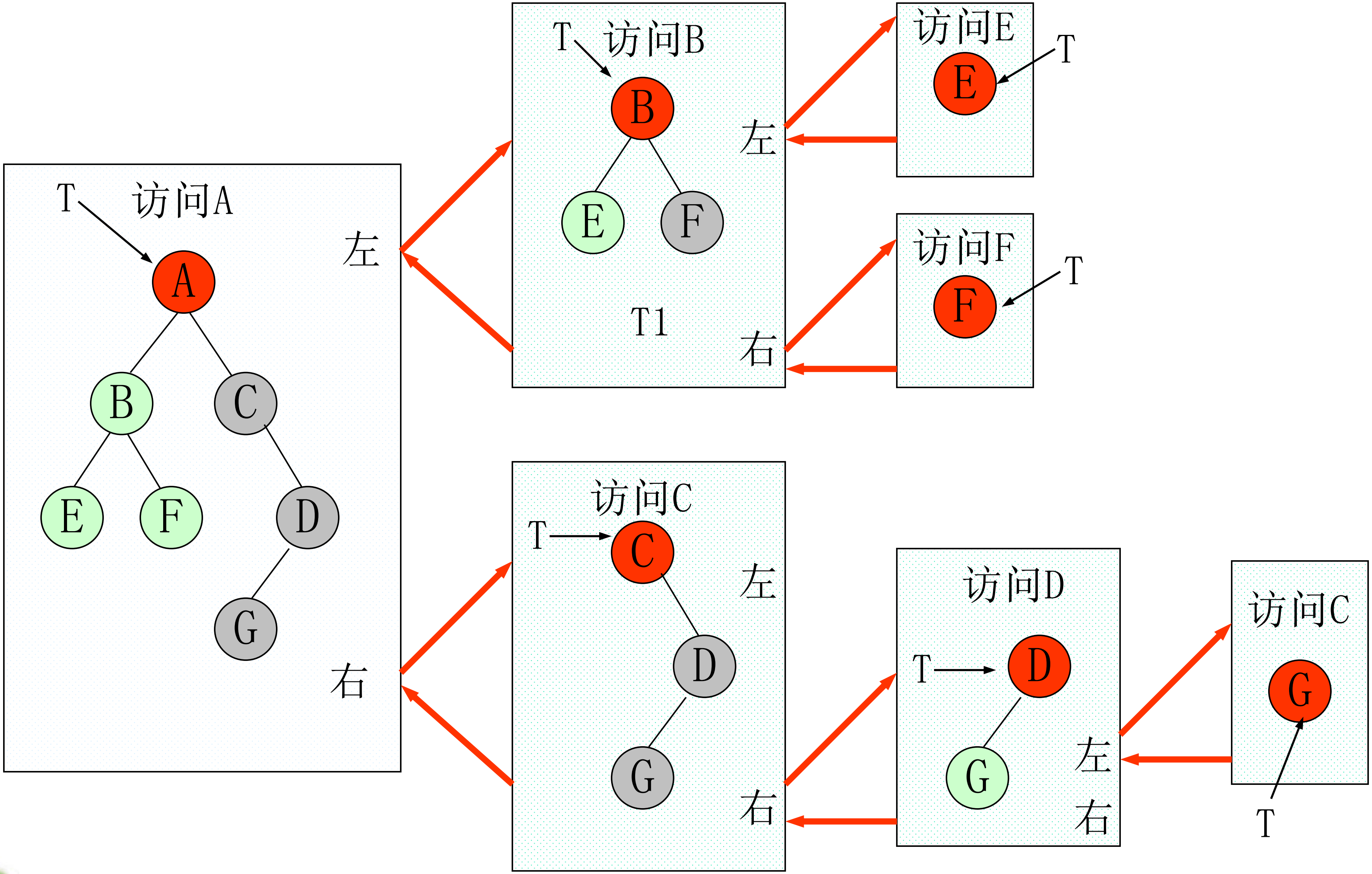
（2）遍历根的左子树；

（3）遍历根的右子树。





前序遍历递归算法过程:



T
访问A
A
B
C
E
F
D
G
左
右
T
访问B
B
E
F
T1
左
右
访问E
E
T
访问F
F
T
访问C
T
C
D
G
左
右
访问D
T
D
G
左
右
访问C
G
T

先序遍历递归算法:

```
typedef  struct BiTNode *BiTree;  //结点指针类型
void  PreOrderTraverse(BiTree T)
//T是指向二叉链表根结点的指针
{
   if (!T)     return;
   else
   {
   printf( “%c” ,T->data);          //访问根指针
   PreOrderTraverse(T->lchild);  //递归访问左子树
   PreOrderTraverse(T->rchild);  //递归访问右子树
   }
  return;
 }
```

2. 中序遍历

中序遍历二叉树递归定义：

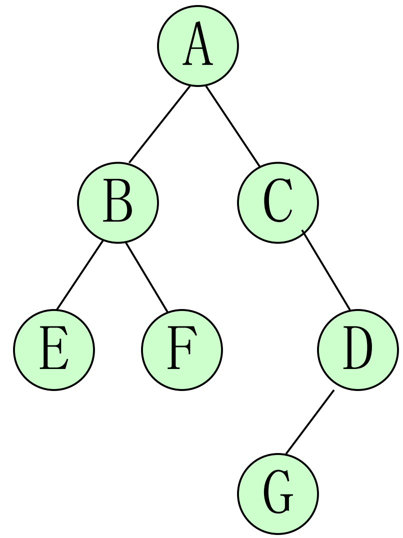
若二叉树为空，则遍历结束；

否则，执行下列步骤：

（1）遍历根的左子树；

（2）访问根结点；

（3）遍历根的右子树。



T

E B F A C G D

中序遍历递归算法

```
typedef  struct BiTNode *BiTree;   //结点指针类型
void  MidOrderTraverse(BiTree   T)
//T是指向二叉链表根结点的指针
{
   if (T)
  {
   MidOrderTraverse(T->lchild);  //递归访问左子树
   printf( "%c" ,T->data);         //访问根指针
   MidOrderTraverse(T->rchild);  //递归访问右子树
   }
  return;
 }
```

3. 后序遍历

后序遍历二叉树递归定义：

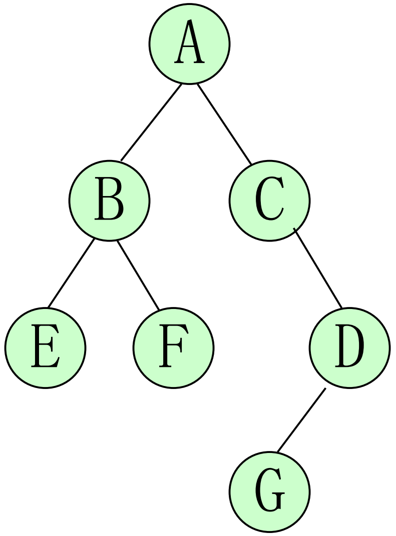
若二叉树为空，则遍历结束；

否则，执行下列步骤：

（1）遍历根的左子树；

（2）遍历根的右子树；

（3）访问根结点。



T

E F B G D C A

后序遍历递归算法

```
typedef  struct BiTNode *BiTree;  //结点指针类型
void  PostOrderTraverse(BiTree   T)
//T是指向二叉链表根结点的指针
{
   if (T)
  {
   PostOrderTraverse(T->lchild);  //递归访问左子树
   PostOrderTraverse(T->rchild);  //递归访问右子树
   printf( “%c” ,T->data);        //访问根指针
   }
  return;
 }
```

4. 非递归算法(以中序遍历为例说明)

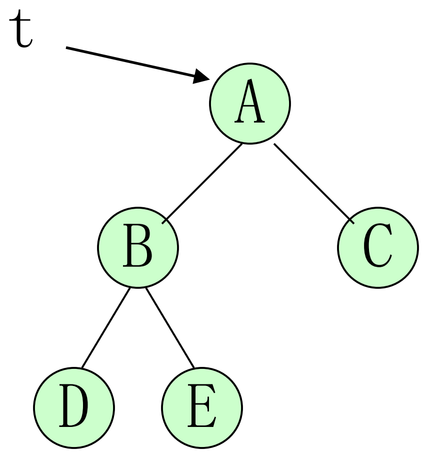
- 递归算法简明精炼，但效率较低，实际应用中常使用非递归；
- 某些高级语言不支持递归；
- 非递归算法思想：

（1）设置一个栈S存放所经过的根结点（指针）信息；初始化S；

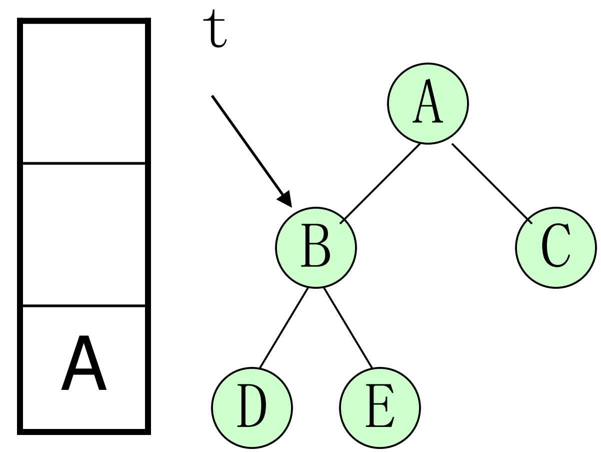
（2）第一次访问到根结点并不访问，而是入栈；

（3）中序遍历它的左子树，左子树遍历结束后，第二次遇到根结点，就将根结点（指针）退栈，并且访问根结点；然后中序遍历它的右子树。

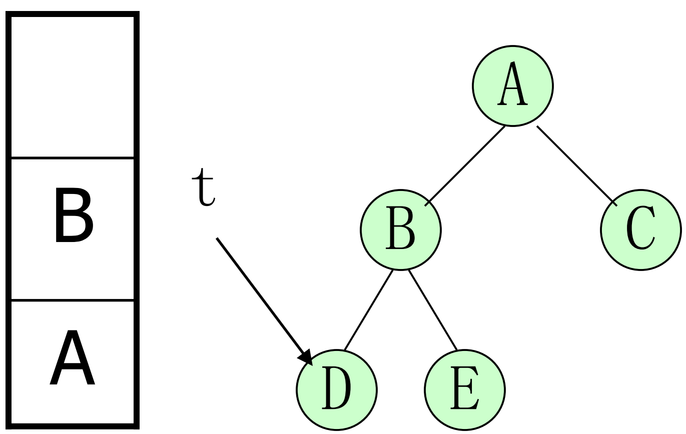
（4） 当需要退栈时，如果栈为空则结束。



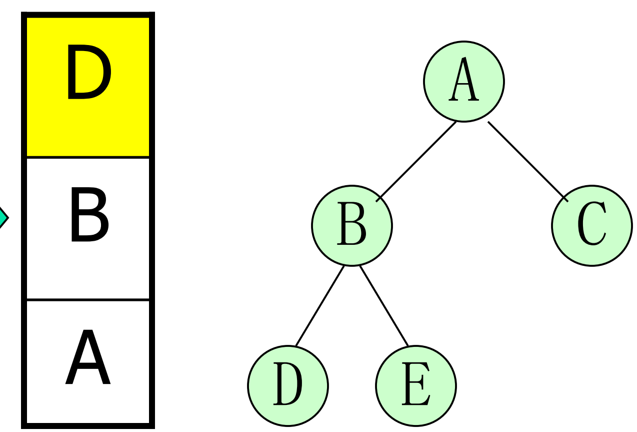
根进栈



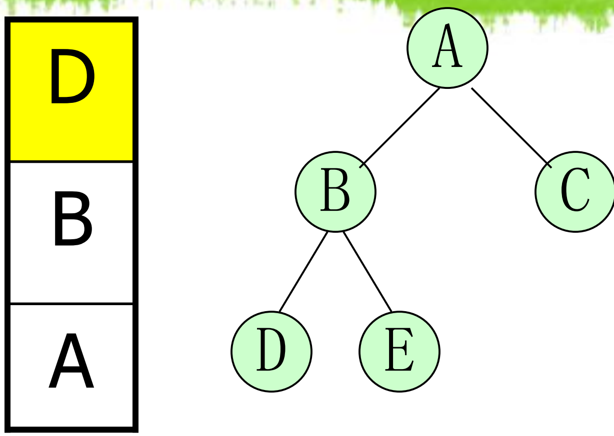
根进栈



根进栈



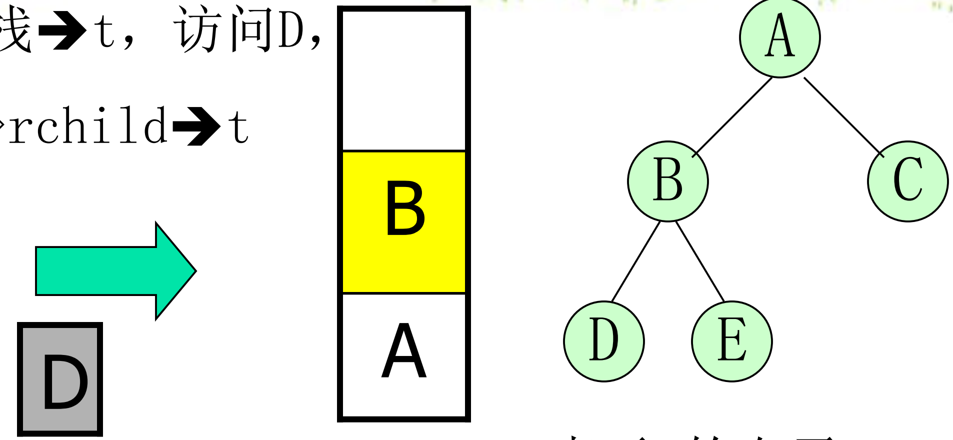
t=NULL，表示D的左子树遍历结束



t=NULL,表示D的左子树遍历结束

退栈➔t，访问D，

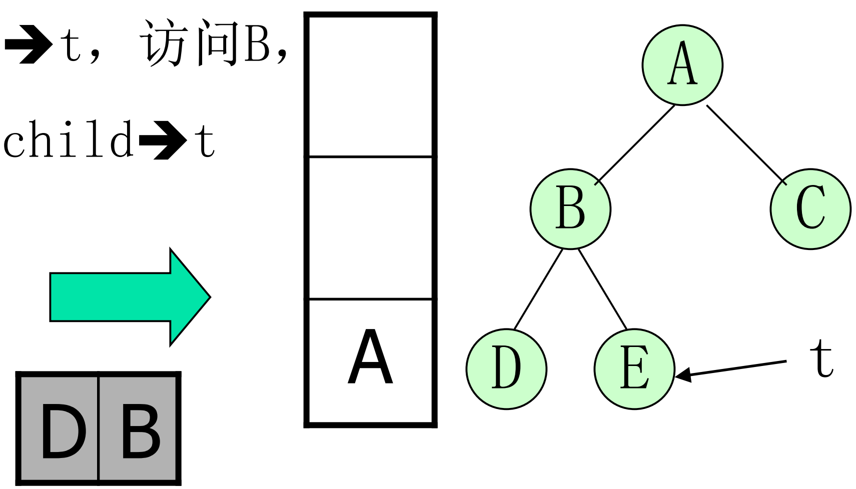
t->rchild➔t



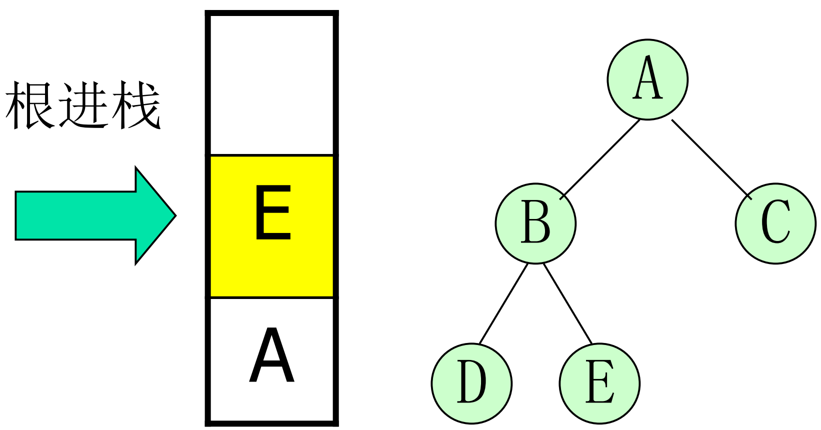
t=NULL,表示D的右子树为空，也即B的左子树遍历结束

退栈➔t，访问B，

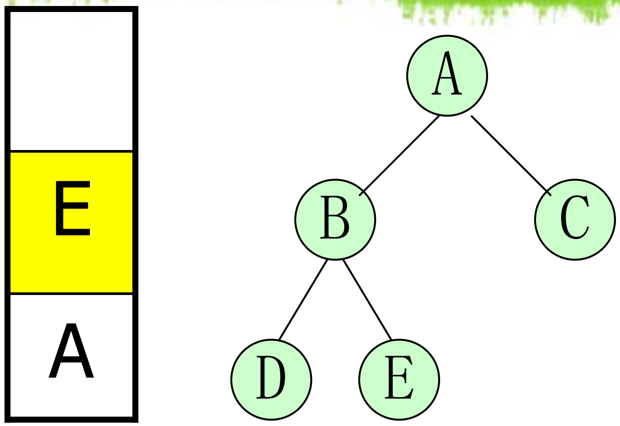
t->rchild➔t



根进栈



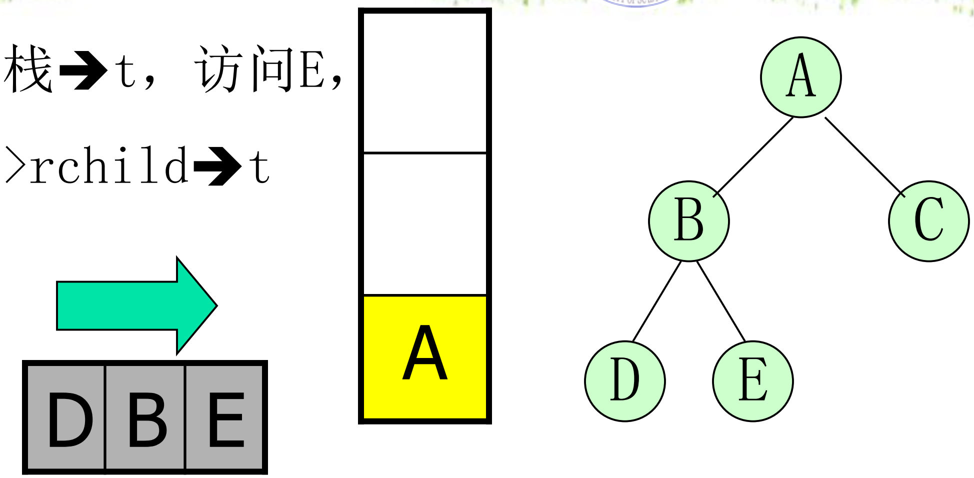
t=NULL,表示E的左子树遍历结束



t=NULL,表示E的左子树遍历结束

退栈→t，访问E，

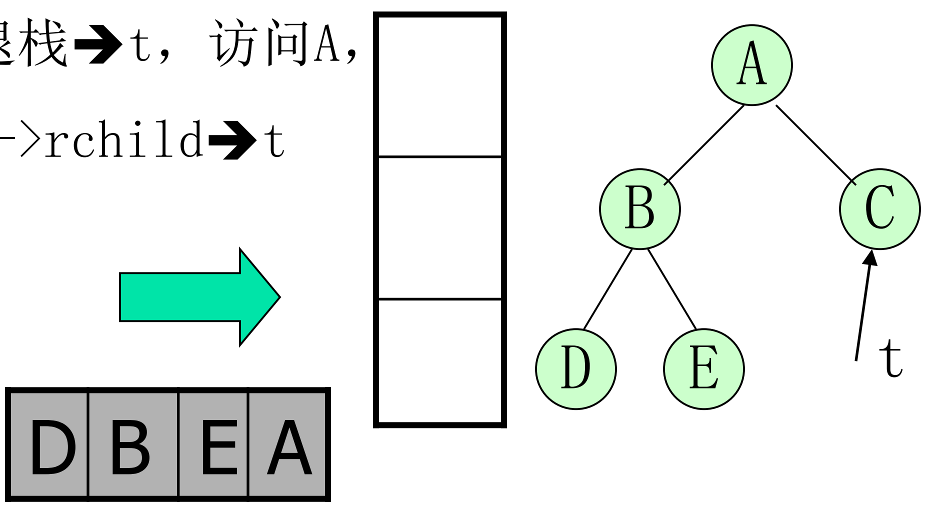
t->rchild→t



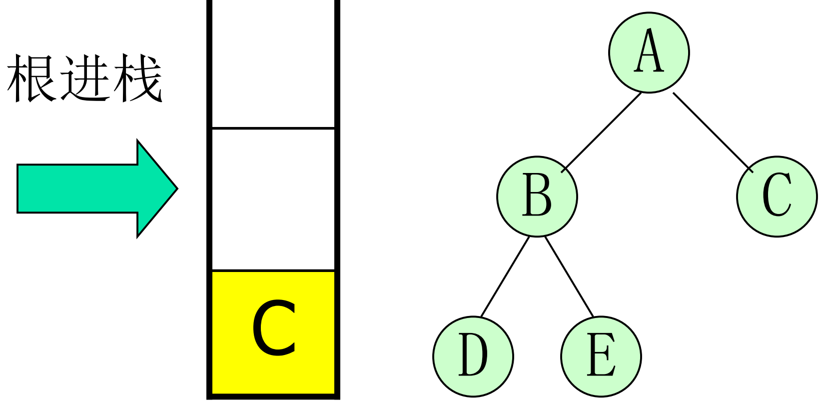
t=NULL， t此时为A的左子树最右结点的右孩子。表示A的左子树遍历结束

退栈→t，访问A，

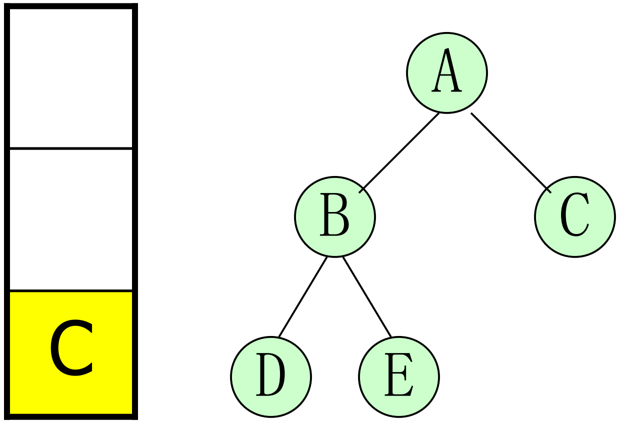
t->rchild→t



根进栈



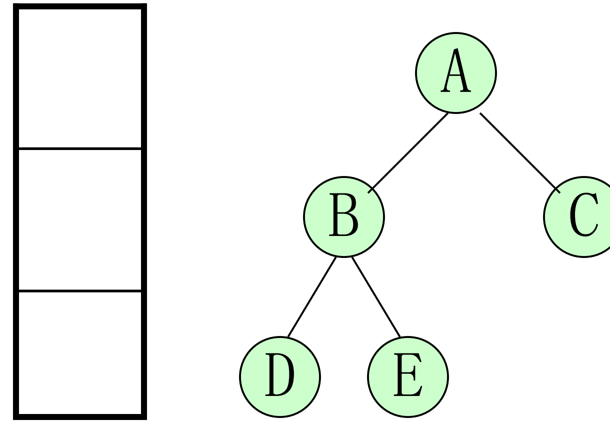
t=NULL,表示C的左子树遍历结束



t=NULL，表示C的左子树遍历结束

退栈➔t，访问C，
t->rchild➔t

D	B	E	A	C



t=NULL，并且栈S为空，遍历结束

中序遍历非递归算法

```
void Midorder(struct BiTNode *t)      //t为根指针
{ struct BiTNode *st[maxleng];//定义指针栈
  int top=0;                          //置空栈
  do{
    while(t)                     //根指针t表示的为非空二叉树
    { if (top==maxleng) exit(OVERFLOW); //栈已满,退出
      st[top++]=t;                    //根指针进栈
      t=t->lchild;                    //t移向左子树
     }   //循环结束表示以栈顶元素的指向为根结点的二叉树
         //的左子树遍历结束
    if (top)                          //为非空栈
    { t=st[--top];                    //弹出根指针
      printf("%c",t->data);           //访问根结点
      t=t->rchild;                    //遍历右子树
     }
   } while(top||t);  //父结点未访问，或右子树未遍历
 }
```

先序遍历非递归算法

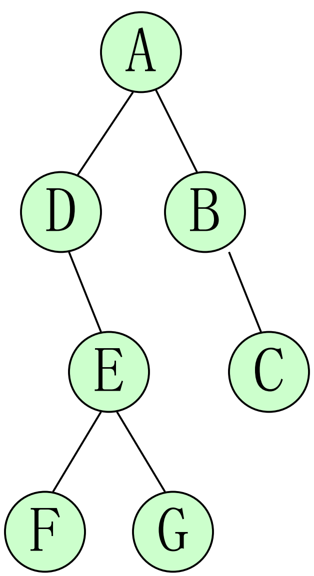
```
void Preorder(struct BiTNode * t){
  struct BiTNode * St[MaxSize], *p;
  int top = 0;                         //置空栈
  if (t! = NULL){
    St[top++] = t;
    while(top){
     p = St[--top]; printf("%c ", p->data);
     if(p->rchild != NULL)
        St[top++] = p->rchild;
     if(p->lchild != NULL)
        St[top++] = p->lchild;
    }
    printf("\n");
  }
}
```

后序遍历非递归算法

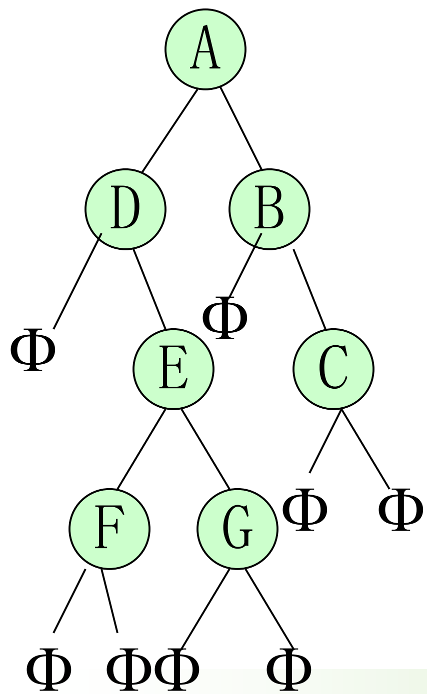
```
void Postorder(struct BiTNode * t){
  struct BiTNode * St[MaxSize], *pre;
  int flag, top = 0;
  if (t != NULL){
     do{
        while(t != NULL){
          St[top++] = t; t = t->lchild;
        }
        pre = NULL; flag = 1;
        while(top && flag){
          t = St[top-1];
          if(t->rchild == pre){
              printf( “%c ” , t->data); top--;  pre = t;
          }
          else{ t=t->rchild; flag = 0;}
        }
     }while(top);
    printf( “\n” );
  }
}
```

6.3.2 建立(生成)二叉树

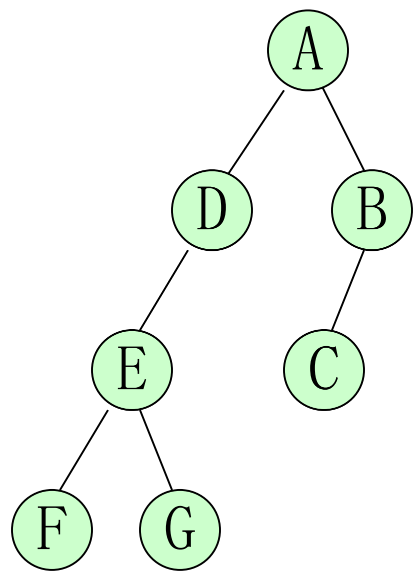
- 二叉树T1的先序序列：
  "ADEFGBC"
  带空二叉树的先序序列：
  "ADΦEFΦΦGΦΦBΦCΦΦ"
  其中：'Φ'表示空二叉树



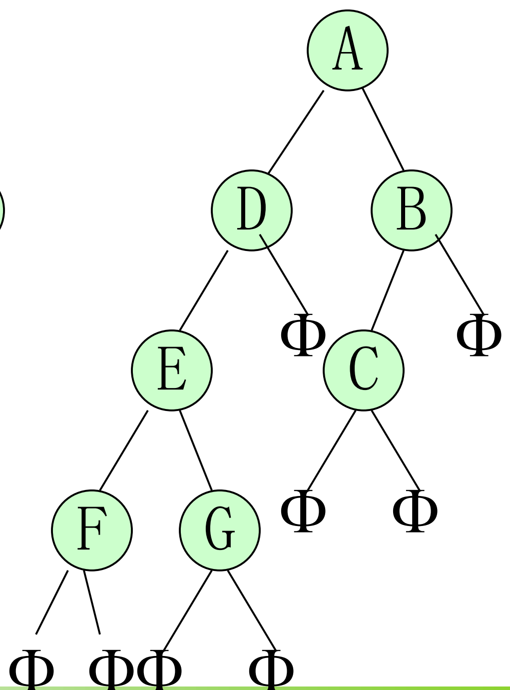
T1



- 二叉树T2的先序序列：
  "ADEFGBC"
  带空二叉树的先序序列：
  "ADEFΦΦGΦΦΦBCΦΦΦ"

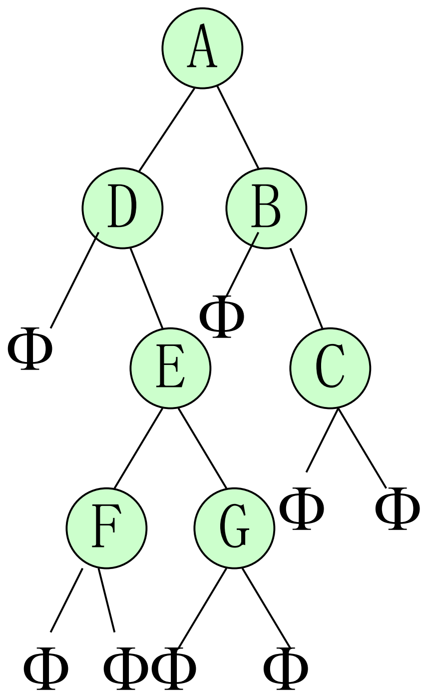


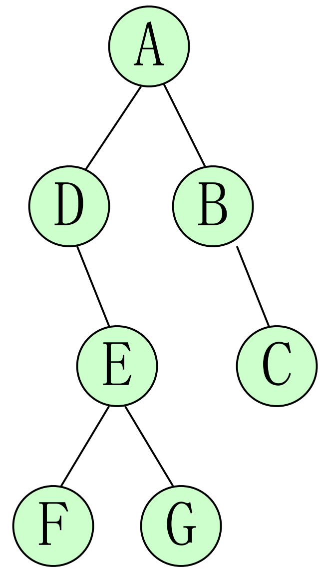
T2



建立方法:

A D Φ E F Φ ΦG Φ ΦB Φ C Φ Φ





T1

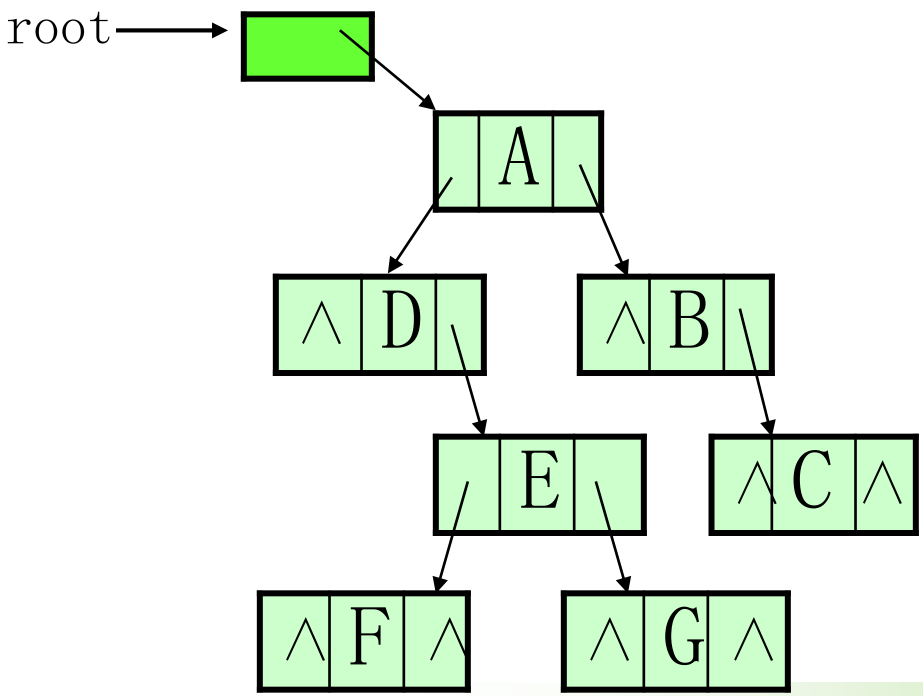
算法：创建二叉树

输入：带空结点的二叉树的先序序列

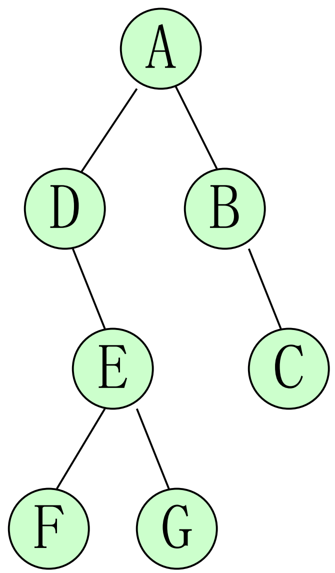
输出：二叉树的根指针

```
#define leng sizeof(BiTNode) //结点所占空间大小
```

假设输入先序序列：ADΦEFΦΦGΦΦBΦCΦΦ



二叉链表



二叉树

实现算法1:

```
void CreatBiTree1(struct BiTNode **root)
//root是指向二叉链表根指针的指针
{ char ch;
  scanf(“%c”,&ch);   /输入一个结点，假定为字符型
  if (ch ==’Φ’) (*root)=NULL;
  else
  {
  (*root)=(struct BiTNode *)malloc(leng);
  (*root)->data=ch;                //生成根结点
  CreatBiTree1(&(*root)->lchild); //递归构造左子树
  CreatBiTree1(&(*root)->rchild); //递归构造右子树
  }
}
```

实现算法2:

```
BiTree CreatBiTree2()
 { char ch;  BiTree root;  //二叉链表根结点指针
   scanf( "%c",&ch);        //输入一个结点
   if (ch ==' Φ ')
     root=NULL;
   else {
     root=(BiTree) malloc(leng);  //生成根结点
     root->data=ch;
     root->lchild=CreatBiTree2(); //递归构造左子树
     root->rchild=CreatBiTree2(); //递归构造右子树
       }
  return root;
 }
```

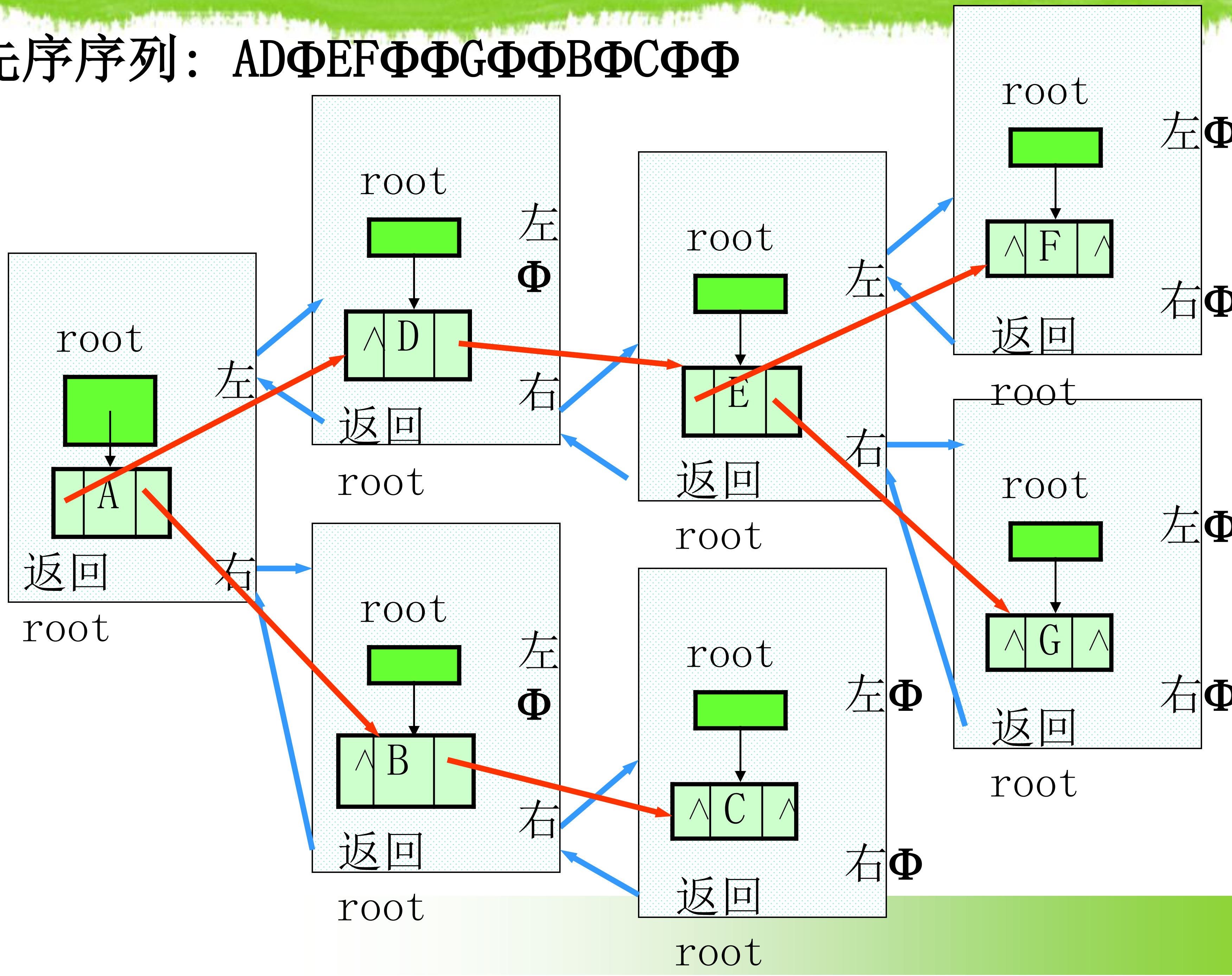
实现算法3:

```
void CreatBiTree3(BiTree  &root)
{ char ch;
  scanf("%c",&ch);              //输入一个结点
  if (ch ==' Φ ')     root=NULL;
  else
  {
    root=(BiTree) malloc(leng);   //生成根结点
    root->data=ch;
    CreatBiTree3(root->lchild);//递归构造左子树
    CreatBiTree3(root->rchild); //递归构造右子树
  }
}
```

主函数(调用方法）：

```
main( )
{
  struct BiTNode *root1,*root2;
  BiTree         root3;
  CreatBiTree1(&root1);                /*算法1*/
  root2=CreatBiTree2();                /*算法2*/
  CreatBiTree3(root3);                 /*算法3*/
}
```

先序序列：ADΦEFΦΦGΦΦBΦCΦΦ



算法4：用非递归算法创建二叉树

输入：　各结点的值及其在满二叉树中的编号

输出：　二叉树根指针

```
#define MAXSIZE 100
BiTree CreatTree()
  { BiTree   s[MAXSIZE+1],root,q;
    int i,j;
    ElemType x;
    printf("i,x=");
    scanf("%d%d",&i,&x);
```

```
while(i!=0)
    { q=(BiTNode *) malloc(sizeof(BiTNode));
      q->data=x;q->lchild=q->rchild=NULL;
      s[i]=q;
      if (i==1) root=q;
      else  { j=i/2;
              if (i%2)    s[j]->rchild=q;
              else        s[j]->lchild=q;
               }
      printf("i,x=");  scanf("%d%d",&i,&x);
    }
  return root;
}
```

6.3.3 线索二叉树

遍历二叉树是按某种规则将非线性结构的二叉树结点线性化。

- 二叉树结点中没有相应前驱和后继的信息。每次遍历时需按规则动态产生。
- n个节点的二叉树：
- 有：n*2 个指针域
- 使用：n-1 个指针。除根以外，每个结点被一个指针指向
- 空指针域数： n*2-(n-1)=n+1

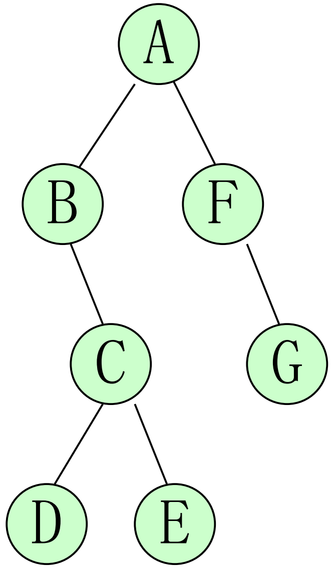
- 当对某二叉树经常按某种规则遍历访问时，可利用空指针域。将空的左指针域指向直接前驱，空的右指针域指向直接后继，非空指针不需要改变。称该处理过程称为**二叉树线索化**。由此可分别得到：

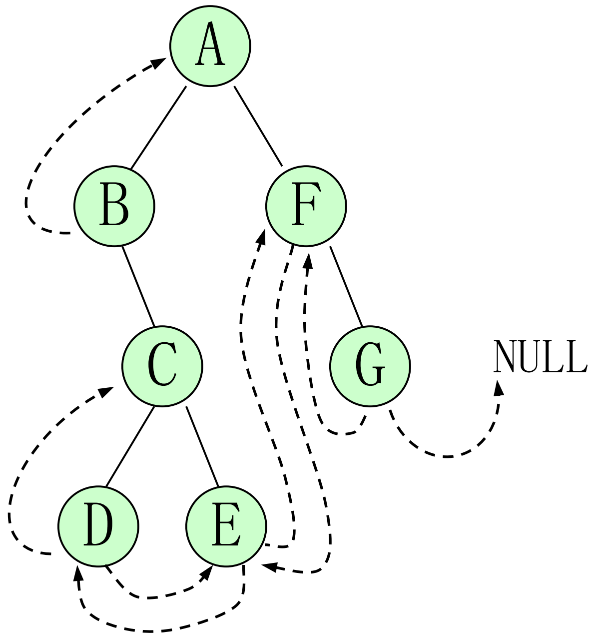
- 前序线索二叉树，中序线索二叉树， 后序线索二叉树

1. 前序线索二叉树:

线索指向前序遍历中前趋、后继的线索二叉树。

例. T的前序序列：A, B, C, D, E, F, G





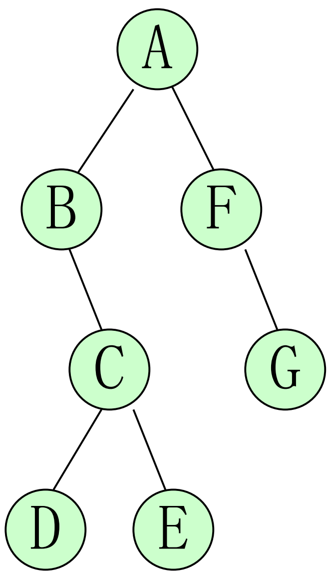
二叉树T

T的前序线索二叉树

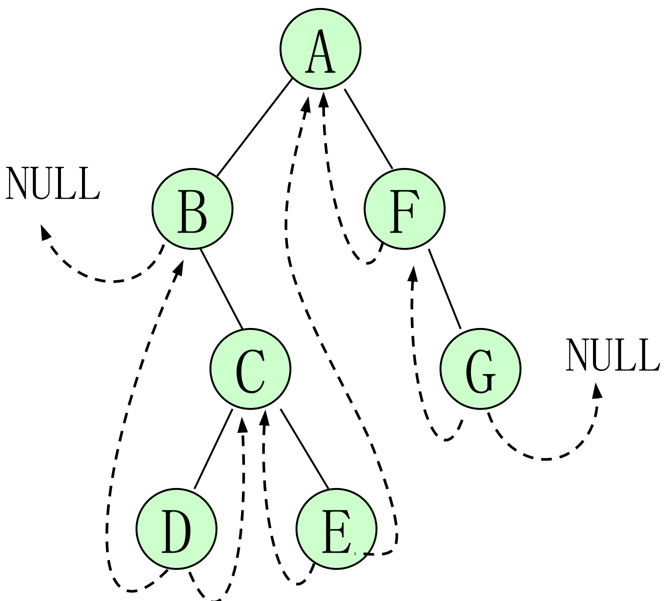
2. 中序线索二叉树:

线索指向中序遍历中前趋、后继的线索二叉树。

例. T的中序序列：B, D, C, E, A, F, G



二叉树T

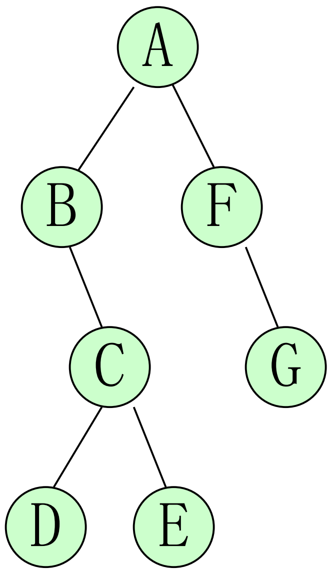


T的中序线索二叉树

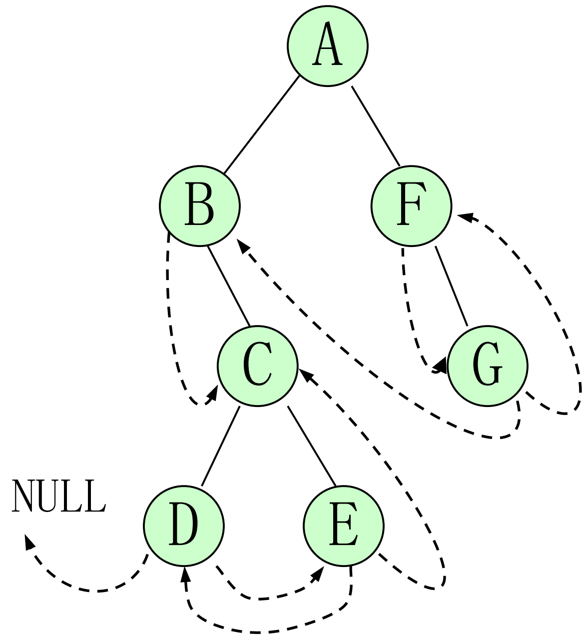
3.后序线索二叉树:

线索指向后序遍历中前趋、后继的线索二叉树。

例. T的后序序列：D,E,C,B,G,F,A



二叉树T

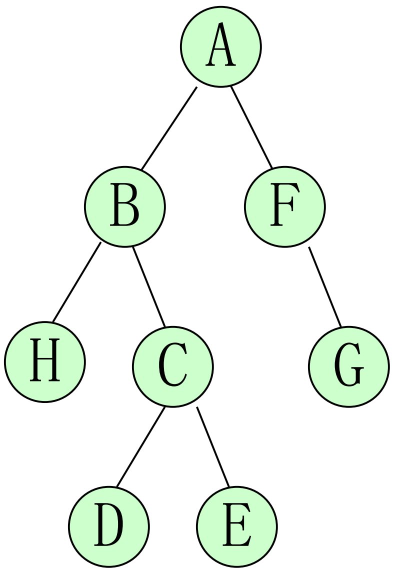


T的后序线索二叉树

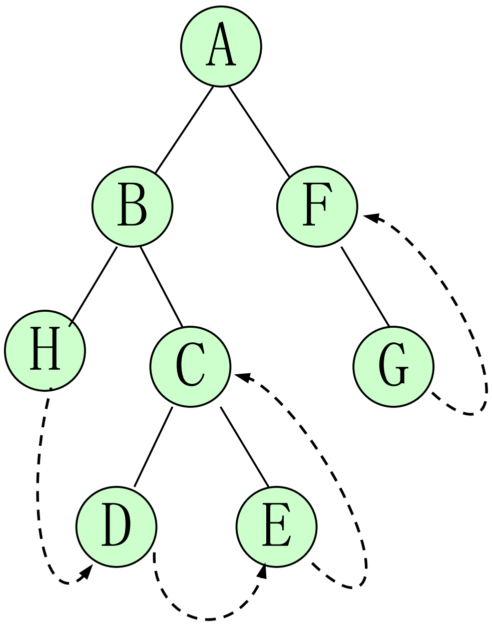
4.后序后继线索二叉树:

只设指向后序遍历中后继线索的线索二叉树。

例. T的后序序列：H,D,E,C,B,G,F,A



二叉树T

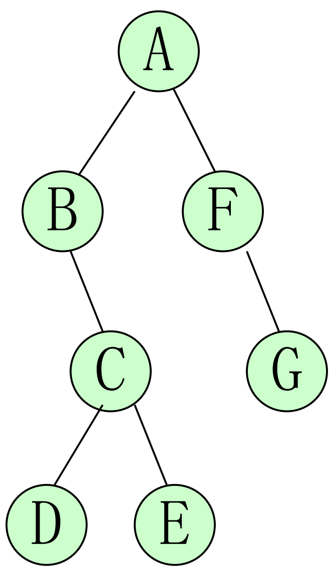


T的后序后继线索二叉树

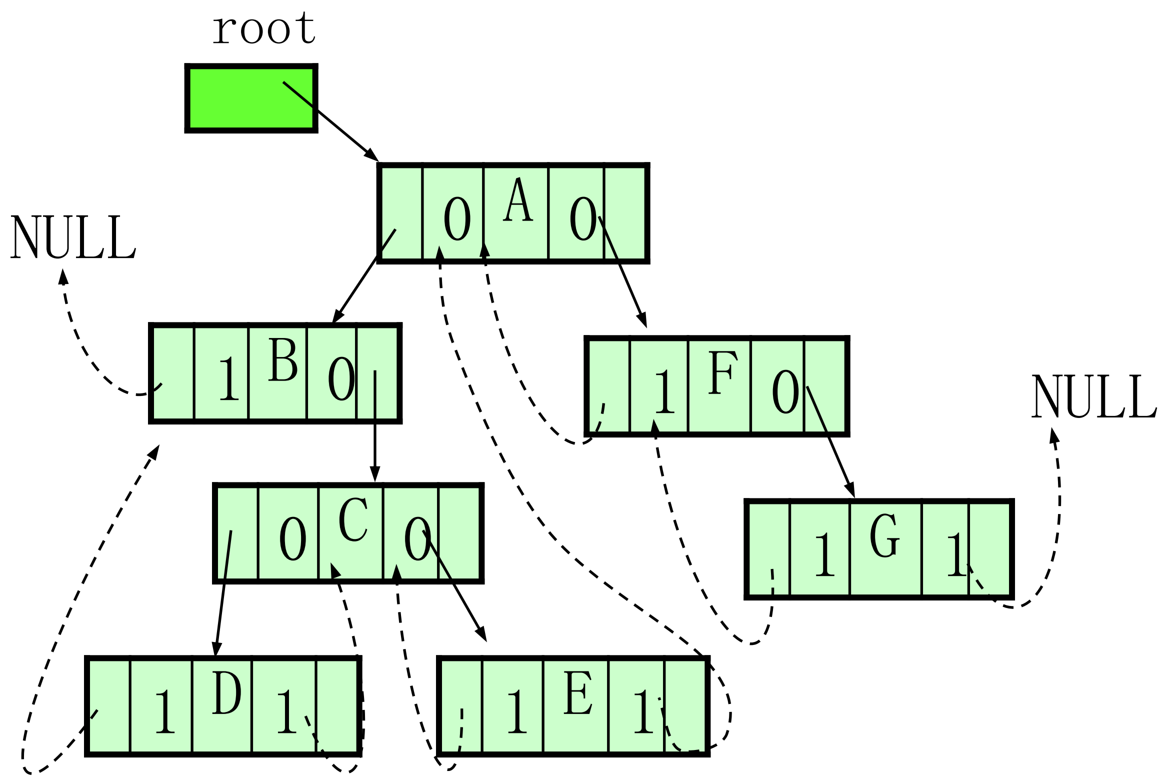
5.线索二叉树的存储结构

(1).结点结构：

lchild	ltag	data	rtag	rchild
	0/1		0/1	
左小孩或前趋	左标志	结点值	右标志	右小孩或后继

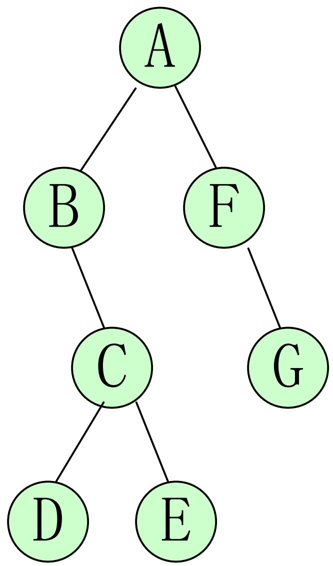


二叉树

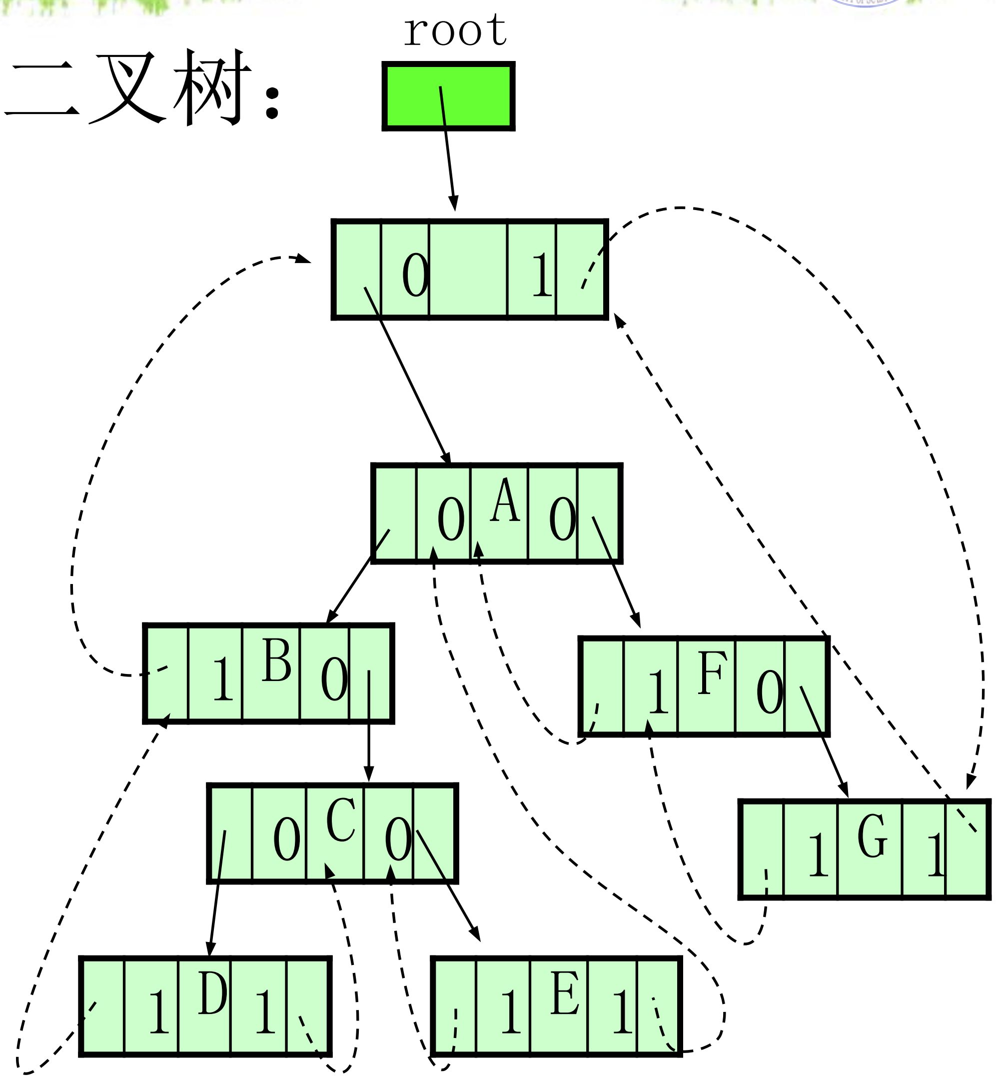


中序线索二叉树链表

带头结点的线索二叉树：



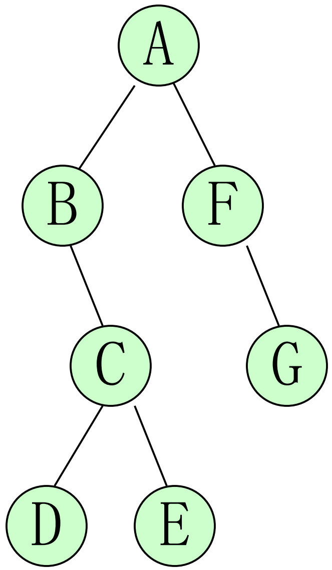
二叉树



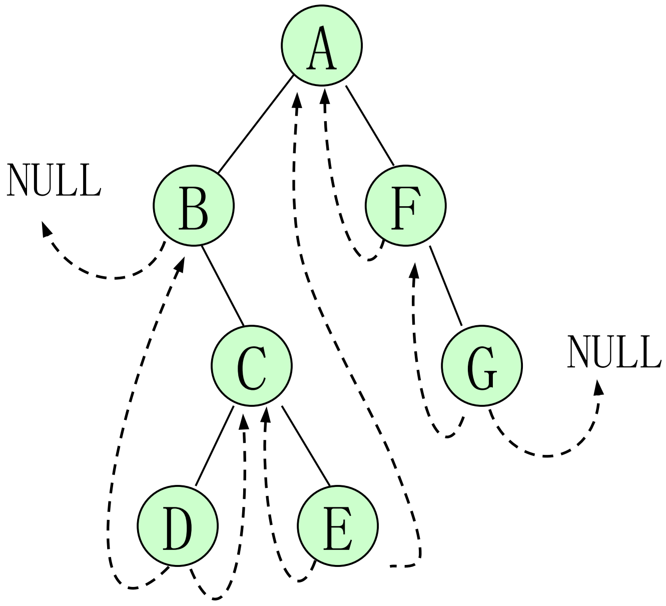
中序线索二叉树链表

(2) 将二叉树线索化（中序）。

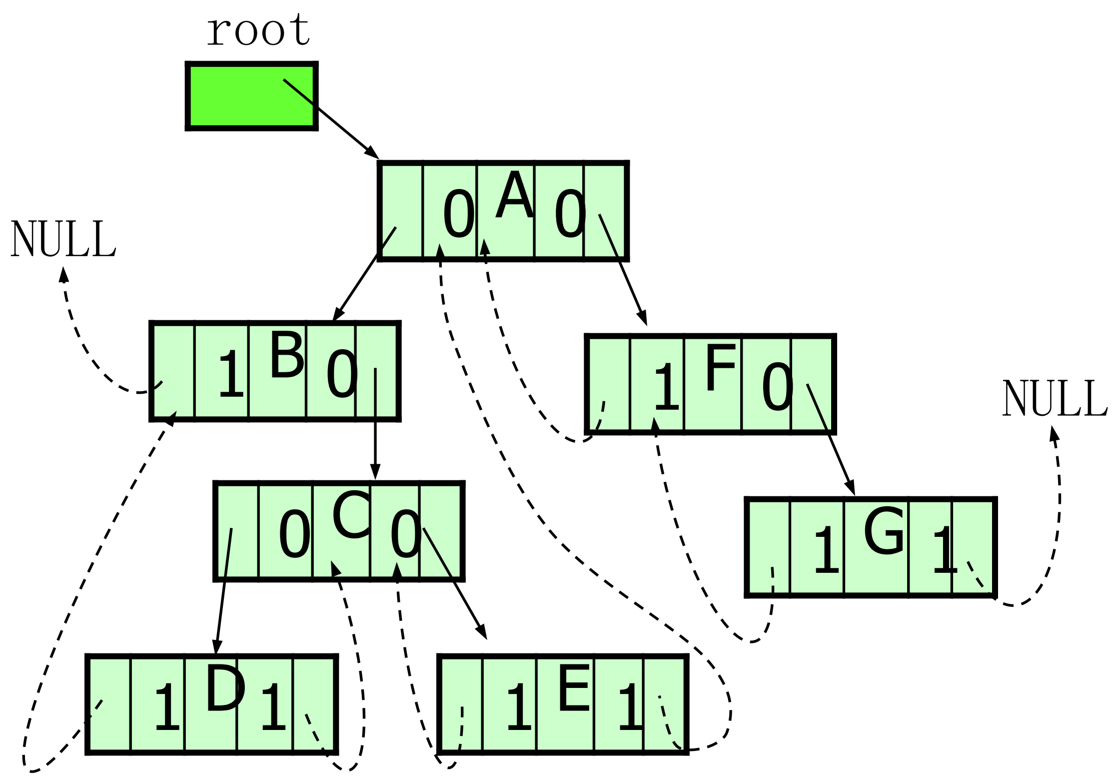
T的中序序列：B, D, C, E, A, F, G



二叉树T



T的中序线索二叉树



中序线索二叉树链表

```
void creat_thread(struct BiTNode *t)
{ struct BiTNode *st[maxleng+1];  //指针栈
  int top=0;                      //置空栈
  struct BiTNode *pre=NULL;       //前驱结点指针
  do
  { while(t)           //根指针t表示的为非空二叉树
    { if (top==maxleng)
            exit(OVERFLOW);       //栈已满,退出
      st[top++]=t;                //根指针进栈
      t=t->lchild;                //t移向左子树
    }
```

```
if (top)                                                    //为非空栈
    { t=st[--top];                                          //弹出根指针
      printf("%c",t->data);                                 //访问根结点
      if (t->lchild!=NULL)  t->ltag=0;                      //左指针为孩子
      else    {
                t->ltag=1;t->lchild=pre; }                  //左指针为线索
      if (pre!=NULL)
          if (pre->rchild!=NULL)
                pre->rtag=0;                                //右指针为孩子
          else {
                pre->rtag=1;                                //右指针为线索
                pre->rchild=t;
                }
      pre=t;                                                //pre与t保持前后
      t=t->rchild;                                          //遍历右子树
    }
  } while(top||t);
 pre->rtag=1;                                               //最后一节点右标记线索
}
```

6. 线索二叉树的遍历

(1). 先序线索二叉树的遍历：

```
void Preorder(struct BiTNode *t)      //t为根指针
{ p=t;
  while (p)
   {  printf("%6c", p->data);
      if (p->ltag==0) //有左孩子时， p移向左孩子结点
          p=p->lchild;
      else            //p移向右孩子或右线索指向的结点
          p=p->rchild;
   }
}
```

(2). 中序线索二叉树的遍历:

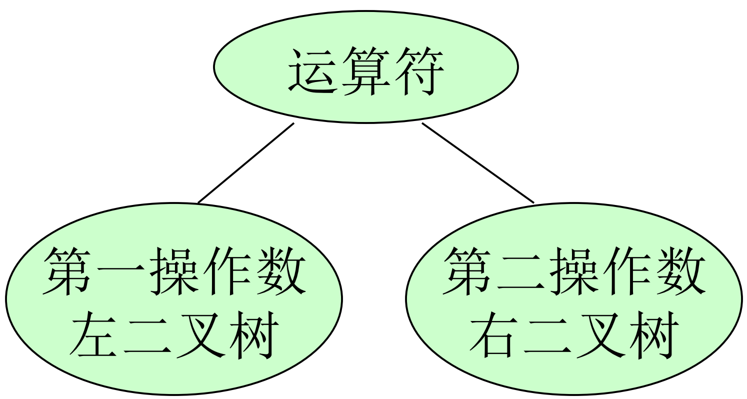
```
void InOrder(struct BiTNode *t)      //t为根指针
{ p=t;
  if (p!=NULL) while (p->lchild!=NULL)
      p=p->lchild;     //查找二叉树的最左结点(第1个结点)
  printf(“%6c”, p->data);
  while (p->rchild!=NULL)                // p有后继结点
     {if (p->rtag==1)  p=p->rchild; //p无右孩子时为线索
      else{p=p->rchild;  //p有右孩子时，取右子树最左结点
           while (p->ltag!=1) p=p->lchild;}
      printf(“%6c”, p->data);
      }
}
```

6.3.4 表达式二叉树

表达式二叉树T =（第一操作数）（运算符）（第二操作数）

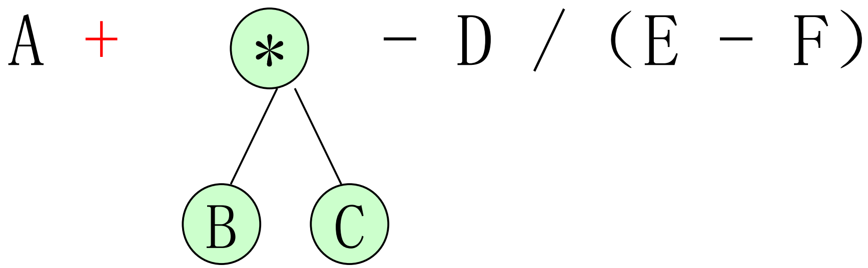
其中：第一操作数、第二操作数也是表达式二叉树，

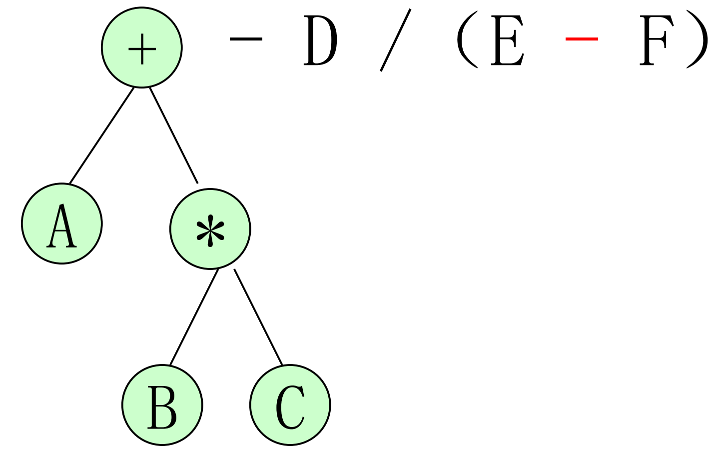
分别为表达式二叉树T的左子树和左子树

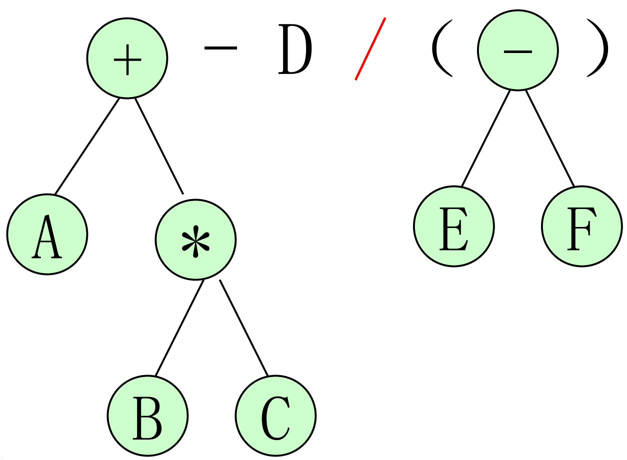


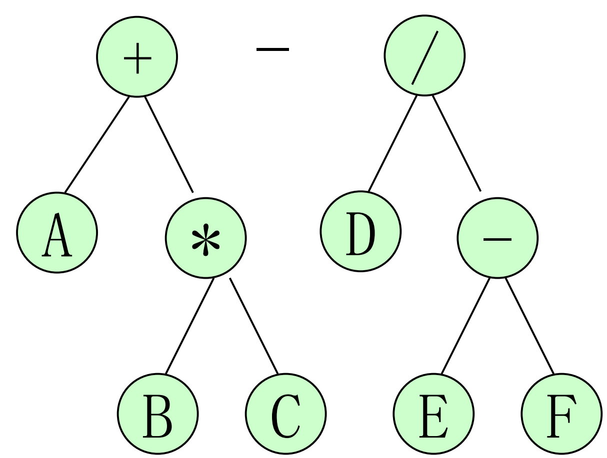
表达式二叉树

例 表达式： A + B * C - D / (E - F)

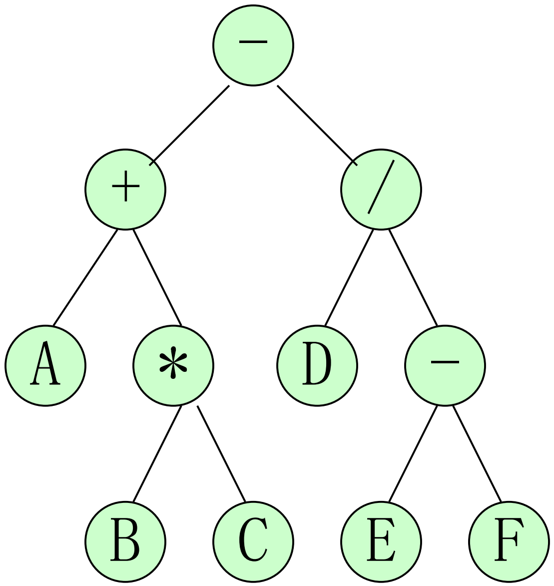








例 表达式： A + B * C - D / (E - F)



表达式二叉树

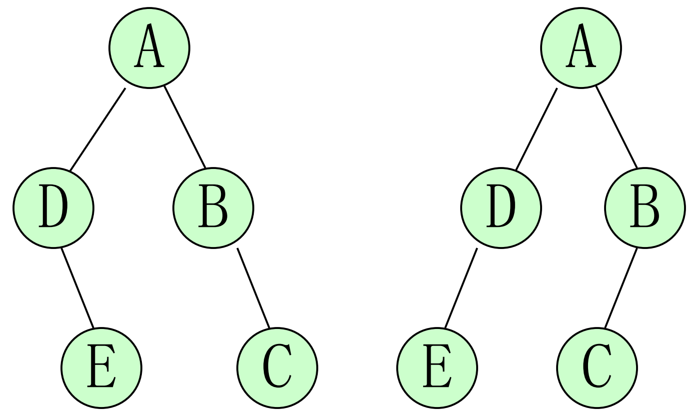
- 前序遍历：- + A * B C / D - E F
  前缀表示, 前缀表达式, 波兰式
- 中序遍历: A + B * C - D / E - F
  中缀表示, 中缀表达式
- 后序遍历: A B C * + D E F - / -
  后缀表示, 后缀表达式， 逆波兰式

6.3.5 中序遍历序列和前(或后)序序列确定唯一棵二叉树

由前序序列：A D E B C

和(或)后序序列：E D C B A

不能确定唯一棵二叉树



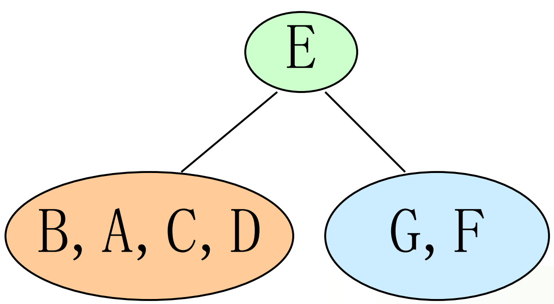
T1　　T2

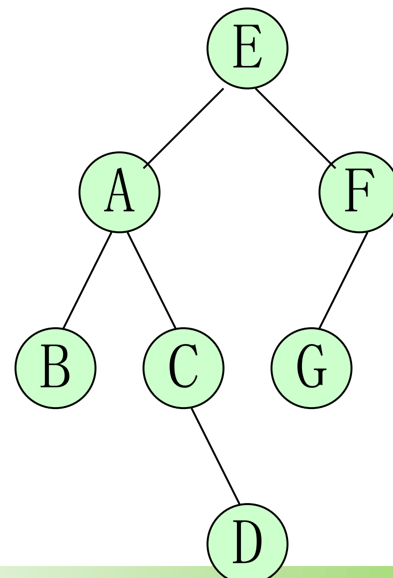
例1. 给定二叉树T的

中序序列：B A C D E G F

前序序列：E A B C D F G

如何求二叉树T?



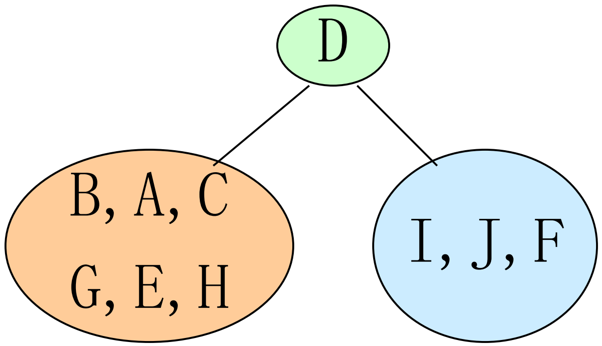


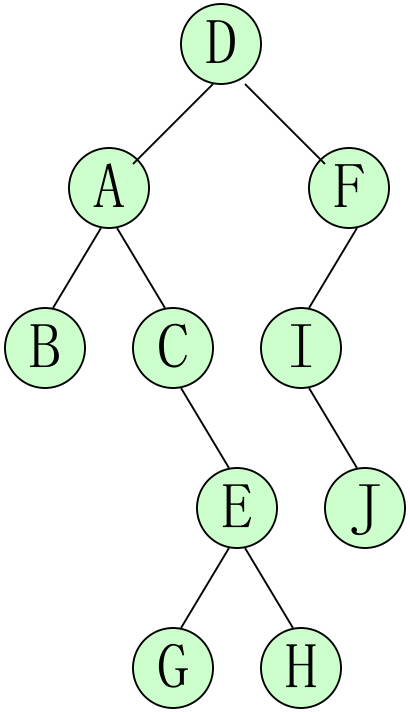
例2. 给定二叉树T的

后序序列：B G H E C A J I F D

中序序列：B A C G E H D I J F

如何求二叉树T？





二叉树T

6.3.6 遍历二叉树的应用

例. 求二叉树中结点的个数

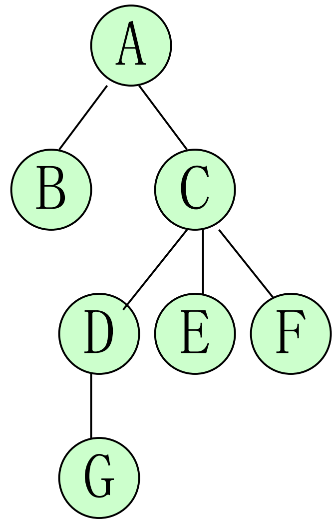
```
int preorder(struct BiTNode *root) //求二叉树中结点的个数
 { int n=0 ;
   if (root)
   { n=1;                            //根结点计数
    n+=preorder(root->lchild);       //递归计算左子树
    n+=preorder(root->rchild);       //递归计算右子树
    }
  return n; }
main()
{ int n;                  //n为计数器,假定二叉树已生成
  n=preorder(root); //root为指针,执行preorder(root)
  printf("%d",n);     //输出n
 }
```

6.4 树和森林

6.4.1 树的存储结构

1. 双亲表示法/数组表示法/顺序表示法

```
struct snode
{ char data;
  int parent;
}t[maxleng+1];
```



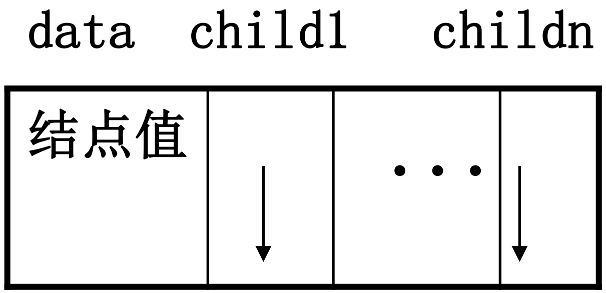
树

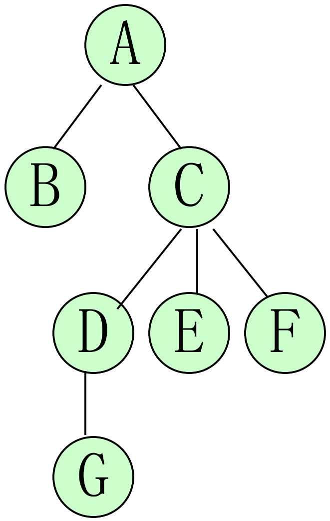
	data	parent
1	A	0
2	B	1
3	C	1
4	D	3
5	E	3
6	F	3
7	G	4

t[1..7]

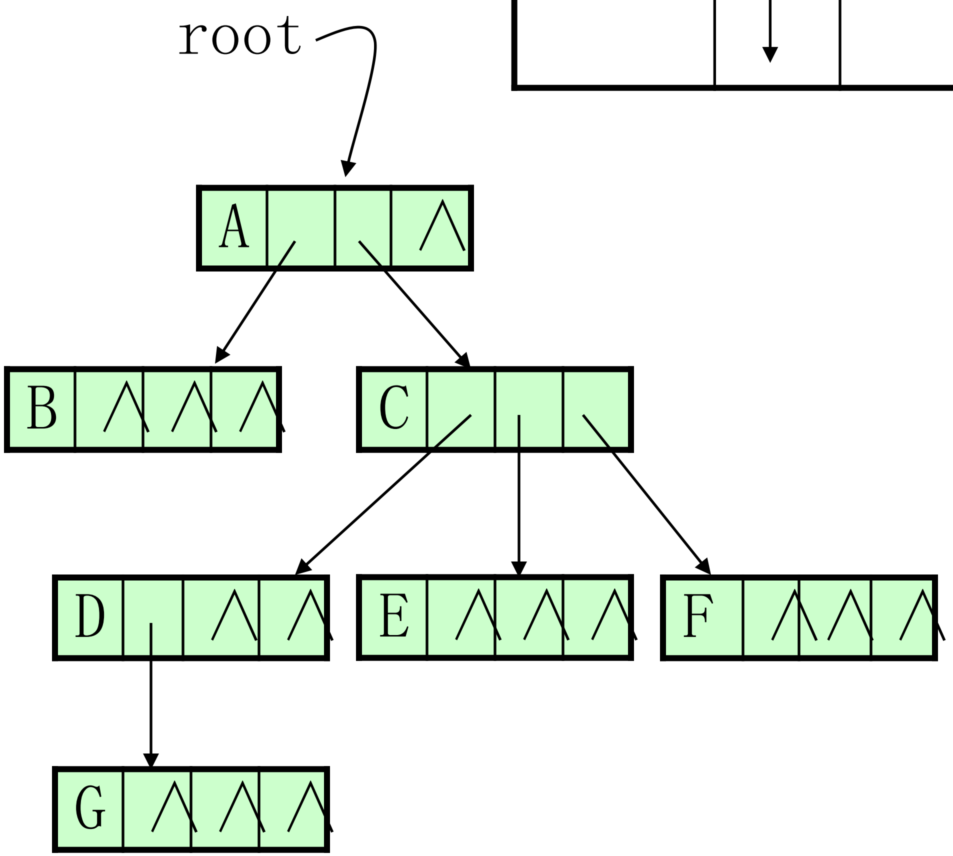
2. 孩子表示法/链接表表示法

(1)固定大小的结点格式，设树T的度为n





树T

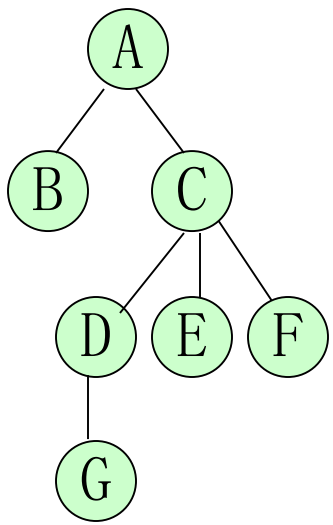


T的链接表

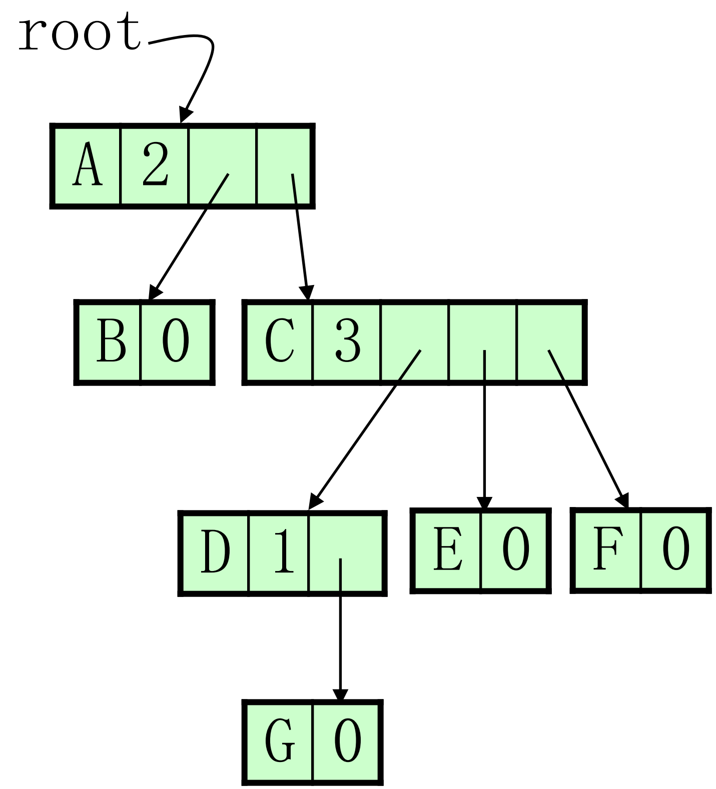
2.孩子表示法/链接表表示法

(2)非固定大小的结点格式

data	degree	child1		childn
结点值	结点的度n		...	

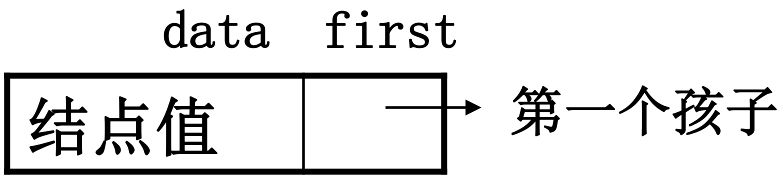


树

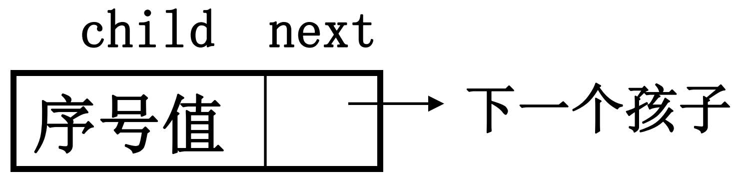


链接表

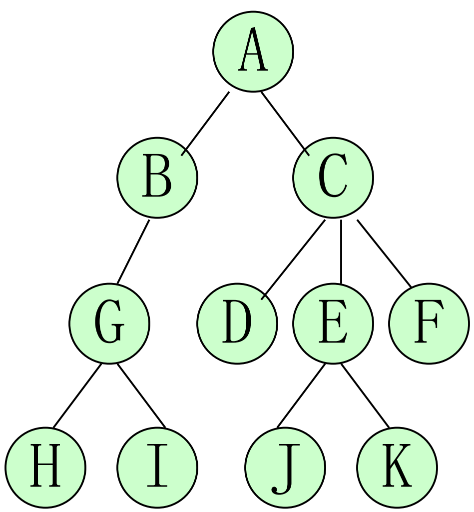
3.孩子链表表示法/单链表表示法



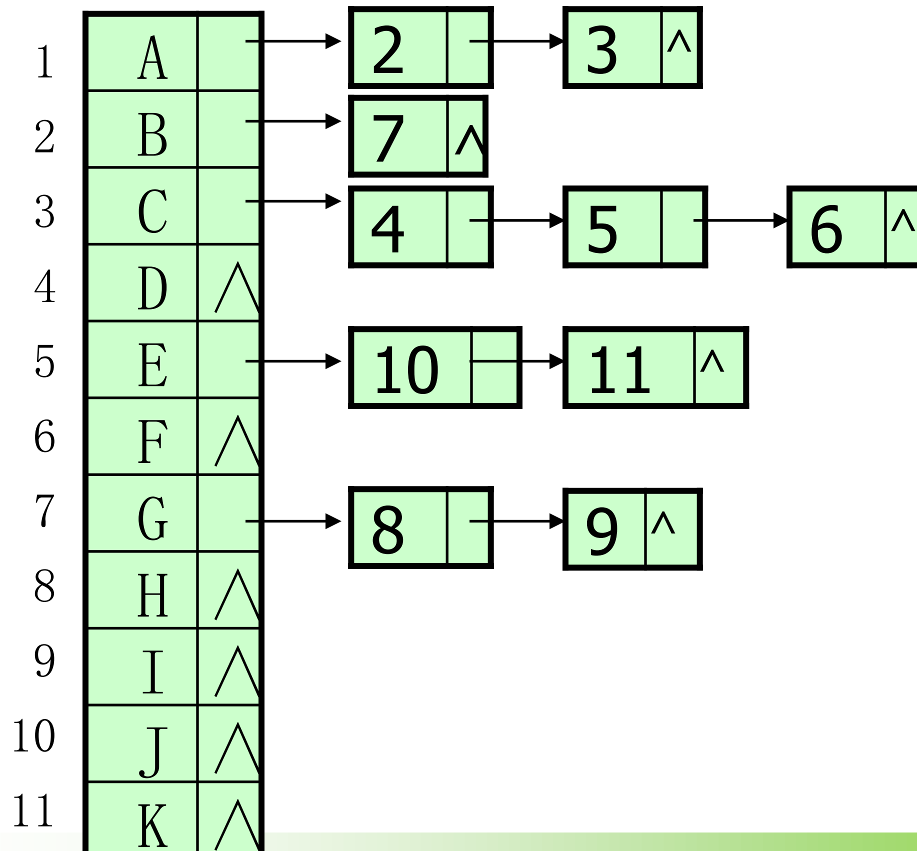
表头结点



孩子结点/表结点

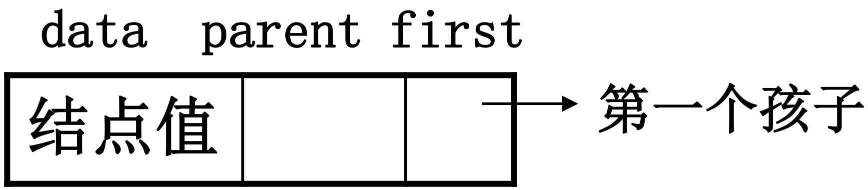


树

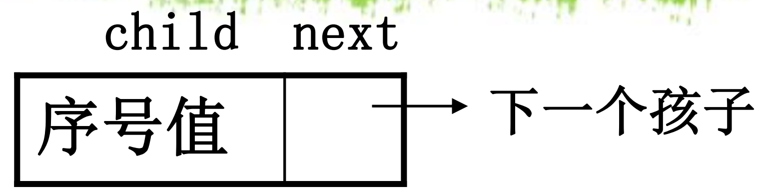


表头结点数组

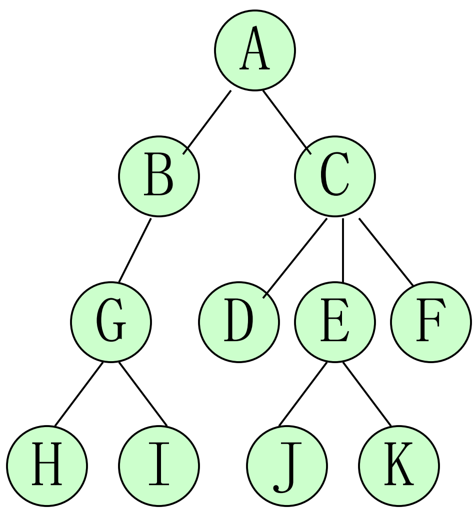
4.带双亲的孩子链表表示法



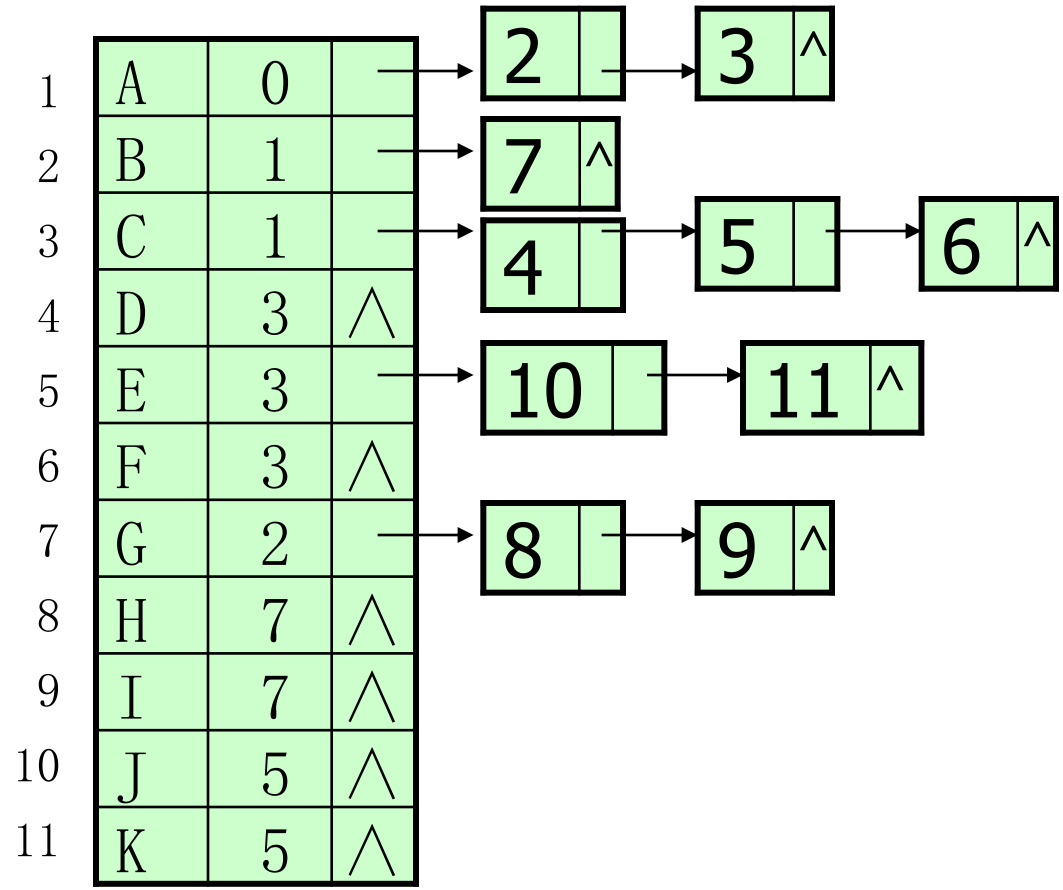
表头结点



孩子结点/表结点



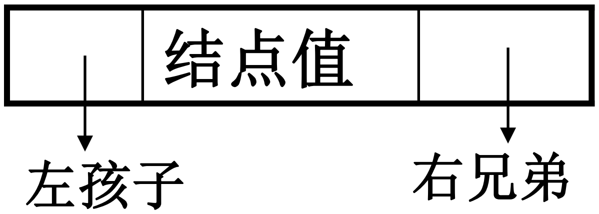
树

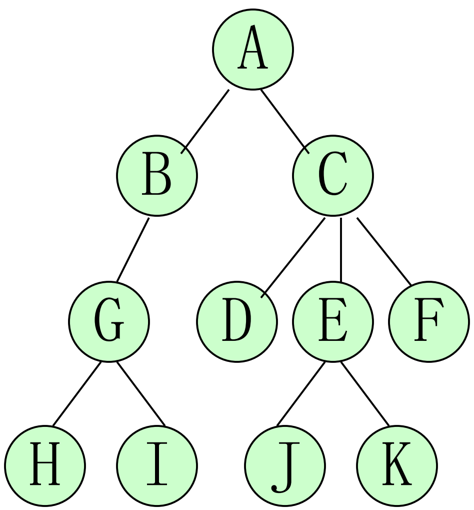


表头结点数组

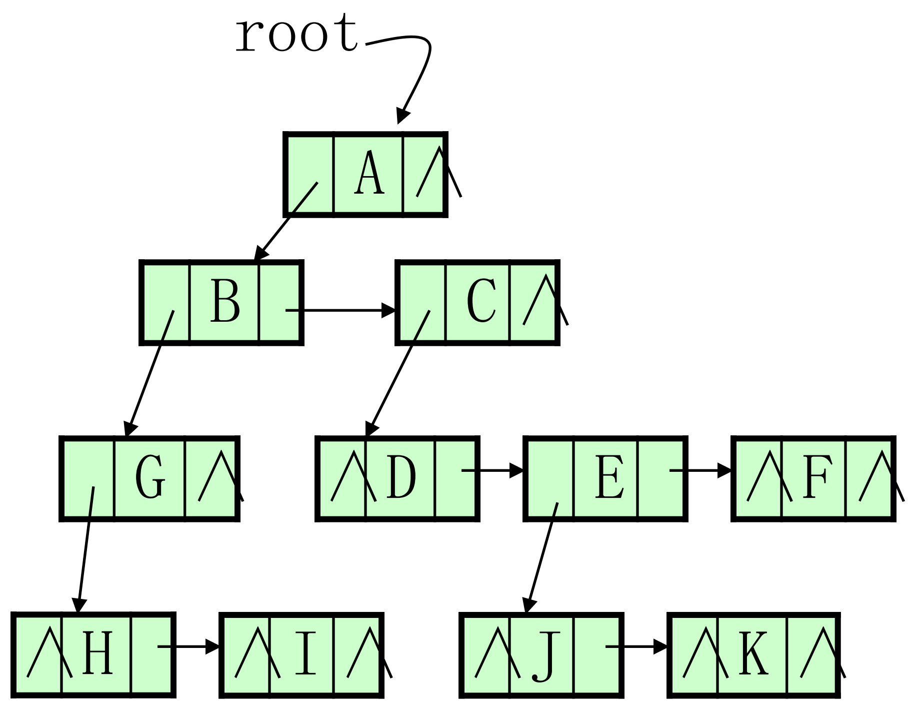
5.孩子兄弟表示法/二叉树表示法/二叉链表

firstchild data nextbrother





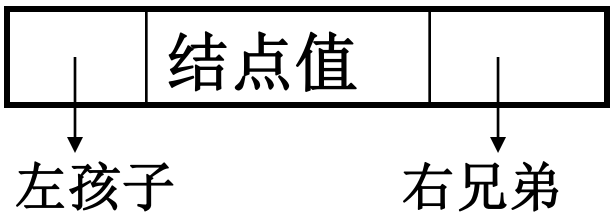
树

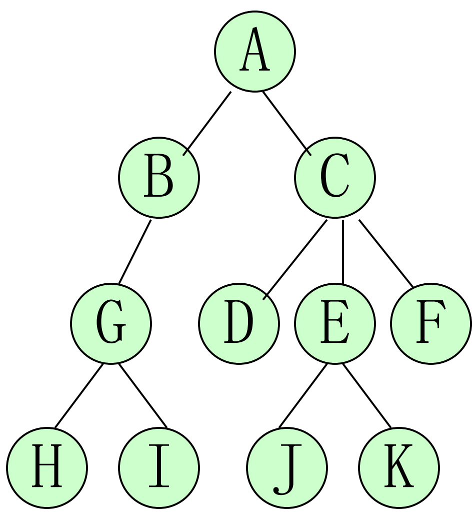


二叉链表

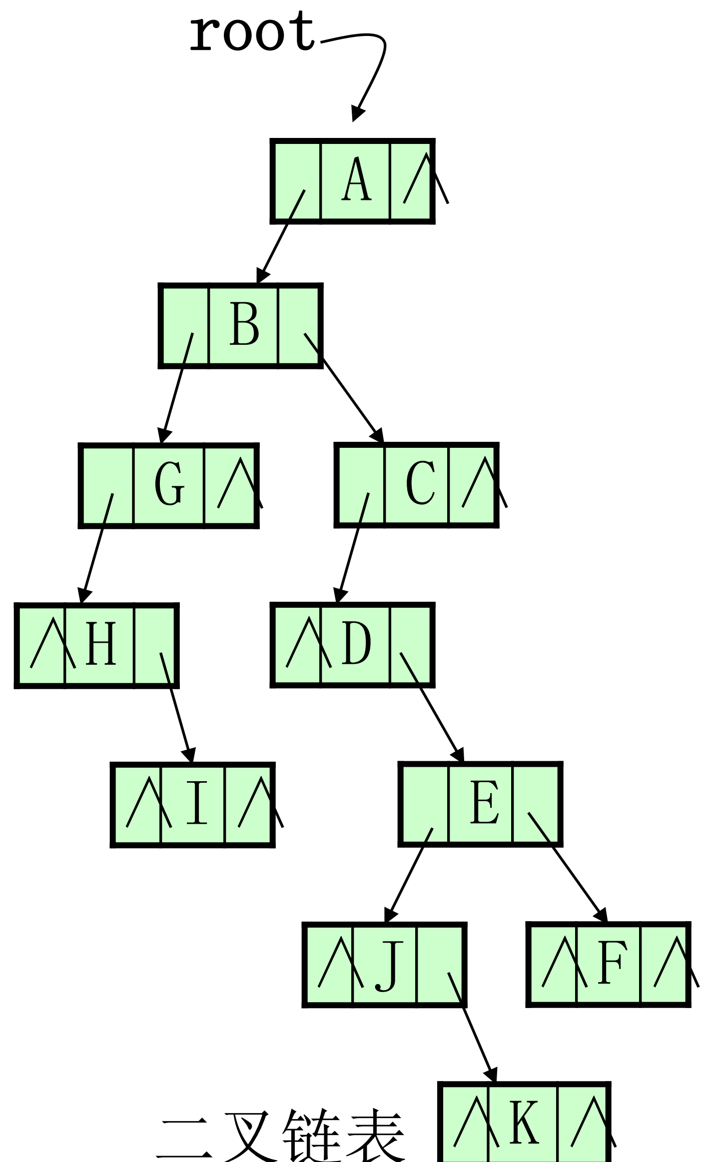
5. 孩子兄弟表示法/二叉树表示法/二叉链表

firstchild data nextbrother





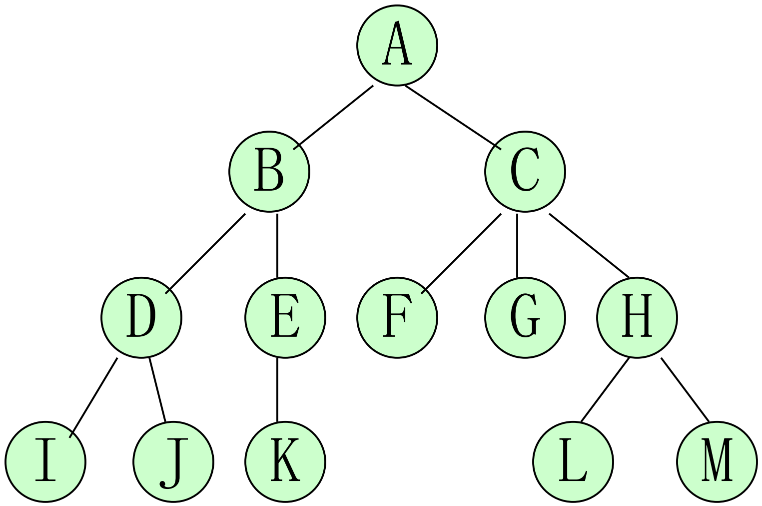
树



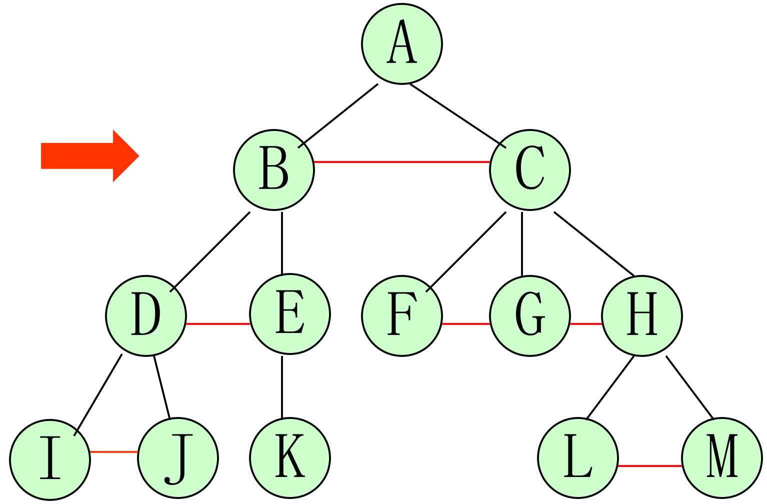
二叉链表

6.4.2 树与二叉树的转换

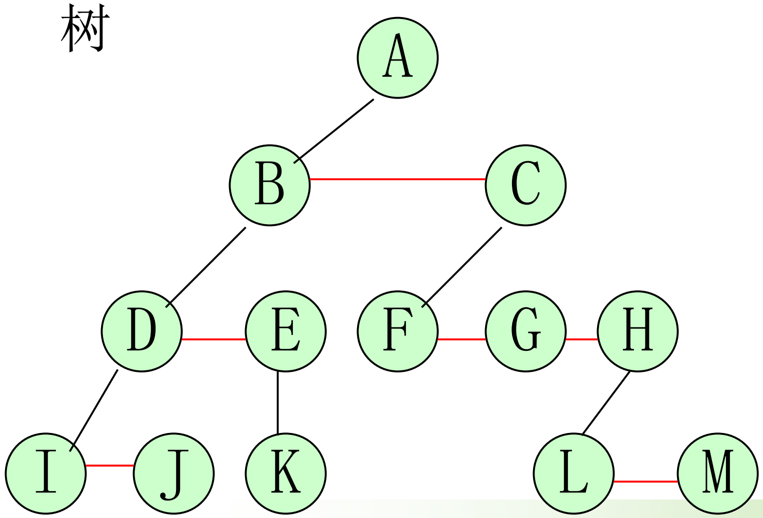
1.树 ➡ 二叉树



树

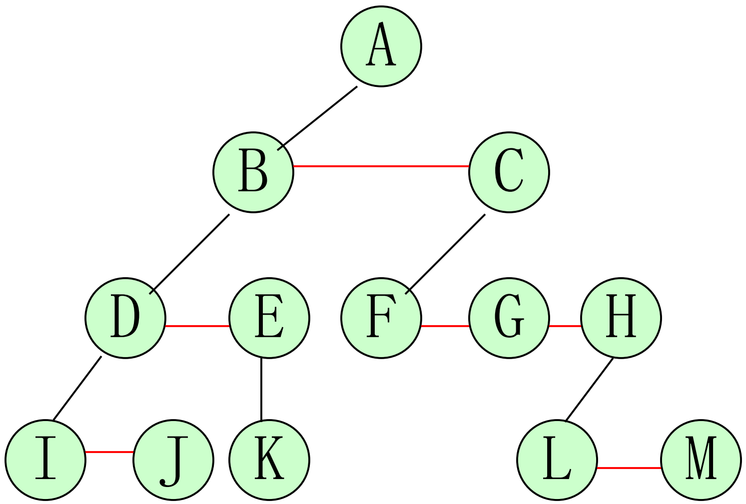


1. 在兄弟之间加连线

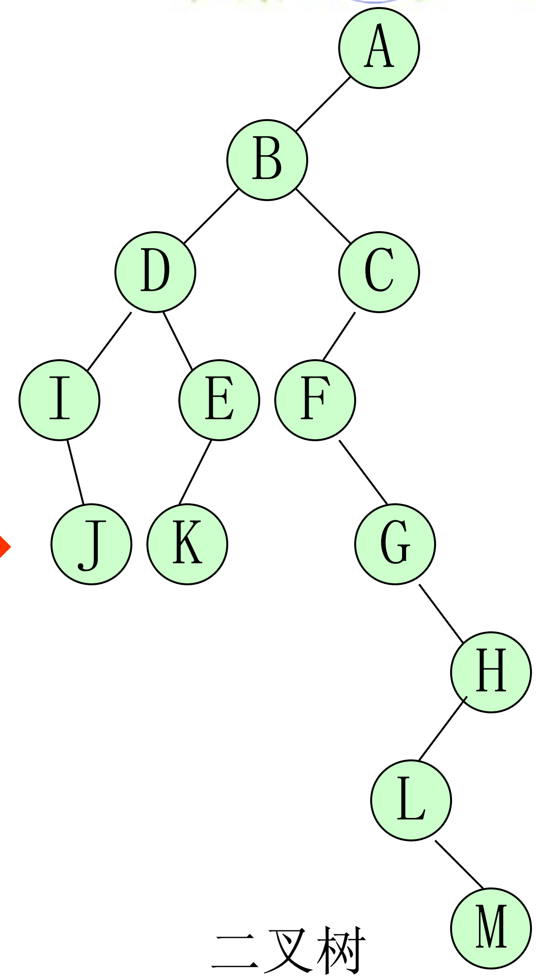


2. 保留根与最左孩之间的连线
删除与其它孩子之间的连线

1. 树 ➡ 二叉树



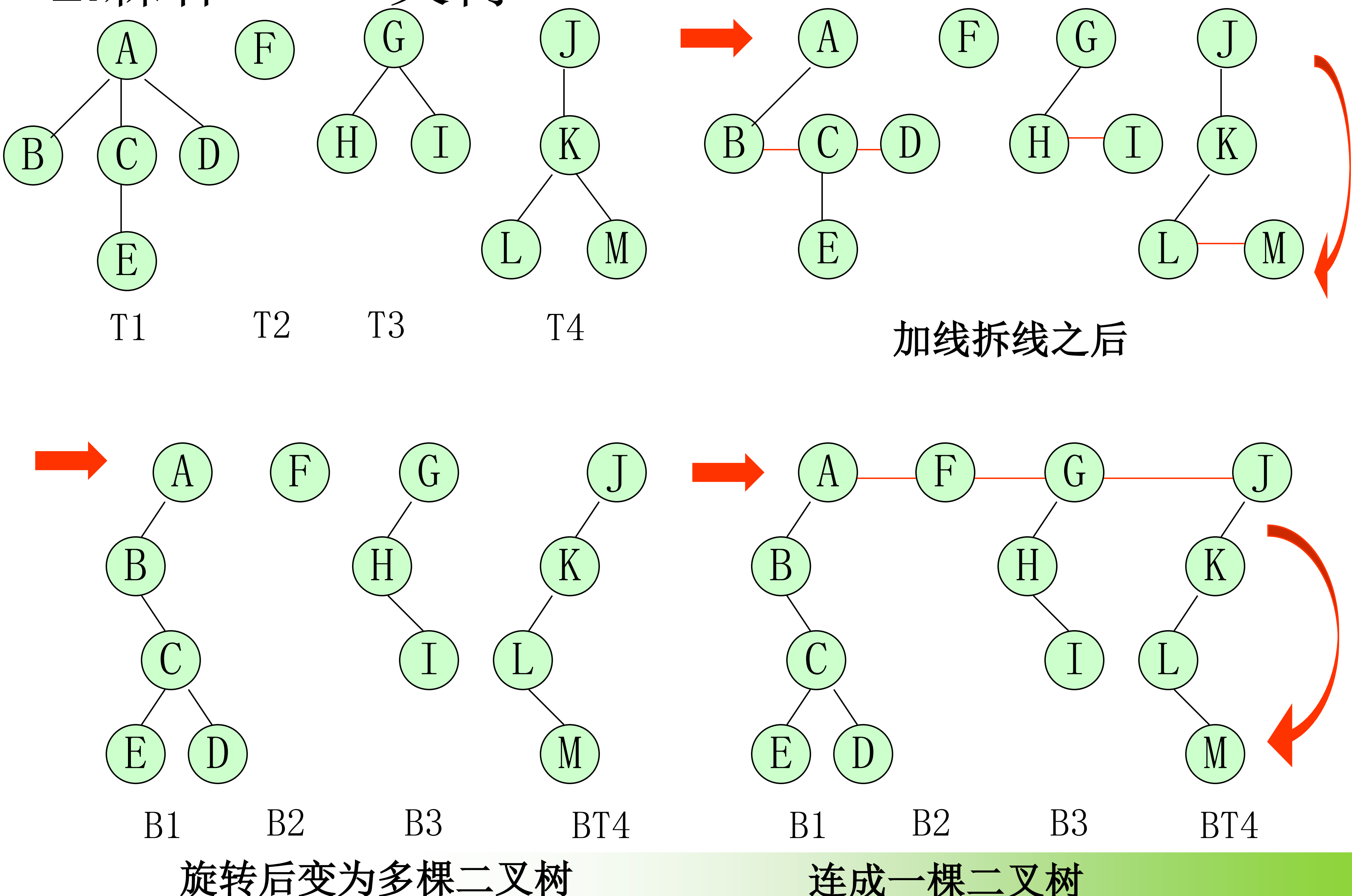
2. 保留根与最左孩之间的连线
   删除与其它孩子之间的连线



二叉树

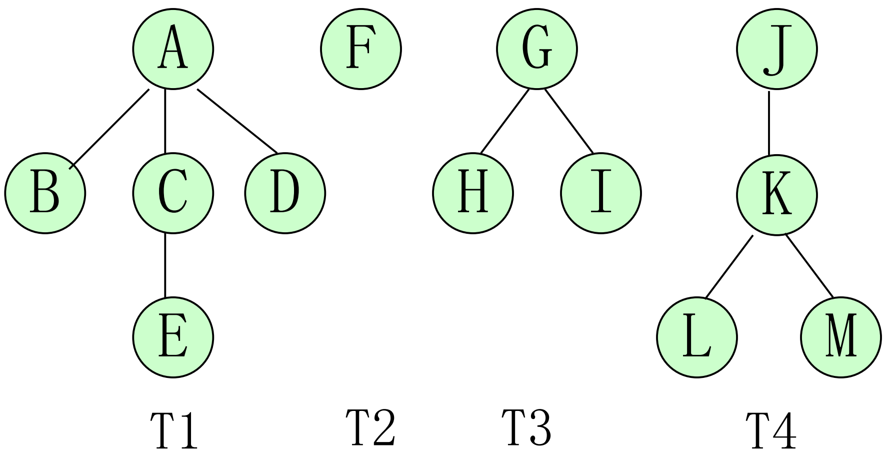
3. 以根为轴心顺时针
   方向旋转45度

2.森林→二叉树

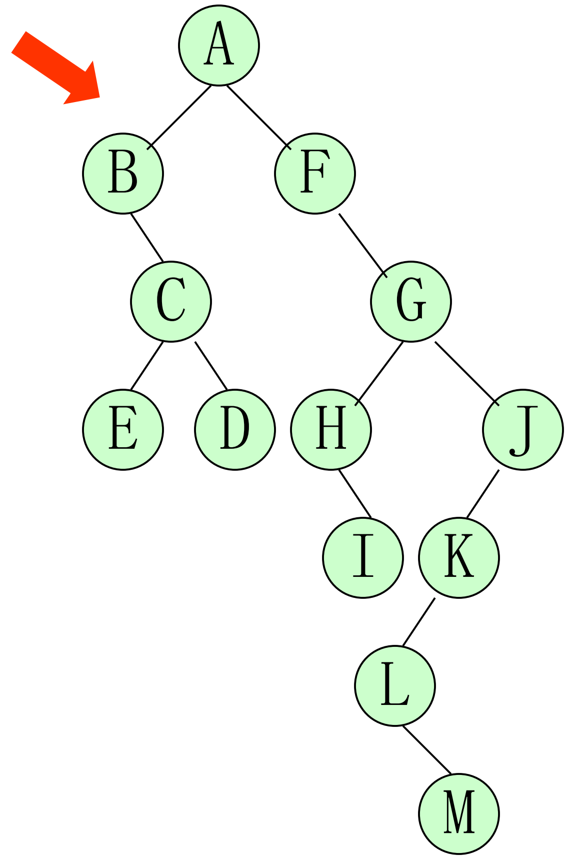


A
F
G
J
B
C
D
H
I
K
E
L
M
T1
T2
T3
T4
加线拆线之后
B1
B2
B3
BT4
旋转后变为多棵二叉树
连成一棵二叉树

2. 森林 → 二叉树



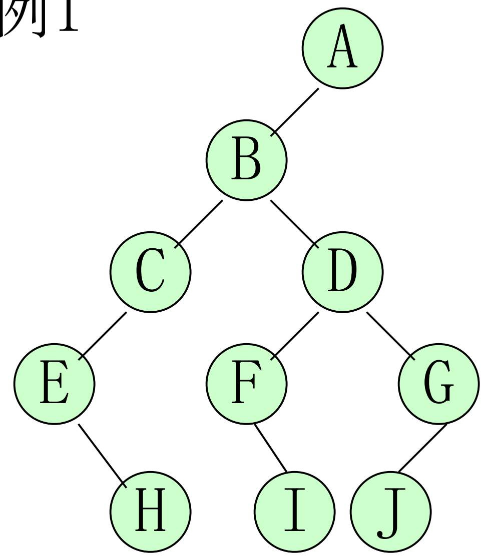
森林F={T1, T2, T3, T4}



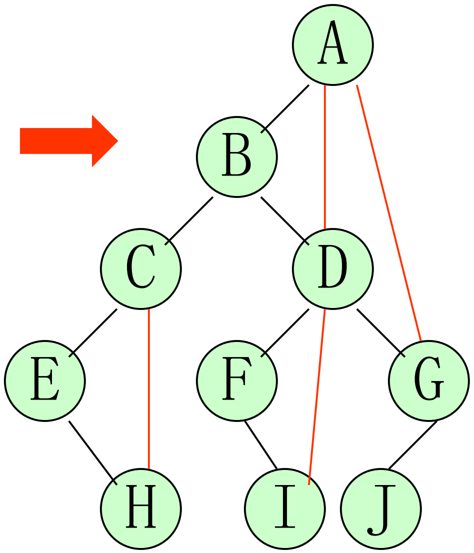
旋转后，变为一棵二叉树

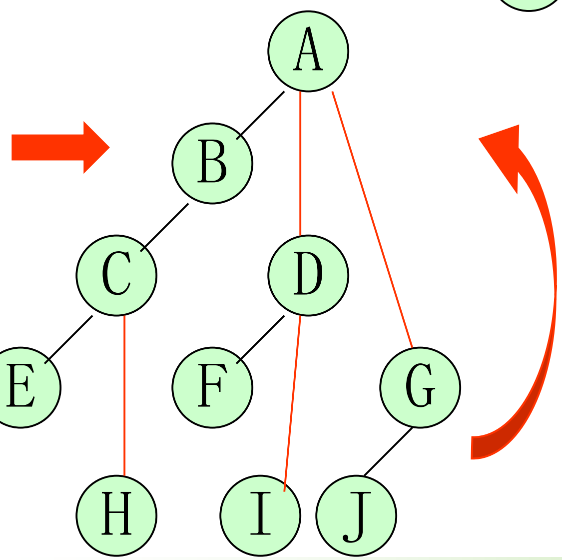
3. 二叉树 → 树

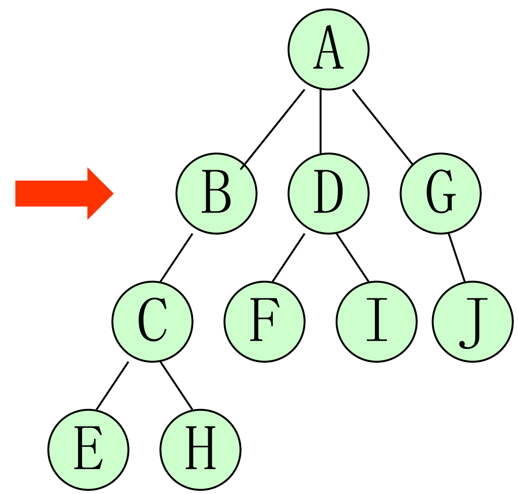
例1



二叉树B



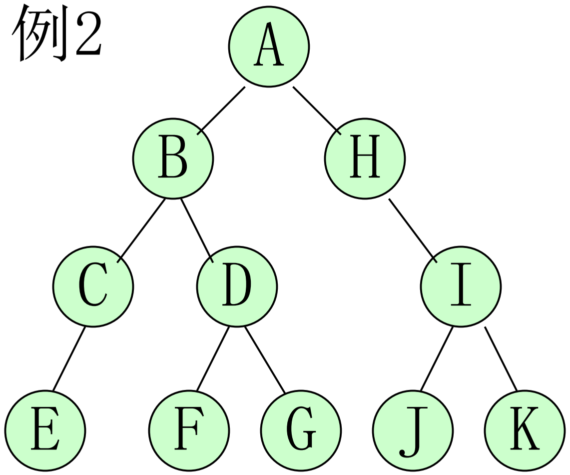




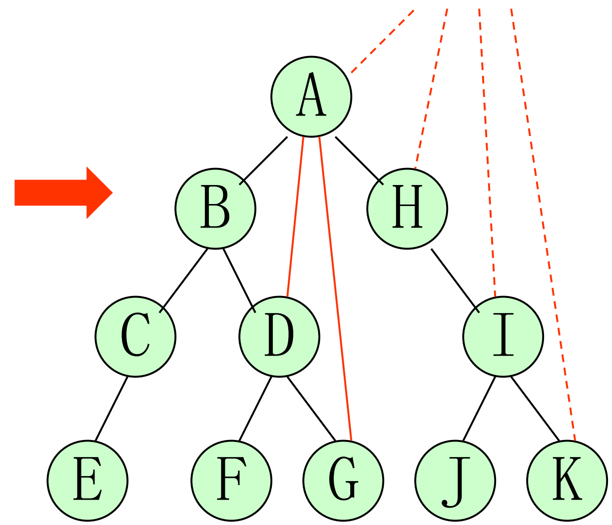
树T

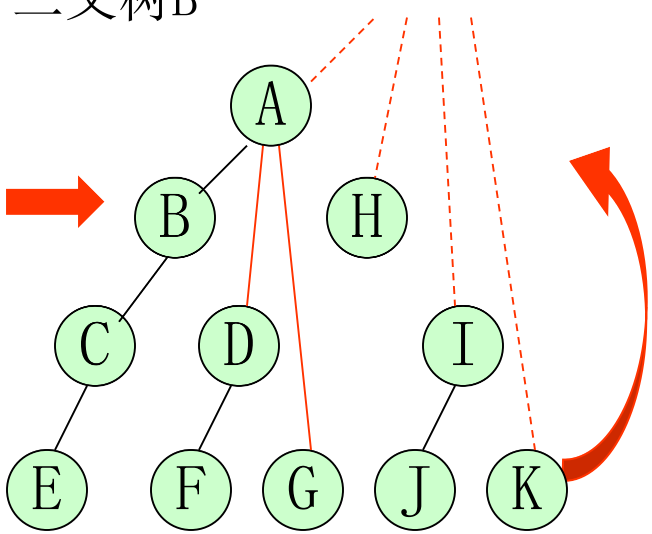
3. 二叉树 → 森林

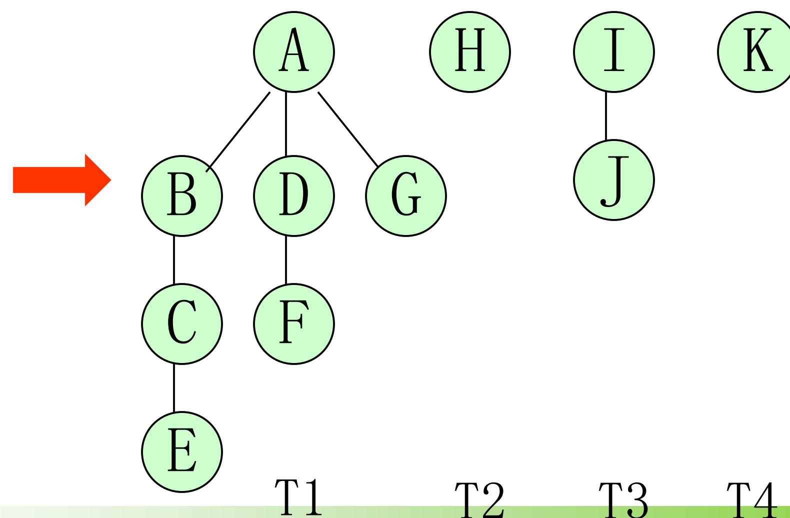
例2



二叉树B

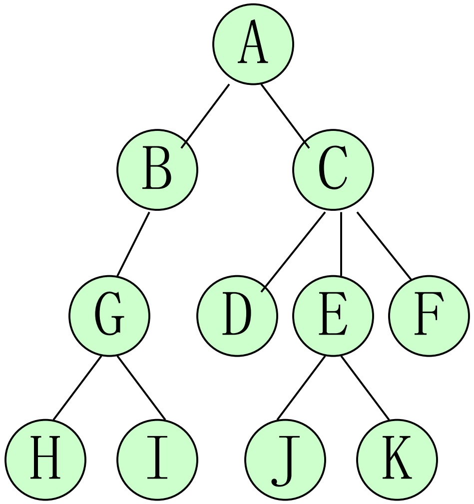






6.4.3 树和森林的遍历

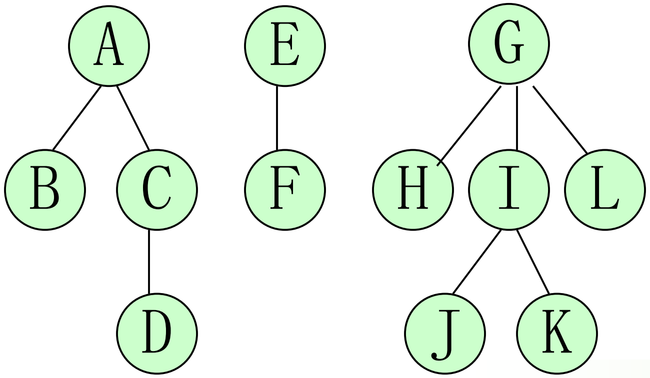
1. 树的遍历



前根遍历：A B G H I C D E J K F

后根遍历：H I G B D J K E F C A

2. 森林的遍历



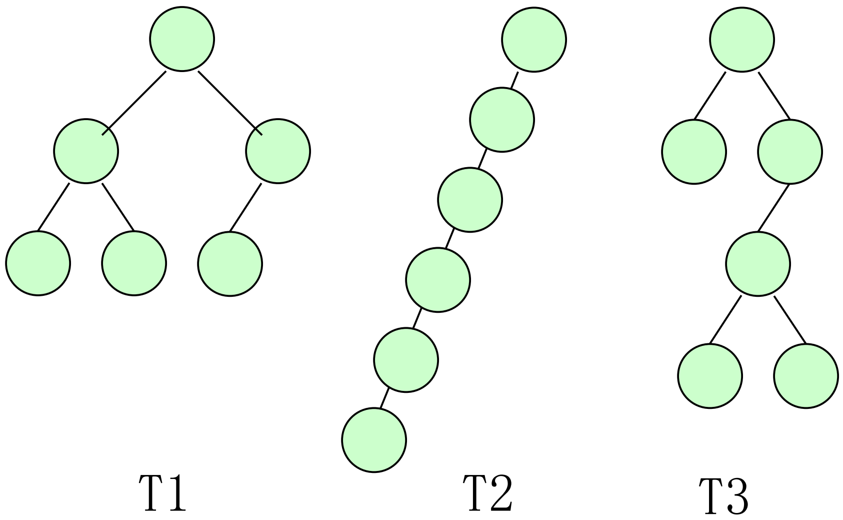
前序遍历：A B C D E F G H I J K L

中序遍历：B D C A F E H J K I L G

（依次对每一棵树后序遍历）

6.6 哈夫曼（Huffman)树及其应用

1. 路径长度----路径上分枝的数目(连线的数目)
2. 树T的路径长度----从树T的根到其余每个结点的路径长度之和，记作PL(T)



PL(T1)=1+1+2+2+2=8

PL(T2)=1+2+3+4+5=15

PL(T3)=1+1+2+3+3=10

- 当n个结点的二叉树为完全二叉树时，PL(T)具有最小值

$\because$ 结点i的层 $= \lfloor \log_2 i \rfloor + 1$

树T的根到结点i的路径长度 = 结点i的层-1

$= \lfloor \log_2 i \rfloor$

$\therefore$ $PL(T) = \lfloor \log_2 1 \rfloor + \lfloor \log_2 2 \rfloor + \ldots + \lfloor \log_2 n \rfloor$

$= \sum_{i=1}^{n} \lfloor \log_2 i \rfloor$

- 当n个结点的二叉树为单枝树时，PL(T)具有最大值：

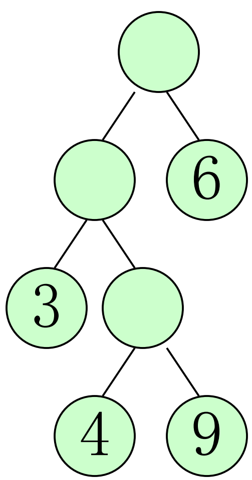
  PL(T) = 1+2+...+(n-1) = n(n-1)/2

3. 树T的带权路径长度----每个叶子的权与根到该叶子的路径长度的乘积之和，记作WPL(T)

$$\mathrm{WPL}(T) = \sum_{k=1}^{n} w_k l_k$$

其中：n --- 树T的叶子数目　　w_k--- 叶子k的权

l_k--- 树T的根到叶子k的路径长度

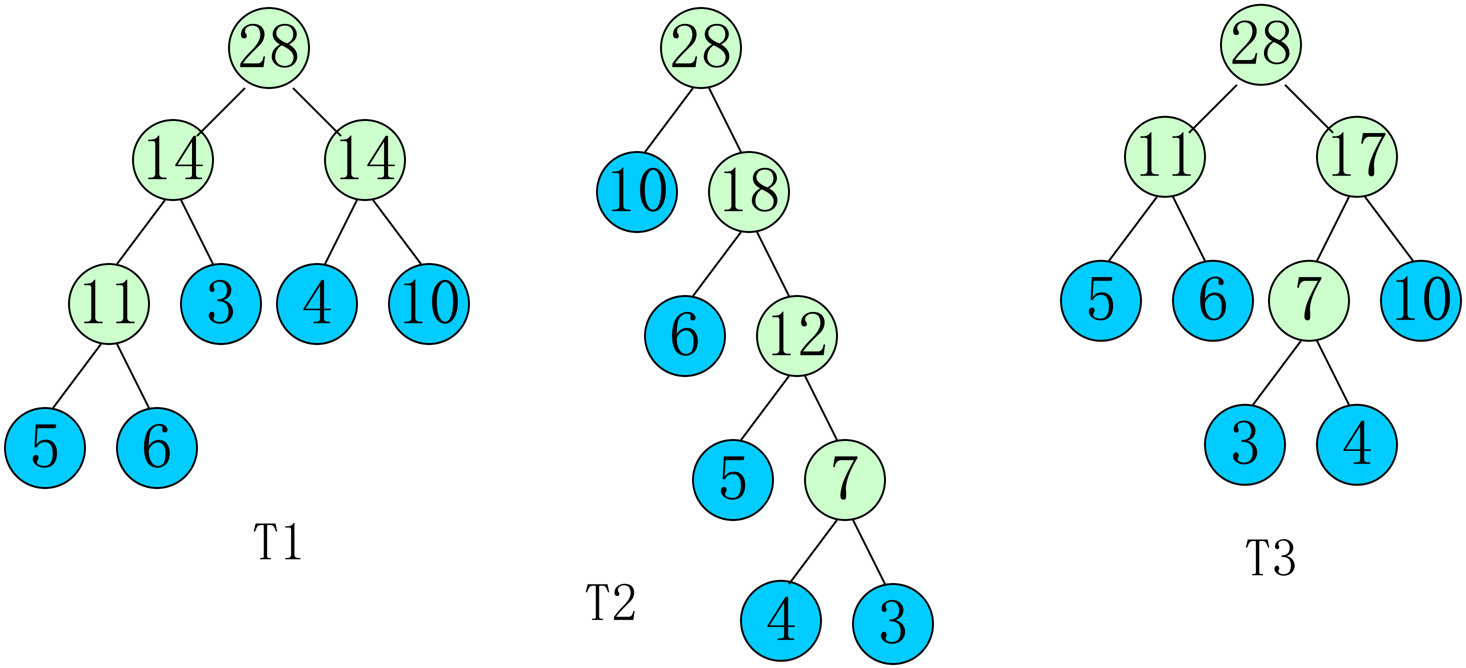


T1

WPL(T)=6*1+3*2+4*3+9*3=51

4. 哈夫曼树/最佳树/最优树----

在具有n个相同叶子的各二叉树中，WPL(T)最小的二叉树。



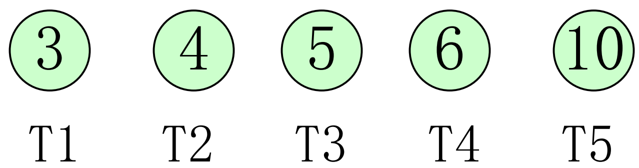
- WPL(T1)=5*3+6*3+3*2+4*2+10*2=67
- WPL(T2)=10*1+6*2+5*3+4*4+3*4=65
- WPL(T3)=5*2+6*2+3*3+4*3+10*2=63

5. 哈夫曼算法

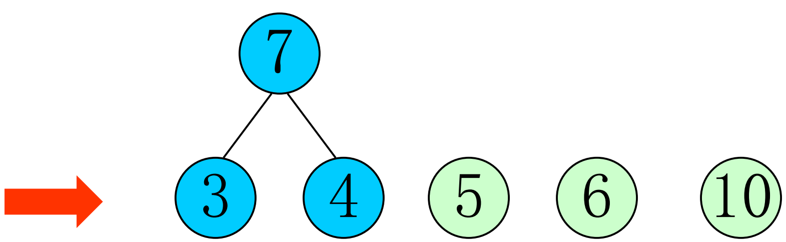
例 给定权集合{4, 5, 3, 6, 10}，构造哈夫曼树

1. 按权值大小排序： 3, 4, 5, 6, 10

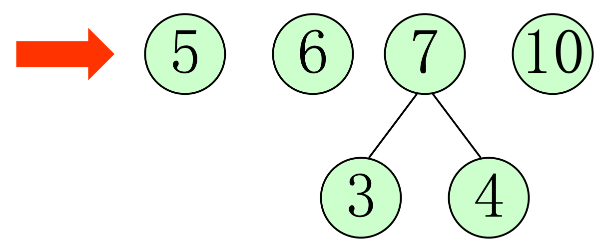
2. 生成森林：



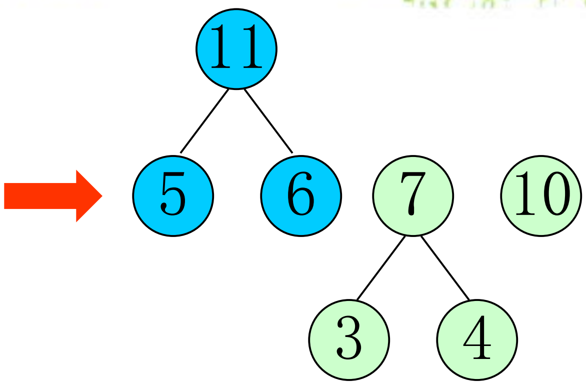
3. 合并两棵权最小的二叉树, 并排序, 直到成为一棵二叉树：



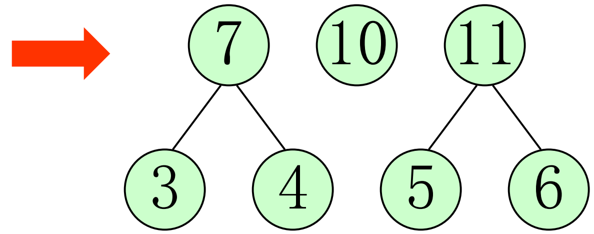
选择合并



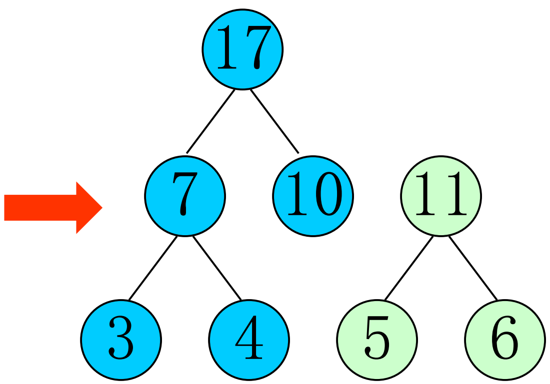
排序



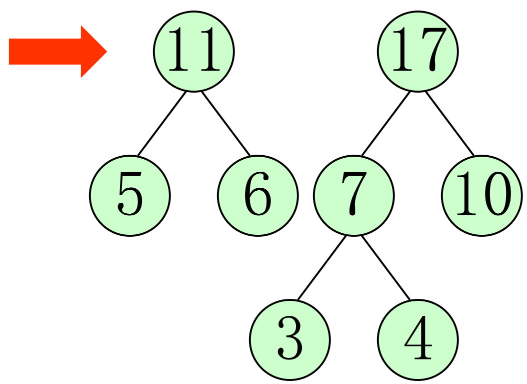
选择合并



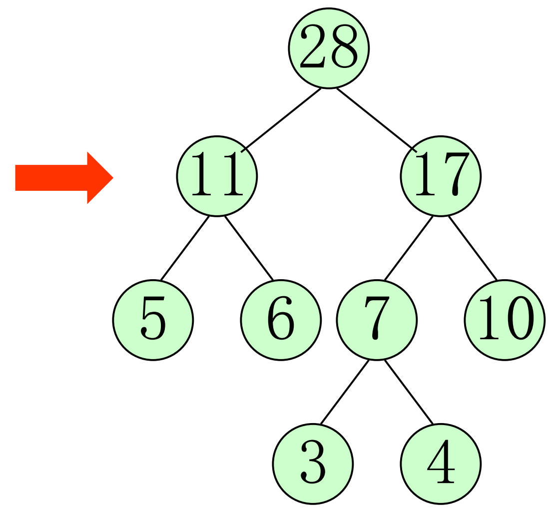
排序



选择合并



排序



哈夫曼树

6. 最小冗余码/哈夫曼码

- ASCII码/定长码

  ab12: 01100001 01100010 00110001 00110010

  97 98 49 50

- 哈夫曼码/不定长码

  能按字符的使用频度，使文本代码的总长度具有最小值。

例. 给定有18个字符组成的文本：

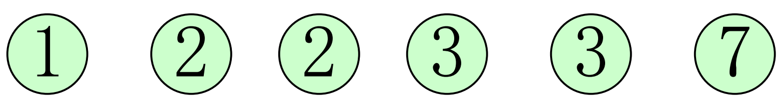
A A D A T A R A E F R T A A F T E R

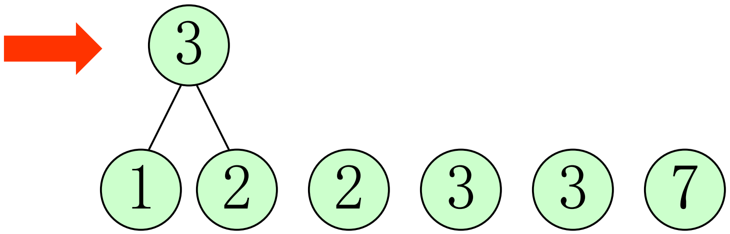
求各字符的哈夫曼码。

(1) 统计：

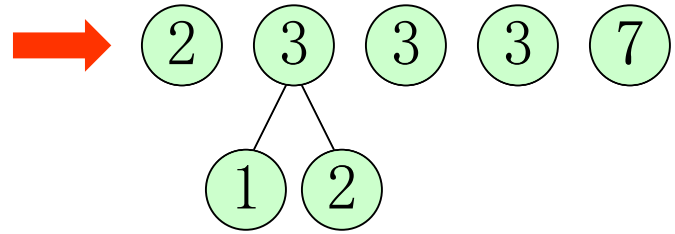
字符	A	D	E	F	T	R
频度	7	1	2	2	3	3

(2) 构造Huffman树：

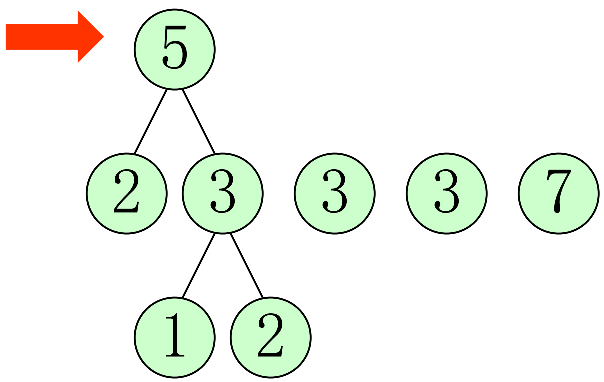




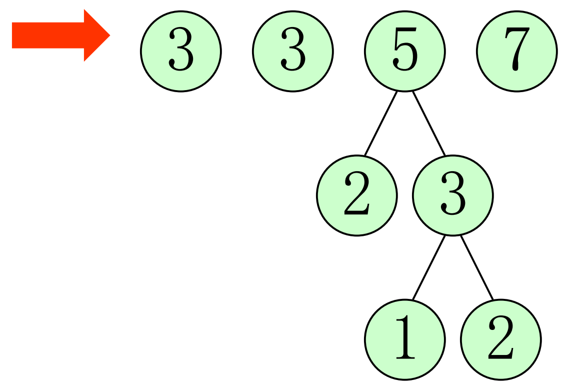
合并1和2



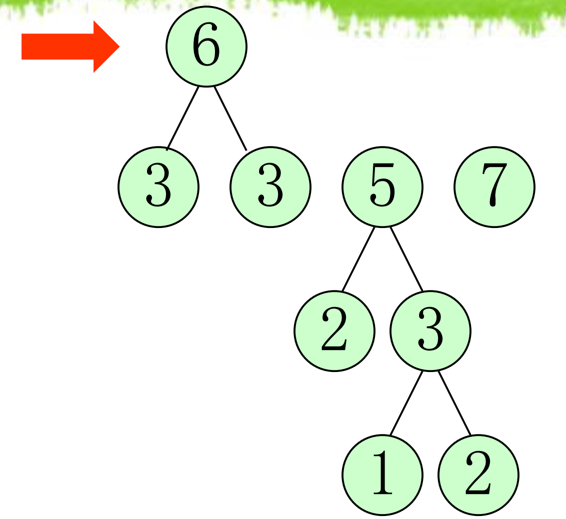
排序



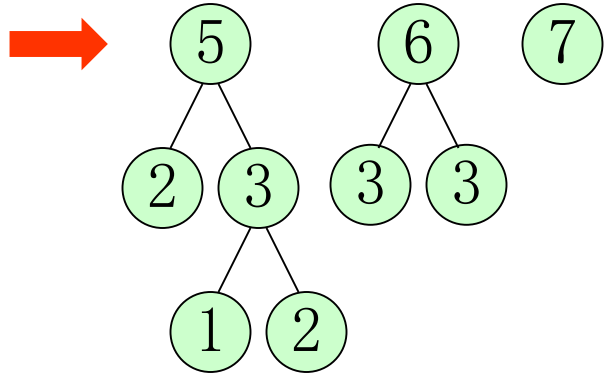
合并2和3



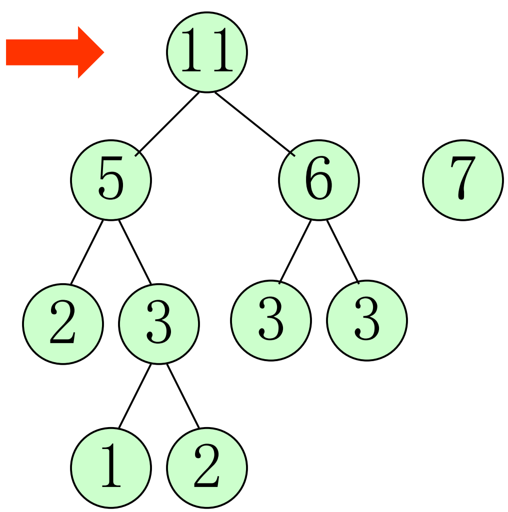
排序



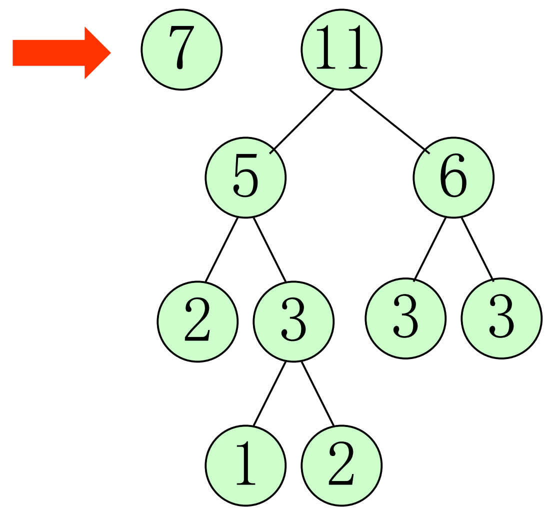
合并3和3



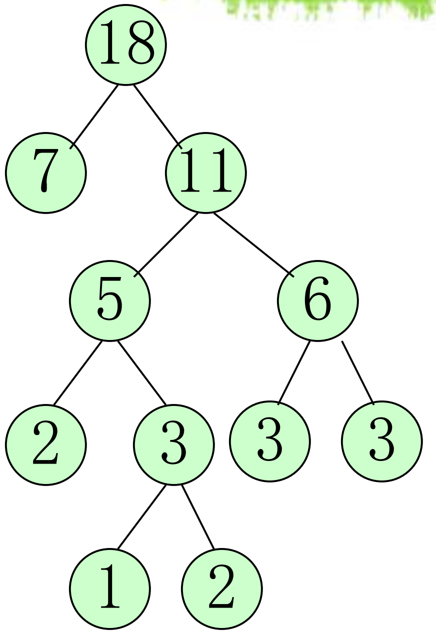
排序



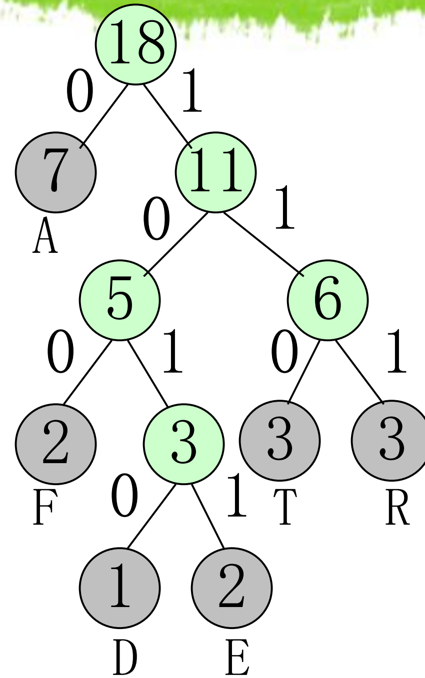
合并5和6



排序



合并7和11得Huffman树



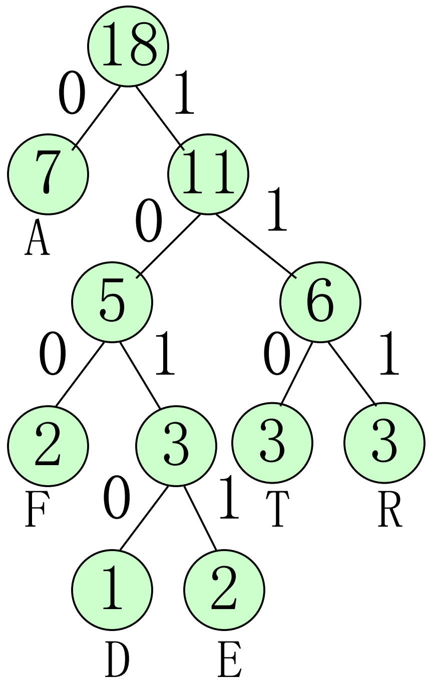
叶结点根据权值附上字符，分支标数的Huffman树

(4) 确定Huffman编码：

字 符	A	D	E	F	T	R
频 度	7	1	2	2	3	3
编 码	0	1010	1011	100	110	111

特点：任一编码不是其它编码的前缀

如何译码？



Huffman树

例. 给定代码序列：

0 0 1 0 0 0 1 1 1 0 1 0 1 0 1 0 1 1 1 10

文本为：A A F A R A D E T

算法处理：

结点说明：

data	parent
lchild	rchild

7	0	5	0	2	0	4	0	3	0								
0	0	0	0	0	0	0	0	0	0								
HT[1]		HT[2]		HT[3]		HT[4]		HT{5]		HT[6]		HT[7]		HT[8]		HT[9]	

7	0	5	0	2	6	4	0	3	6	5	0						
0	0	0	0	0	0	0	0	0	0	3	5						
HT[1]		HT[2]		HT[3]		HT[4]		HT{5]		HT[6]		HT[7]		HT[8]		HT[9]	

7	0	5	7	2	6	4	7	3	6	5	0	9	0				
0	0	0	0	0	0	0	0	0	0	3	5	4	2				
HT[1]		HT[2]		HT[3]		HT[4]		HT{5]		HT[6]		HT[7]		HT[8]		HT[9]	

7	8	5	7	2	6	4	7	3	6	5	8	9	0	12	0		
0	0	0	0	0	0	0	0	0	0	3	5	4	2	6	1		
HT[1]		HT[2]		HT[3]		HT[4]		HT{5]		HT[6]		HT[7]		HT[8]		HT[9]	

7	8	5	7	2	6	4	7	3	6	5	8	9	9	12	9	21	0
0	0	0	0	0	0	0	0	0	0	3	5	4	2	6	1	7	8
HT[1]		HT[2]		HT[3]		HT[4]		HT{5]		HT[6]		HT[7]		HT[8]		HT[9]	

HT[1]的编码反序列为： 11　　则编码为：11

HT[2]的编码反序列为： 10　　则编码为：01

同理得：　HT[3]：100　　　HT[4]：00

HT[5]：101