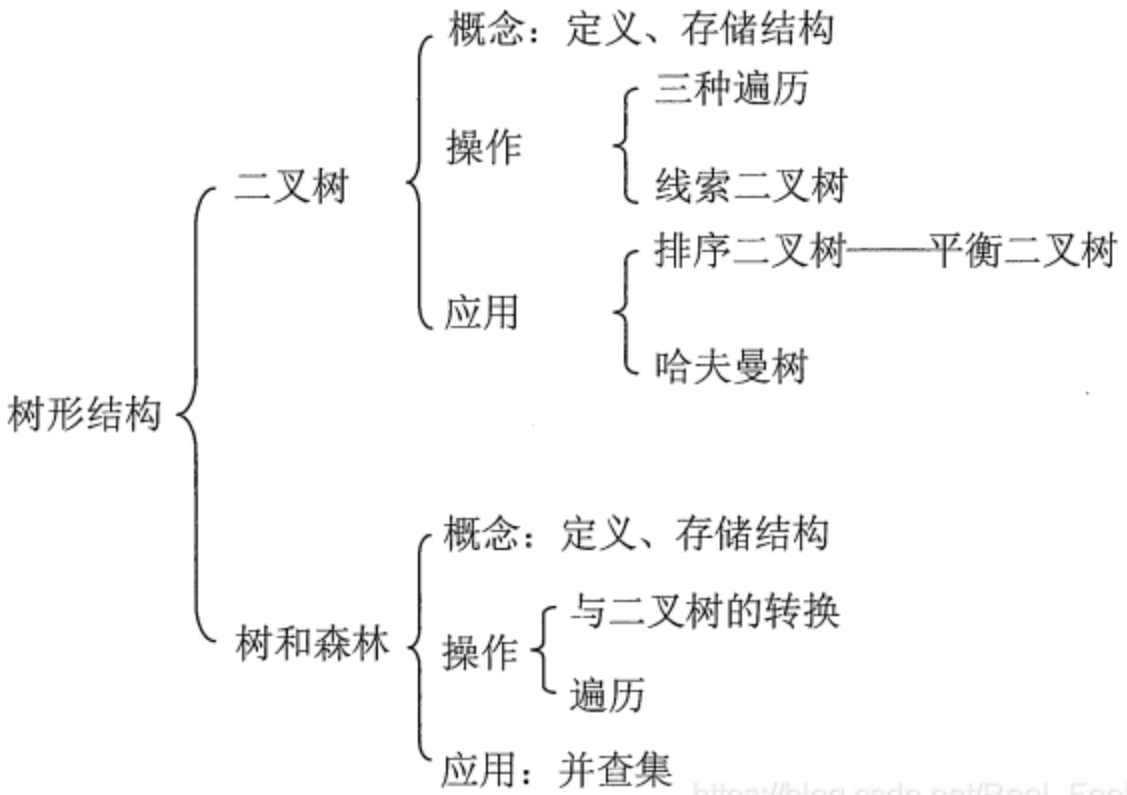
树

【知识框架】



一、树的基本概念

1、树的定义

树是n（n>=0）个结点的有限集。当n = 0时，称为空树。在任意一棵非空树中应满足：

1. 有且仅有一个特定的称为根的结点。
2. 当n>1时，其余节点可分为m（m>0）个互不相交的有限集T1,T2,…,Tm，其中每个集合本身又是一棵树，并且称为根的子树。

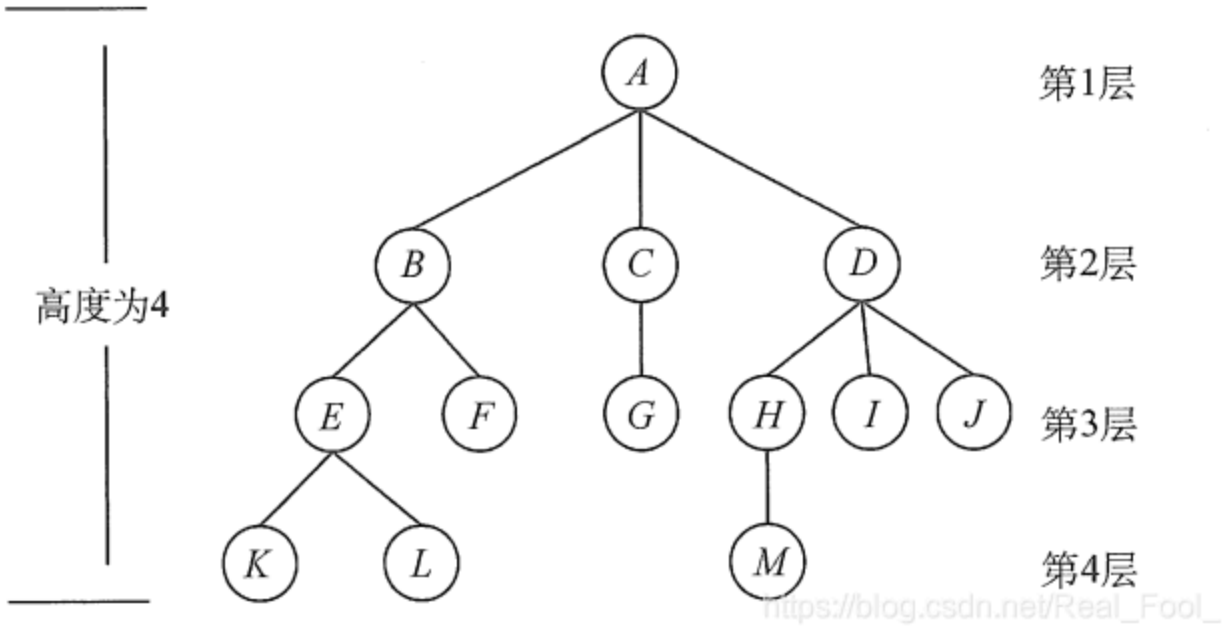
显然，树的定义是递归的，即在树的定义中又用到了自身，树是一种递归的数据结构。树作为一种逻辑结构，同时也是一种分层结构，具有以下两个特点：

1. 树的根结点没有前驱，除根结点外的所有结点有且只有一个前驱。
2. 树中所有结点可以有零个或多个后继。

因此n个结点的树中有n-1条边。

2、基本术语

下面结合图示来说明一下树的一些基本术语和概念。



1. 考虑结点K。根A到结点K的唯一路径上的任意结点,称为结点K的**祖先**。如结点B是结点K的祖先,而结点K是结点B的**子孙**。路径上最接近结点K的结点E称为K的**双亲**,而K为结点E的**孩子**。根A是树中唯一没有双亲的结点。有相同双亲的结点称为**兄弟**,如结点K和结点L有相同的双亲E,即K和L为兄弟。
2. 树中一个结点的孩子个数称为该**结点的度**,树中结点的最大度数称为**树的度**。如结点B的度为2,结点D的度为3,树的度为3。
3. 度大于0的结点称为**分支结点**(又称**非终端结点**);度为0(没有子女结点)的结点称为**叶子结点**(又称**终端结点**)。在分支结点中,每个结点的分支数就是该结点的度。
4. 结点的深度、高度和层次。
   结点的层次从树根开始定义,根结点为第1层,它的子结点为第2层,以此类推。双亲在同一层的结点互为**堂兄弟**,图中结点G与E,F,H,I,J互为堂兄弟。
   结点的深度是从根结点开始自顶向下逐层累加的。
   结点的高度是从叶结点开始自底向上逐层累加的。
   树的高度(或**深度**)是树中结点的最大层数。图中树的高度为4。
5. 有序树和无序树。树中结点的各子树从左到右是有次序的,不能互换,称该树为**有序树**,否则称为**无序树**。假设图为有序树,若将子结点位置互换,则变成一棵不同的树。
6. 路径和路径长度。树中两个结点之间的**路径**是由这两个结点之间所经过的结点序列构成的,而**路径长度**是路径上所经过的边的个数。
   注意:由于树中的分支是有向的,即从双亲指向孩子,所以树中的路径是从上向下的,同一双亲的两个孩子之间不存在路径。

7. 森林。**森林**是m (m≥0)棵互不相交的树的集合。森林的概念与树的概念十分相近，因为只要把树的根结点删去就成了森林。反之，只要给m棵独立的树加上一个结点，并把这m棵树作为该结点的子树，则森林就变成了树。

注意:上述概念无须刻意记忆， 根据实例理解即可。

3、树的性质

树具有如下最基本的性质:

1. 树中的结点数等于所有结点的度数加1.
2. 度为 m m m的树中第 i i i层上至多有 m i − 1 m^{i-1} mi−1个结点 （ i > = 1 i>=1 i>=1）
3. 高度为 h h h的 m m m叉树至多有 (m h − 1) / (m − 1) (m^h-1)/(m-1) (mh−1)/(m−1)个结点。
4. 具有 n n n个结点的 m m m叉树的最小高度为

$$logm(n(m-1)+1)$$

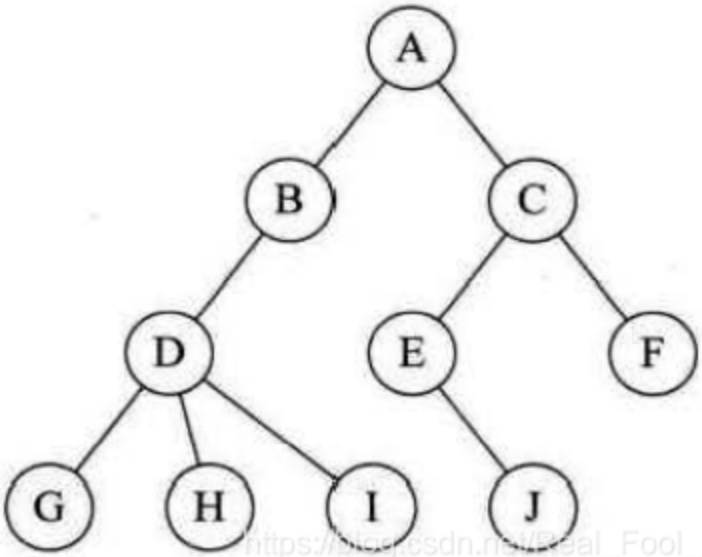
$$log\_m(n(m-1)+1)$$

$$logm(n(m-1)+1)$$

。

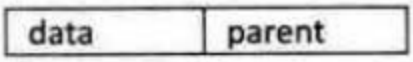
二、树的存储结构

在介绍以下三种存储结构的过程中，我们都以下面这个树为例子。



1、双亲表示法

我们假设以一组连续空间存储树的结点，同时**在每个结点中，附设一个指示器指示其双亲结点到链表中的位置**。也就是说，每个结点除了知道自已是谁以外，还知道它的双亲在哪里。



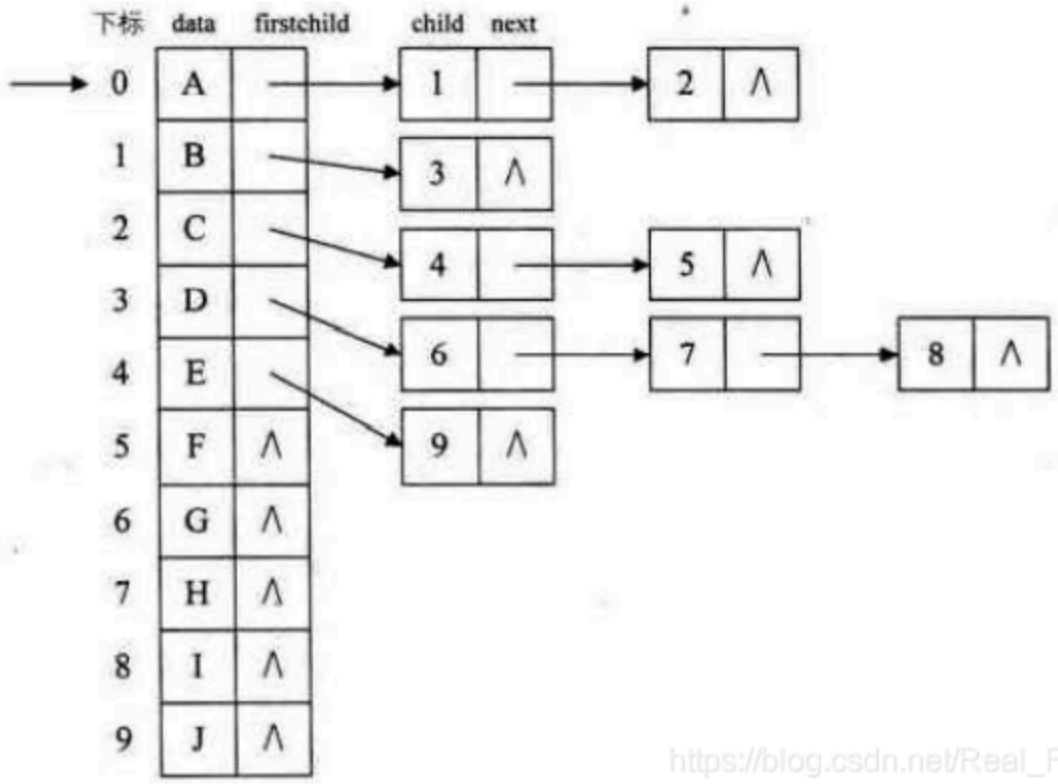
其中data是数据域，存储结点的数据信息。而parent是指针域，存储该结点的双亲在数组中的下标。
以下是我们的双亲表示法的结点结构定义代码。

```
/*树的双亲表示法结点结构定义*/
#define MAX_TREE_SIZE 100
typedef int TElemType;  //树结点的数据类型，目前暂定为整型
/*结点结构*/
typedef struct PTNode{
        TElemType data; //结点数据
        int parent;     //双亲位置
}PTNode;
/*树结构*/
typedef struct{
        PTNode nodes[MAX_TREE_SIZE];    //结点数组
        int r, n;       //根的位置和结点数
}PTree;
```

这样的存储结构，我们可以根据结点的parent 指针很容易找到它的双亲结点，所用的时间复杂度为0(1)，直到parent为-1时，表示找到了树结点的根。可如果我们要知道结点的孩子是什么，对不起，请遍历整个结构才行。

2、孩子表示法

具体办法是，**把每个结点的孩子结点排列起来，以单链表作存储结构，则n个结点有n个孩子链表，如果是叶子结点则此单链表为空。然后n个头指针又组成-一个线性表，采用顺序存储结构，存放进一个一维数组中**，如图所示。



为此，设计两种结点结构，一个是孩子链表的孩子结点。

child	next

其中child是数据域，用来存储某个结点在表头数组中的下标。next 是指针域，用来存储指向某结点的下一个孩子结点的指针。

另一个是表头数组的表头结点。

data	firstchild

其中data是数据域，存储某结点的数据信息。firstchild 是头指针域，存储该结点的孩子链表的头指针。

以下是我们的孩子表示法的结构定义代码。

```
/*树的孩子表示法结构定义*/
#define MAX_TREE_SIZE 100
/*孩子结点*/
typedef struct CTNode{
        int child;
        struct CTNode *next;
}*ChildPtr;
/*表头结点*/
typedef struct{
        TElemType data;
        ChildPtr firstchild;
}CTBox;
/*树结构*/
typedef struct{
        CTBox nodes[MAX_TREE_SIZE];     //结点数组
        int r, n;       //根的位置和结点数
}
```

这样的结构对于我们要查找某个结点的某个孩子，或者找某个结点的兄弟，只需要查找这个结点的孩子单链表即可。对于遍历整棵树也是很方便的，对头结点的数组循环即可。
但是，这也存在着问题，我如何知道某个结点的双亲是谁呢?比较麻烦，需要整棵树遍历才行，难道就不可以把双亲表示法和孩子表示法综合一下吗? 当然是可以，这个读者可自己尝试结合一下，在次不做赘述。

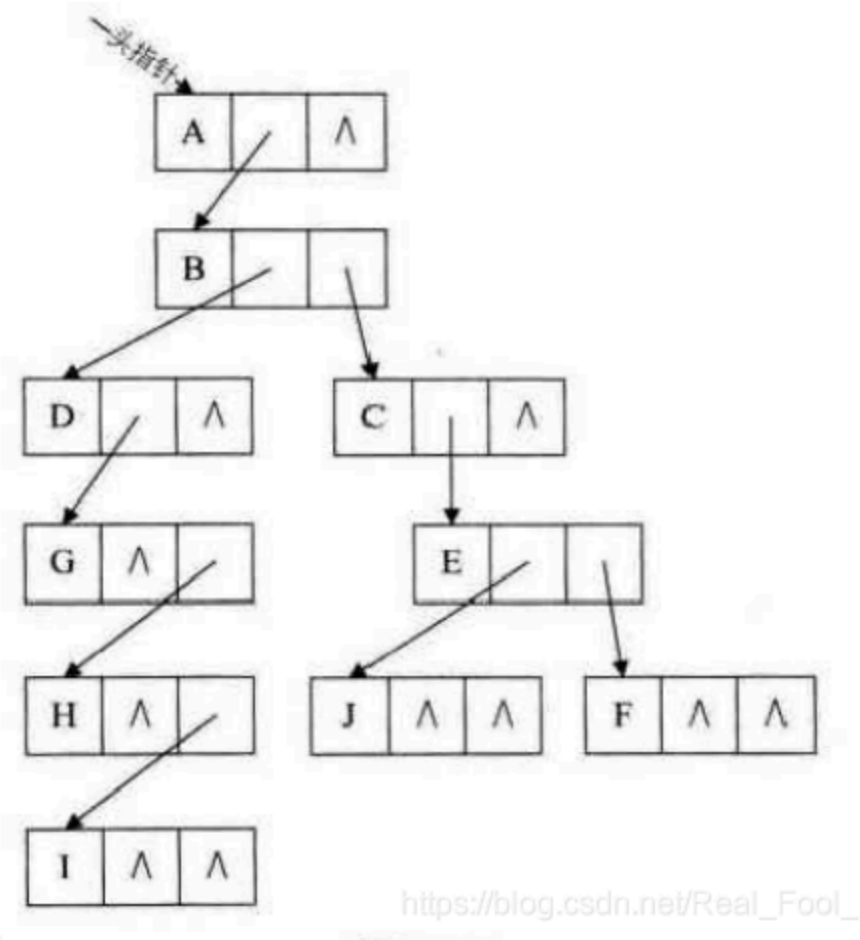
3、孩子兄弟表示法

刚才我们分别从双亲的角度和从孩子的角度研究树的存储结构，如果我们从树结点的兄弟的角度又会如何呢?当然，对于树这样的层级结构来说，只研究结点的兄弟是不行的，我们观察后发现，**任意一棵树， 它的结点的第一个孩子如果存在就是唯一的，它的右兄弟如果存在也是唯一的。 因此，我们设置两个指针，分别指向该结点的第一个孩子和此结点的右兄弟。**
结点的结构如下：

其中data是数据域，firstchild 为指针域，存储该结点的第一个孩子结点的存储地址，rightsib 是指针域，存储该结点的右兄弟结点的存储地址。
这种表示法，给查找某个结点的某个孩子带来了方便。
结构定义代码如下。

```
/*树的孩子兄弟表示法结构定义*/
typedef struct CSNode{
        TElemtype data;
        struct CSNode *firstchild, *rightsib;
} CSNode, *CSTree;
```

于是通过这种结构，我们就把原来的树变成了这个样子：



这不就是个二叉树么？
没错，其实这个表示法的最大好处就是它把一棵复杂的树变成了一棵**二叉树**。
接下来，我们详细介绍二叉树。

二叉树

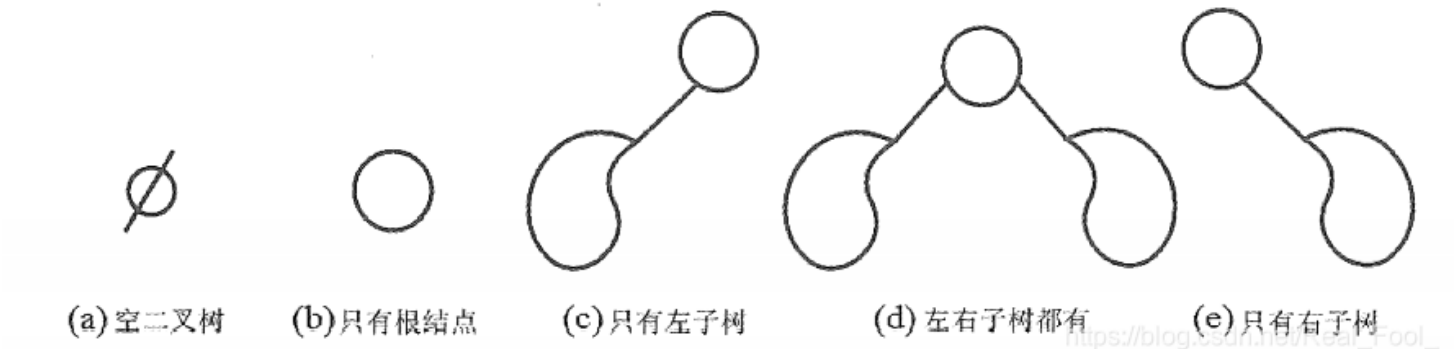
一、二叉树的概念

1、二叉树的定义

二叉树是另一种树形结构，其特点是每个结点至多只有两棵子树(即二叉树中不存在度大于2的结点)，并且二叉树的子树有左右之分，其次序不能任意颠倒。
与树相似，二叉树也以递归的形式定义。二叉树是n (n≥0) 个结点的有限集合:

1. 或者为空二叉树，即n=0。
2. 或者由一个根结点和两个互不相交的被称为根的左子树和右子树组成。左子树和右子树又分别是一棵二叉树。

二叉树是有序树，若将其左、右子树颠倒，则成为另一棵不同的二叉树。即使树中结点只有一棵子树，也要区分它是左子树还是右子树。二叉树的5种基本形态如图所示。



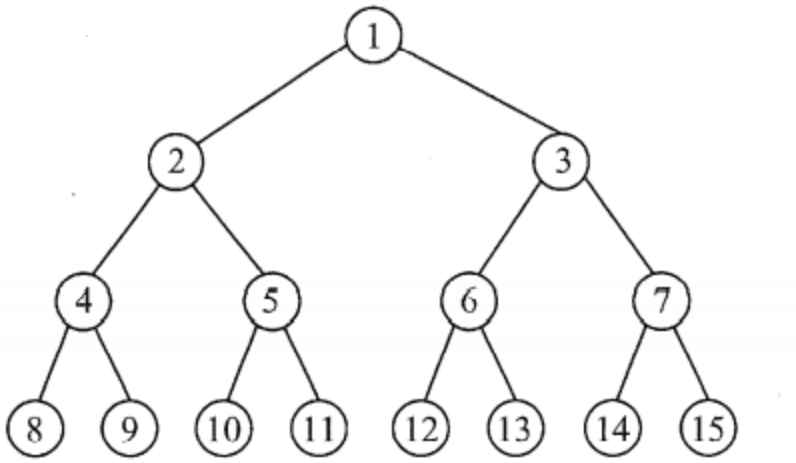
2、几个特殊的二叉树

（1）斜树

所有的结点都只有左子树的二叉树叫左斜树。所有结点都是只有右子树的二叉树叫右斜树。这两者统称为斜树。

（2）满二叉树

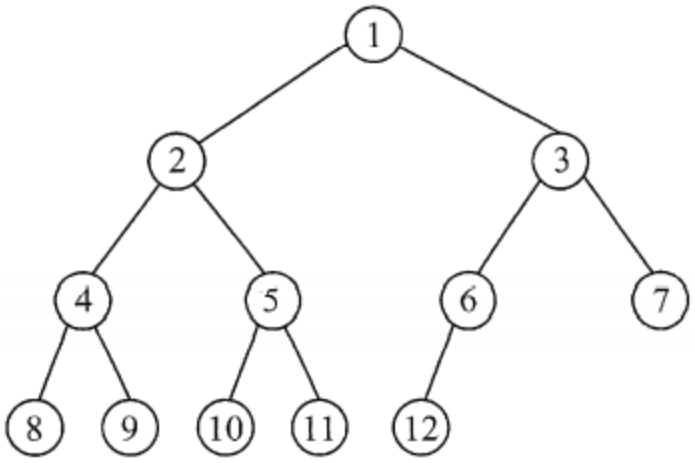
一棵高度为 h h h，且含有 2 h − 1 2^h-1 2h−1个结点的二叉树称为满二叉树，即树中的每层都含有最多的结点。满二叉树的叶子结点都集中在二叉树的最下一层，并且除叶子结点之外的每个结点度数均为 2 2 2。可以对满二叉树按层序编号:约定编号从根结点(根结点编号为 1 1 1)起，自上而下，自左向右。这样，每个结点对应一个编号,对于编号为i的结点，若有双亲，则其双亲为 i / 2 i/2 i/2,若有左孩子，则左孩子为 2 i 2i 2i;若有右孩子，则右孩子为 2 i + 1 2i+1 2i+1。



(a) 满二叉树

（3）完全二叉树

高度为 h h h、有 n n n个结点的二叉树，当且仅当其每个结点都与高度为 h h h的满二叉树中编号为1~n的结点一一对应时，称为完全二叉树，如图所示。其特点如下:



(b) 完全二叉树

1. 若 i ≤ n / 2 i≤n/2 i≤n/2, 则结点 i i i为分支结点，否则为叶子结点。
2. 叶子结点只可能在层次最大的两层上出现。对于最大层次中的叶子结点，都依次排列在该层最左边的位置上。
3. 若有度为 1 1 1的结点，则只可能有一个，且该结点只有左孩子而无右孩子(重要特征)。
4. 按层序编号后，一旦出现某结点(编号为 i i i)为叶子结点或只有左孩子，则编号大于 i i i的结点均为叶子结点。
5. 若 n n n为奇数，则每个分支结点都有左孩子和右孩子;若 n n n为偶数，则编号最大的分支结点(编号为 n / 2 n/2 n/2)只有左孩子，没有右孩子，其余分支结点左、右孩子都有。

（4）二叉排序树

左子树上所有结点的关键字均小于根结点的关键字;右子树上的所有结点的关键字均大于根结点的关键字;左子树和右子树又各是一棵二叉排序树。

（5）平衡二叉树

树上任一结点的左子树和右子树的深度之差不超过1。

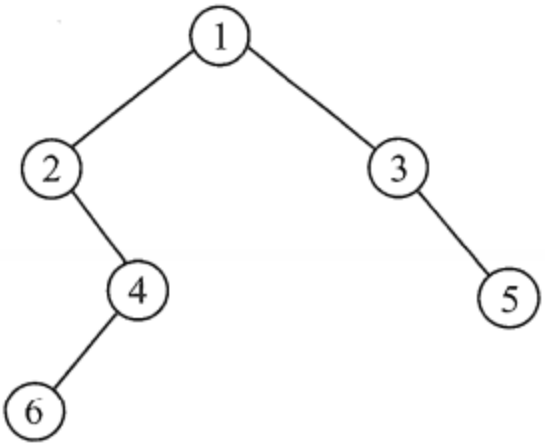
3、二叉树的性质

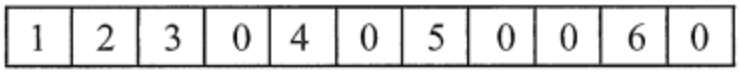
1. 任意一棵树，若结点数量为 n n n,则边的数量为 n − 1 n-1 n−1。
2. 非空二叉树上的叶子结点数等于度为 2 2 2的结点数加 1 1 1，即 n o = n 2 + 1 n_o=n_2+ 1 no=n2+1。
3. 非空二叉树上第 k k k层上至多有 2 k − 1 2^{k-1} 2k−1个结点 (k ≥ 1) (k≥1) (k≥1)。
4. 高度为 h h h的二叉树至多有 2 h − 1 2^h-1 2h−1个结点 (h ≥ 1) (h≥1) (h≥1)。
5. 对完全二叉树按从上到下、从左到右的顺序依次编号 1 , 2.. ∗ , n 1,2..*,n 1,2..∗,n,则有以下关系:
   - i > 1 i>1 i>1时，结点 i i i的双亲的编号为 i / 2 i/2 i/2,即当 i i i为偶数时， 它是双亲的左孩子;当i为奇数时，它是双亲的右孩子。
   - 当 2 i ≤ n 2i≤n 2i≤n时，结点 i i i的左孩子编号为 2 i 2i 2i, 否则无左孩子。
   - 当 2 i + 1 ≤ n 2i+1≤n 2i+1≤n时，结点 i i i的右孩子编号为 2 i + 1 2i+1 2i+1，否则无右孩子。
   - 结点 i i i所在层次(深度)为 { l o g 2 i } + 1 \{log_2i\}+ 1 {log2i}+1。
6. 具有 n n n个 (n > 0) (n>0) (n>0)结点的完全二叉树的高度为 { l o g 2 n } + 1 \{log_2n\}+1 {log2n}+1。

4、二叉树的存储结构

（1）顺序存储结构

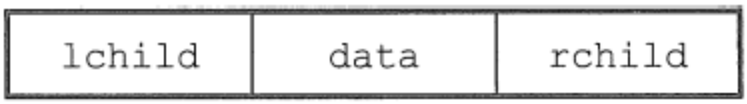
二叉树的顺序存储是指用一组地址连续的存储单元依次自上而下、自左至右存储完全二叉树上的结点元素，即将完全二叉树上编号为 i i i的结点元素存储在一维数组下标为 i − 1 i-1 i−1的分量中。
依据二叉树的性质，完全二叉树和满二叉树采用顺序存储比较合适，树中结点的序号可以唯一地反映结点之间的逻辑关系，这样既能最大可能地节省存储空间，又能利用数组元素的下标值确定结点在二叉树中的位置，以及结点之间的关系。
但对于一般的二叉树，为了让数组下标能反映二叉树中结点之间的逻辑关系，只能添加一些并不存在的空结点，让其每个结点与完全二叉树上的结点相对照，再存储到一维数组的相应分量中。然而，在最坏情况下，一个高度为 h h h且只有 h h h个结点的单支树却需要占据近 2 h − 1 2h-1 2h−1个存储单元。二叉树的顺序存储结构如图所示，其中0表示并不存在的空结点。





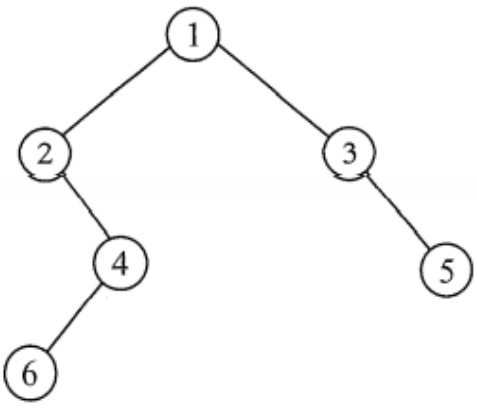
（2）链式存储结构

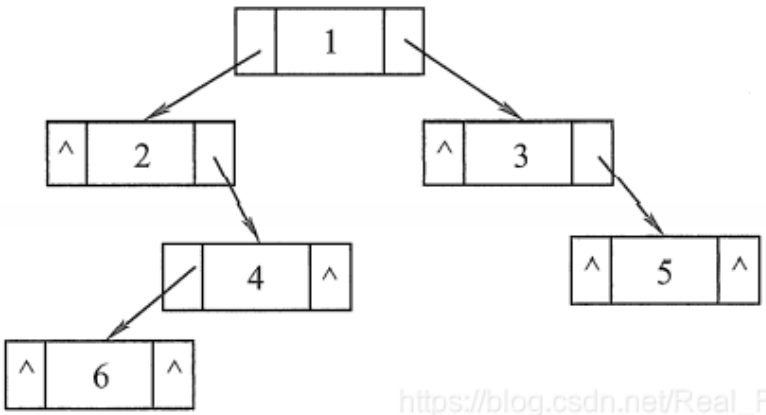
既然顺序存储适用性不强，我们就要考虑链式存储结构。**二叉树每个结点最多有两个孩子，所以为它设计一个数据域和两个指针域是比较自然的想法，我们称这样的链表叫做二叉链表。**



其中data是数据域，lchild 和rchild都是指针域，分别存放指向左孩子和右孩子的指针。
以下是我们的二叉链表的结点结构定义代码。

```
/*二叉树的二叉链表结点构造定义*/
/*结点结构*/
typedef struct BiTNode{
        TElemType data; //结点数据
        struct BiTNode *lchild, *rchild;        //左右孩子指针
} BiTNode, *BiTree;
```





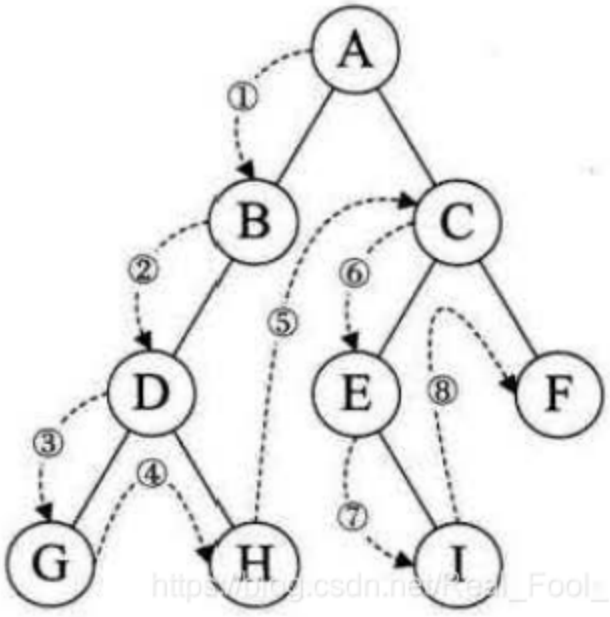
容易验证，**在含有 n n n个结点的二叉链表中，含有 n + 1 n + 1 n+1个空链域。**

二、遍历二叉树

二叉树的遍历(traversing binary tree)是指从根结点出发，按照某种次序依次访问二叉树中所有结点，使得每个结点被访问一次且仅被访问一次。

1、先序遍历

> 先序遍历(PreOrder) 的操作过程如下：
> 若二叉树为空，则什么也不做，否则，
> 1)访问根结点;
> 2)先序遍历左子树;
> 3)先序遍历右子树。

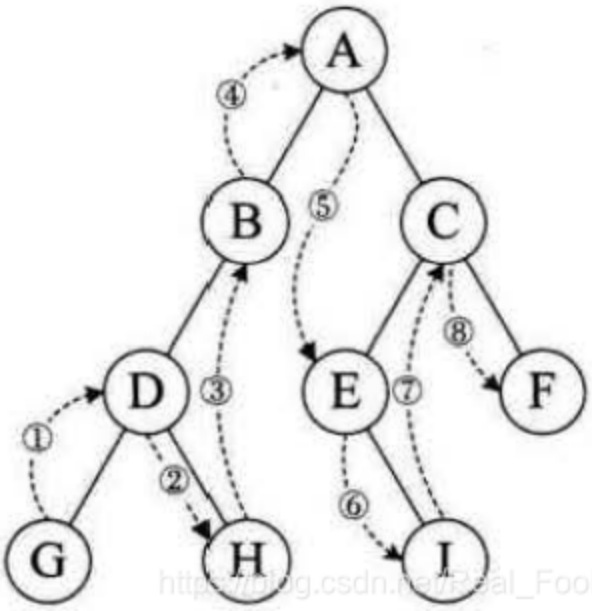


对应的递归算法如下:

```
void PreOrder(BiTree T){
        if(T != NULL){
                visit(T);       //访问根节点
                PreOrder(T->lchild);    //递归遍历左子树
                PreOrder(T->rchild);    //递归遍历右子树
        }
}
```

2、中序遍历

> 中序遍历(InOrder)的操作过程如下：
> 若二叉树为空，则什么也不做，否则，
> 1)中序遍历左子树;
> 2)访问根结点;
> 3)中序遍历右子树。

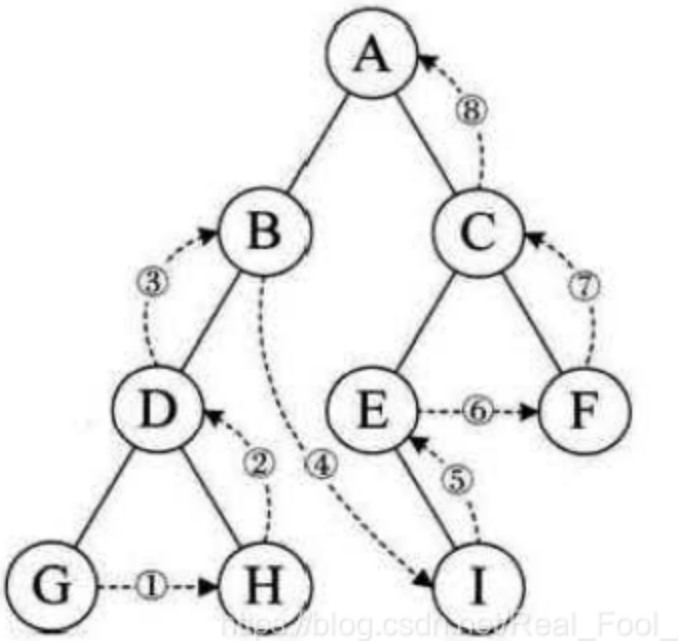


对应的递归算法如下:

```
void InOrder(BiTree T){
        if(T != NULL){
                InOrder(T->lchild);     //递归遍历左子树
                visit(T);       //访问根结点
                InOrder(T->rchild);     //递归遍历右子树
        }
}
```

3、后序遍历

> 后序遍历(PostOrder) 的操作过程如下：
> 若二叉树为空，则什么也不做，否则，
> 1)后序遍历左子树;
> 2)后序遍历右子树;
> 3)访问根结点。



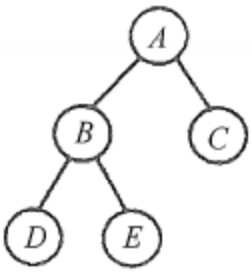
对应的递归算法如下:

```
void PostOrder(BiTree T){
        if(T != NULL){
                PostOrder(T->lchild);    //递归遍历左子树
                PostOrder(T->rchild);    //递归遍历右子树
                visit(T);        //访问根结点
        }
}
```

三种遍历算法中,递归遍历左、右子树的顺序都是固定的，只是访问根结点的顺序不同。不管采用哪种遍历算法，每个结点都访问一次且仅访问一次，故时间复杂度都是O(n)。在递归遍历中，递归工作栈的栈深恰好为树的深度，所以在最坏情况下，二叉树是有n个结点且深度为n的单支树，遍历算法的空间复杂度为O(n)。

4、递归算法和非递归算法的转换

我们以下图的树为例子。



(a)二叉树

（1）中序遍历的非递归算法

借助栈，我们来分析中序遍历的访问过程:

1. 沿着根的左孩子，依次入栈，直到左孩子为空，说明已找到可以输出的结点，此时栈内元素依次为ABD。
2. 栈顶元素出栈并访问:若其右孩子为空，继续执行步骤2;若其右孩子不空,将右子树转执行步骤1。

栈顶D出栈并访问，它是中序序列的第一个结点; D右孩子为空，栈顶B出栈并访问; B右孩子不空，将其右孩子E入栈，E左孩子为空，栈顶E出栈并访问; E右孩子为空，栈顶A出栈并访问; A右孩子不空，将其右孩子C入栈，C左孩子为空，栈顶C出栈并访问。由此得到中序序列DBEAC。
根据分析可以写出中序遍历的非递归算法如下:

```
void InOrder2(BiTree T){
        InitStack(S);    //初始化栈S
        BiTree p = T;    //p是遍历指针
        while(p || !IsEmpty(S)){        //栈不空或p不空时循环
                if(p){
                        Push(S, p);     //当前节点入栈
                        p = p->lchild;  //左孩子不空，一直向左走
                }else{
                        Pop(S, p);      //栈顶元素出栈
                        visit(p);       //访问出栈结点
                        p = p->rchild;  //向右子树走，p赋值为当前结点的右孩子
                }
        }
}
```

（2）先序遍历的非递归算法

先序遍历和中序遍历的基本思想是类似的，只需把访问结点操作放在入栈操作的前面。先序遍历的非递归算法如下:

```
void PreOrder2(BiTree T){
        InitStack(S);    //初始化栈S
        BiTree p = T;    //p是遍历指针
        while(p || !IsEmpty(S)){        //栈不空或p不空时循环
                if(p){
                        visit(p);       //访问出栈结点
                        Push(S, p);     //当前节点入栈
                        p = p->lchild;  //左孩子不空，一直向左走
                }else{
                        Pop(S, p);      //栈顶元素出栈
                        p = p->rchild;  //向右子树走，p赋值为当前结点的右孩子
                }
        }
}
```

（3）后序遍历的非递归算法

后序遍历的非递归实现是三种遍历方法中最难的。因为在后序遍历中，要保证左孩了和右孩子都已被访问并且左孩子在右孩子前访问才能访问根结点，这就为流程的控制带来了难题。

算法思想:后序非递归遍历二叉树是先访问左子树，再访问右子树，最后访问根结点。

1. 沿着根的左孩子，依次入栈，直到左孩子为空。此时栈内元素依次为ABD。
2. 读栈顶元素:若其右孩子不空且未被访问过，将右子树转执行①;否则，栈顶元素出栈并访问。

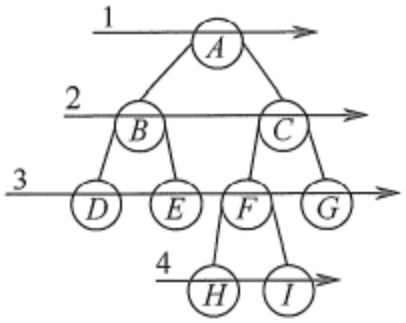
栈顶D的右孩子为空，出栈并访问，它是后序序列的第一个结点;栈顶B的右孩子不空且未被访问过，E入栈，栈顶E的左右孩子均为空，出栈并访问;栈顶B的右孩子不空但已被访问，B出栈并访问;栈项A的右孩子不空且未被访问过，C入栈,栈项C的左右孩子均为空，出栈并访问;栈顶A的右孩子不空但已被访问，A出栈并访问。由此得到后序序列DEBCA。
在上述思想的第②步中，必须分清返回时是从左子树返回的还是从右子树返回的，因此设定一个辅助指针r,指向最近访问过的结点。也可在结点中增加一个标志域，记录是否已被访问。

后序遍历的非递归算法如下:

```
void PostOrder2(BiTree T){
        InitStack(S);
        p = T;
        r = NULL;
        while(p || !IsEmpty(S)){
                if(p){  //走到最左边
                        push(S, p);
                        p = p->lchild;
                }else{  //向右
                        GetTop(S, p);   //读栈顶元素（非出栈）
                        //若右子树存在，且未被访问过
                        if(p->rchild && p->rchild != r){
                                p = p->rchild;  //转向右
                                push(S, p);     //压入栈
                                p = p->lchild;  //再走到最左
                        }else{  //否则，弹出结点并访问
                                pop(S, p);      //将结点弹出
                                visit(p->data); //访问该结点
                                r = p;  //记录最近访问过的结点
                                p = NULL;
                        }
                }
        }
}
```

5、层次遍历

下图为二叉树的层次遍历，即按照箭头所指方向，按照1,2,3, 4的层次顺序，对二叉树中的各个结点进行访问。



要进行层次遍历，需要借助一个队列。先将二叉树根结点入队，然后出队，访问出队结点，若它有左子树，则将左子树根结点入队;若它有右子树，则将右子树根结点入队。然后出队，访问出队结…如此反复，直至队列为空。
二叉树的层次遍历算法如下:

```
void LevelOrder(BiTree T){
        InitQueue(Q);   //初始化辅助队列
        BiTree p;
        EnQueue(Q, T);  //将根节点入队
        while(!IsEmpty(Q)){     //队列不空则循环
                DeQueue(Q, p);  //队头结点出队
                visit(p);       //访问出队结点
                if(p->lchild != NULL){
                        EnQueue(Q, p->lchild);  //左子树不空，则左子树根节点入队
                }
                if(p->rchild != NULL){
                        EnQueue(Q, p->rchild);  //右子树不空，则右子树根节点入队
                }
        }
}
```

6、由遍历序列构造二叉树

由二叉树的先序序列和中序序列可以唯一地确定一棵二叉树。

在先序遍历序列中,第一个结点一定是二叉树的根结点;而在中序遍历中,根结点必然将中序序列分割成两个子序列,前一个子序列是根结点的左子树的中序序列,后一个子序列是根结点的右子树的中序序列。根据这两个子序列,在先序序列中找到对应的左子序列和右子序列。在先序序列中,左子序列的第一个结点是左子树的根结点,右子序列的第一个结点是右子树的根结点。

如此递归地进行下去,便能唯一地确定这棵二叉树

同理,**由二叉树的后序序列和中序序列也可以唯一地确定一棵二叉树。**

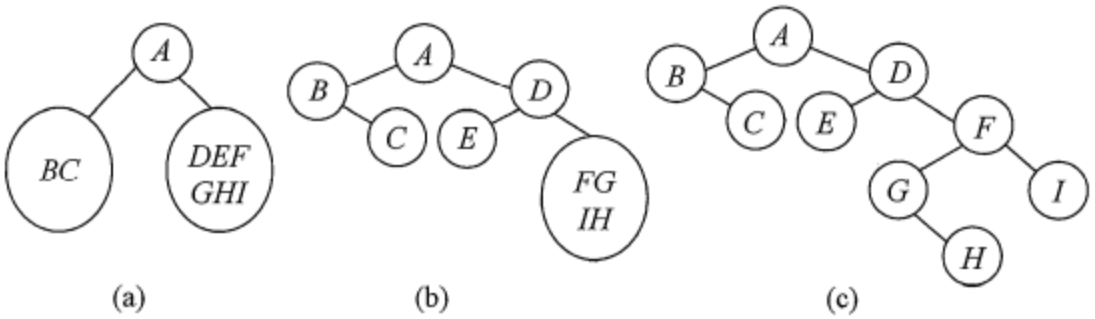
因为后序序列的最后一个结点就如同先序序列的第一个结点,可以将中序序列分割成两个子序列,然后采用类似的方法递归地进行划分,进而得到一棵二叉树。

由二叉树的层序序列和中序序列也可以唯一地确定一棵二叉树。

要注意的是,若只知道二叉树的先序序列和后序序列,则无法唯一确定一棵二叉树。

例如,求先序序列(ABCDEFGH)和中序序列(BCAEDGHFI) 所确定的二叉树

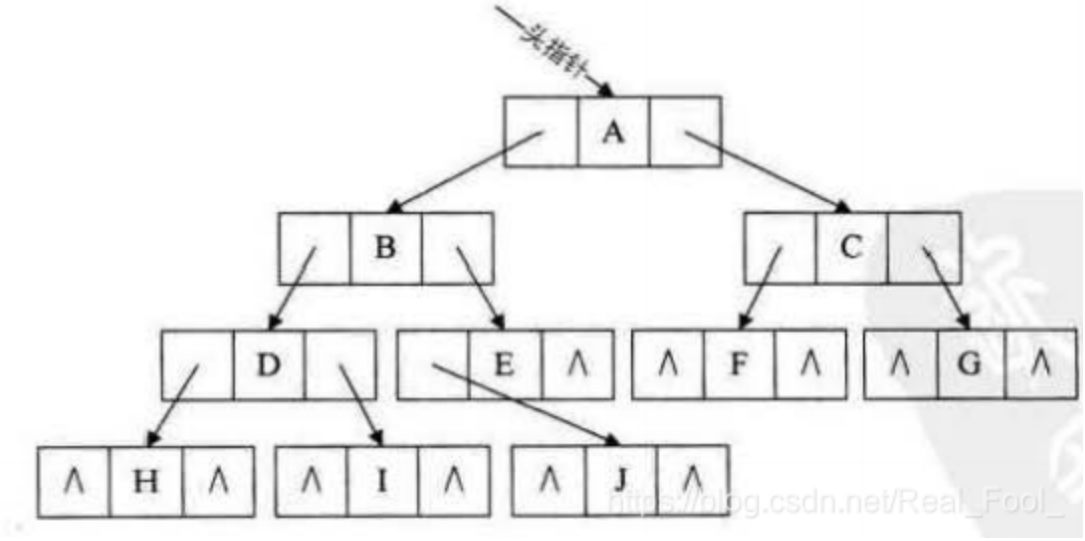
首先,由先序序列可知A为二叉树的根结点。中序序列中A之前的BC为左子树的中序序列,EDGHFI为右子树的中序序列。然后由先序序列可知B是左子树的根结点,D是右子树的根结点。以此类推,就能将剩下的结点继续分解下去,最后得到的二叉树如图©所示。



三、线索二叉树

1、线索二叉树原理

遍历二叉树是以一定的规则将二叉树中的结点排列成一个线性序列，从而得到几种遍历序列，使得该序列中的每个结点(第一个和最后一个结点除外)都有一个直接前驱和直接后继。
传统的二叉链表存储仅能体现一种父子关系，不能直接得到结点在遍历中的前驱或后继。



首先我们要来看看这空指针有多少个呢?对于一个有n个结点的二叉链表，每个结点有指向左右孩子的两个指针域，所以一共是2n个指针域。而n个结点的二叉树一共有n-1 条分支线数，也就是说，其实是存在2n- (n-1) =n+1个空指针域。

由此设想能否利用这些空指针来存放指向其前驱或后继的指针?这样就可以像遍历单链表那样方便地遍历二叉树。引入线索二叉树正是为了加快查找结点前驱和后继的速度。
我们把这种指向前驱和后继的指针称为线索，加上线索的二叉链表称为线索链表，相应的二叉树就称为线索二叉树(Threaded Binary Tree)。

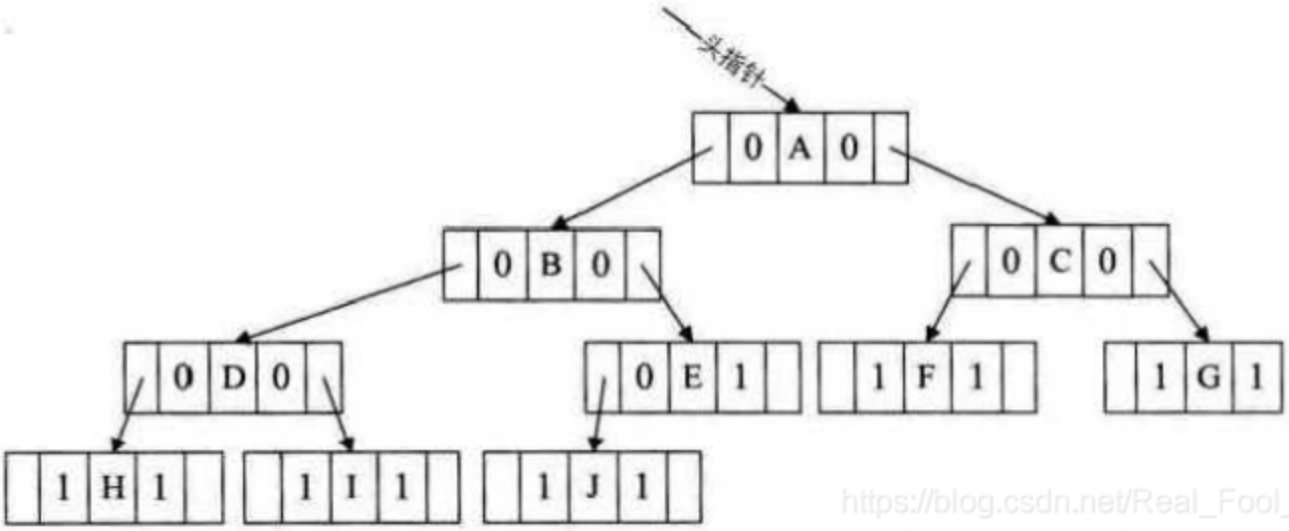
其结点结构如下所示：

lchild	ltag	data	rtag	rchild

其中

- ltag为0时指向该结点的左孩子，为1时指向该结点的前驱。
- rtag为0时指向该结点的右孩子，为1时指向该结点的后继。

因此对于上图的二叉链表图可以修改为下图的样子。



2、线索二叉树的结构实现

二叉树的线索存储结构代码如下：

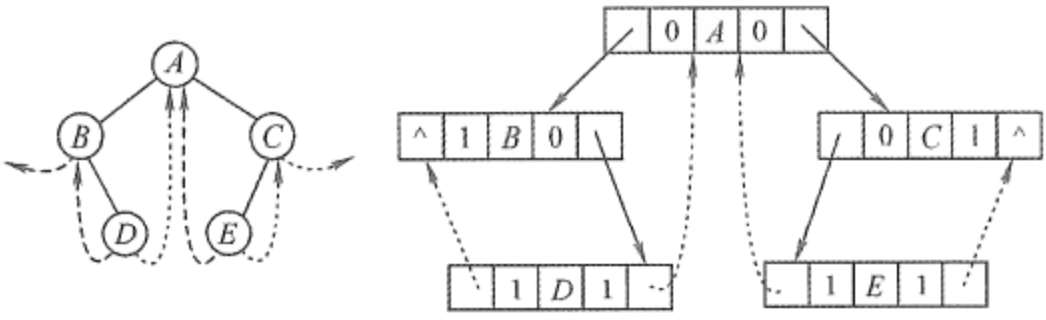
```
typedef struct ThreadNode{
        ElemType data;  //数据元素
        struct ThreadNode *lchild, *rchild;     //左、右孩子指针
        int ltag, rtag; //左、右线索标志
}ThreadNode, *ThreadTree;
```

3、二叉树的线索化

二叉树的线索化是将二叉链表中的空指针改为指向前驱或后继的线索。而前驱或后继的信息只有在遍历时才能得到，**因此线索化的实质就是遍历一次二叉树，线索化的过程就是在遍历的过程中修改空指针的过程。**

（1）中序线索二叉树

以中序线索二叉树的建立为例。附设指针pre指向刚刚访问过的结点，指针p指向正在访问的结点，即pre指向p的前驱。在中序遍历的过程中，检查p的左指针是否为空，若为空就将它指向pre;检查pre的右指针是否为空，若为空就将它指向p，如下图所示。



通过中序遍历对二叉树线索化的递归算法如下:

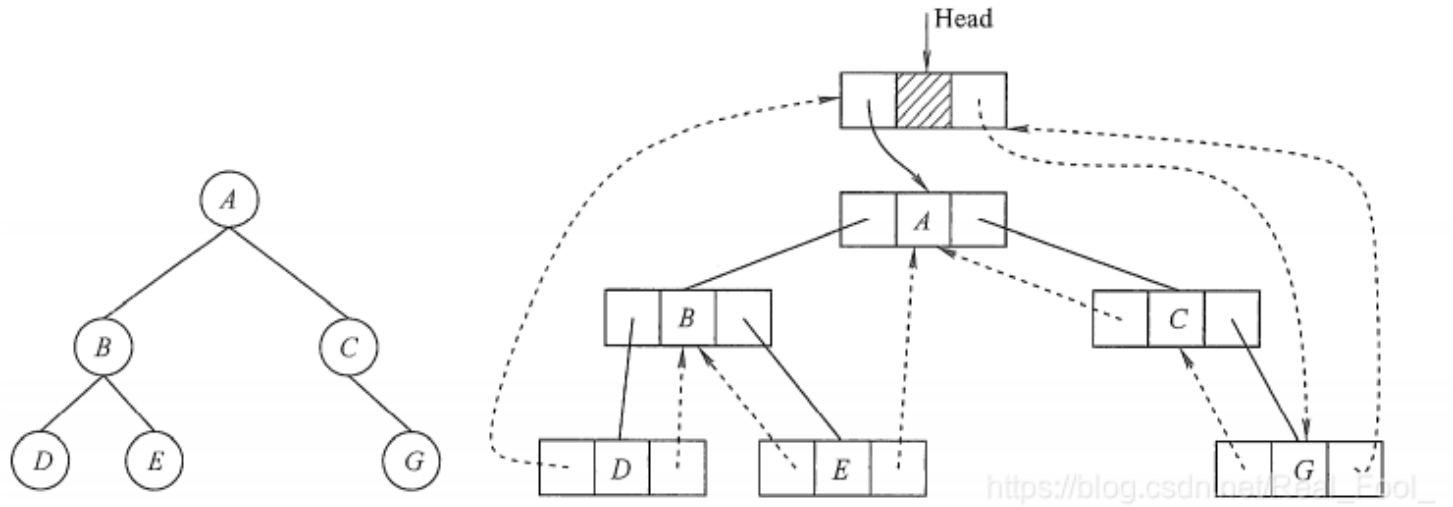
```
void InThread(ThreadTree p, ThreadTree pre){
        if(p != NULL){
                InThread(p->lchild, pre);        //递归，线索化左子树
                if(p->lchild == NULL){  //左子树为空，建立前驱线索
                        p->lchild = pre;
                        p->ltag = 1;
                }
                if(pre != NULL && pre->rchild == NULL){
                        pre->rchild = p;         //建立前驱结点的后继线索
                        pre->rtag = 1;
                }
                pre = p;         //标记当前结点成为刚刚访问过的结点
                InThread(p->rchild, pre);        //递归，线索化右子树
        }
}
```

你会发现，除了中间的代码，和二叉树中序遍历的递归代码几乎完全一样。只不过将本是访问结点的功能改成了线索化的功能。

通过中序遍历建立中序线索二叉树的主过程算法如下:

```
void CreateInThread(ThreadTree T){
        ThreadTree pre = NULL;
        if(T != NULL){
                InThread(T, pre);        //线索化二叉树
                pre->rchild = NULL;      //处理遍历的最后一个结点
                pre->rtag = 1;
        }
}
```

为了方便,可以在二叉树的线索链表上也添加一个头结点，令其lchild域的指针指向二叉树的根结点，其rchild域的指针指向中序遍历时访问的最后一个结点;令二叉树中序序列中的第一个结点的lchild域指针和最后一个结点的rchild域指针均指向头结点。这好比为二叉树建立了一个双向线索链表，方便从前往后或从后往前对线索二叉树进行遍历，如下图所示。



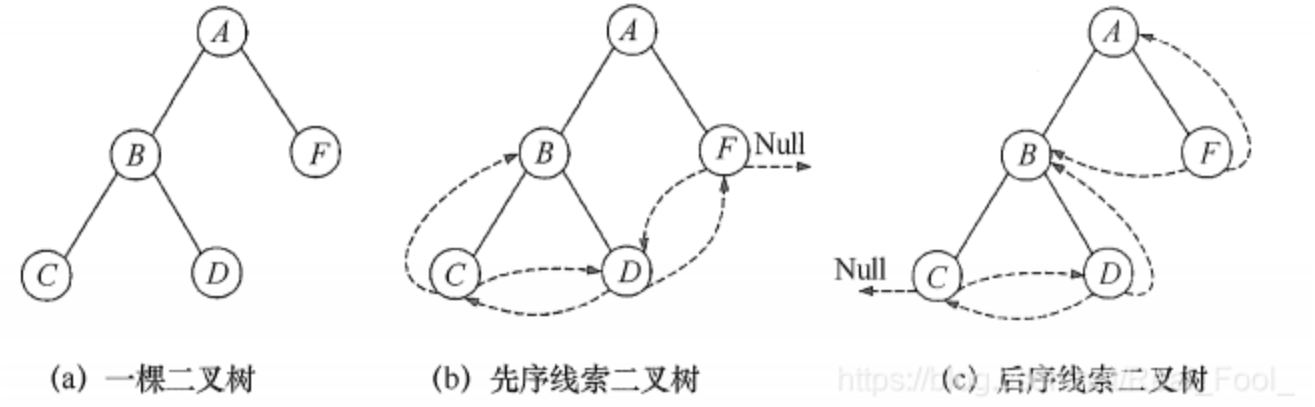
遍历的代码如下：

```
/*T指向头结点，头结点左链lchild指向根结点，头结点右链rchild指向中序遍
的最后一个结点。中序遍历二叉线索链表表示的二叉树T*/
void InOrderTraverse_Thr(BiThrTree T){
        BiThrTree p;
        p = T->lchild;  //p指向根结点
        //空树或遍历结束时，p==T（最后一个结点指向根结点）
        while(p != T){
                //当ltag==0时循环到中序序列第一个结点
                while(p->ltag == 0){
                        p = p->lchild;  //p指向p的左子树
                }
                visit(p);       //访问该结点
                //后继线索为1且不是指向头指针
                while(p->rtag == 1 && p->rchild != T){
                        p = p->rchild;  //p指向p的后继
                        visit(p);       //访问该节点
                }
                //p进至其右子树根，开始对右子树根进行遍历
                p = p->rchild;
        }
}
```

从这段代码也可以看出，它等于是一个链表的扫描,所以时间复杂度为0(n)。
由于它充分利用了空指针域的空间(这等于节省了空间)，又保证了创建时的一次遍历就可以终生受用前驱后继的信息(这意味着节省了时间)。所以在实际问题中，如果所用的二叉树需经常遍历或查找结点时需要某种遍历序列中的前驱和后继，那么采用线索二叉链表的存储结构就是非常不错的选择。

（2）先序和后序线索二叉树

上面给出了建立中序线索二叉树的代码，建立先序线索二叉树和后序线索二叉树的代码类似，只需变动线索化改造的代码段与调用线索化左右子树递归函数的位置。
以图(a)的二叉树为例，其先序序列为ABCDF，后序序列为CDBFA，可得出其先序和后序线索二叉树分别如图(b)和(c)所示：



如何在先序线索二叉树中找结点的后继?如果有左孩子，则左孩子就是其后继;如果无左孩子但有右孩子，则右孩子就是其后继;如果为叶结点，则右链域直接指示了结点的后继。
在后序线索二叉树中找结点的后继较为复杂，可分3种情况:①若结点x是二叉树的根，则其后继为空;②若结点x是其双亲的右孩子，或是其双亲的左孩子且其双亲没有右子树，则其后继即为双亲;③若结点x是其双亲的左孩子，且其双亲有右子树，则其后继为双亲的右子树上按后序遍历列出的第一个结点。图(c)中找结点B的后继无法通过链域找到，可见在后序线索二叉树上找后继时需知道结点双亲，即需采用带标志域的三叉链表作为存储结构。

四、树、森林与二叉树的转化

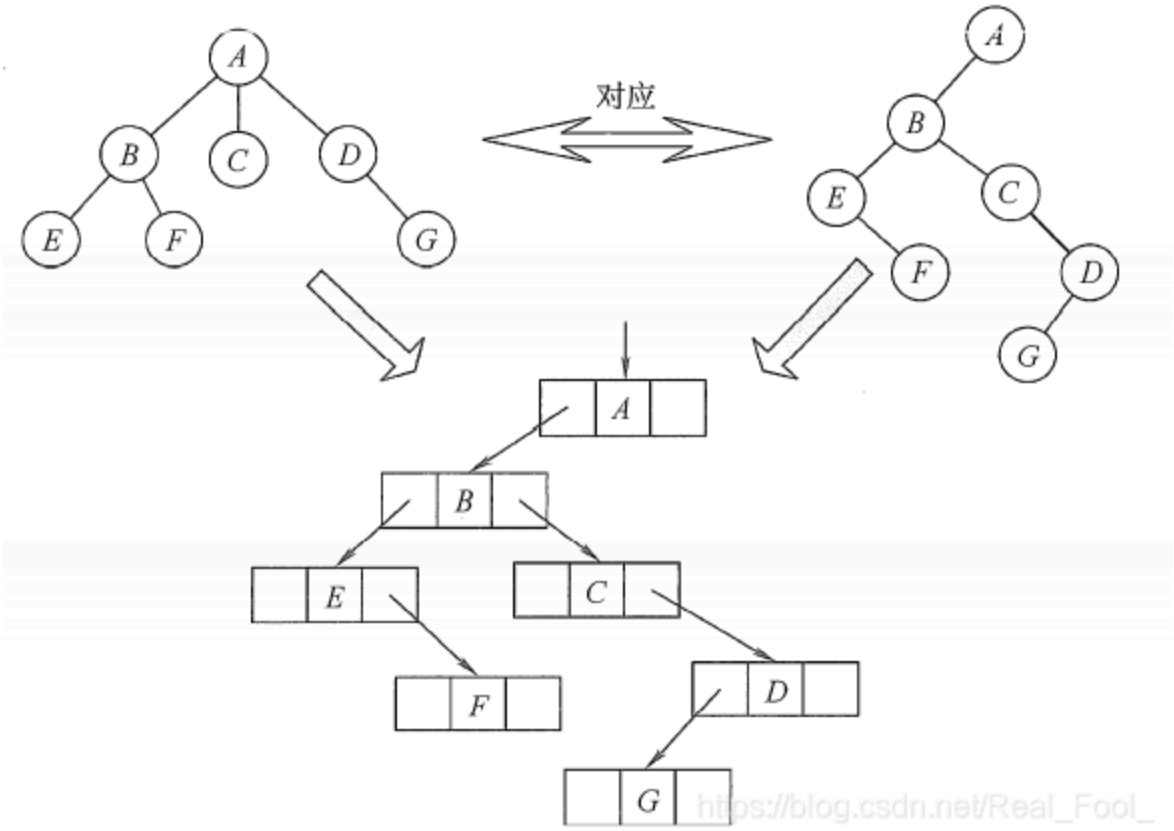
在讲树的存储结构时，我们提到了树的孩子兄弟法可以将一棵树用二叉链表进行存储，所以借助二叉链表，树和二叉树可以相互进行转换。从物理结构来看，它们的二叉链表也是相同的，只是解释不太一样而已。 因此，只要我们设定一定的规则，用二叉树来表示树，甚至表示森林都是可以的，森林与二叉树也可以互相进行转换。

1、树转换为二叉树

> 树转换为二义树的规则:每个结点左指针指向它的第一个孩子，右指针指向它在树中的相邻右兄弟，这个规则又称“左孩子右兄弟”。由于根结点没有兄弟，所以对应的二叉树没有右子树。

树转换成二叉树的画法:

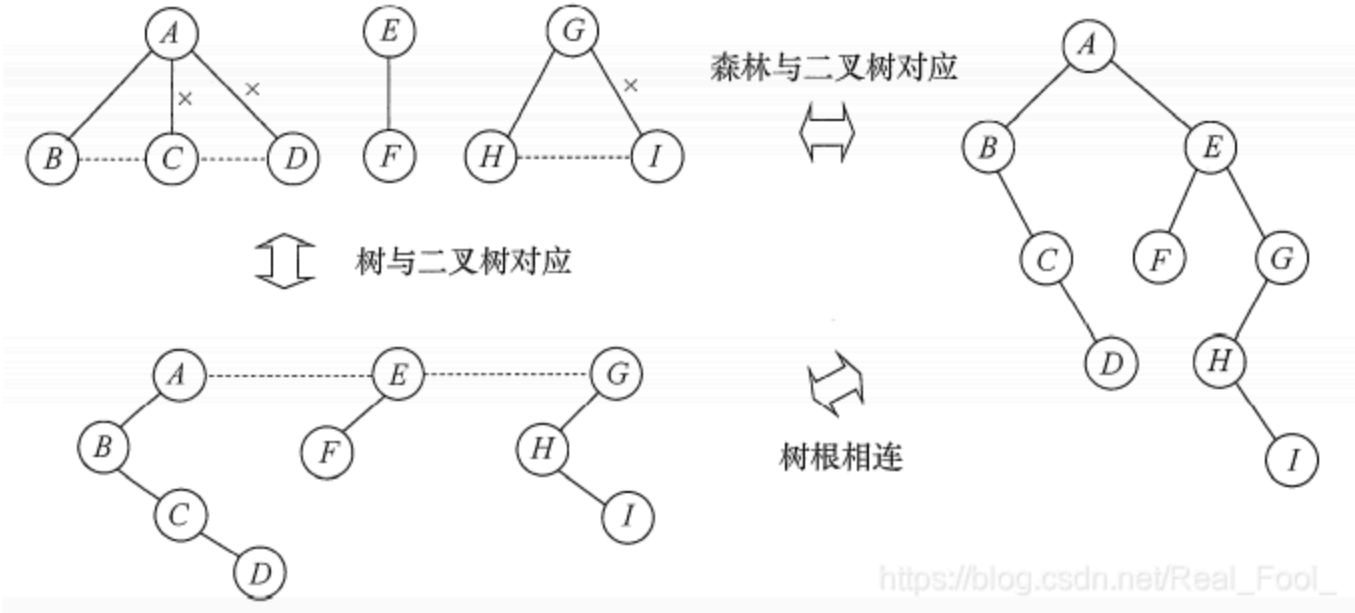
1. 在兄弟结点之间加一连线;
2. 对每个结点，只保留它与第一个孩子的连线，而与其他孩子的连线全部抹掉;
3. 以树根为轴心，顺时针旋转45°。



2、森林转化为二叉树

森林是由若干棵树组成的，所以完全可以理解为,森林中的每一棵树都是兄弟，可以按照兄弟的处理办法来操作。
森林转换成二叉树的画法:

1. 将森林中的每棵树转换成相应的二叉树;
2. 每棵树的根也可视为兄弟关系，在每棵树的根之间加一根连线;
3. 以第一棵树的根为轴心顺时针旋转45°。



至于二叉树转换为树或者二叉树转换为森林只不过是上面步骤的逆过程，在此不做赘述。

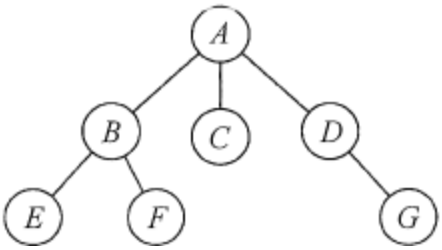
五、树和森林的遍历

1、树的遍历

树的遍历是指用某种方式访问树中的每个结点，且仅访问一次。主要有两种方式:

1. 先根遍历。若树非空，先访问根结点，再依次遍历根结点的每棵子树，遍历子树时仍遵循先根后子树的规则。其遍历序列与这棵树相应二叉树的先序序列相同。
2. 后根遍历。若树非空，先依次遍历根结点的每棵子树，再访问根结点，遍历子树时仍遵循先子树后根的规则。其遍历序列与这棵树相应二叉树的中序序列相同。

下图的树的先根遍历序列为ABEFCDG,后根遍历序列为EFBCGDA。



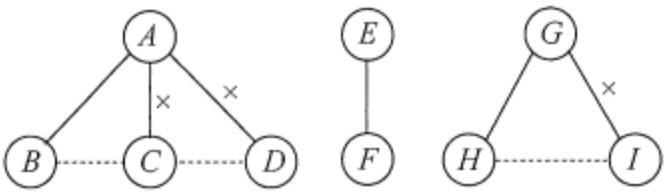
另外，树也有层次遍历，与二叉树的层次遍历思想基本相同，即按层序依次访问各结点。

2、森林的遍历

按照森林和树相互递归的定义，可得到森林的两种遍历方法。

1. 先序遍历森林。若森林为非空，则按如下规则进行遍历:
   ●访问森林中第一棵树的根结点。
   ●先序遍历第一棵树中根结点的子树森林。
   ●先序遍历除去第一棵树之后剩余的树构成的森林。
2. 后序遍历森林。森林为非空时，按如下规则进行遍历:
   ●后序遍历森林中第一棵树的根结点的子树森林。
   ●访问第一棵树的根结点。
   ●后序遍历除去第一棵树之后剩余的树构成的森林。

图5.17的森林的先序遍历序列为ABCDEFGHI，后序遍历序列为BCDAFEHIG。



当森林转换成二叉树时，其第一棵树的子树森林转换成左子树，剩余树的森林转换成右子树，可知森林的先序和后序遍历即为其对应二叉树的先序和中序遍历。

树与二叉树的应用

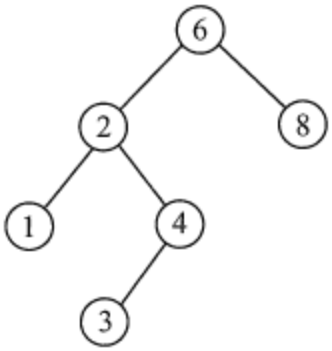
一、二叉排序树

1、定义

二叉排序树(也称二叉查找树)或者是一棵空树，或者是具有下列特性的二叉树:

1. 若左子树非空，则左子树上所有结点的值均小于根结点的值。
2. 若右子树非空，则右子树上所有结点的值均大于根结点的值。
3. 左、右子树也分别是一棵二叉排序树。

根据二叉排序树的定义，左子树结点值<根结点值<右子树结点值，所以对二叉排序树进行中序遍历，可以得到一个递增的有序序列。例如，下图所示二叉排序树的中序遍历序列为123468。



2、二叉排序树的常见操作

构造一个二叉树的结构：

```
/*二叉树的二叉链表结点结构定义*/
typedef struct BiTNode
{
        int data;       //结点数据
        struct BiTNode *lchild, *rchild;        //左右孩子指针
} BiTNode, *BiTree;
```

(1) 查找操作

```
/*
递归查找二叉排序树T中是否存在key
指针f指向T的双亲，其初始调用值为NULL
若查找成功，则指针p指向该数据元素结点，并返回TRUE
否则指针p指向查找路径上访问的最后一个结点并返回FALSE
*/
bool SearchBST(BiTree T, int key, BiTree f, BiTree *p){
        if(!T){
                *p = f;
                return FALSE;
        }else if(key == T->data){
                //查找成功
                *p = T;
                return TRUE;
        }else if(key < T->data){
                return SearchBST(T->lchild, key, T, p); //在左子树继续查找
        }else{
                return SearchBST(T->rchild, key, T, p); //在右子树继续查找
        }
}
```

(2) 插入操作

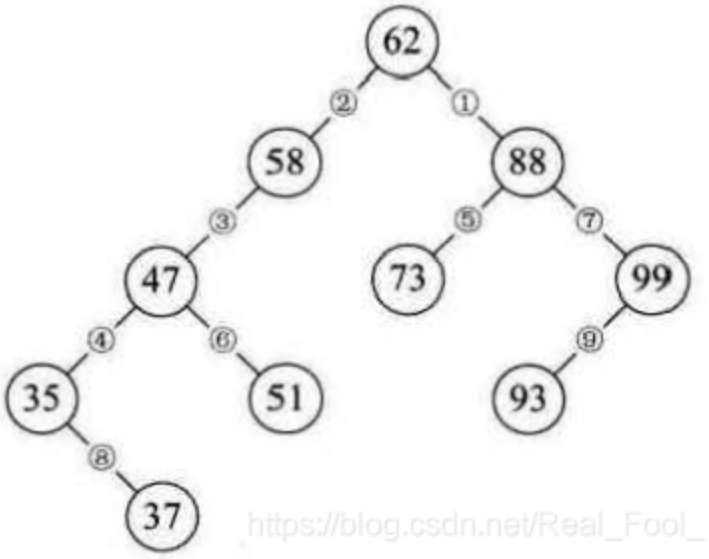
有了二叉排序树的查找函数，那么所谓的二叉排序树的插入，其实也就是将关键字放到树中的合适位置而已。

```
/*
当二叉排序树T中不存在关键字等于key的数据元素时
插入key并返回TRUE，否则返回FALSE
*/
bool InsertBST(BiTree *T, int key){
        BiTree p, s;
        if(!SearchBST(*T, key, NULL, &p)){
                //查找不成功
                s = (BiTree)malloc(sizeof(BiTNode));
                s->data = key;
                s->lchild = s->rchild = NULL;
                if(!p){
                        *T = s; //插入s为新的根节点
                }else if(key < p->data){
                        p->lchild = s;  //插入s为左孩子
                }else{
                        p->rchild = s;  //插入s为右孩子
                }
                return TRUE;
                }else{
                        return FALSE;   //树种已有关键字相同的结点，不再插入
                }
}
```

有了二叉排序树的插入代码，我们要实现二叉排序树的构建就非常容易了，几个例子：

```
int i;
int a[10] = {62, 88, 58, 47, 35, 73, 51, 99, 37, 93};
BiTree T = NULL;
for(i = 0; i<10; i++){
        InsertBST(&T, a[i]);
}
```

上面的代码就可以创建一棵下图这样的树。



（3）删除操作

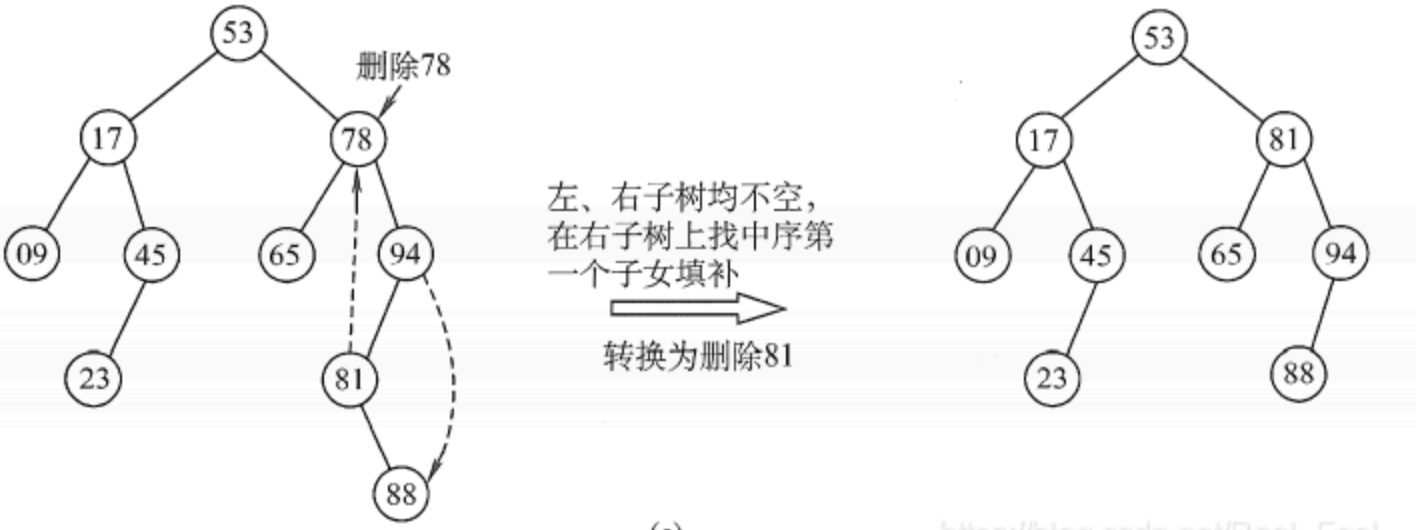
二叉排序树的查找和插入都很简单，但是删除操作就要复杂一些，此时要删除的结点有三种情况：

1. 叶子结点；
2. 仅有左或右子树的结点；
3. 左右子树都有的结点；

前两种情况都很简单，第一种只需删除该结点不需要做其他操作；第二种删除后需让被删除结点的直接后继接替它的位置；**复杂就复杂在第三种，此时我们需要遍历得到被删除结点的直接前驱或者直接后继来接替它的位置，然后再删除。**
第三种情况如下图所示：

(a)

(b)



(c)

代码如下：

```
/*
若二叉排序树T中存在关键字等于key的数据元素时，则删除该数据元素结点，
并返回TRUE;否则返回FALSE
*/
bool DeleteBST(BiTree *T, int key){
        if(!*T){
                return FALSE;
        }else{
                if(key == (*T)->data){
                        //找到关键字等于key的数据元素
                        return Delete(T);
                }else if(key < (*T) -> data){
                        return DeleteBST((*T) -> lchild, key);
                }else{
                        return DeleteBST((*T) -> rchild, key);
                }
        }
}
```

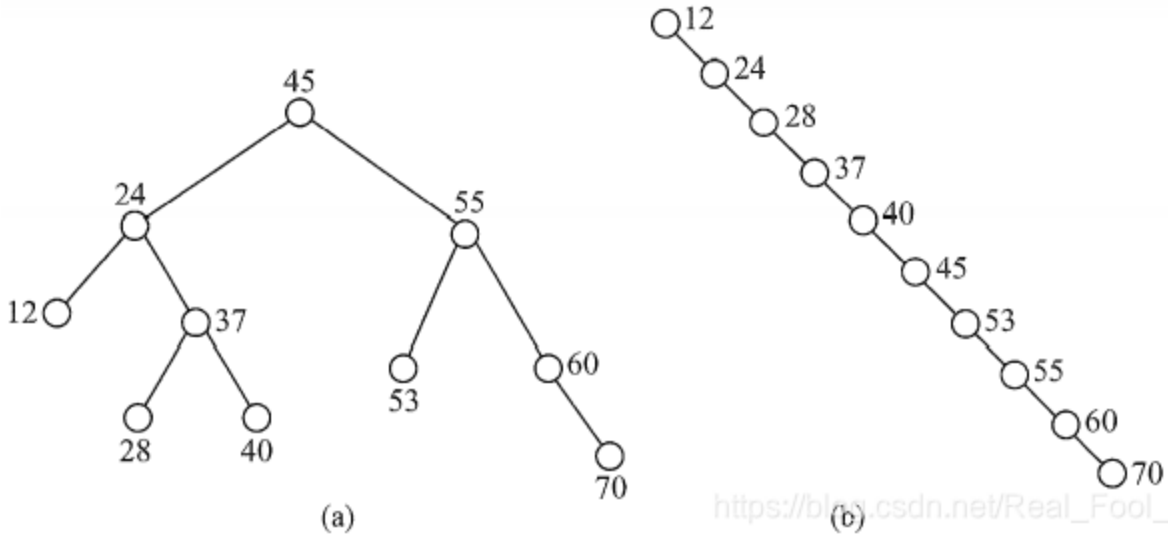
下面是Delete()方法：

```
/*从二叉排序树中删除结点p，并重接它的左或右子树。*/
bool Delete(BiTree *p){
        BiTree q, s;
        if(p->rchild == NULL){
                //右子树为空则只需重接它的左子树
                q = *p;
                *p = (*p)->lchild;
                free(q);
        }else if((*p)->lchild == NULL){
                //左子树为空则只需重接它的右子树
                q = *p;
                *p = (*p)->rchild;
                free(q);
        }else{
                //左右子树均不空
                q = *p;
                s = (*p)->lchild;       //先转左
                while(s->rchild){//然后向右到尽头，找待删结点的前驱
                        q = s;
                        s = s->rchild;
                }
                //此时s指向被删结点的直接前驱，p指向s的父母节点
                p->data = s->data;      //被删除结点的值替换成它的直接前驱的值
                if(q != *p){
                        q->rchild = s->lchild;  //重接q的右子树
                }else{
                        q->lchild = s->lchild;  //重接q的左子树
                }
                pree(s);
        }
        return TRUE;
}
```

3、小结（引申出平衡二叉树）

二叉排序树的优点明显，插入删除的时间性能比较好。而对于二叉排序树的查找，走的就是从根结点到要查找的结点的路径，其比较次数等于给定值的结点在二叉排序树的层数。极端情况，最少为1次，即根结点就是要找的结点，最多也不会超过树的深度。也就是说，**二叉排序树的查找性能取决于二叉排序树的形状**。可问题就在于，二叉排序树的形状是不确定的。
例如 { 62 , 88 , 58 , 47 , 35 , 73 , 51 , 99 , 37 , 93 } \{62,88,58,47,35,73,51,99,37,93\} {62,88,58,47,35,73,51,99,37,93}这样的数组，我们可以构建如下左图的二叉排序树。但如果数组元素的

次序是从小到大有序，如{35,37,47,51,58,62,73,88,93,99},则二叉排序树就成了极端的右斜树，如下面右图的二叉排序树：

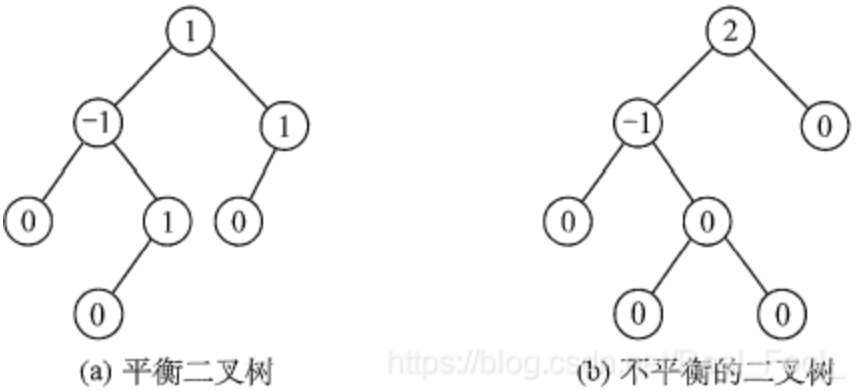


也就是说，我们希望二叉排序树是比较平衡的，即其深度与完全二叉树相同，那么查找的时间复杂也就为 O (l o g n) O(logn) O(logn)，近似于折半查找。
不平衡的最坏情况就是像上面右图的斜树，查找时间复杂度为 O (n) O(n) O(n)，这等同于顺序查找。
因此，如果我们希望对一个集合按二叉排序树查找，最好是把它构建成一棵**平衡的二叉排序树**。

二、平衡二叉树

1、定义

平衡二叉树(Self-Balancing Binary Search Tree 或 Height-Balanced Binary Search Tree)是一种二叉排序树，其中每一个节点的左子树和右子树的高度差至多等于1。
它是一种高度平衡的二叉排序树。它要么是一棵空树， 要么它的左子树和右子树都是平衡二叉树，且左子树和右子树的深度之差的绝对值不超过1。我们**将二叉树上结点的左子树深度减去右子树深度的值称为平衡因子BF (Balance Factor)**， 那么平衡二叉树上所有结点的平衡因子只可能是-1、0和1。只要二叉树上有一个结点的平衡因子的绝对值大于1，则该二叉树就是不平衡的。



2、平衡二叉树的查找

在平衡二叉树上进行查找的过程与二叉排序树的相同。因此，在查找过程中，与给定值进行比较的关键字个数不超过树的深度。假设以 n h n_h nh表示深度为 h h h的平衡树中含有的最少结点数。显然，有 n 0 = 0 , n 1 = 1 , n 2 = 2 n_0=0,n_1=1,n_2=2 n0=0,n1=1,n2=2，并且有 n h = n h − 1 + n h − 2 + 1 n_h=n_{h-1}+n_{h-2}+1 nh=nh−1+nh−2+1。可以证明，含有 n n n个结点的平衡二叉树的最大深度为 O (l o g 2 n) O(log2n) O(log2n)，因此平衡二叉树的平均查找长度为 O (l o g 2 n) O(log2n) O(log2n) 如下图所示。

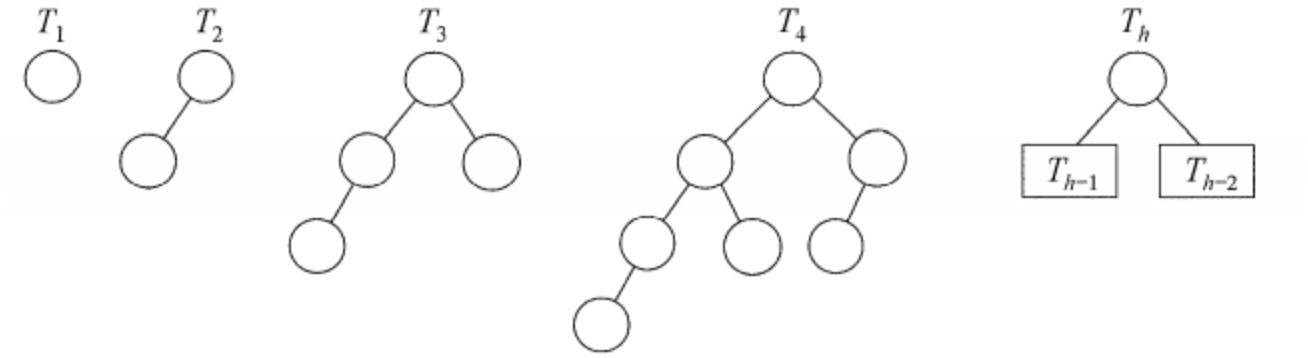
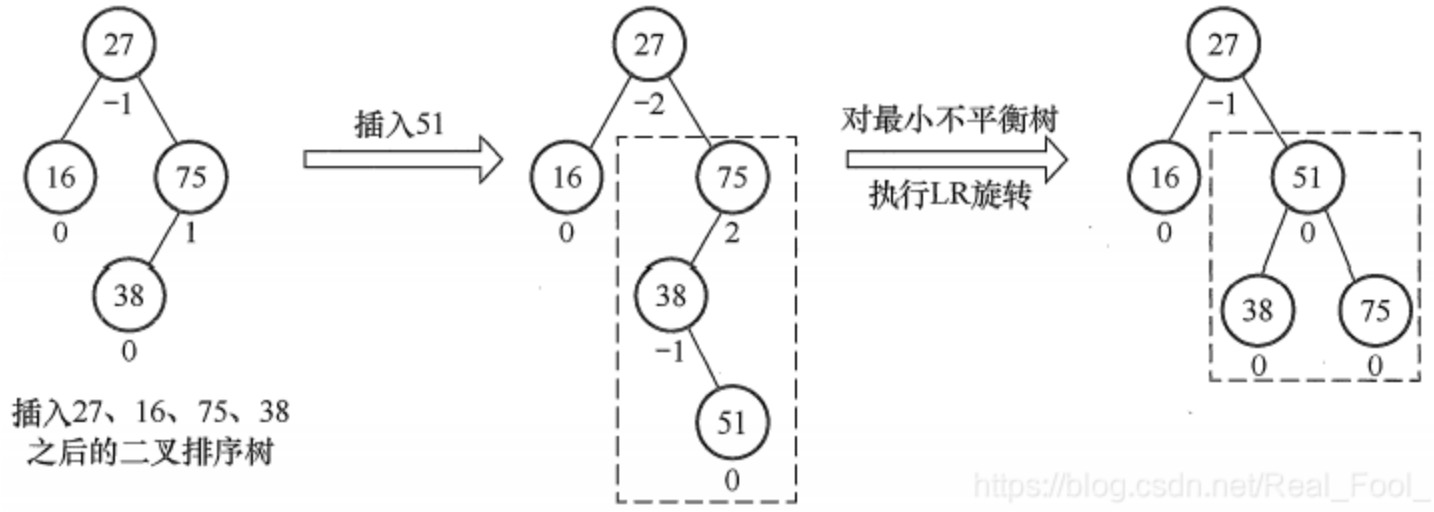


图 5.33　结点个数 n 最少的平衡二叉树

3、平衡二叉树的插入

二叉排序树保证平衡的基本思想如下：每当在二叉排序树中插入(或删除)一个结点时，首先检查其插入路径上的结点是否因为此次操作而导致了不平衡。若导致了不平衡，则先找到插入路径上离插入结点最近的平衡因子的绝对值大于1的结点A，再对以A为根的子树，在保持二叉排序树特性的前提下，调整各结点的位置关系，使之重新达到平衡。

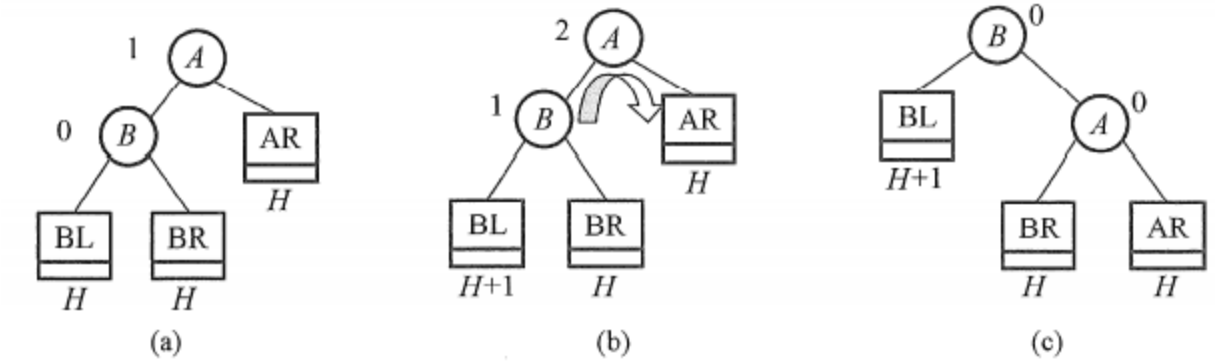
注意:每次调整的对象都是最小不平衡子树，即以插入路径上离插入结点最近的平衡因子的绝对值大于1的结点作为根的子树。下图中的虚线框内为最小不平衡子树。



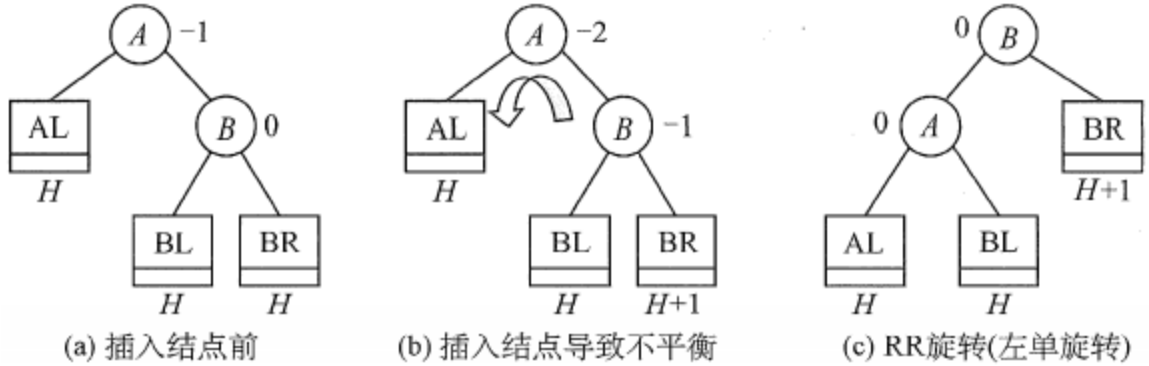
平衡二叉树的插入过程的前半部分与二叉排序树相同，但在新结点插入后，若造成查找路径上的某个结

点不再平衡，则需要做出相应的调整。可将调整的规律归纳为下列4种情况：

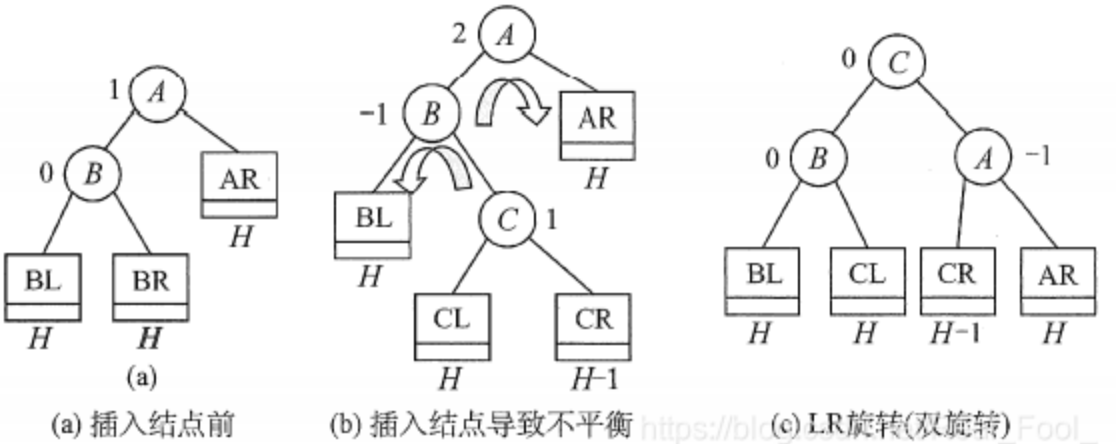
1. **LL平衡旋转(右单旋转)**。由于在结点A的左孩子(L)的左子树(L)上插入了新结点，A的平衡因子由1增至2，导致以A为根的子树失去平衡，需要一次向右的旋转操作。将A的左孩子B向右上旋转代替A成为根结点，将A结点向右下旋转成为B的右子树的根结点，而B的原右子树则作为A结点的左子树。如下图所示，结点旁的数值代表结点的平衡因子，而用方块表示相应结点的子树，下方数值代表该子树的高度。



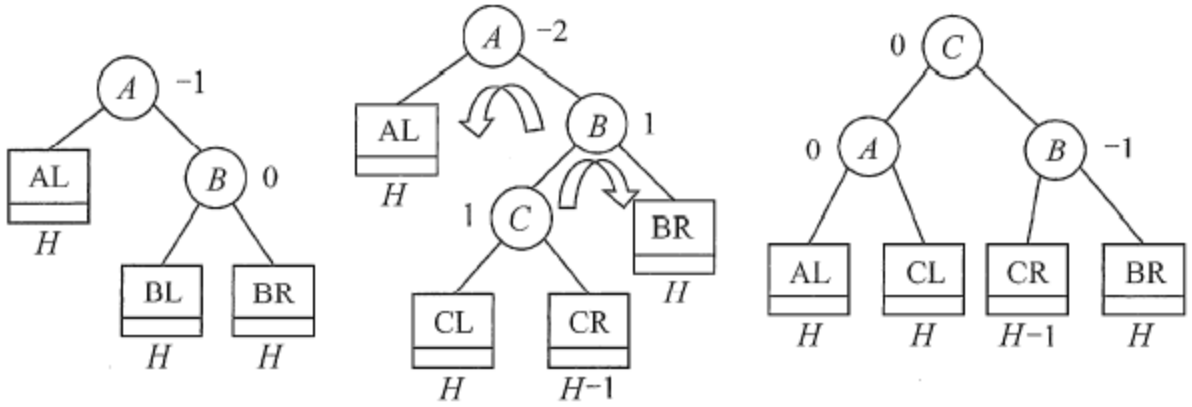
2. **RR平衡旋转(左单旋转)**。由于在结点A的右孩子®的右子树®上插入了 新结点，A的平衡因子由-1减至-2，导致以A为根的子树失去平衡，需要一次向左的旋转操作。将A的右孩子B向左上旋转代替A成为根结点，将A结点向左下旋转成为B的左子树的根结点，而B的原左子树则作为A结点的右子树。



3. **LR平衡旋转(先左后右双旋转)**。由于在A的左孩子(L)的右子树®上插入新结点，A的平衡因子由1增至2，导致以A为根的子树失去平衡，需要进行两次旋转操作，先左旋转后右旋转。先将A结点的左孩子B的右子树的根结点C向左上旋转提升到B结点的位置（即进行一次**RR平衡旋转(左单旋转)**），然后再把该C结点向右上旋转提升到A结点的位置（即进行一次**LL平衡旋转(右单旋转)**）。



4. **RL平衡旋转(先右后左双旋转)**。由于在A的右孩子®的左子树(L)上插入新结点，A的平衡因子由-1减至-2，导致以A为根的子树失去平衡，需要进行两次旋转操作，先右旋转后左旋转。先将A结点的右孩子B的左子树的根结点C向右上旋转提升到B结点的位置（即进行一次**LL平衡旋转(右单旋转)**），然后再把该C结点向左上旋转提升到A结点的位置（即进行一次**RR平衡旋转(左单旋转)**）。

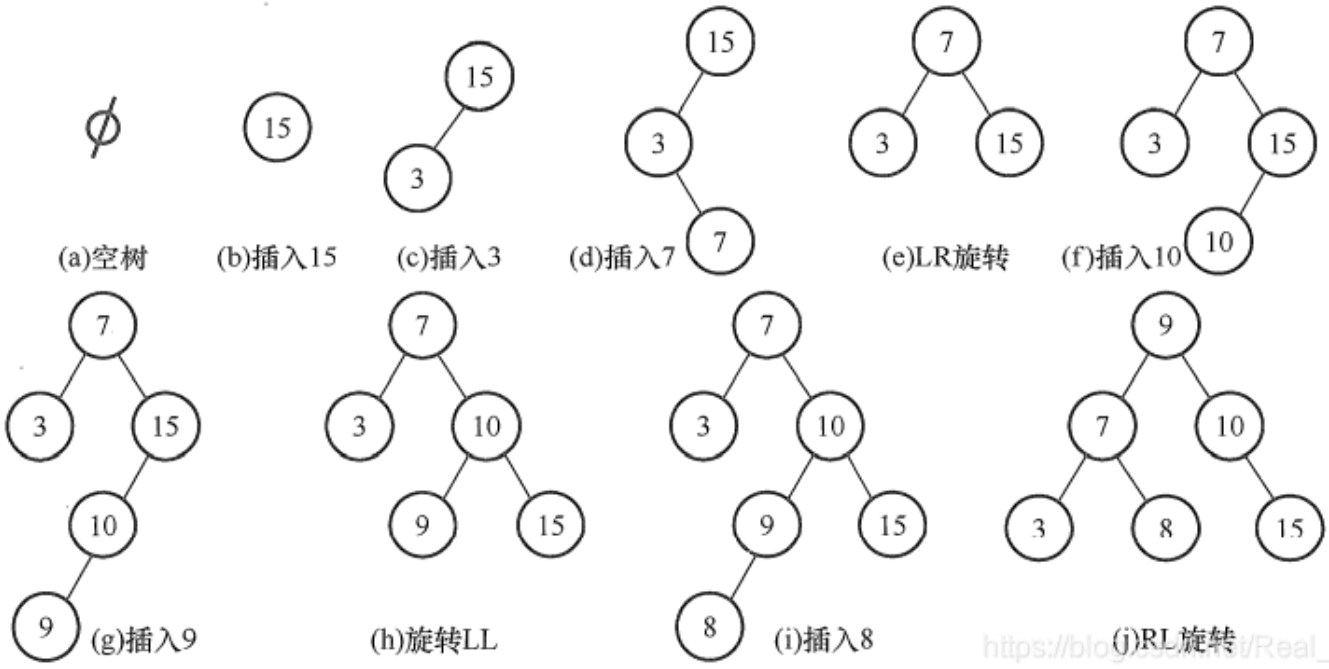


(a) 插入结点前　(b) 插入结点导致不平衡　(c) RL旋转(双旋转)

注意: LR和RL旋转时，新结点究竟是插入C的左子树还是插入C的右子树不影响旋转过程，而上图中是以插入C的左子树中为例。

举个例子:

假设关键字序列为 15 , 3 , 7 , 10 , 9 , 8 {15,3, 7, 10, 9, 8} 15,3,7,10,9,8，通过该序列生成平衡二叉树的过程如下图所示。

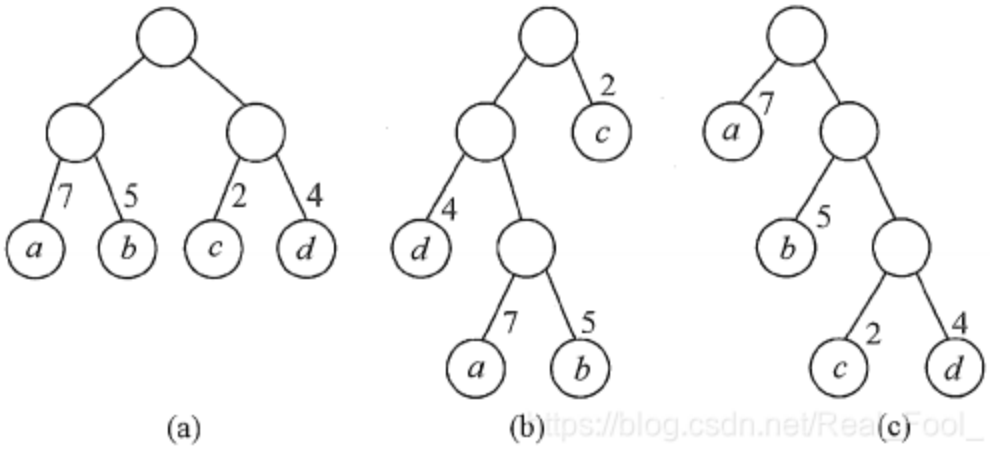


二叉排序树还有另外的平衡算法，如红黑树(Red Black Tree)等，与平衡二叉树(AVL树)相比各有优势。

三、哈夫曼树和哈夫曼编码

1、哈夫曼树的定义和原理

在许多应用中，树中结点常常被赋予一个表示某种意义的数值，称为该结点的**权**。从树的根到任意结点的路径长度(经过的边数)与该结点上权值的乘积，称为该结点的**带权路径长度**。树中所有叶结点的带权路径长度之和称为该树的**带权路径长度**，记为 W P L = ∑ i = 1 n w i l i WPL = \displaystyle\sum_{i=1}^{n} w_il_i WPL=i=1∑nwili式中， w i w_i wi是第i个叶结点所带的权值， l i l_i li是该叶结点到根结点的路径长度。
在含有n个带权叶结点的二叉树中，其中带权路径长度(WPL)最小的二叉树称为**哈夫曼树**，也称**最优二叉树**。例如，下图中的3棵二叉树都有4个叶子结点a, b,c,d,分别带权7,5,2,4，它们的带权路径长度分别为



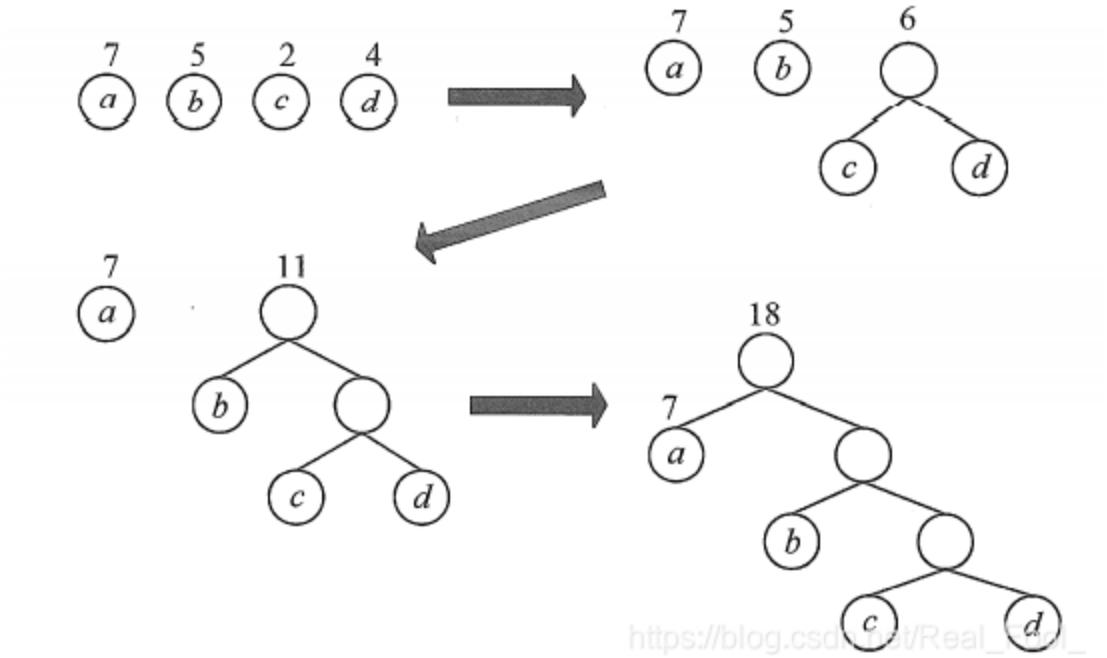
a. WPL = 7x2 + 5x2 + 2x2 + 4x2 = 36。
b. WPL = 4x2 + 7x3 + 5x3 + 2x1 = 46。
c. WPL = 7x1 + 5x2 + 2x3 + 4x3 = 35。
其中，图c树的WPL最小。可以验证，它恰好为哈夫曼树。

2、哈夫曼树的构造

步骤：

1. 先把有权值的叶子结点按照从大到小（从小到大也可以）的顺序排列成一个有序序列。
2. 取最后两个最小权值的结点作为一个新节点的两个子结点，注意相对较小的是左孩子。
3. 用第2步构造的新结点替掉它的两个子节点，插入有序序列中，保持从大到小排列。
4. 重复步骤2到步骤3，直到根节点出现。

看图就清晰了，如下图所示：



3、哈夫曼编码

赫夫曼当前研究这种最优树的目的是为了解决当年远距离通信(主要是电报)的数据传输的最优化问题。

哈夫曼编码是一种被广泛应用而且非常有效的数据压缩编码。

比如我们有一段文字内容为“ BADCADFEED”要网络传输给别人，显然用二进制的数字(0和1)来表示是很自然的想法。我们现在这段文字只有六个字母ABCDEF，那么我们可以用相应的二进制数据表示，如下表所示：

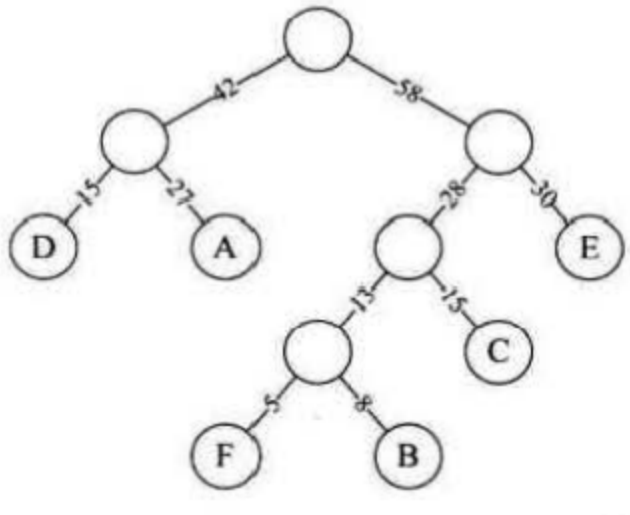
字母	A	B	C	D	E	F
二进制字符	000	001	010	011	100	101

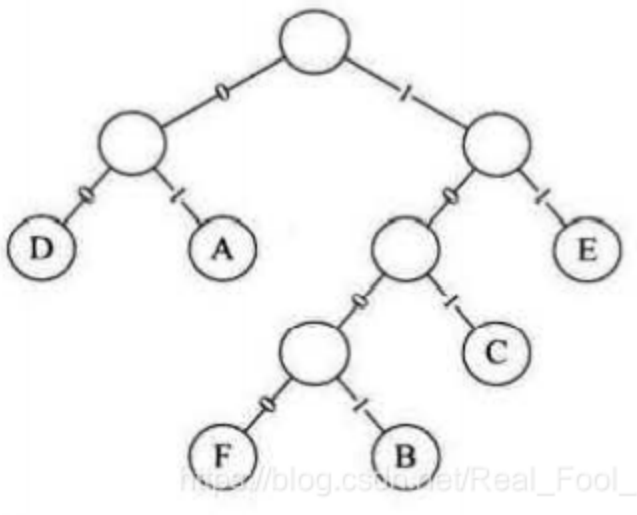
这样按照固定长度编码编码后就是“001000011010000011101100100011”，对方接收时可以按照3位一分来译码。如果一篇文章很长，这样的二进制串也将非常的可怕。而且事实上，不管是英文、中文或是其他语言，字母或汉字的出现频率是不相同的。

假设六个字母的频率为A 27,B 8,C 15,D 15,E 30,F 5,合起来正好是

100%。那就意味着，我们完全可以重新按照赫夫曼树来规划它们。

下图左图为构造赫夫曼树的过程的权值显示。右图为将权值左分支改为0，右分支改为1后的赫夫曼树。





这棵哈夫曼树的WPL为:
W P L = 2 ∗ (15 + 27 + 30) + 3 ∗ 15 + 4 ∗ (5 + 8) = 241 WPL=2*(15+27+30) + 3*15 + 4*(5+8)=241
WPL=2∗(15+27+30)+3∗15+4∗(5+8)=241
此时，我们对这六个字母用其从树根到叶子所经过路径的0或1来编码，可以得到如下表所示这样的定义。

字母	A	B	C	D	E	F
二进制字符	01	1001	101	00	11	1000

若没有一个编码是另一个编码的前缀，则称这样的编码为前缀编码。
我们将文字内容为“ BADCADFEED”再次编码，对比可以看到结果串变小了。

- 原编码二进制串: 000011000011101100100011 (共 30个字符)
- 新编码二进制串: 10100101010111100(共25个字符)

也就是说，我们的数据被压缩了，节约了大约17%的存储或传输成本。

> 注意:
> 0和1究竟是表示左子树还是右子树没有明确规定。左、右孩子结点的顺序是任意的，所以构造出的哈夫曼树并不唯一，但各哈夫曼树的带权路径长度WPL相同且为最优。此外，如有若干权值相同的结点，则构造出的哈夫曼树更可能不同，但WPL必然相同且是最优的。

@title: 树
@date: 2025-01-08 19:00:00
@version: 1.0.0

@author：软件工程宋浩元