`visit函数`用来对当前结点的某种操作

```
void visit(BiTNode * p)//访问当前结点的数据
{
    cout << p->value << " ";
}
```

二叉树的前序遍历

- 递归版

```
void PreOrder(BiTree T)//前序遍历
{
    if(T!=NULL)
    {
        cout << T->value << " ";//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
    }
}
```

- 非递归版

对于非递归遍历可参考一下递归遍历，在递归遍历中，除去`visit()`函数，三种遍历完全一致，所以**访问路径**也应该一致

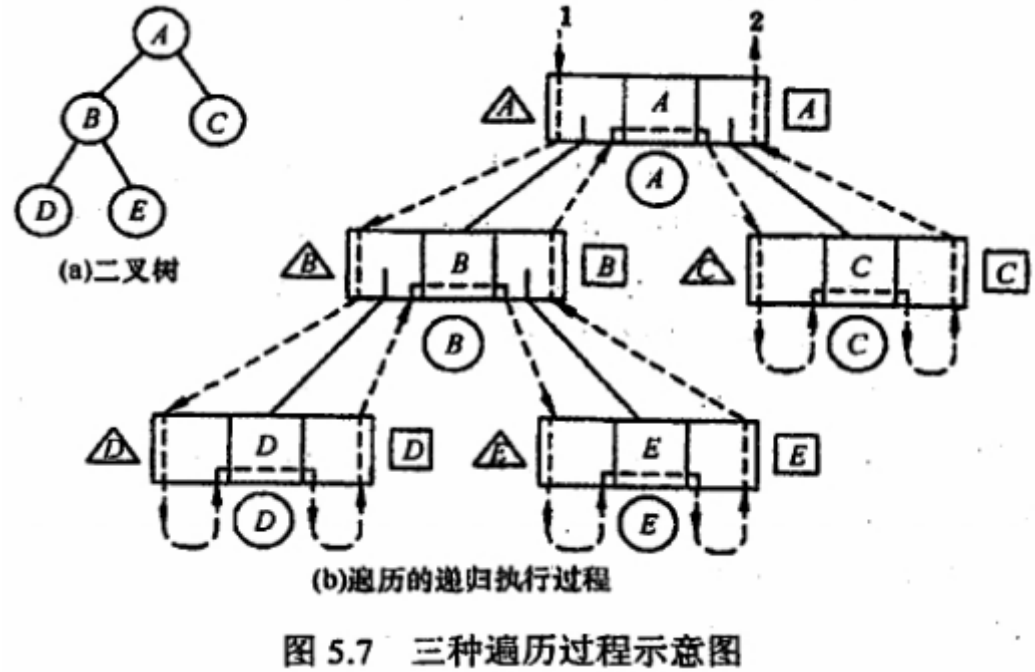


图 5.7　三种遍历过程示意图

可以看到三种遍历的**访问路径完全一致**，只是访问结点时机不同，前序遍历在**第一次到结点时**(从`根结点`出发)就进行访问(**第一次**到达该结点)，中序遍历为**第一次回到结点**(从`左子树`返回)进行访问(**第二次**到达该结点)，后序遍历为**第二次回到结点**(从`右子树`返回)进行访问(**第三次**到达该结点)。叶子结点同样对其左右孩子进行访问。

> 访问结点的时机取决于是哪种遍历(前序遍历、中序遍历、后序遍历)

> 1)**访问结点** P，并将结点 P入栈;
>
> 2)判断结点 P的左孩子是否为空，若为空，则取栈顶结点并进行出栈操作，并将栈顶结点的右孩子置为当前的结点 P，循环至1);若不为空，则将 P的左孩子置为当前的结点 P;
>
> 3)直到 P为 $NULL$并且栈为空，则遍历结束。

```
//C++中STL的栈
while(p!=NULL||s.empty()==0)//p不空或栈不空
{
    if(p!=NULL)//一路向左
    {
        s.push(p);//当前结点入栈
        p=p->lchild;//左孩子不空,一直向左走
    }
    else//出栈,并转向出栈结点的右子树
    {
        p=s.top();//获取栈顶
        s.pop();//栈顶元素出栈

        p=p->rchild;//向右子树走,p赋值为当前出栈元素的右孩子
    }
}
```

故前序遍历的非递归如下:

```
void PreOrder_Non_Recursive(BiTree T)//前序遍历非递归
{
    stack<BiTNode* > s;//此处为了方便使用c++中的stl

    BiTNode *p=T;

    while(p!=NULL||s.empty()==0)//p不空或栈不空
    {
        if(p!=NULL)//一路向左
        {
            visit(p);// ***访问当前结点***
            s.push(p);//当前结点入栈
            p=p->lchild;//左孩子不空,一直向左走
        }
        else//出栈,并转向出栈结点的右子树
        {
            p=s.top();//获取栈顶
            s.pop();//栈顶元素出栈

            p=p->rchild;//向右子树走,p赋值为当前出栈元素的右孩子
        }
    }
}
```

二叉树的中序遍历

- 递归版

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

- 非递归版

> 1)若其左孩子不为空，则将 P入栈并将 P的左孩子置为当前的 P，然后对当前结点 P再进行相同的处理；
>
> 2)若其左孩子为空，则取栈顶元素并进行出栈操作，**访问该栈顶结点**，然后将当前的 P置为栈顶结点的右孩子；
>
> 3)直到 P为 $NULL$并且栈为空则遍历结束。

```
void InOrder_Non_Recursive(BiTree T)//中序遍历非递归
{
    stack<BiTNode* > s;//此处为了方便使用c++中的stl

    BiTNode *p=T;

    while(p!=NULL||s.empty()==0)//p不空或栈不空
    {
        if(p!=NULL)//一路向左
        {
            s.push(p);//当前结点入栈
            p=p->lchild;//左孩子不空,一直向左走
        }
        else//出栈,并转向出栈结点的右子树
        {
            p=s.top();//获取栈顶
            s.pop();//栈顶元素出栈

            visit(p);// ***访问当前结点***
            p=p->rchild;//向右子树走,p赋值为当前出栈元素的右孩子
        }
    }
}
```

二叉树的后序遍历

- 递归版

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

- 非递归版

后序遍历的非递归实现是三种遍历方式中最难的一种。因为在后序遍历中，要保证左孩子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。

第一种思路：对于任一结点 P，将其入栈，然后沿其左子树一直往下搜索，直到搜索到没有左孩子的结点，此时该结点出现在栈顶，但是此时不能将其出栈并访问， 因此其右孩子还未被访问。所以接下来按照相同的规则对其右子树进行相同的处理，当访问完其右孩子时，该结点又出现在栈顶，此时可以将其出栈并访问。这样就保证了正确的访问顺序。可以看出，在这个过程中，每个结点都两次出现在栈顶，只有在第二次出现在栈顶时，才能访问它。因此需要多**设置一个变量标识该结点是否是第一次出现在栈顶。**

```
typedef struct StackNode{
    BiTNode *bitnode;
    bool isFirst;
}StackNode;

void PostOrder_Non_Recursive1(BiTree T)//后序遍历非递归
{
    stack<StackNode* > s;//此处为了方便使用c++中的stl

    BiTNode *p=T;

    while(p!=NULL||s.empty()==0)//p不空或栈不空
    {
        if(p!=NULL)//一路向左
        {
            StackNode *new_node=(StackNode *)malloc(sizeof(StackNode));
            new_node->bitnode=p;
            new_node->isFirst=true;//标记第一次访问

            //s.push(p);//当前结点入栈
            s.push(new_node);

            p=p->lchild;//左孩子不空,一直向左走
        }
        else//出栈,并转向出栈结点的右子树
        {
```

```
            StackNode *temp_node=s.top();//获取栈顶
            s.pop();//栈顶元素出栈

            if(temp_node->isFirst==true)//表示是第一次出现在栈顶(从左子树返回)
            {
                temp_node->isFirst=false;
                s.push(temp_node);
                p=temp_node->bitnode->rchild;//向右子树走,p赋值为当前出栈元素的右孩子
            }
            else//第二次出现在栈顶
            {
                visit(temp_node->bitnode);// ***访问当前结点***
                p=NULL;//结点访问完后,重置p指针
            }
        }
    }
}
```

第二种思路：要保证根结点在左孩子和右孩子访问之后才能访问，因此对于任一结点 P，先将其入栈。如果 P不存在左孩子和右孩子，则可以直接访问它；或者 P存在左孩子或者右孩子，但是其左孩子和右孩子都已被访问过了，则同样可以直接访问该结点。若非上述两种情况，则将 P的右孩子和左孩子依次入栈，这样就保证了每次取栈顶元素的时候，左孩子在右孩子前面被访问，左孩子和右孩子都在根结点前面被访问。用**辅助指针**记录最近访问过的结点。

```
void PostOrder_Non_Recursive2(BiTree T)//后序遍历非递归
{
    stack<BiTNode* > s;//此处为了方便使用c++中的stl

    BiTNode *p=T;
    BiTNode *pre=NULL;

    while(p!=NULL||s.empty()==0)//p不空或栈不空
    {
        if(p!=NULL)//一路向左
        {
            s.push(p);//当前结点入栈
            p=p->lchild;//左孩子不空,一直向左走
        }
        else//向右
        {
            p=s.top();//获取栈顶
            if(p->rchild!=NULL&&p->rchild!=pre)//若右子树存在,且未被访问
                p=p->rchild;//转向右
            else//否则,弹出结点并访问
            {
                s.pop();
                visit(p);// ***访问当前结点***
                pre=p;
                p=NULL;//结点访问完,重置p指针
            }
        }
```

```
    }
}
```

二叉树的层序遍历

```
void LevelOrder(BiTree T)//层次遍历
{
    queue<BiTNode *> q;//此处为了方便使用c++中的stl

    q.push(T);//将根结点入队

    while(q.empty()==0)
    {
        BiTNode *p=q.front();//队头结点出队
        q.pop();
        visit(p);//访问队头元素

        if(p->lchild!=NULL)//左子树不空,将左子树根结点入队
            q.push(p->lchild);
        if(p->rchild!=NULL)//右子树不空,将右子树根结点入队
            q.push(p->rchild);
    }
}
```

由遍历序列构造二叉树

- 由二叉树的`先序序列`和**中序序列**可以唯一确定一棵二叉树.
- 由二叉树的`后序序列`和**中序序列**可以唯一确定一棵二叉树.
- 由二叉树的`层序序列`和**中序序列**可以唯一确定一棵二叉树.

前序、后序、层序序列的两两组合**无法唯一确定**一棵二叉树

线索二叉树

核心

```
ThreadNode *pre=NULL;//指向当前访问结点的前驱

void visit(ThreadNode *q)//访问当前结点的数据
{
    if(q->lchild==NULL)//左子树为空建立前驱线索
    {
        q->lchild=pre;
        q->ltag=1;
    }
    if(pre!=NULL&&pre->rchild==NULL)//建立前驱结点的后继结点
    {
        pre->rchild=q;
        pre->rtag=1;
    }
```

```
    pre=q;
}
```

- 前序线索化

```
void PreThread(ThreadTree T)//前序线索化
{
    if(T!=NULL)
    {
        visit(T);//访问根节点
        if(T->ltag==0)
            PreThread(T->lchild);//递归遍历左子树
        if(T->rtag==0) //注意王道此处未加,实际应加上
            PreThread(T->rchild);//递归遍历右子树
    }
}

void CreatePreThread(ThreadTree T)
{
    pre=NULL;//pre初始为NULL
    if(T!=NULL)//非空二叉树才能线索化
    {
        PreThread(T);//前序线索化二叉树
        if(pre->rchild==NULL)//处理遍历的最后一个结点(此处if可省略)
            pre->rtag=1;
    }
}
```

- 中序线索化

```
void InThread(ThreadTree T)//中序线索化
{
    if(T!=NULL)
    {
        InThread(T->lchild);//递归遍历左子树
        visit(T);//访问根节点
        InThread(T->rchild);//递归遍历右子树
    }
}

void CreateInThread(ThreadTree T)
{
    pre=NULL;//pre初始为NULL
    if(T!=NULL)//非空二叉树才能线索化
    {
        InThread(T);//中序线索化二叉树
        if(pre->rchild==NULL)//处理遍历的最后一个结点(此处if可省略)
            pre->rtag=1;
    }
}
```

- 后序线索化

```
void PostThread(ThreadTree T)//后序线索化
{
    if(T!=NULL)
    {
        PostThread(T->lchild);//递归遍历左子树
        PostThread(T->rchild);//递归遍历右子树
        visit(T);//访问根节点
    }
}

void CreatePostThread(ThreadTree T)
{
    pre=NULL;//pre初始为NULL
    if(T!=NULL)//非空二叉树才能线索化
    {
        PostThread(T);//后序线索化二叉树
        if(pre->rchild==NULL)//处理遍历的最后一个结点(此处if可省略)
            pre->rtag=1;
    }
}
```

在线索二叉树中找前驱后继

- 中序线索二叉树找**中序后继**并实现中序遍历

> 1.若 $p \rightarrow rtag = 1$,则 $next = p \rightarrow rchild$(有后继线索);
>
> 2.若 $p \rightarrow rtag = 0$,则 $next = p$的**右子树中最左下结点**.

```
//找到以p为根的子树中,第一个被中序遍历的结点
ThreadNode* Firstnode(ThreadNode *p)
{
    //循环找到最左下结点(不一定是叶节点)
    while(p->ltag==0)
        p=p->lchild;
    return p;
}

//在中序线索二叉树中找到结点p的后继结点
ThreadNode* Nextnode(ThreadNode *p)
{
    if(p->rtag==0)//右子树中最左下结点
        return Firstnode(p->rchild);
    else return p->rchild;//rtag==1直接返回后继线索
}


void Inorder(ThreadNode* T)//利用线索二叉树实现中序遍历
{
    for(ThreadNode *p=Firstnode(T);p!=NULL;p=Nextnode(p))
```

```
        cout << p->value << " ";
}
```

- 中序线索二叉树找**中序前驱**并实现逆向中序遍历

> 1.若 $p \rightarrow ltag = 1$,则 $pre = p \rightarrow lchild$(有前驱线索);
>
> 2.若 $p \rightarrow ltag = 0$,则 $pre = p$的**左子树中最右下结点**.

```
//找到以p为根的子树中,最后一个被中序遍历的结点
ThreadNode* Lastnode(ThreadNode *p)
{
    //循环找到最右下结点(不一定是叶节点)
    while(p->rtag==0)
        p=p->rchild;
    return p;
}

//在中序线索二叉树中找到结点p的前驱结点
ThreadNode* Prenode(ThreadNode *p)
{
    if(p->ltag==0)//左子树中最右下结点
        return Lastnode(p->lchild);
    else return p->lchild;//ltag==1直接返回前驱线索
}

void RevInorder(ThreadNode* T)//利用线索二叉树实现逆向中序遍历
{
    for(ThreadNode *p=Lastnode(T);p!=NULL;p=Prenode(p))
        cout << p->value << " ";
}
```

- 先序线索二叉树找先序后继

> 1.若 $p \rightarrow rtag = 1$,则 $next = p \rightarrow rchild$(有后继线索);
>
> 2.若 $p \rightarrow rtag = 0$,则:
>
> 当 p有左孩子,则先序后继为**左孩子**;当 p没有左孩子,则先序后继为**右孩子**.

```
//在先序线索二叉树中找到结点p的后继结点
ThreadNode* Nextnode(ThreadNode *p)
{
    if(p->rtag==0)
    {
        if(p->lchild!=NULL&&p->ltag==0)
            return p->lchild;
        else if(p->rchild!=NULL&&p->rtag==0)
            return p->rchild;
        else return NULL;
    }
    else return p->rchild;//rtag==1直接返回后继线索
}
```

```
14
15
16  void Preorder(ThreadNode* T)//利用线索二叉树实现先序遍历
17  {
18      for(ThreadNode *p=T;p!=NULL;p=Nextnode(p))
19          cout << p->value << " ";
20  }
```

- 先序线索二叉树找先序前驱(**无法实现**,`无指向父结点的指针`的条件下,由于左右子树中的结点只能是根的后继,不可能是前驱)

基于**三叉链表**可实现先序前驱的查找

> 1.如果能找到 p的父结点,且 p是左孩子,此时 p的父结点是 p的前驱;
>
> 2.如果能找到 p的父结点,且 p是右孩子,其左兄弟为空,此时 p的父结点是 p的前驱;
>
> 3.如果能找到 p的父结点,且 p是右孩子,其左兄弟非空,此时 p的前驱为**左兄弟子树中最后一个被先序遍历的结点**;
>
> 4.如果 p是根结点,则 p无先序前驱.

- 后序线索二叉树找后序前驱

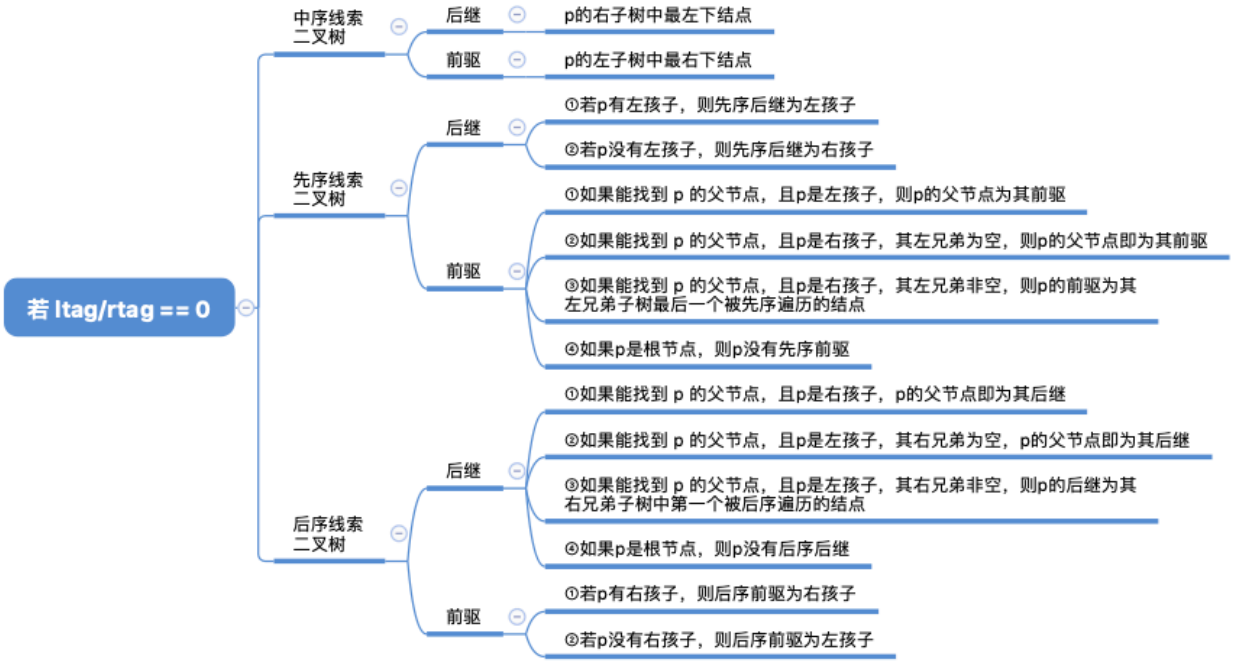
> 1.若 $p \rightarrow ltag = 1$,则 $pre = p \rightarrow lchild$;
>
> 2.若 $p \rightarrow ltag = 0$,则:
>
> 当 p有右孩子,则后序前驱为右孩子;当 p没有右孩子,则后序前驱为左孩子.

```
1   //在后序线索二叉树中找到结点p的后继结点
2   ThreadNode* Prenode(ThreadNode *p)
3   {
4       if(p->ltag==0)
5       {
6           if(p->rchild!=NULL&&p->rtag==0)
7               return p->rchild;
8           else if(p->lchild!=NULL&&p->ltag==0)
9               return p->lchild;
10          else return NULL;
11      }
12      else return p->lchild;//ltag==1直接返回前驱线索
13  }
14
15  void RevInorder(ThreadNode* T)//利用线索二叉树实现逆向后序遍历
16  {
17      for(ThreadNode *p=T;p!=NULL;p=Prenode(p))
18          cout << p->value << " ";
19  }
```

- 后序线索二叉树找后序后继(**无法实现**,`无指向父结点的指针`的条件下,左右子树中的结点只可能是根的前驱,不可能是后继)

基于**三叉链表**可实现后序后继的查找

> 1.如果能找到 p的父结点,且 p是右孩子,此时 p的父结点是 p的后继;
>
> 2.如果能找到 p的父结点,且 p是左孩子,其右兄弟为空,此时 p的父结点是 p的后继;
>
> 3.如果能找到 p的父结点,且 p是左孩子,其右兄弟非空,此时 p的前驱为**右兄弟子树中第一个被后序遍历的结点**;
>
> 4.如果 p是根结点,则 p无后序后继.



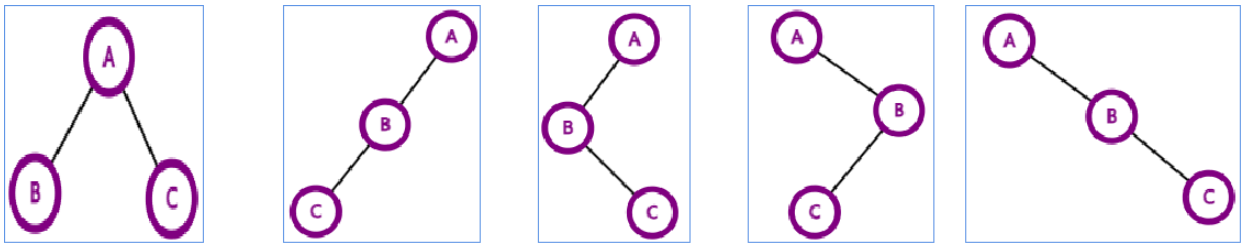
	中序线索二叉树	先序线索二叉树	后序线索二叉树
找前驱	✓	×	✓
找后继	✓	✓	×

除非用三叉链表，或者用土办法从根开始遍历寻找

> 前序为 A, B, C,后序为 C, B, A的二叉树共有().
>
> A. 1棵
>
> B. 2棵
>
> C. 3棵
>
> D. 4棵

当树的结点较少时,可通过枚举的方式进行判断.

前序为 A, B, C的二叉树有 5种结果(符合**卡特兰数** $\frac{C_{2n}^{n}}{n+1}$)



- 在第一个二叉树中,后序遍历为 BCA,不满足要求;
- 在第二个二叉树中,后序遍历为 CBA,满足要求;
- 在第三个二叉树中,后序遍历为 CBA,满足要求;
- 在第四个二叉树中,后序遍历为 CBA,满足要求;
- 在第五个二叉树中,后序遍历为 CBA,满足要求;

共有 4个满足条件的二叉树

> **扩展**:一棵非空二叉树的先序遍历序列与后序遍历序列**正好相反**,则二叉树一定满足
>
> - 只有一个叶结点;
> - 高度等于结点数;
> - 每个结点的左子树`或者`右子树为空.
>
> 若先序遍历序列与后序遍历序列**正好相同**,则只有可能为`只有根节点`

> 一棵二叉树的前序遍历序列为 1234567,它的中序遍历序列可能是()
>
> A. 3124567
>
> B. 1234567
>
> C. 4135627
>
> D. 1463572

> **方法**:
>
> - 硬算，把每个答案都代入一遍，如果出现矛盾的话，结果不正确。
> - 转化成入栈出栈问题。
>
> 1.一棵二叉树的前序遍历结果，就是`前序遍历`时候元素**入栈顺序。**
>
> 2.一颗二叉树的中序、后序遍历的结果，就是`中序遍历`、`后序遍历`遍历时候**元素出栈顺序。**

- 在选项 A中,当第一个元素 3出栈时, $1, 2$已在栈中,而当第二个元素 1出栈,而栈顶元素是 2,不满足条件;
- 在选项 B中,依次入栈一个元素,出栈一个元素,满足条件;
- 在选项 C中,当第一个元素 4出栈时, $1, 2, 3$已在栈中,而当第二个元素 1出栈,而栈顶元素是 3,不满足条件;
- 在选项 D中, 1入栈出栈,第二个元素 4出栈时, $2, 3$已在栈中,第三个元素 6出栈时, $2, 3, 5$已在栈中第四个元素 3出栈时,栈顶元素是 5,不满足条件.

> 已知一棵二叉树的后序序列为 $DABEC$,中序序列为 $DEBAC$,则先序序列为().

A. $ACBED$

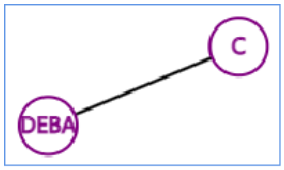
B. $DECAB$

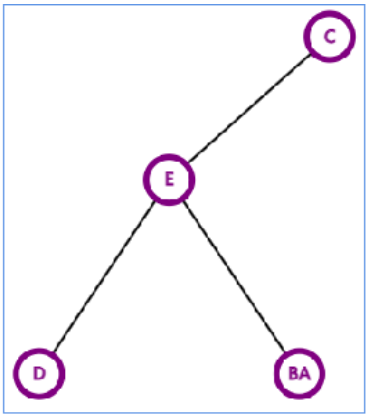
C. $DEABC$

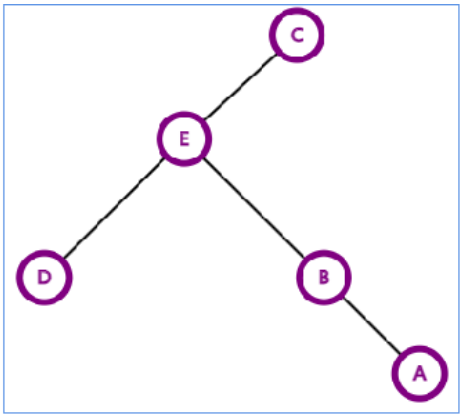
D. $CEDBA$

后序遍历是 `左右根` 的顺序,故从后序序列的最右端开始

- 从后序的倒数第一个元素 C开始,在中序遍历中, $DEBA$在 C的左子树, C的右子树为空;
- 从后序的倒数第二个元素 E,在中序遍历中, D在 E的左子树, BA在 E的右子树;(由于 D已安排好,故 D不再考虑)
- 从后序的倒数第三个元素 B,在中序遍历中, B的左子树为空, A在 B的右子树(由于 A已安排好,故 A不再考虑).







故前序序列为 $CEDBA$

已知一棵二叉树的先序遍历结果为 $ABCDEF$,中序遍历为 $CBAEDF$,则后序遍历的结果为()

A. $CBEFDA$

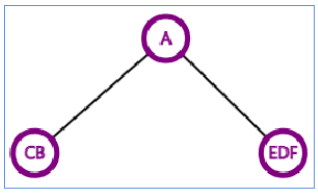
B. $FEDCBA$

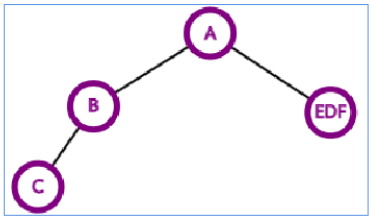
C. $CBEDFA$

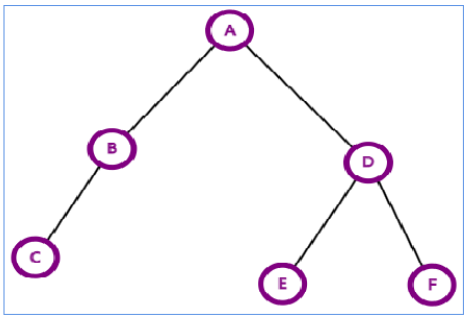
D. 不确定

先序遍历是 `根左右` 的顺序,故从先序序列的最左端开始

- 从先序的第一个元素 A开始,在中序遍历中, CB在 A的左子树, EDF在 A的右子树;
- 从先序的第二个元素 B开始,在中序遍历中, C在 B的左子树, B的右子树为空;(由于 C已安排好,故 C不再考虑)
- 从先序的第四个元素 D开始,在中序遍历中, E在 D的左子树, F在 D的右子树(由于 E, F已安排好,故 E, F不再考虑).





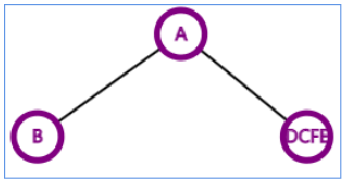


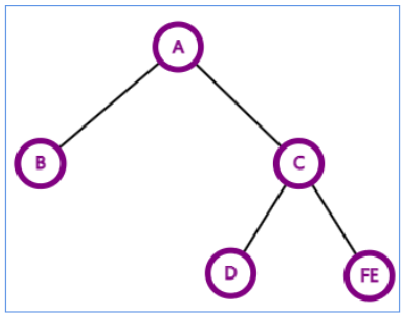
故后序序列为 $CBEFDA$

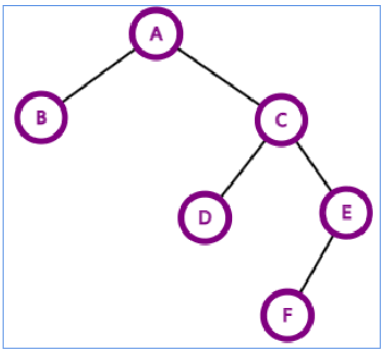
> 已知一棵二叉树的层序遍历结果为 $ABCDEF$,中序遍历为 $BADCFE$,则后序遍历的结果为()
>
> A. $ACBEDF$
>
> B. $ABCDEF$
>
> C. $BDFECA$
>
> D. $FCEDBA$

层序遍历和先序遍历类似,也是从头开始

- 从层序的第一个元素 A开始,在中序遍历中, B在 A的左子树, $DCFE$在 A的右子树(由于 B已安排好,故 B不再考虑);
- 从层序的第三个元素 C开始,在中序遍历中, D在 C的左子树, FE在 C的右子树;(由于 D已安排好,故 D不再考虑);
- 从层序的第五个元素 E开始,在中序遍历中, F在 E的左子树, E的右子树为空(由于 F已安排好,故 F不再考虑).

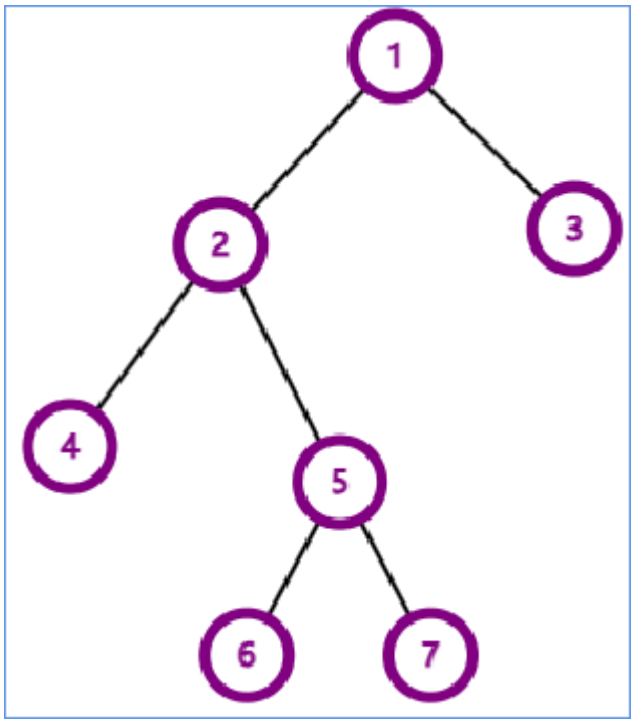






故先序序列为 $ABCDEF$

> **2009统考真题**：给定二叉树如图所示。设 N代表二叉树的根， L代表根结点的左子树， R代表根结点的右子树。若遍历后的结点序列是 3175624,则其遍历方式是().



A. LRN

B. NRL

C. RLN

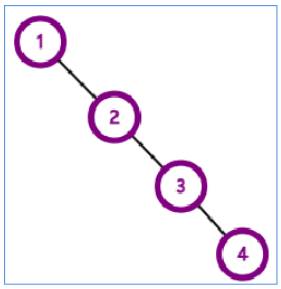
D. RNL

- 选项 A, 按照`左右根`(后序遍历),后序序列为 4675231,不满足题中序列 3175624;
- 选项 B,按照`根右左`,序列为 1325764,不满足题中序列 3175624;
- 选项 C,按照`右左根`,序列为 3765421,不满足题中序列 3175624;
- 选项 D,按照`右根左`,序列为 3175624,满足题中序列 3175624;

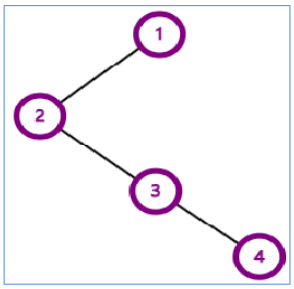
> **2011统考真题**：若一棵二叉树的前序遍历序列和后序遍历序列分别为 $1,2,3,4$和 $4,3,2,1$,则该二叉树的中序遍历序列不会是().
>
> A. $1,2,3,4$
>
> B. $2,3,4,1$
>
> C. $3,2,4,1$
>
> D. $4,3,2,1$

可先将中序遍历和前序(或后序)遍历**构成**一棵二叉树,再用对应的后序(或前序)去**验证**二叉树是否满足要求.

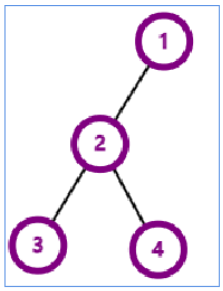
下图是根据前序和中序分别构造的二叉树



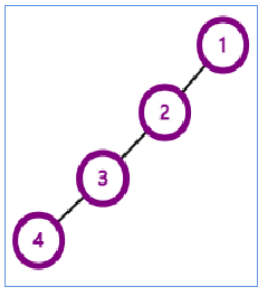
A



B



C



D

- 选项 A,用后序遍历去验证满足条件;
- 选项 B,用后序遍历去验证满足条件;
- 选项 C,后序序列为 $3,4,2,1$,不满足条件;
- 选项 D,用后序遍历去验证满足条件;

此题另可发现先序遍历序列与后序遍历序列**正好相反**,此时必然满足只有一个叶节点,故 C不满足条件.

> **2012统考真题**：若一棵二叉树的前序遍历序列为 a,e,b,d,c,后序遍历序列为 b,c,d,e,a,则根结点的孩子结点(A).
>
> A. 只有 e
>
> B. 有 e,b
>
> C. 有 e,c
>
> D. 无法确定

结论:前序序列和后序序列不能唯一确定一棵二叉树,但可以确定二叉树中结点的祖先关系:当两个结点的前序序列为 XY,后序序列为 YX,则 X为 Y的祖先。

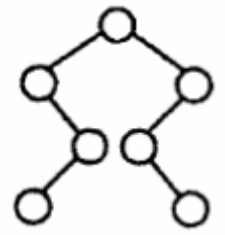
故 前序序列 a,e和后序序列 e,a,可得 a是 e的祖先, e是 a的孩子结点;同理前序序列 d,c和后序序列 c,d,可得 d是 c的祖先;而前序序列 e,b,d,c和后序序列 b,c,d,e中, e是 b,c,d的祖先.

综上, e是根结点 a的孩子结点.

> **2015统考真题**：先序序列为 a,b,c,d的不同二叉树的个数是()
>
> A. 13
>
> B. 14
>
> C. 15
>
> D. 16

根据二叉树前序遍历和中序遍历的递归算法中递归工作栈的状态变化得出：前序序列和中序序列的关系相当于以**前序序列**为`入栈次序`，以**中序序列**为`出栈次序`。因为前序序列和中序序列可以唯一地确定一棵二叉树，所以题意相当于`以序列a,b,c,d为入栈次序，则出栈序列的个数为`,对于 n个不同元素进 栈，出栈序列的个数为 $\frac{C_{2n}^{n}}{n+1}$(**卡特兰数**),故结果为 $\frac{C_{8}^{4}}{5}=\frac{8\times7\times6\times5}{5\times4\times3\times2}=14$

> **2017统考真题**：某二叉树的树形如右图所示，其后序序列为 e,a,c,b,d,g,f,树中与结点 a同层的结点是()



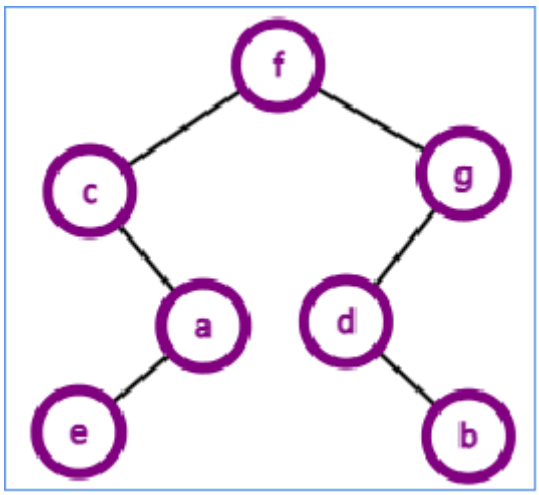
A. c

B. d

C. f

D. g

- 由后序遍历(**左右根**)和图知:根结点为 f,左子树有 3个结点 e, a, c,右子树有 3个结点 b, d, g;
- 同理, f的左子树的根结点为 c, c的右子树的根结点为 a, a的左子树的根结点为 e;
- 同理, f的右子树的根结点为 g, g的左子树的根结点为 d, d的右子树的根结点为 b.



由图知,树中与结点 a同层的结点是 d

2022统考真题：若结点 p与 q在二叉树 T的中序遍历序列中相邻，且 p在 q之前，则下列 p与 q的关系中，不可能的是(B)

Ⅰ. q是 p的双亲

Ⅱ. q是 p的右孩子

Ⅲ. q是 p的右兄弟

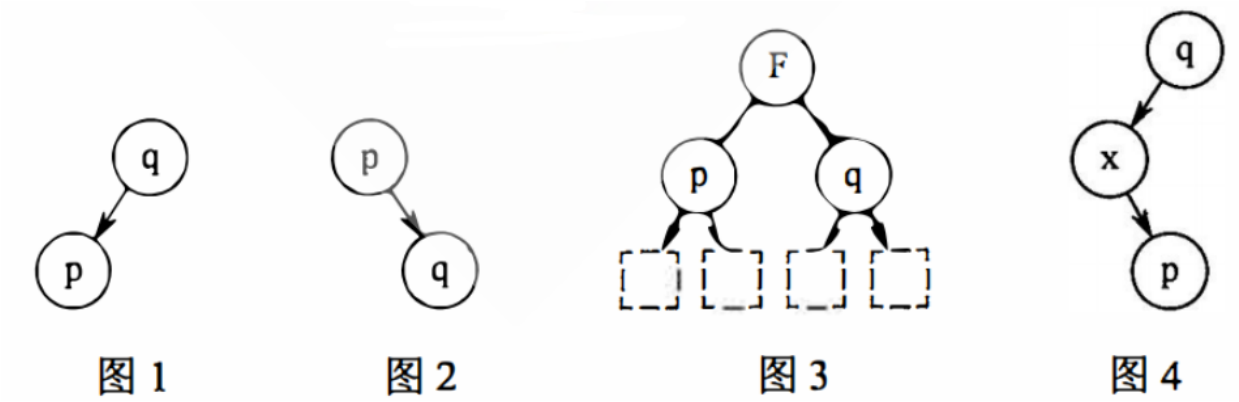
Ⅳ. q是 p的双亲的双亲

A. 仅 Ⅰ

B. 仅 Ⅲ

C. 仅 Ⅱ、 Ⅲ

D. 仅 Ⅱ、 Ⅳ



- 图 1中, q是 p的双亲,中序序列为`{p,q}`,满足条件, I 可能;
- 图 2中, q是 p的右孩子,中序序列为`{p,q}`,满足条件,II可能;
- 图 3中, q是 p的右兄弟,一定先访问 p,再访问 F,最后访问 q,即 p和 q不可能相邻出现,不满足条件,III不可能;
- 图 4中, q是 p的双亲的双亲,中序序列为`{x,p,q}`,满足条件,IV可能;

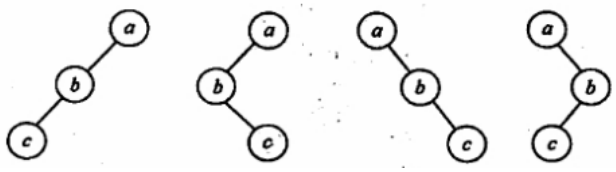
5.3.1 & 5.3.2

> 若某非空二叉树的先序序列和后序序列**正好相反**，则该二叉树的形态是什么?
>
> 若某非空二叉树的先序序列和后序序列**正好相同**，则该二叉树的形态是什么?

可参考前文的**扩展**:一棵非空二叉树的先序遍历序列与后序遍历序列**正好相反**,则二叉树一定满足

- 只有一个叶结点;
- 高度等于结点数;
- 每个结点的左子树`或者`右子树为空.

若先序遍历序列与后序遍历序列**正好相同**,则只有可能为`只有根节点`

> - **相反**
>
> **二叉树的先序序列是 NLR，后序序列是 LRN。要使 NLR = NRL（后序序列反序）成立，L 或 R 应为空，这样的二叉树每层只有一个结点，即二叉树的形态是其高度等于结点个数。以 3 个结点 a, b, c 为例，其形态如下图所示。**
>
> 
>
>
> - **相同**
>
> **二叉树的先序序列是 NLR，后序序列是 LRN。要使 NLR = LRN 成立，L 和 R 应均为空，所以满足条件的二叉树只有一个根结点。**

5.3.3

> 编写后序遍历二叉树的非递归算法。

后序遍历的非递归实现是三种遍历方式中最难的一种。因为在后序遍历中，要保证左孩子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。

第一种思路：对于任一结点 P，将其入栈，然后沿其左子树一直往下搜索，直到搜索到没有左孩子的结点，此时该结点出现在栈顶，但是此时不能将其出栈并访问， 因此其右孩子还未被访问。所以接下来按照相同的规则对其右子树进行相同的处理，当访问完其右孩子时，该结点又出现在栈顶，此时可以将其出栈并访问。这样就保证了正确的访问顺序。可以看出，在这个过程中，每个结点都两次出现在栈顶，只有在第二次出现在栈顶时，才能访问它。因此需要多**设置一个变量标识该结点是否是第一次出现在栈顶。**

```
typedef struct StackNode{
    BiTNode *bitnode;
    bool isFirst;
}StackNode;

void PostOrder_Non_Recursive1(BiTree T)//后序遍历非递归
{
    stack<StackNode* > s;//此处为了方便使用c++中的stl

    BiTNode *p=T;

    while(p!=NULL||s.empty()==0)//p不空或栈不空
    {
        if(p!=NULL)//一路向左
        {
            StackNode *new_node=(StackNode *)malloc(sizeof(StackNode));
            new_node->bitnode=p;
            new_node->isFirst=true;//标记第一次访问

            //s.push(p);//当前结点入栈
            s.push(new_node);

            p=p->lchild;//左孩子不空,一直向左走
        }
        else//出栈,并转向出栈结点的右子树
        {
            StackNode *temp_node=s.top();//获取栈顶
            s.pop();//栈顶元素出栈

            if(temp_node->isFirst==true)//表示是第一次出现在栈顶(从左子树返回)
            {
                temp_node->isFirst=false;
                s.push(temp_node);
                p=temp_node->bitnode->rchild;//向右子树走,p赋值为当前出栈元素的右孩子
            }
            else//第二次出现在栈顶
            {
                visit(temp_node->bitnode);// ***访问当前结点***
                p=NULL;//结点访问完后,重置p指针
```

```
            }
        }
    }
}
```

第二种思路：要保证根结点在左孩子和右孩子访问之后才能访问，因此对于任一结点 P，先将其入栈。如果 P不存在左孩子和右孩子，则可以直接访问它；或者 P存在左孩子或者右孩子，但是其左孩子和右孩子都已被访问过了，则同样可以直接访问该结点。若非上述两种情况，则将 P的右孩子和左孩子依次入栈，这样就保证了每次取栈顶元素的时候，左孩子在右孩子前面被访问，左孩子和右孩子都在根结点前面被访问。用**辅助指针**记录最近访问过的结点。

```
void PostOrder_Non_Recursive2(BiTree T)//后序遍历非递归
{
    stack<BiTNode* > s;//此处为了方便使用c++中的stl

    BiTNode *p=T;
    BiTNode *pre=NULL;

    while(p!=NULL||s.empty()==0)//p不空或栈不空
    {
        if(p!=NULL)//一路向左
        {
            s.push(p);//当前结点入栈
            p=p->lchild;//左孩子不空,一直向左走
        }
        else//向右
        {
            p=s.top();//获取栈顶
            if(p->rchild!=NULL&&p->rchild!=pre)//若右子树存在,且未被访问
                p=p->rchild;//转向右
            else//否则,弹出结点并访问
            {
                s.pop();
                visit(p);// ***访问当前结点***
                pre=p;
                p=NULL;//结点访问完,重置p指针
            }
        }
    }
}
```

5.3.4

> 试给出二叉树的自下而上、从右到左的层次遍历算法。

用正常的**层次遍历算法**，唯一区别就是变为逆向输出，很容易联想到**栈**，可以将输出语句改为压栈语句，等层次遍历完之后再输出栈元素，即为结果。

```
void visit(BiTNode * p)//访问当前结点的数据
{
    cout << p->value << " ";
```

```
}

void LevelOrder(BiTree T)//层次遍历
{
    queue<BiTNode *> q;//此处为了方便使用c++中的stl

    q.push(T);//将根结点入队

    while(q.empty()==0)
    {
        BiTNode *p=q.front();//队头结点出队
        q.pop();
        visit(p);//访问队头元素

        if(p->lchild!=NULL)//左子树不空,将左子树根结点入队
            q.push(p->lchild);
        if(p->rchild!=NULL)//右子树不空,将右子树根结点入队
            q.push(p->rchild);
    }
}

void Reverse_LevelOrder(BiTree T)
{
    queue<BiTNode *> q;//此处为了方便使用c++中的stl
    stack<BiTNode *> sta;

    q.push(T);//将根结点入队

    while(q.empty()==0)
    {
        BiTNode *p=q.front();//队头结点出队

        sta.push(p);//入栈
        q.pop();

        if(p->lchild!=NULL)//左子树不空,将左子树根结点入队
            q.push(p->lchild);
        if(p->rchild!=NULL)//右子树不空,将右子树根结点入队
            q.push(p->rchild);
    }

    while(sta.size()>0)
    {
        BiTNode *p=sta.top();
        visit(p);//访问栈顶元素
        sta.pop();
    }
}
```

5.3.5

> 假设二叉树采用二叉链表存储结构，设计一个非递归算法求二叉树的高度。

- 递归做法(非题目要求)

```
int treeDepth(BiTree T)
{
    if(T==NULL)
        return 0;
    else
    {
        int l=treeDepth(T->lchild);
        int r=treeDepth(T->rchild);
        return l>r?l+1:r+1;
    }
}
```

- 非递归做法

```
int treeDepth(BiTree T)
{
    if(T==NULL)
        return 0;

    int maxdepth=0;

    queue<node> q;//此处为了方便使用c++中的stl

    q.push({T,1});//将根结点入队

    while(q.empty()==0)
    {
        node temp=q.front();//队头结点出队
        q.pop();

        maxdepth=max(maxdepth,temp.depth);//更新值
        maxdepth=temp.depth;//由于是层序遍历,必然深度越大的,后访问,故可直接更新

        if(temp.p->lchild!=NULL)//左子树不空,将左子树根结点入队
            q.push({temp.p->lchild,temp.depth+1});
        if(temp.p->rchild!=NULL)//右子树不空,将右子树根结点入队
            q.push({temp.p->rchild,temp.depth+1});
    }

    return maxdepth;
}
```

5.3.6

> 设一棵二叉树中各结点的值互不相同，其先序遍历序列和中序遍历序列分别存于两个一维数组$A[1\cdots n]$和$B[1\cdots n]$中，试编写算法建立该二叉树的二叉链表。

根据二叉树的先序遍历序列和中序遍历序列可以创建一棵唯一的二叉树。先序遍历的第一个结点，是二叉树的根节点，在中序遍历找到根结点后，可以知道根节点的左右子树的结点和左右子树的结点数(用$left_size$和$right_size$表示），然后递归分别建立其左右子树，依次扫描二叉树先序序列，然后在中序序列中找到该结点从而确定该结点下的左右子树，直到左右子树的结点数为0时($left_size=0$和$right_size=0$)，二叉树建立完毕。

```
BiTree CreateBiTree_by_Pre_and_In(ElemType a[],ElemType b[],int la,int ra,int lb,int rb)
{
    BiTNode *root=(BiTNode *)malloc(sizeof(BiTNode));

    root->value=a[la];//当前先序第一个元素

    int pos=lb;

    while(b[pos]!=a[la])//找到当前段中序遍历等于当前先序的第一个元素
        pos++;

    int left_size=pos-lb;//左子树长度
    int right_size=rb-pos;//右子树长度

    if(left_size>0)//建立左子树
        root->lchild=CreateBiTree_by_Pre_and_In(a, b, la+1, la+left_size, lb,lb+left_size-1);
    else root->lchild=NULL;//左子树为空

    if(right_size>0)
        root->rchild=CreateBiTree_by_Pre_and_In(a, b, ra-right_size+1, ra, rb-right_size+1, rb);
    else root->rchild=NULL;

    return root;
}

ElemType pre_list[MaxSize];
ElemType in_list[MaxSize];

BiTree T=CreateBiTree_by_Pre_and_In(pre_list, in_list, 1, n, 1, n);
```

5.3.7

> 二叉树按二叉链表形式存储，写一个判别给定二叉树是否是完全二叉树的算法。

采用层次遍历。只有一下两种情况出现时，一棵树才不是完全二叉树

- 一个节点的左子树为空，右子树非空

- 在n层遇到过非空节点，然后在$n+1$层又遇到了非空节点

```
bool isCompleteTree(BiTree T)
{
    if(T==NULL)
        return true;
    queue<BiTNode *> q;//此处为了方便使用c++中的stl

    q.push(T);//将根结点入队

    bool flag=false;//用来标记是否遇到空
    while(q.empty()==0)
    {
        int cnt=q.size();//当前层的个数

        while(cnt--)
        {
            BiTNode *p=q.front();//队头结点出队
            q.pop();

            if(p->lchild==NULL&&p->rchild!=NULL)//左子树空，右子树不空
                return false;

            if(flag==true&&(p->lchild!=NULL||p->rchild!=NULL))//曾经遇到过空，又遇到了非空
                return false;

            if(p->lchild==NULL||p->rchild==NULL)//遇到空,更新flag
                flag=true;

            if(p->lchild!=NULL)//左子树不空,将左子树根结点入队
                q.push(p->lchild);
            if(p->rchild!=NULL)//右子树不空,将右子树根结点入队
                q.push(p->rchild);
        }
    }
    return true;
}
```

5.3.8

> 假设二叉树采用二叉链表存储结构存储，试设计一个算法，计算一棵给定二叉树的所有双分支结点个数。

方法一:通过 4种遍历方式,访问根节点时判断是否左右子树都存在

```
int ans=0;

void visit(BiTNode * p)//访问当前结点的数据
{
    //cout << p->value << " ";
```

```
    if(p->lchild!=NULL&&p->rchild!=NULL)
        ans++;
}

void PreOrder(BiTree T)//前序遍历
{
    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
    }
}


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


void PostOrder(BiTree T)//后序遍历
{
    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
    }
}

void LevelOrder(BiTree T)//层次遍历
{
    queue<BiTNode *> q;//此处为了方便使用c++中的stl

    q.push(T);//将根结点入队

    while(q.empty()==0)
    {
        BiTNode *p=q.front();//队头结点出队
        q.pop();
        visit(p);//访问队头元素

        if(p->lchild!=NULL)//左子树不空,将左子树根结点入队
            q.push(p->lchild);
        if(p->rchild!=NULL)//右子树不空,将右子树根结点入队
            q.push(p->rchild);
```

```
    }
}
```

方法二:通过递归式**计算双分支结点个数** $F(p)$

- 当结点 p为 $NULL$时, $F(p)=0$;
- 当结点 p为**双分支结点**时(左右节点都不为空), $F(p)=F(p\rightarrow lchild)+F(p\rightarrow rchild)+1$;
- 反之(即结点 p为**单分支结点**或**叶子结点**), $F(p)=F(p\rightarrow lchild)+F(p\rightarrow rchild)$.

```c
int Calc_Double_Brach(BiTree T)
{
    if(T==NULL)//结点为空
        return 0;
    else if(T->lchild!=NULL&&T->rchild!=NULL)//当前是双分支结点
        return Calc_Double_Brach(T->lchild)+Calc_Double_Brach(T->rchild)+1;
    else return Calc_Double_Brach(T->lchild)+Calc_Double_Brach(T->rchild);//叶子结点或单分子结点
}
```

5.3.9

> 设树 B是一棵采用链式结构存储的二叉树，编写一个把树 B中所有结点的左、右子树进行交换的函数。

采用递归算法实现交换二叉树的左、右子树,首先交换**左孩子**的左、右子树,再交换**右孩子**的左、右子树,最后交换当前根结点的左、右孩子,当结点为空时递归结束

```c
void Swap_Left_Right_Tree(BiTree T)
{
    if(T!=NULL)
    {
        Swap_Left_Right_Tree(T->lchild);//处理左孩子
        Swap_Left_Right_Tree(T->rchild);//处理右孩子

        //处理当前根结点
        BiTNode *temp=T->lchild;
        T->lchild=T->rchild;
        T->rchild=temp;
    }
}
```

5.3.10

> 假设二叉树采用二叉链存储结构存储，设计一个算法，求先序遍历序列中第 k($1\leq k\leq$二叉树中结点个数)个结点的值。

用全局变量 cnt记录当前访问的个数,当先序遍历访问到第 k个结点时,输出它的值.

```c
void visit(BiTNode * p)//访问当前结点的数据
{
```

```
    cout << p->value << " ";
}

int cnt=0;
void PreOrder(BiTree T,int k)//前序遍历
{
    if(T!=NULL)
    {
        cnt++;//个数+1
        if(cnt==k)//当到达第 k个元素访问
        {
            visit(T);//访问根节点
            return ;
        }
        PreOrder(T->lchild,k);//递归遍历左子树
        PreOrder(T->rchild,k);//递归遍历右子树
    }
}
```

5.3.11

> 已知二叉树以二叉链表存储，编写算法完成:对于树中每个元素值为 x的结点，删去以它为根的子树，并释放相应的空间.

```
void ReleaseTree(BiTree t)//释放二叉树空间
{
    if(t!=NULL){
        ReleaseTree(t->lchild);
        ReleaseTree(t->rchild);
        free(t);
    }
}

void Delete_X(BiTree &T,ElemType e)
{
    if(T==NULL)
        return ;

    if(T->value==e)//若T的value为想要删除的元素,则进行删除
    {
        ReleaseTree(T);//删除包括根节点
        T=NULL;//手动赋值为NULL
    }

    if(T!=NULL)
    {
        Delete_X(T->lchild, e);
        Delete_X(T->rchild, e);
    }
}
```

5.3.12

> 在二叉树中查找值为 x的结点，试编写算法（用 C语言）打印值为 x的结点的所有祖先，假设值为 x 的结点不多于一个。

- **递归版**

```
void visit(BiTNode * p)//访问当前结点的数据
{
    cout << p->value << " ";
}

int Search_Ancestor(BiTree T,ElemType e)
{
    if(T==NULL)//空结点返回
        return 0;
    if(T->value==e)
        return 1;

    if(Search_Ancestor(T->lchild, e)==1||Search_Ancestor(T->rchild, e)==1)//左子树或右子树找到
    {
        visit(T);
        return 1;
    }
    else return 0;
}
```

- **非递归版**

采用非递归后序遍历,最后访问根节点,访问到值为 x的结点时,栈中所有元素均为该结点的祖先,依次出栈即可

```
typedef struct StackNode{
    BiTNode *bitnode;
    bool isFirst;
}StackNode;

void Search_Ancestor(BiTree T,ElemType e)
{
    if(T==NULL)//空结点返回
        return ;
    stack<StackNode* > sta;//此处为了方便使用c++中的stl

    BiTNode *p=T;

    while(p!=NULL||sta.empty()==0)//p不空或栈不空
    {
        if(p!=NULL)//一路向左
        {
            StackNode *new_node=(StackNode *)malloc(sizeof(StackNode));
            new_node->bitnode=p;
            new_node->isFirst=true;//标记第一次访问
```

```
            //s.push(p);//当前结点入栈
            sta.push(new_node);

            p=p->lchild;//左孩子不空,一直向左走
        }
        else//出栈,并转向出栈结点的右子树
        {
            StackNode *temp_node=sta.top();//获取栈顶
            sta.pop();//栈顶元素出栈

            if(temp_node->isFirst==true)//表示是第一次出现在栈顶(从左子树返回)
            {
                temp_node->isFirst=false;
                sta.push(temp_node);
                p=temp_node->bitnode->rchild;//向右子树走,p赋值为当前出栈元素的右孩子
            }
            else//第二次出现在栈顶
            {
                p=NULL;//结点访问完后,重置p指针
            }
        }

        if(p!=NULL&&p->value==e)
        {
            while(sta.size()>0)
            {
                StackNode *temp_node=sta.top();//获取栈顶
                visit(temp_node->bitnode);
                sta.pop();//栈顶元素出栈
            }
            return ;
        }
    }
}
```

5.3.13

> 设一棵二叉树的结点结构为($LLINK, INFO, RLINK$)，$ROOT$为指向该二叉树根结点的指针，p和q分别为指向该二叉树中任意两个结点的指针，试编写算法`ANCESTOR(ROOT,p,q,r)`，找到p和q的最近公共祖先结点r.

- **递归做法**

```
BiTNode* Search_Common_Ancestor(BiTree T,BiTNode *p,BiTNode *q)
{
    if(T==NULL||p==NULL||q==NULL)
        return NULL;
    if(p==T||q==T)
        return T;

    BiTNode *left_lca=Search_Common_Ancestor(T->lchild, p, q);
```

```
    BiTNode *right_lca=Search_Common_Ancestor(T->rchild, p, q);

    if(left_lca!=NULL&&right_lca!=NULL)
        return T;
    else if(left_lca==NULL)
        return right_lca;
    else
        return left_lca;
}
```

- **非递归做法**

后序遍历最后访问根结点，即在递归算法中，根是压在栈底的。本题要找 p 和 q 的最近公共祖先结点 r，不失一般性，设 p 在 q 的左边。算法思想：采用后序非递归算法，栈中存放二叉树结点的指针，当访问到某结点时，栈中所有元素均为该结点的祖先。后序遍历必然先遍历到结点 p，栈中元素均为 p 的祖先。先将栈复制到另一辅助栈中。继续遍历到结点 q 时，将栈中元素从栈顶开始逐个到辅助栈中去匹配，第一个匹配（即相等）的元素就是结点 p 和 q 的最近公共祖先。

```
typedef struct StackNode{
    BiTNode *bitnode;
    bool isFirst;
}StackNode;

//此处设p1在p2左边
BiTNode* Search_Common_Ancestor(BiTree T,BiTNode *p1,BiTNode *p2)
{
    if(p1==p2)
        return p1;
    stack<StackNode* > s;//此处为了方便使用c++中的stl

    stack<StackNode* > temp1,temp2;//存储两个的祖先的栈

    vector<BiTNode*> v1,v2;//分别存储各自的祖先

    int flag=0;//标记是否经过p
    BiTNode *p=T;

    while(p!=NULL||s.empty()==0)//p不空或栈不空
    {
        if(p!=NULL)//一路向左
        {
            StackNode *new_node=(StackNode *)malloc(sizeof(StackNode));
            new_node->bitnode=p;
            new_node->isFirst=true;//标记第一次访问

            //s.push(p);//当前结点入栈
            s.push(new_node);
            if(flag==0)
                temp1.push(new_node);
            temp2.push(new_node);

```

```
            p=p->lchild;//左孩子不空,一直向左走
        }
        else//出栈,并转向出栈结点的右子树
        {

            StackNode *temp_node=s.top();//获取栈顶
            s.pop();//栈顶元素出栈
            if(flag==0)
                temp1.pop();
            temp2.pop();

            if(temp_node->isFirst==true)//表示是第一次出现在栈顶(从左子树返回)
            {
                temp_node->isFirst=false;
                s.push(temp_node);
                if(flag==0)
                    temp1.push(temp_node);
                temp2.push(temp_node);

                p=temp_node->bitnode->rchild;//向右子树走,p赋值为当前出栈元素的右孩子
            }
            else//第二次出现在栈顶
            {
                if(temp_node->bitnode==p1)
                {
                    flag=1;
                    while(temp1.size()>0)
                        v1.push_back(temp1.top()->bitnode),temp1.pop();
                }

                if(temp_node->bitnode==p2)
                {
                    while(temp2.size()>0)
                        v2.push_back(temp2.top()->bitnode),temp2.pop();

                    // for(int i=0;i<v1.size();i++)
                        // cout << v1[i]->value << " ";
                    // cout << endl;
                    // for(int i=0;i<v2.size();i++)
                        // cout << v2[i]-> value << " ";
                    // cout << endl;
                    for(int i=0;i<v1.size();i++)
                        for(int j=0;j<v2.size();j++)
                            if(v1[i]==v2[j])
                                return v1[i];
                }
                p=NULL;//结点访问完后,重置p指针
            }
        }
    }
    return NULL;
}
```

5.3.14

> 假设二叉树采用二叉链表存储结构，设计一个算法，求非空二叉树 b的宽度（即具有结点数最多的那一层的结点个数).

层序遍历找到每层中的结点个数即可,用结构体保存当前结点指针和层数

```
typedef struct QueueNode{
    BiTNode *bitnode;
    int level;
}QueueNode;


int Calc_Width(BiTree T)//层次遍历
{
    if(T==NULL)
        return 0;

    queue<QueueNode> q;//此处为了方便使用c++中的stl
    QueueNode temp={T,1};
    q.push(temp);//将根结点入队

    int last=0,maxx=0;//记录上一层

    int tot=0;

    while(q.empty()==0)
    {
        QueueNode new_node=q.front();//队头结点出队
        q.pop();


        if(last!=new_node.level)//更新上一层的最大值
        {
            if(tot>maxx)
                maxx=tot;
            tot=1;
            last=new_node.level;
        }
        else tot++;

        if(new_node.bitnode->lchild!=NULL)//左子树不空,将左子树根结点入队
        {
            temp.bitnode=new_node.bitnode->lchild;
            temp.level=new_node.level+1;
            q.push(temp);
        }

        if(new_node.bitnode->rchild!=NULL)//右子树不空,将右子树根结点入队
        {
            temp.bitnode=new_node.bitnode->rchild;
            temp.level=new_node.level+1;
```

```
            q.push(temp);
        }
    }

    maxx=max(maxx,tot);
    return maxx;
}
```

5.3.15

> 设有一棵满二叉树（所有结点值均不同），已知其先序序列为 pre，设计一个算法求其后序序列 $post$。

对满二叉树,任意一个结点的左、右子树均**含有相等**的结点数,同时,先序序列的第一个结点作为后序序列的最后一个结点.递归实现,每一次递归的结果就是将先序序列的第一个结点放到后序序列的最后一个结点,直至这个二叉树的递归完成.

```
void Pre_To_Post(ElemType pre[],ElemType post[],int l1,int r1,int l2,int r2)
{
    if(l1<=r1)
    {
        post[r2]=pre[l1];//先序的第一个元素是后序最后一个元素
        int mid=(r1-l1)/2;
        Pre_To_Post(pre, post, l1+1, l1+mid, l2, l2+mid-1);//左子树
        //左子树元素个数为mid-1(刨去根)
        Pre_To_Post(pre, post, l1+mid+1, r1, l2+mid, r2-1);//右子树
        //后序最后一个元素被占据
    }
}
```

5.3.16

> 设计一个算法将二叉树的叶结点按从左到右的顺序连成一个单链表，表头指针为 $head$.二叉树按二叉链表方式存储，链接时用叶结点的右指针城来存放单链表指针。

设置前驱结点指针 pre,初始为空.第一个叶结点由指针 $head$指向,遍历(前序,中序和后序都可以)到叶子结点,将它前驱的 $rchild$指向它,最后一个叶子结点的 $rchild$为空.

```
LinkList head,pre=NULL;

LinkList Get_Leaf_List(BiTree T)//中序遍历
{
    if(T!=NULL)
    {
        Get_Leaf_List(T->lchild);//递归遍历左子树

        if(T->lchild==NULL&&T->rchild==NULL)
        {
            if(pre==NULL)//第一个叶子结点
                head=T,pre=T;
            else
            {
```

```
                pre->rchild=T;
                pre=T;
            }
        }

        Get_Leaf_List(T->rchild);//递归遍历右子树
    }
    return head;
}
```

5.3.17

> 试设计判断两棵二叉树是否相似的算法。所谓二叉树 T_1和 T_2相似，指的是 T_1和 T_2都是空的二叉树或都只有一个根结点;或 T_1的左子树和 T_2的左子树是相似的，且 T_1的右子树和 T_2的右子树是相似的.

若 T_1和 T_2都是空树,则相似;若有一个为空另一个不空,则必然不相似;否则**递归**比较它们的左、右子树是否相似.

```
bool isSimilar(BiTree T1,BiTree T2)
{
    if(T1==NULL&&T2==NULL)//均为空
        return true;
    else if(T1==NULL||T2==NULL)//有一个树为空另一个树不空
        return false;
    else
    {
        bool left_sim=isSimilar(T1->lchild, T2->lchild);//左子树是否相似
        bool right_sim=isSimilar(T1->rchild, T2->rchild);//右子树是否相似
        return left_sim&&right_sim;//两者都成立时相似
    }
}
```

5.3.18

> 写出在中序线索二叉树里查找指定结点在后序的前驱结点的算法.

在二叉树后序序列中，对于结点 p，其前驱依次有可能是：

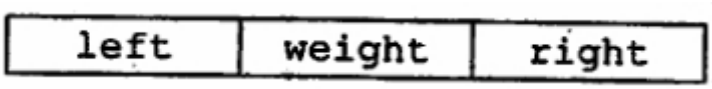
- p的**右孩子**
- 没有右孩子，那就可能是**左孩子**
- 没有孩子，那就可能是其**父结点的左孩子**
- 否则,可能是其**爷爷结点的左孩子**
- ...
- 以此类推(即找**最近的祖先结点的左孩子**)。

中序线索二叉树中， p无左孩子，则其左指针域指向**其父**，故可向上访问，直到有一个祖先有左孩子，则这个左孩子一定是后序遍历 p的前驱。

```
ThreadNode * Get_PostPre_By_InOrder(ThreadNode * p)
{
```

```
    if(p==NULL)
        return NULL;
    if(p->rtag==0)//有右孩子,则右孩子是它的前驱
        return p->rchild;
    else if(p->ltag==0)//只有左孩子,则右孩子是它的前驱
        return p->lchild;

    while(p!=NULL&&p->ltag==1)//p的最近的<祖先的左孩子>
        p=p->lchild;

    if(p!=NULL)
        return p->lchild;
    else return NULL;
}
```

5.3.19

> **2014统考真题**：二叉树的带权路径长度(WPL)是二叉树中所有叶结点的带权路径长度之和。给定一棵二叉树 T，采用二叉链表存储，结点结构为
>
> 
>
>
> 其中叶节点的 $weight$域保存该结点的非负权值,设 $root$为指向 T的根结点的指针,请设计求 T的 WPL的算法,要求:
>
> 1)给出算法的基本设计思想;
>
> 2)使用 C 或 C++ 语言,给出二叉树结点的数据类型定义;
>
> 3)根据设计思想,采用 C 或 C++ 语言描述算法,关键之处给出注释.

1)可以递归或非递归实现

- 递归(**前序、中序和后序遍历**)

① 基于先序递归遍历的算法思想是用一个 static 变量记录 wpl，把每个结点的深度作为递归函数的一个参数传递，算法步骤如下：
若该结点是叶结点，则变量 wpl 加上该结点的深度与权值之积。
若该结点是非叶结点，则左子树不为空时，对左子树调用递归算法，右子树不为空，对右子树调用递归算法，深度参数均为本结点的深度参数加 1。
最后返回计算出的 wpl 即可。

- 非递归(**层序遍历**)

基于层次遍历的算法思想是使用队列进行层次遍历，并记录当前的层数：
当遍历到叶结点时，累计 wpl。
当遍历到非叶结点时，把该结点的子树加入队列
当某结点为该层的最后一个结点时，层数自增 1。
队列空时遍历结束，返回 wpl。

2)

```
typedef int ElemType;

typedef struct BiTNode{
    ElemType weight;//结点中的顺序元素
    struct BiTNode *lchild,*rchild;
    //struct BiTNode *parent;
}BiTNode,*BiTree;
```

3)

- 递归

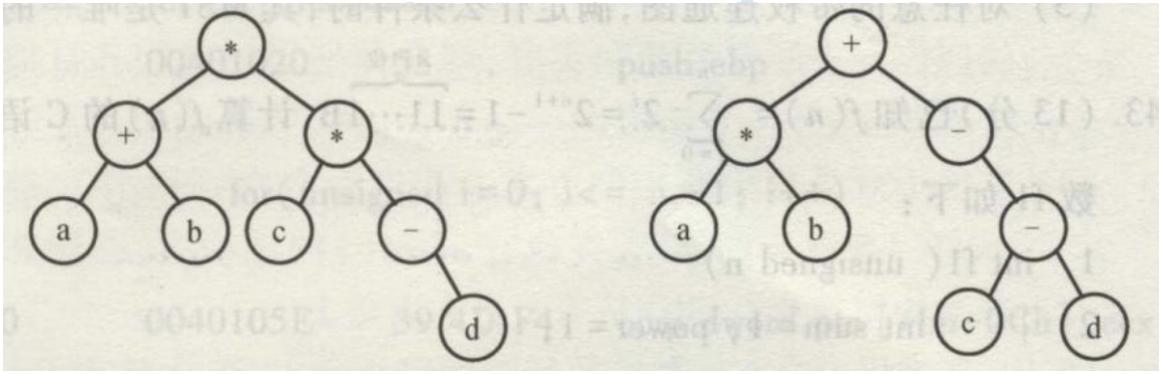
```
int wpl_res=0;

void Wpl_PreOrder(BiTree T,int depth)
{
    if(T!=NULL)
    {
        if(T->lchild==NULL&&T->rchild==NULL)//叶子结点
            wpl_res+=(depth-1)*T->weight;//注意是路径,不是深度

        if(T->lchild!=NULL)
            Wpl_PreOrder(T->lchild, depth+1);
        if(T->rchild!=NULL)
            Wpl_PreOrder(T->rchild, depth+1);
    }

}

int Calc_Wpl(BiTree T)
{
    wpl_res=0;
    Wpl_PreOrder(T,1);
    return wpl_res;
}
```

- 非递归

```
typedef struct QueueNode{
    BiTNode *bitnode;
    int level;
}QueueNode;

int Calc_Wpl(BiTree T)
{
    int wpl_res=0;

    if(T==NULL)
        return 0;
```

```
12
13      queue<QueueNode> q;//此处为了方便使用c++中的stl
14      QueueNode temp={T,1};
15      q.push(temp);//将根结点入队
16
17
18      while(q.empty()==0)
19      {
20          QueueNode new_node=q.front();//队头结点出队
21          q.pop();
22
23          if(new_node.bitnode->lchild==NULL&&new_node.bitnode->rchild==NULL)
24              wpl_res+=(new_node.level-1)*new_node.bitnode->weight;
25
26          if(new_node.bitnode->lchild!=NULL)//左子树不空,将左子树根结点入队
27          {
28              temp.bitnode=new_node.bitnode->lchild;
29              temp.level=new_node.level+1;
30              q.push(temp);
31          }
32
33          if(new_node.bitnode->rchild!=NULL)//右子树不空,将右子树根结点入队
34          {
35              temp.bitnode=new_node.bitnode->rchild;
36              temp.level=new_node.level+1;
37              q.push(temp);
38          }
39      }
40      return wpl_res;
41  }
```

5.3.20

> **2017统考真题**：请设计一个算法，将给定的表达式树（二叉树）转换为等价的中缀表达式(通过括号反映操作符的计算次序）并输出。例如，当下列两棵表达式树作为算法的输入时:
>
> 
>
>
> 输出的等价中缀表达式分别为`(a+b)*(c*(-d))`和`(a*b)+(-(c-d))`。 二叉树结点定义如下:

```c
typedef struct node{
    char data[10]; //存储操作数或操作符
    struct node *left, *right;
}
BTree;
```

> 要求:
>
> 1)给出算法的基本设计思想;
>
> 2)根据设计思想,采用 `C` 或 `C++` 语言描述算法,关键之处给出注释.

1)表达式树的中序序列加上必要的括号即为等价的中缀表达式。可以基于二叉树的**中序遍历**策略得到所需的表达式。表达式树中分支结点所对应的子表达式的计算次序，由该分支结点所处的位置决定。为得到正确的中缀表达式， 需要在生成遍历序列的同时，在适当位置增加必要的括号。显然，表达式的最外层（对应根结点）及操作数（对应叶结点）不需要添加括号。

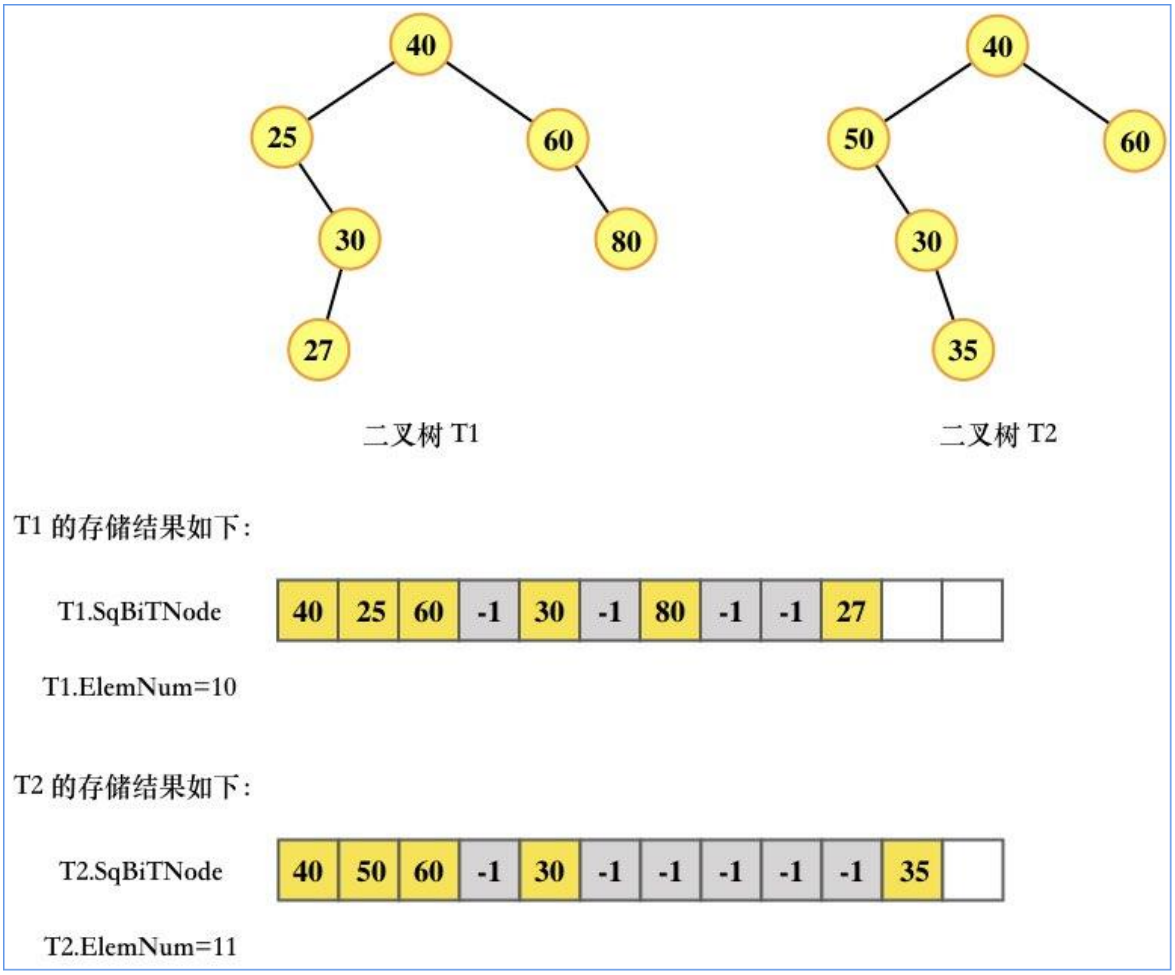
2)除**根结点和叶结点**外，遍历到其他结点时在遍历 其左子树之前加上左括号，在遍历完右子树后加上右括号。

```c
void visit(BTree *p)//访问当前结点的数据
{
    printf(" %s ",p->data);
}

void InOrder(BTree *T,int depth)//中序遍历
{
    if(T==NULL)
        return ;
    if(T->left!=NULL||T->right)
    {
        if(depth>1)
            cout << "(";
        InOrder(T->left,depth+1);//递归遍历左子树
        visit(T);//访问根节点
        InOrder(T->right,depth+1);//递归遍历右子树
        if(depth>1)
            cout << ")";
    }
    else visit(T);
}

InOrder(T,1);
```

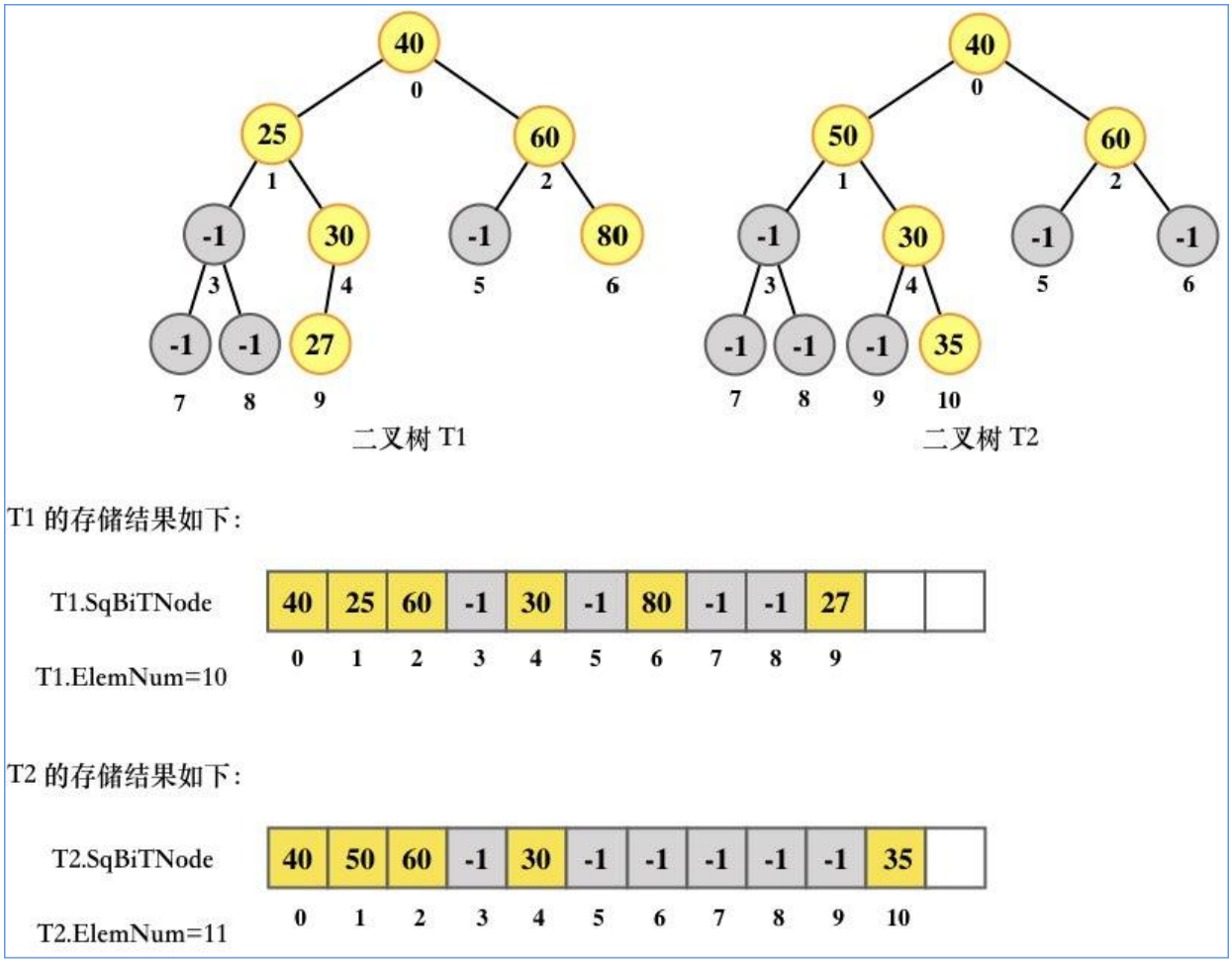
5.3.21

> **2022统考真题**：已知非空二叉树 T的结点值均为正数，采用顺序存储方式保存，数据结构定义如下：

```
typedef struct {                    // MAX_SIZE为已定义常量
    int SqBiTNode[MAX_SIZE];    // 保存二叉树结点值的数组
    int ELemNum;                    // 实际占用的数组元素个数
}SqBiTree;
```

> T中不存在的结点在数组$SqBiTNode$中用-1表示。例如，对于下图所示的两棵非空二叉树T_1和T_2：
>
> 
>
>
> 请设计一个尽可能高效的算法，判定一颗采用这种方式存储的二叉树是否为二叉搜索树，若是，则返回$true$,否则，返回$false$.要求：
>
> 1)给出算法的基本设计思想。
>
> 2)根据设计思想，采用`C`或`C++`语言描述算法，关键之处给出注释。

将其填充为**完全二叉树**，按照层序遍历顺序填写编号：



如果根结点下标为 0,设编号为 i的结点,它的左孩子结点编号为 $2\times i+1$,它的右孩子结点编号为 $2\times i+2$.

- **方法一**:递归

利用二叉搜索树的性质: 设 x是二叉搜索树中的一个结点,如果 y是 x的**左子树**中的一个结点,那么 y的关键字 $\leq x$的关键字;如果 y是 x的**右子树**中的一个结点,那么 y的关键字 $\geq x$的关键字.

```
bool check(SqBiTree T,int id,ElemType minx,ElemType maxx)
{
    if(id>=T.ElemNum||T.SqBiTNode[id]==-1)//越界或者为空结点
        return true;
    if(T.SqBiTNode[id]<=minx||T.SqBiTNode[id]>=maxx)
        return false;
    return check(T,2*id+1,minx,T.SqBiTNode[id])&&check(T, 2*id+2,
T.SqBiTNode[id], maxx);
}

bool isValidBST(SqBiTree T)
{
    return check(T,0,INT_MIN,INT_MAX);
}
```

- **方法二**:中序遍历

利用二叉搜索树的性质的推论：中序遍历为**单调递增序列。**

```cpp
void InOrder(SqBiTree T,int id,vector<ElemType> &res)
{
    if(id>=T.ElemNum||T.SqBiTNode[id]==-1)//越界或者为空结点
        return ;
    InOrder(T, 2*id+1, res);
    res.push_back(T.SqBiTNode[id]);
    InOrder(T, 2*id+2, res);
}

bool isValidBST(SqBiTree T)
{
    vector<ElemType> res;

    InOrder(T, 0, res);

    int minx=res[0];//记录前驱结点的值
    for(int i=1;i<res.size();i++)//检查中序遍历是否单调
    {
        if(res[i]<minx)
            return false;
        minx=res[i];
    }
    return true;
}
```

- 优化**空间复杂度**,使用一个变量 $prev$记录前驱,由于非空二叉树 T的结点值均为正数,初始化 $prev=0$.

```cpp
bool InOrder(SqBiTree T,int id,ElemType &prev)
{
    if(id>=T.ElemNum||T.SqBiTNode[id]==-1)//越界或者为空结点
        return true;
    if(InOrder(T, 2*id+1, prev)==false)
    //此处不能写成return InOrder(T, 2*id+1, prev);
    //当是true的时候需看右孩子
        return false;
    if(T.SqBiTNode[id]<prev)
        return false;
    prev=T.SqBiTNode[id];
    return InOrder(T, 2*id+2, prev);
}

bool isValidBST(SqBiTree T)
{
    ElemType prev=0;
    return InOrder(T, 0, prev);
}
```