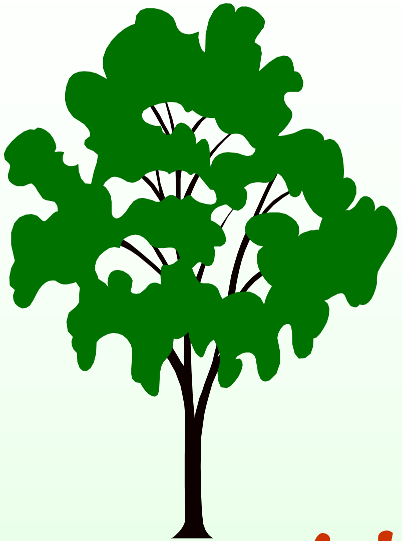
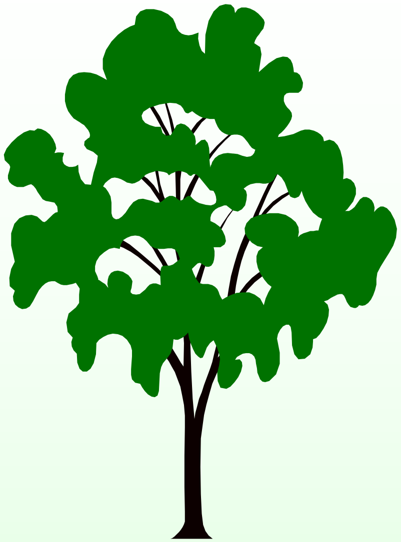


6.3
遍历二叉树和线索二叉树



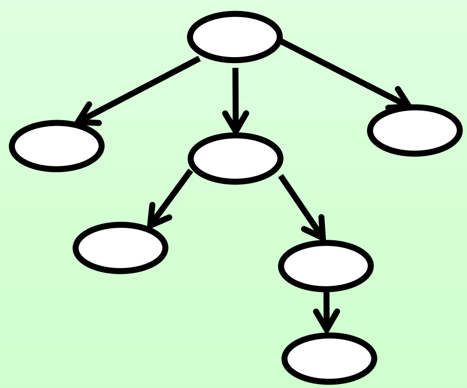
6.3.1 遍历二叉树

1. 基本概念

遍历：顺着某一条搜索路径**巡访**二叉树中的每个结点，使得每个结点均被**访问一次**，而且**仅被访问一次**。

“**访问**”的含义可以很广，如：输出结点的信息等。

遍历目的



遍历

结点访问序列

非线性 线性化

"二叉树"由三个基本单元组成：**根结点、左子树和右子树**。若能依次遍历这三部分，就遍历了整个二叉树。

设用L、D、R分别表示遍历左子树、访问根结点、遍历右子树，则可有以下 6 种遍历二叉树的方案：

DLR、LDR、LRD —— 先左后右

DRL、RDL、RLD —— 先右后左

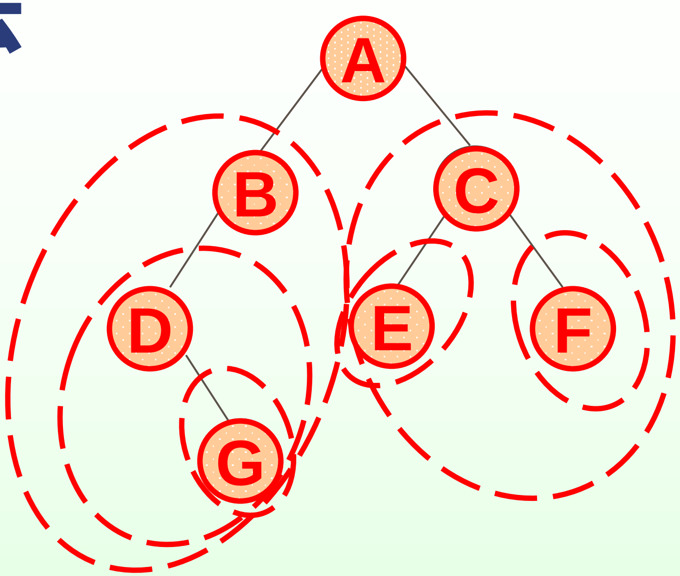
2. 先左后右的遍历算法

DLR：前（根）序的遍历算法

LDR：中（根）序的遍历算法

LRD：后（根）序的遍历算法

2.1 前（根）序的遍历算法



若二叉树为空树，则空操作；

否则：

（1）D：访问根结点；

（2）L：前序遍历左子树；

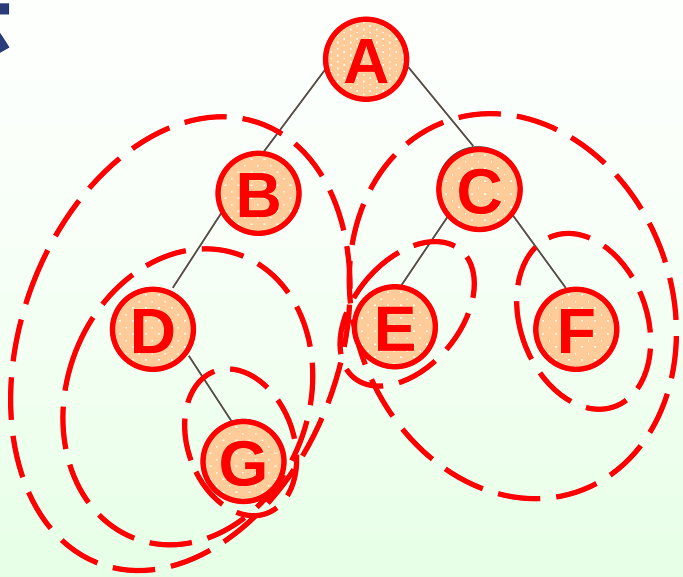
（3）R：前序遍历右子树。

先序遍历结果：A B D G C E F

2.1前（根）序的遍历算法

```
void preorder (NODE  *p)
{   // 前序遍历二叉树
    if (p!=NULL)
    {      printf("%c",p->data);
           preorder(p->lchild);
           preorder(p->rchild);
     }
}
```

2.2 中（根）序的遍历算法



若二叉树为空树，则空操作；

否则：

（1）L：中序遍历左子树；

（2）D：访问根结点；

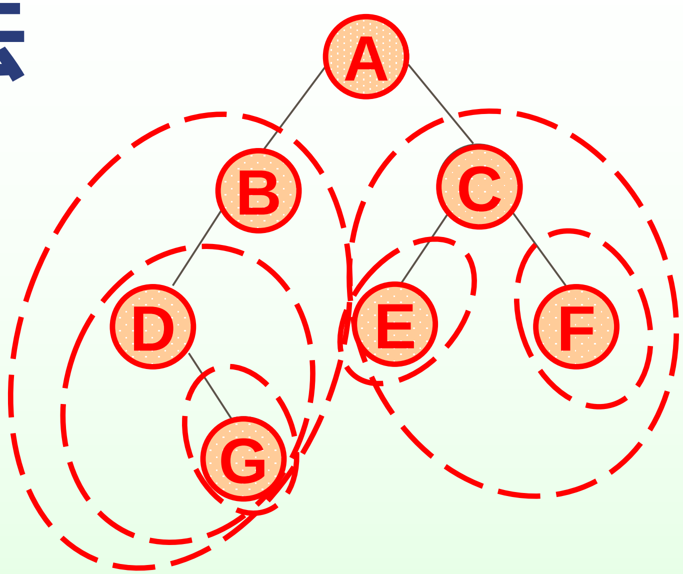
（3）R：中序遍历右子树。

中序遍历结果：*D G B A E C F*

2.2 中（根）序的遍历算法

```
void  inorder (NODE  *p)
{   // 中序遍历二叉树
     if (p!=NULL)
     {     inorder(p->lchild);
           printf(“%c”,p->data);
           inorder(p->rchild);
      }
}
```

2.3 后（根）序的遍历算法



若二叉树为空树，则空操作；

否则：

（1）L：后序遍历左子树；

（2）R：后序遍历右子树。

（3）D：访问根结点；

后序遍历结果：G D B E F C A

2.3 后（根）序的遍历算法

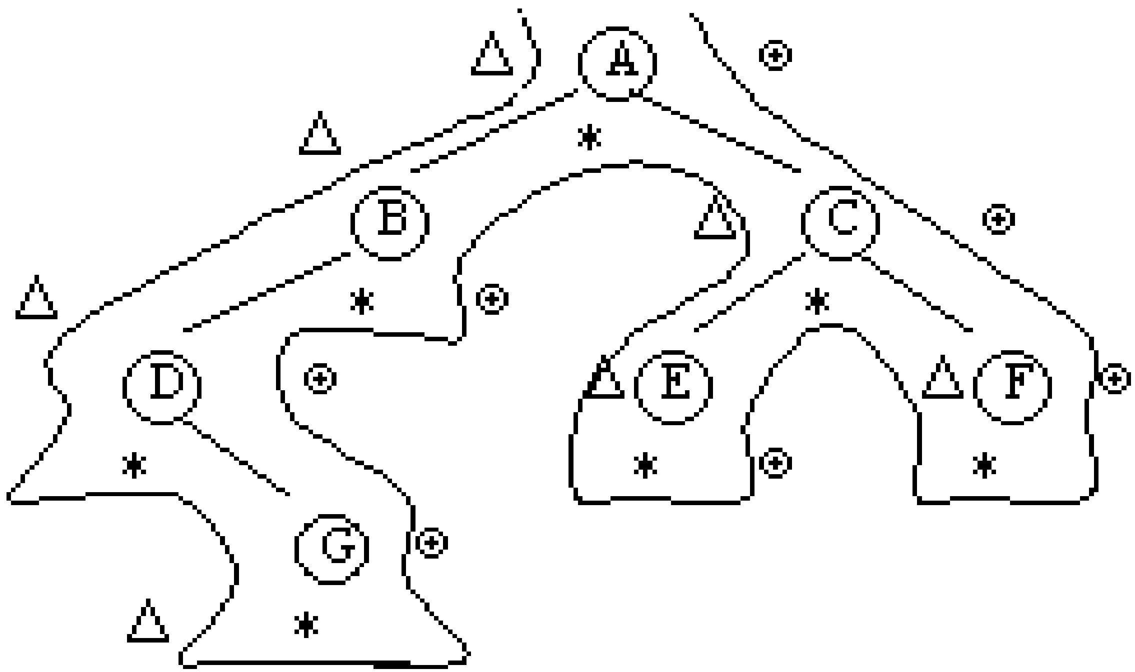
```
void postorder (NODE  *p)
{   // 后序遍历二叉树
      if (p!=NULL)
      {     postorder(p->lchild);
            postorder(p->rchild);
            printf("%c",p->data);
       }
}
```

```
void preorder (NODE  *p)
{  // 前序遍历二叉树
   if (p!=NULL)
   {   visit(p->data);
       preorder(p->lchild);
       preorder(p->rchild);
   }
}
```

```
void  inorder (NODE  *p)
{  // 中序遍历二叉树
    if (p!=NULL)
    {    inorder(p->lchild);
         visit(p->data);
         inorder(p->rchild);
     }
}
```

```
void postorder (NODE  *p)
{  // 后序遍历二叉树
   if (p!=NULL)
   {   postorder(p->lchild);
       postorder(p->rchild);
       visit(p->data);
    }
}
```

语句都是一样的，只是顺序不同！

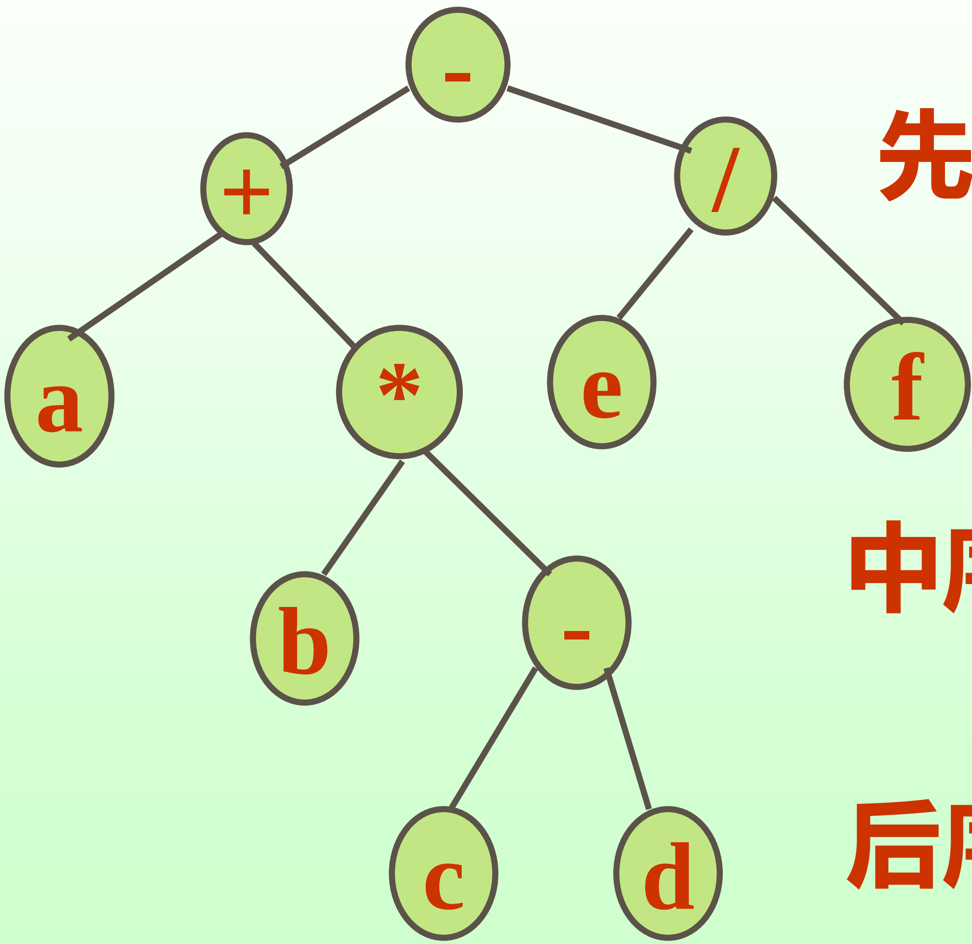


遍历路线示意图

进行前序、中序和后序遍历都是从根结点A开始的，且在遍历过程中**经过结点的路线是一样的**，只是**访问的时机不同**而已。

3. 表达式的二叉树表示

如表达式：a+b*(c-d)-e/f



先序序列：前缀(波兰式)

- + a * b – c d / e f

中序序列：中缀表示

a + b * c – d – e / f

后序序列：后缀(逆波兰式)

a b c d - * + e f / -

4. 遍历算法的应用举例

1）输出二叉树中的结点(先序遍历)

2）统计二叉树中叶子结点的个数(先序遍历)

3）求二叉树的深度（后序遍历）

4.1 输出二叉树的结点

```
void Preorder (BiTree root)
{      if (root!=NULL)
       {
              printf(root->data);
              Preorder(root->lchild);
              Preorder(root->rchild);
       }
}
```

4.2 统计二叉树中叶子结点的数目

算法基本思想:

先序(或中序或后序)遍历二叉树，在遍历过程中查找叶子结点，并计数。

由此，**需在遍历算法中增添一个“计数”的参数**，并将算法中“访问结点”的操作改为：**若是叶子，则计数器增1。**

4.2 统计二叉树中叶子结点的数目（算法）

```
int  LeafCount=0;
void  LeafNum(BiTree bt)   //先序遍历
{    if (bt!=NULL)
     {
          if(bt->LChild==NULL&&bt->RChild==NULL)
               LeafCount++;
          LeafNum(bt->LChild);
          LeafNum(bt->RChild);
     }
}
```

4.3 求二叉树的深度（后序遍历）

算法基本思想:

首先分析二叉树的深度和它的左、右子树深度之间的关系。

从二叉树深度的定义可知，二叉树的深度应为其左、右子树深度的最大值加1。

由此，需先分别求得左、右子树的深度，算法中“访问结点”的操作为：求得左、右子树深度的最大值，然后加 1。

4.3 求二叉树的深度（后序遍历）

```
int PostTreeDepth(BiTree bt)
{   int hl,hr,max;
    if (bt!=NULL)
    {  hl=PostTreeDepth(bt->LChild);
       hr=PostTreeDepth(bt->RChild);
       max=hl>hr?hl:hr;
       return(max+1);
    }
    else return(0);    /* 如果是空树，则返回0 */
}
```

Non-recursive traversal algorithm ★
二叉树遍历的非递归算法

★**先序遍历**的基本思想：

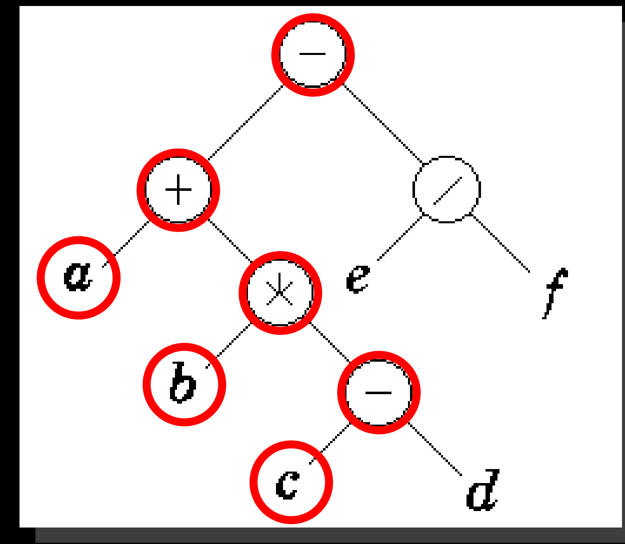
遇到一个结点，访问之，接着把它压入栈中，然后去遍历它的左子树。遍历完它的左子树后，从栈顶弹出这个结点，然后再去遍历它的右子树。

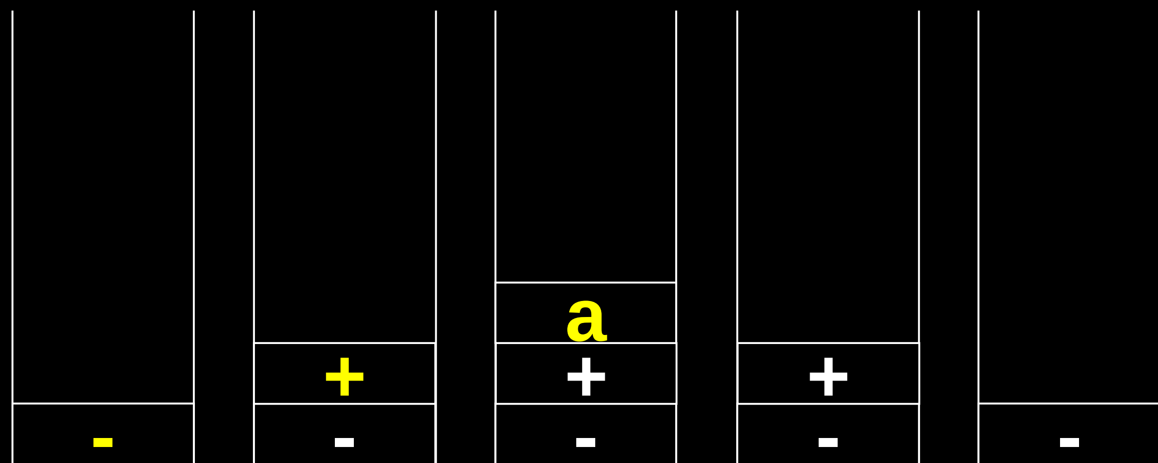
★**中序遍历**的基本思想：

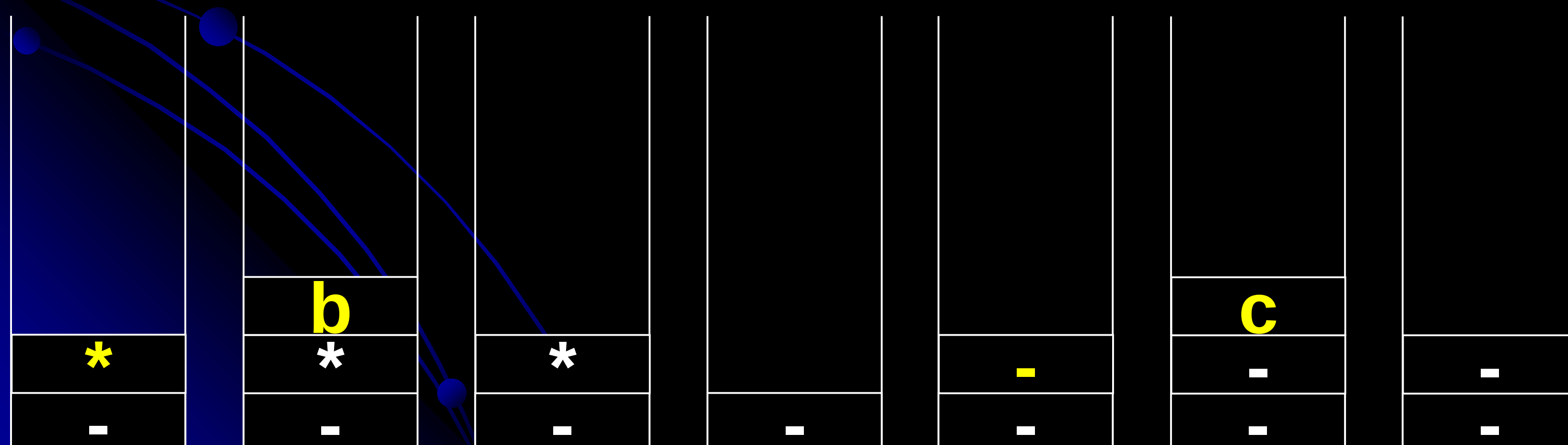
遇到一个结点，就把它压入栈中，去遍历它的左子树，遍历它的左子树后，从栈顶弹出这个结点并访问之，然后再去遍历它的右子树。

Stack state in PreOrderTraversal

先序遍历中栈的变化

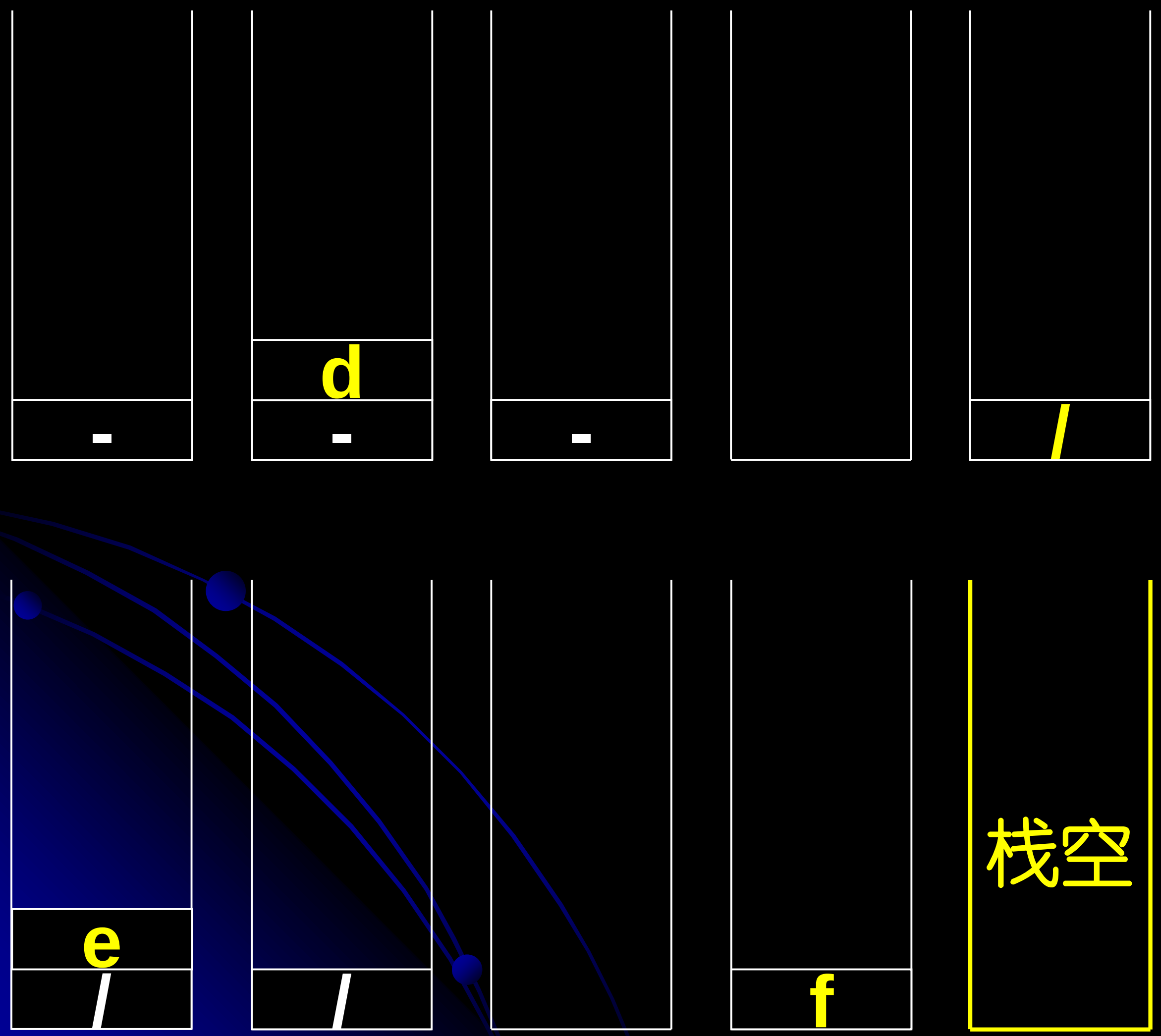


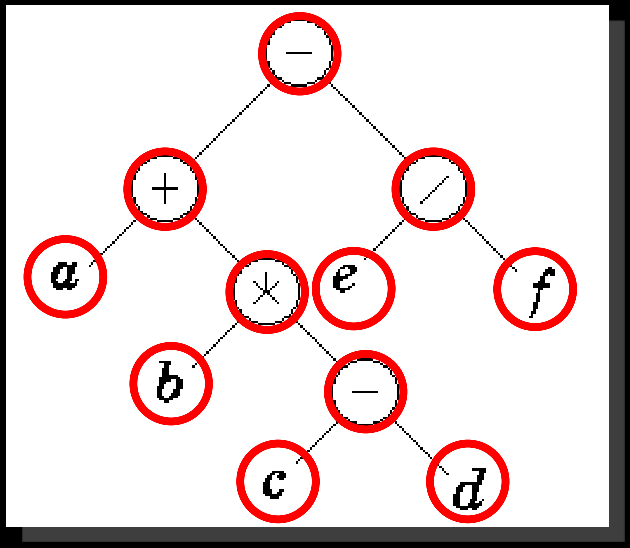




Stack state in PreOrderTraversal

先序遍历中栈的变化



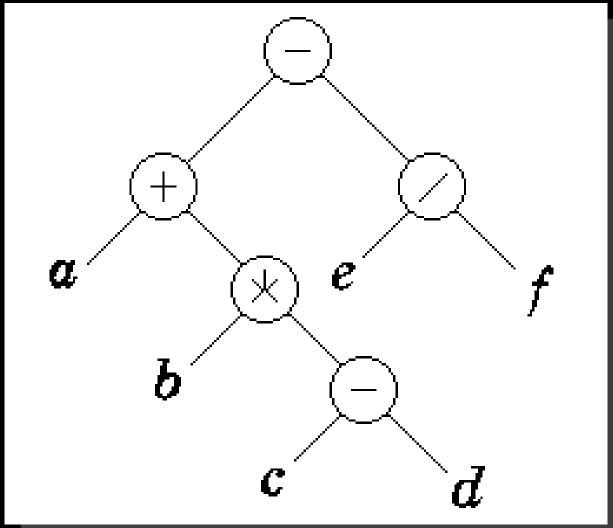


//Preorder Traversal

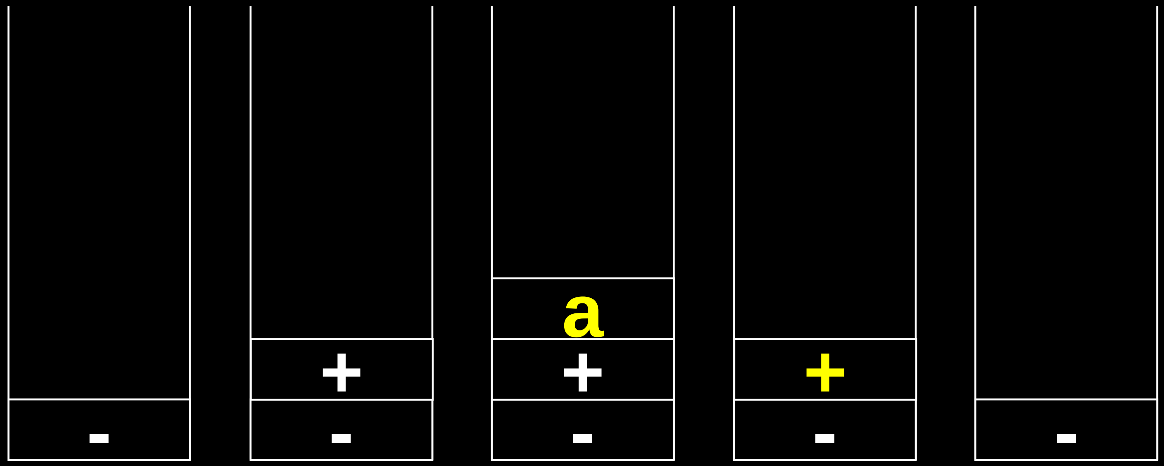
```
void PreOrderTraverse (PBinTree T ) {
    Stack S; PBinTree p;
    StackEmpty( S );   p = T;
    do {
        while ( p ) {
            printf ( p->info);
            Push( S, p );
            p = p->lchild;
        }
        if ( !IsEmpty( S ) ) {
            p = getTop( S );
            Pop( S );
            p = p->rchild;
        }
    } while ( p || !IsEmpty( S ) );
}
```

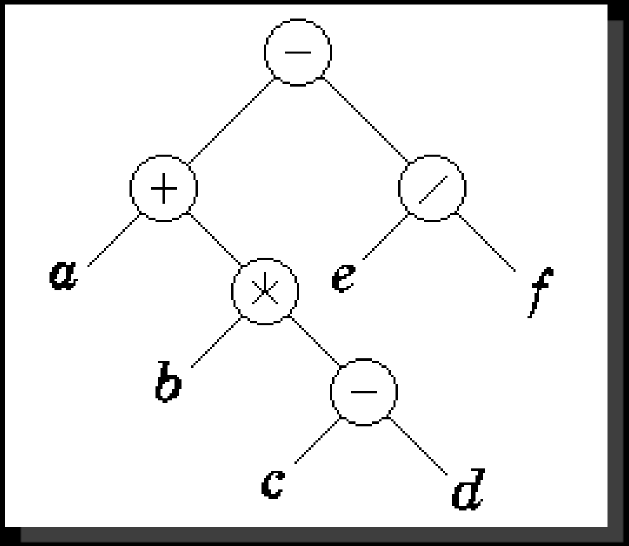
```
struct StackNode {
    PBinTree ptr;
};
```

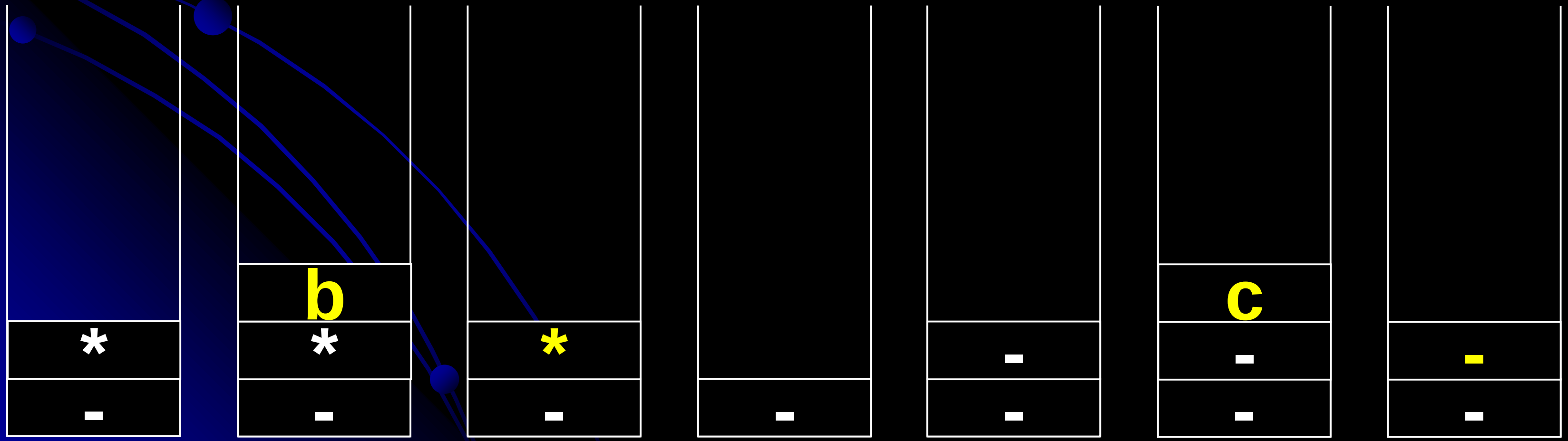
ptr



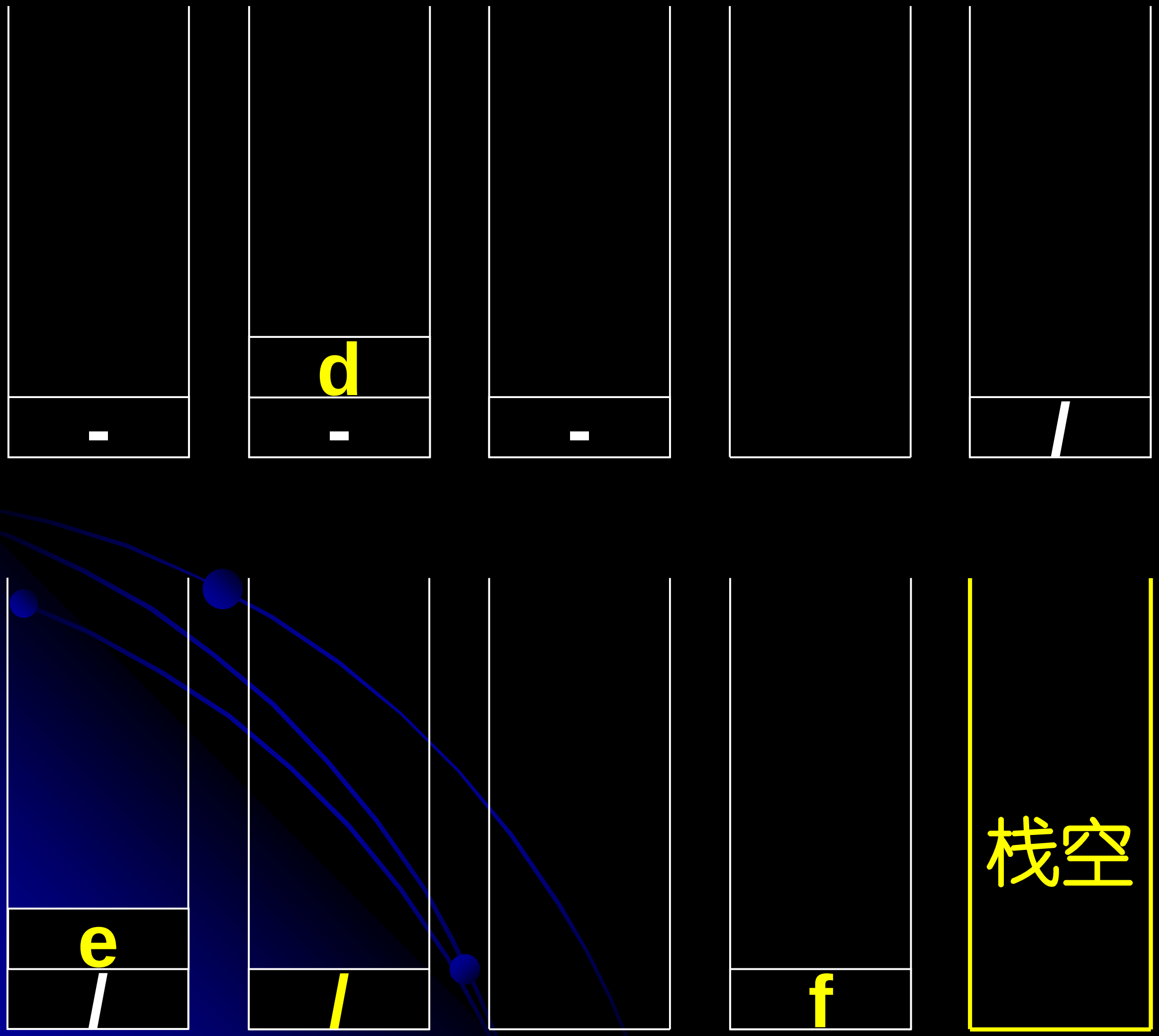
Stack state in InOrderTraversal
中序遍历中栈的变化

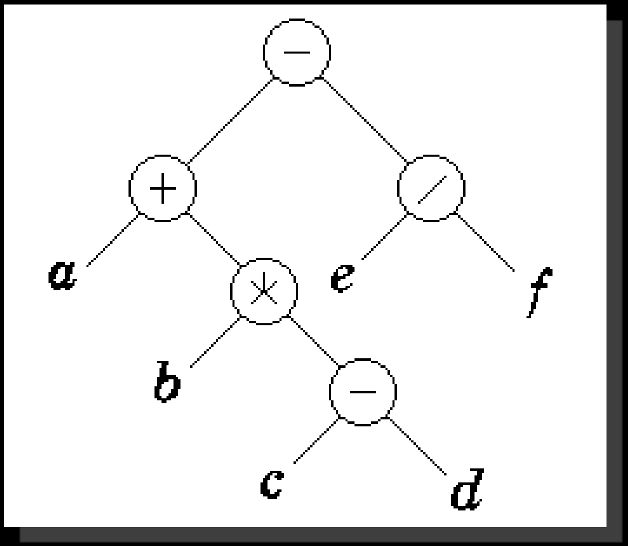






Stack state in InOrderTraversal
中序遍历中栈的变化

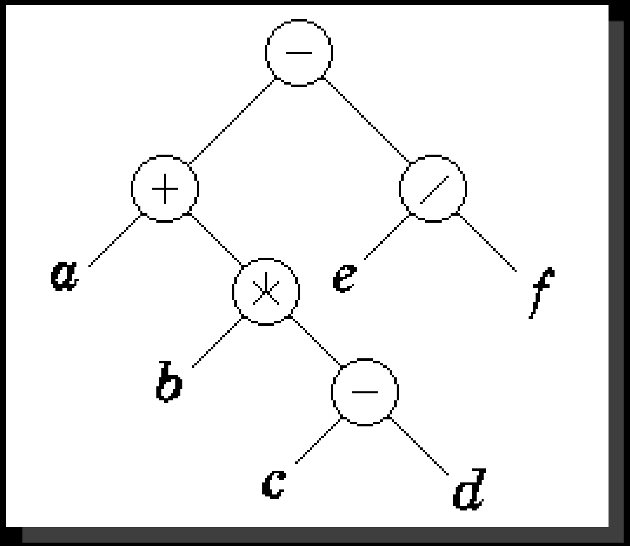




//Inorder Traversal

```
void InOrderTraverse (PBinTree T ) {
    Stack S; PBinTree  p;
    StackEmpty( S );   p = T;
    do {
        while ( p ) {
            Push( S, p );
            p = p->lchild;
        }
        if ( !IsEmpty( S ) ) {
            p = getTop( S );
            Pop( S );
            printf ( p->info);
            p = p->rchild;
        }
    } while ( p || !IsEmpty( S ) );
}
```

ptr



后序遍历的基本思想是：

后序非递归算法比较复杂，每个结点要等到它们左、右子树都被遍历完后才得以访问，所以**在去遍历它的左、右子树之前都需要进栈**。

当它出栈时，需要判断是从左子树回来（即刚遍历完左子树，此时需要去遍历右子树），还是从右子树回来（即刚遍历完右子树，此时可以访问这个结点）。因此，进栈的结点需要伴随着一个标记***tag***。

$$tag=\begin{cases} L & \text{表明遍历它的左子树} \\ R & \text{表明遍历它的右子树} \end{cases}$$

```
typedef struct StackNode {
    enum tag { L, R };
    PBinTree ptr;
} StackNode;
```

ptr	*tag* (*L*/*R*)

后序遍历根为 *T* 的二叉树, 存储结构为二叉链表, *S* 是存储所经过二叉树结点的工作栈。其中, *tag* 是结点标记。当 *tag* = *L* 时, 表示刚才是在左子树中, 从左子树退出时, 还要去遍历右子树。当 *tag* = *R* 时, 表示刚才是在右子树中, 在从右子树中退出时, 去访问位于栈顶的结点。

//Postorder Traversal

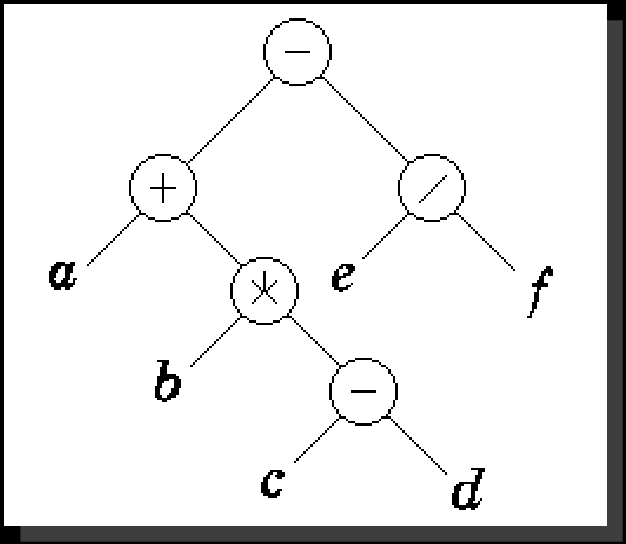
```
void PostOrderTraverse (PBinTree T ) {
Stack S; PBinTree  p;   StackNode w;
    StackEmpty( S );    p = T;
    do {
        while ( p ) {
            w.ptr = p;   w.tag = L;   Push( S, w);
            p = p->lchild;
        }
```

a	L
+	L
-	L

a	R
+	L
-	L

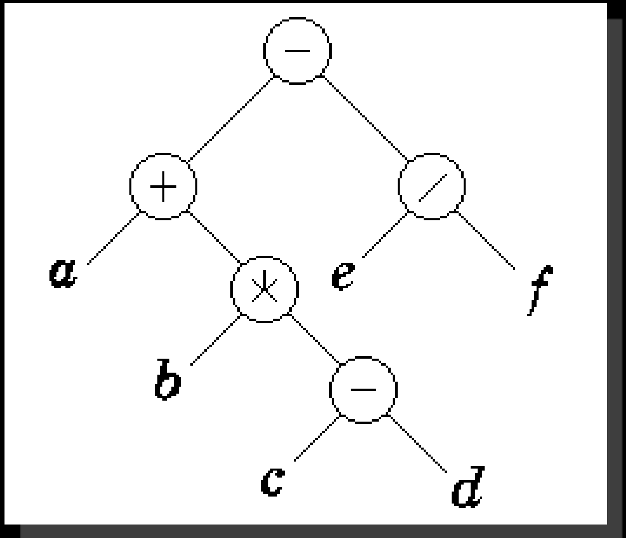
+	L
-	L

+	R
-	L



```
        int continue = 1;
        while ( continue && ! IsEmpty( S ) ) {
            w = getTop( S );   Pop( S );
            p = w.ptr;
            switch (w.tag) {
                case L :  w.tag = R;   Push( S, w);
                          continue = 0;
                          p = p->rchild;
                          break;
                case R :  printf (p->info);
                          break;
            }
        }
    } while ( p || !IsEmpty( S ) );
}
```

Stack state in InOrderTraversal
后序遍历中栈的变化



-	L

+	L
-	L

a	L
+	L
-	L

a	R
+	L
-	L

a	R
+	L
-	L

+	L
-	L

+	R
-	L

b	L
*	L
+	R
-	L

b	R
*	L
+	R
-	L

b	R
*	L
+	R
-	L

*	R
+	R
-	L

-	L
*	R
+	R
-	L

c	L
-	L
*	R
+	R
-	L

c	R
-	L
*	R
+	R
-	L

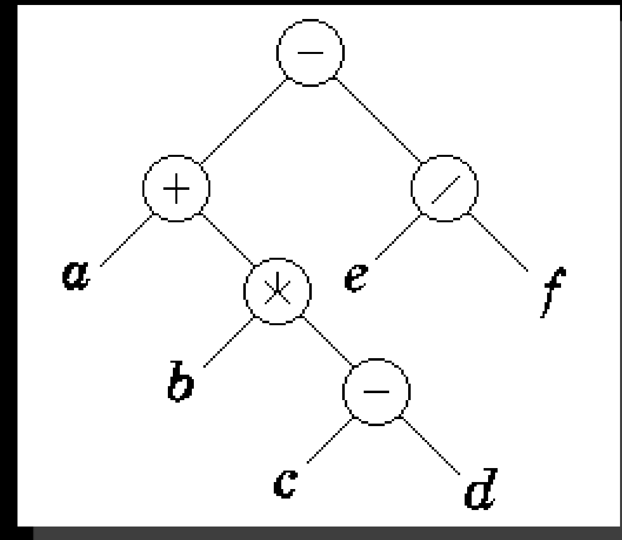
c	R
-	L
*	R
+	R
-	L

-	L
*	R
+	R
-	L

-	R
*	R
+	R
-	L

d	L
-	R
*	R
+	R
-	L

Stack state in InOrderTraversal
后序遍历中栈的变化



d	R
-	R
*	R
+	R
-	L

d	R
-	R
*	R
+	R
-	L

-	R
*	R
+	R
-	L

*	R
+	R
-	L

+	R
-	L

-	R

/	L
-	R

e	L
/	L
-	R

e	R
/	L
-	R

e	R
/	L
-	R

/	L
-	R

/	R
-	R

f	L
/	R
-	R

f	R
/	R
-	R

f	R
/	R
-	R

/	R
-	R

-	R

栈空

6. 二叉树的层次遍历

所谓二叉树的层次遍历，是指从二叉树的第一层（根结点）开始，从上至下逐层遍历，在同一层中，则按从左到右的顺序对结点逐个访问。

在进行层次遍历时，可设置一个队列结构，遍历从二叉树的根结点开始，首先将根结点指针入队列，然后从队头取出一个元素，每取一个元素，执行下面两个操作：

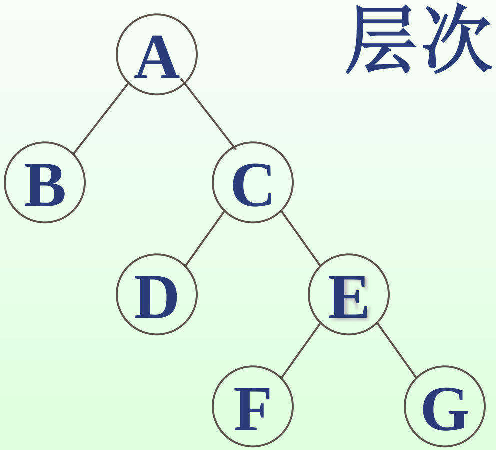
（1）访问该元素所指结点；

（2）若该元素所指结点的左、右孩子结点非空，则将该元素所指结点的左孩子指针和右孩子指针顺序入队。

此过程不断进行，当队列为空时，二叉树的层次遍历结束。

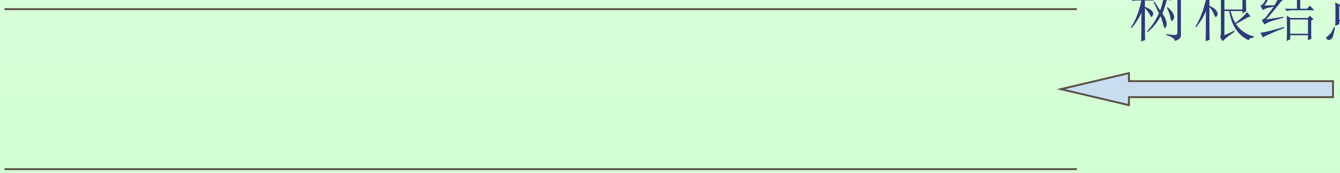
层次遍历

❿ 先根，后子树；先左子树，后右子树



层次遍历序列:

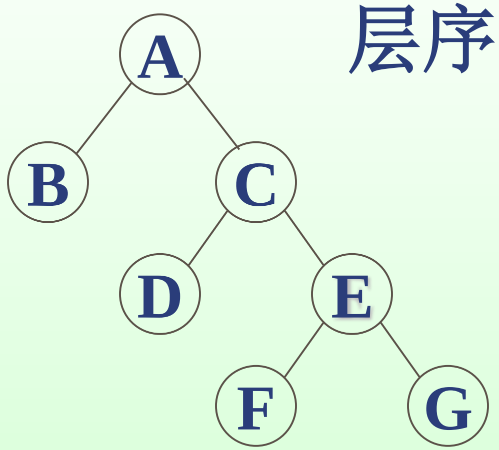
树根结点A入队



队头 队列

层次遍历

⑩ 先根，后子树；先左子树，后右子树



层序遍历序列:

A出队

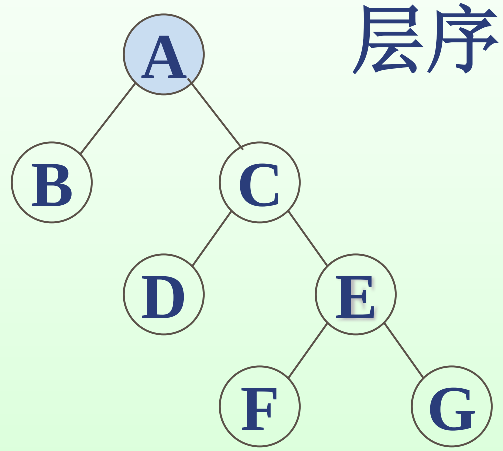
A

队头　　队列

层次遍历

- ⑩ 先根，后子树；先左子树，后右子树

层序遍历序列: A



A的子树

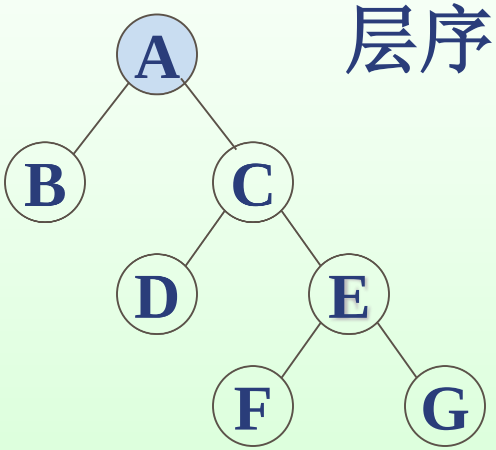
根结点B、C入队

队头

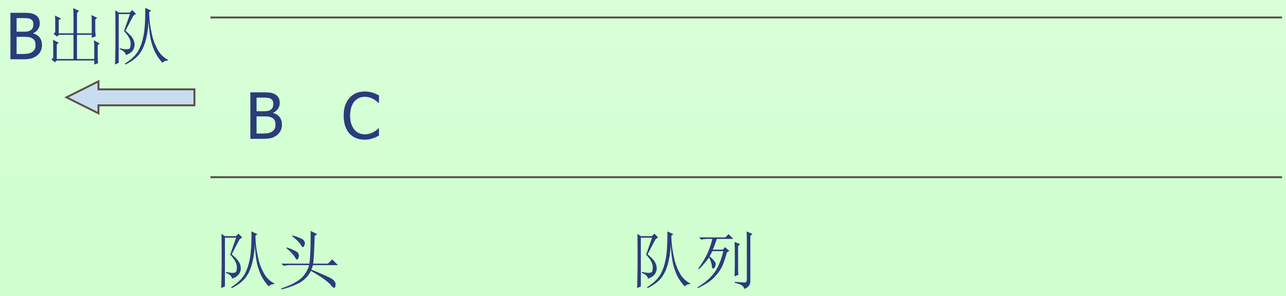
队列

层次遍历

⑩ 先根，后子树；先左子树，后右子树

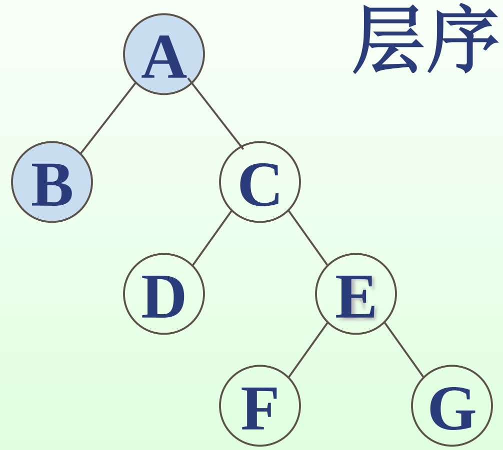


层序遍历序列: A

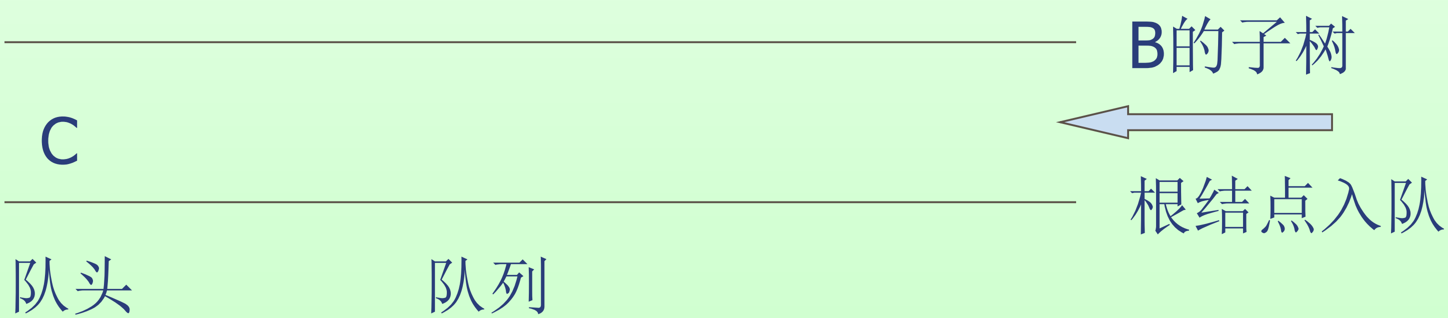


层次遍历

❿ 先根，后子树；先左子树，后右子树



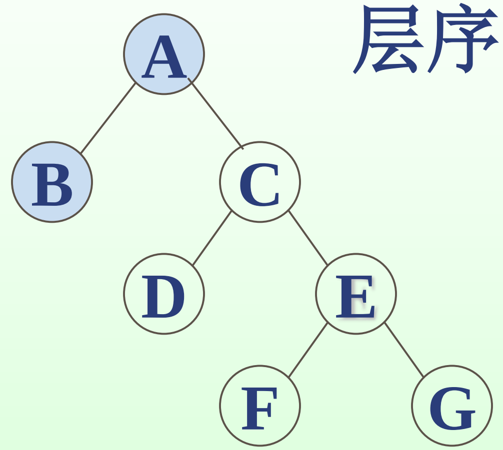
层序遍历序列: AB



层次遍历

⑩ 先根，后子树；先左子树，后右子树

层序遍历序列: AB



C出队

C

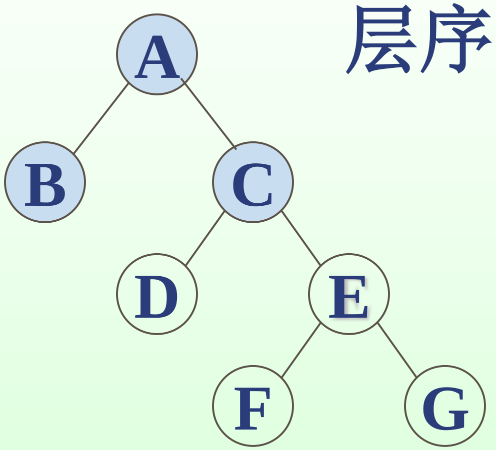
队头

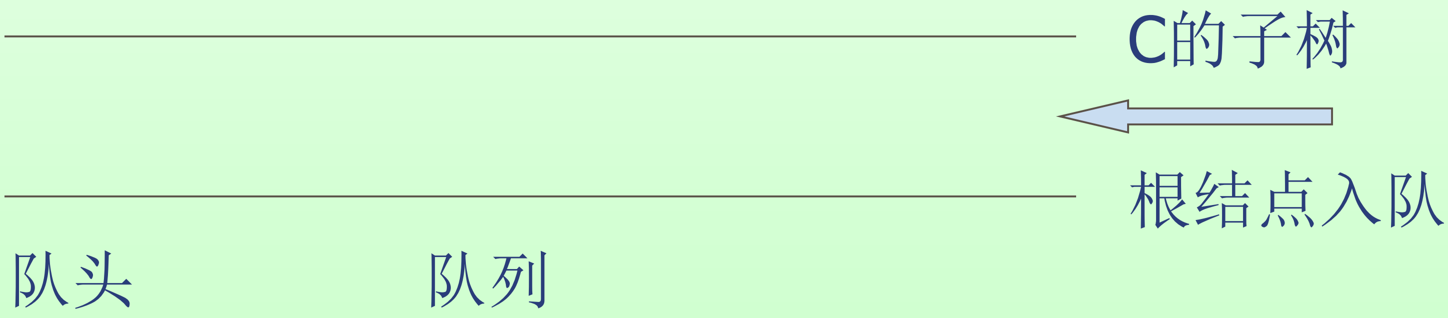
队列

层次遍历

⑩先根，后子树；先左子树，后右子树

层序遍历序列: ABC

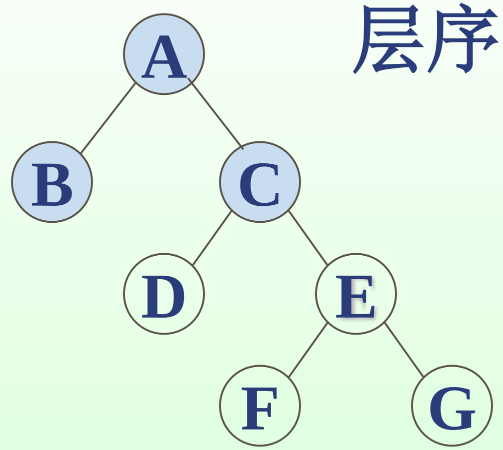


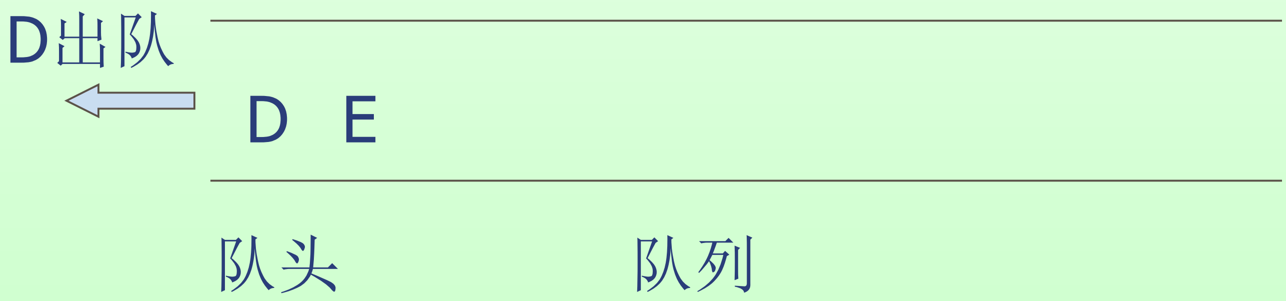


层次遍历

❿ 先根，后子树；先左子树，后右子树

层序遍历序列: ABC

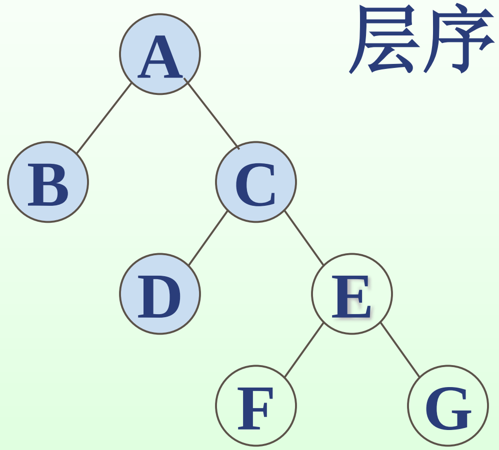




层次遍历

⑩ 先根，后子树；先左子树，后右子树

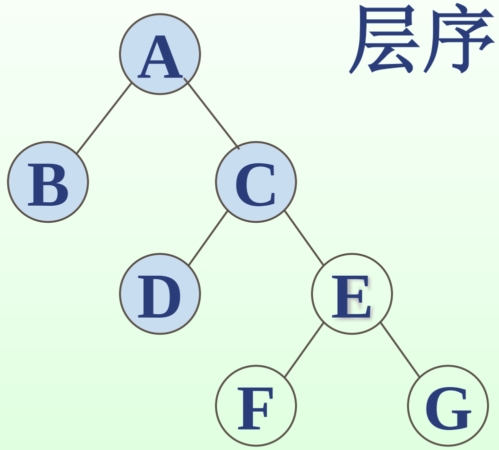
层序遍历序列: ABCD



层次遍历

❿ 先根，后子树；先左子树，后右子树

层序遍历序列: ABCD



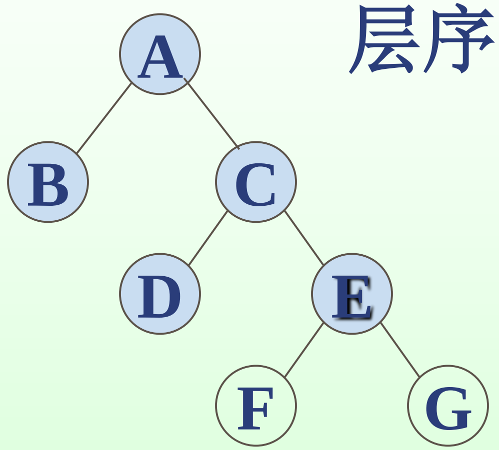
E出队

E

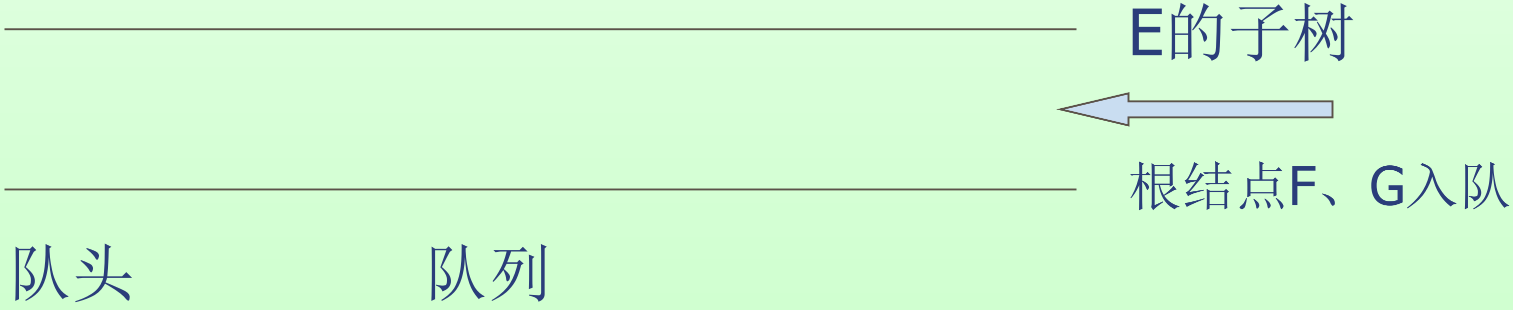
队头 队列

层次遍历

❿ 先根，后子树；先左子树，后右子树



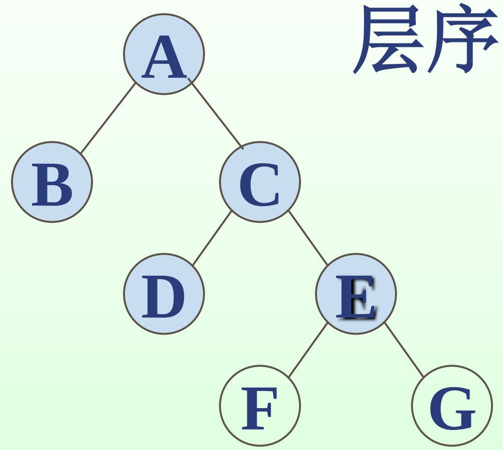
层序遍历序列: ABCDE



层次遍历

⑩ 先根，后子树；先左子树，后右子树

层序遍历序列: ABCDE



F出队

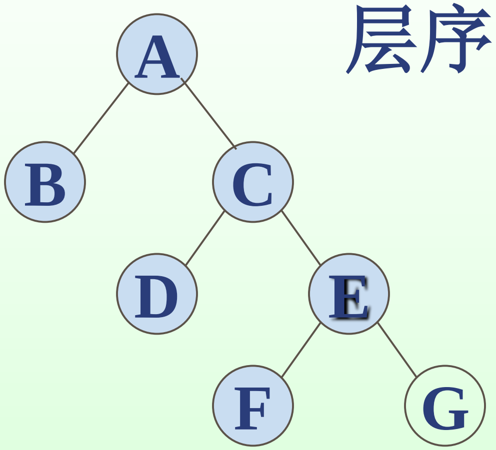
F G

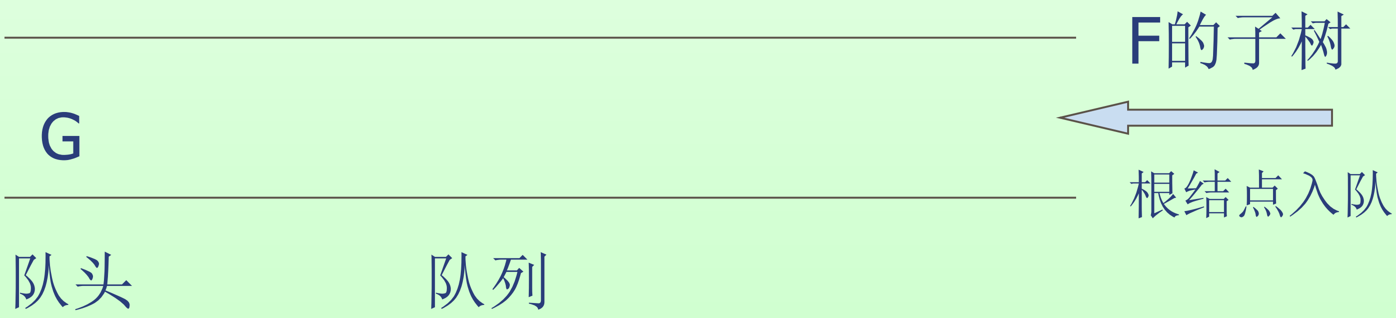
队头 队列

层次遍历

⑩ 先根，后子树；先左子树，后右子树

层序遍历序列: ABCDEF

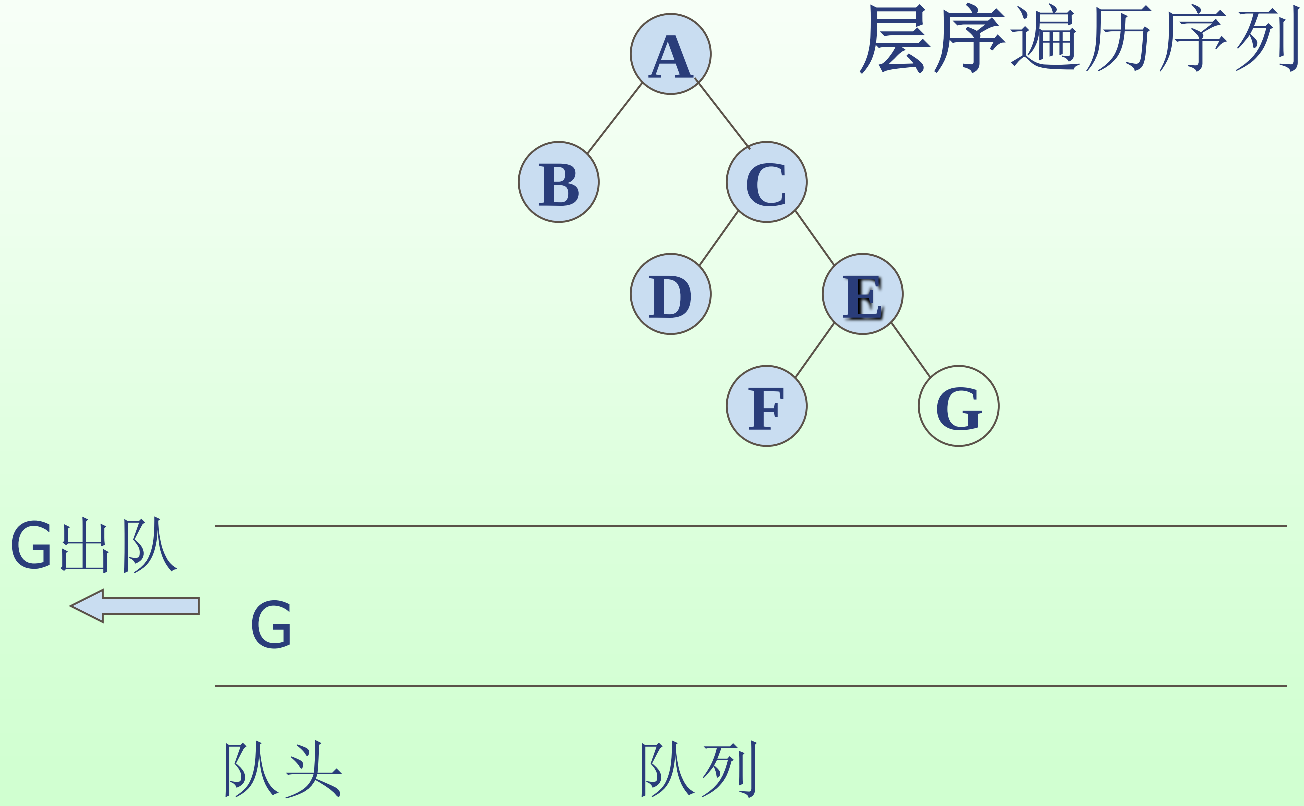




层次遍历

❿ 先根，后子树；先左子树，后右子树

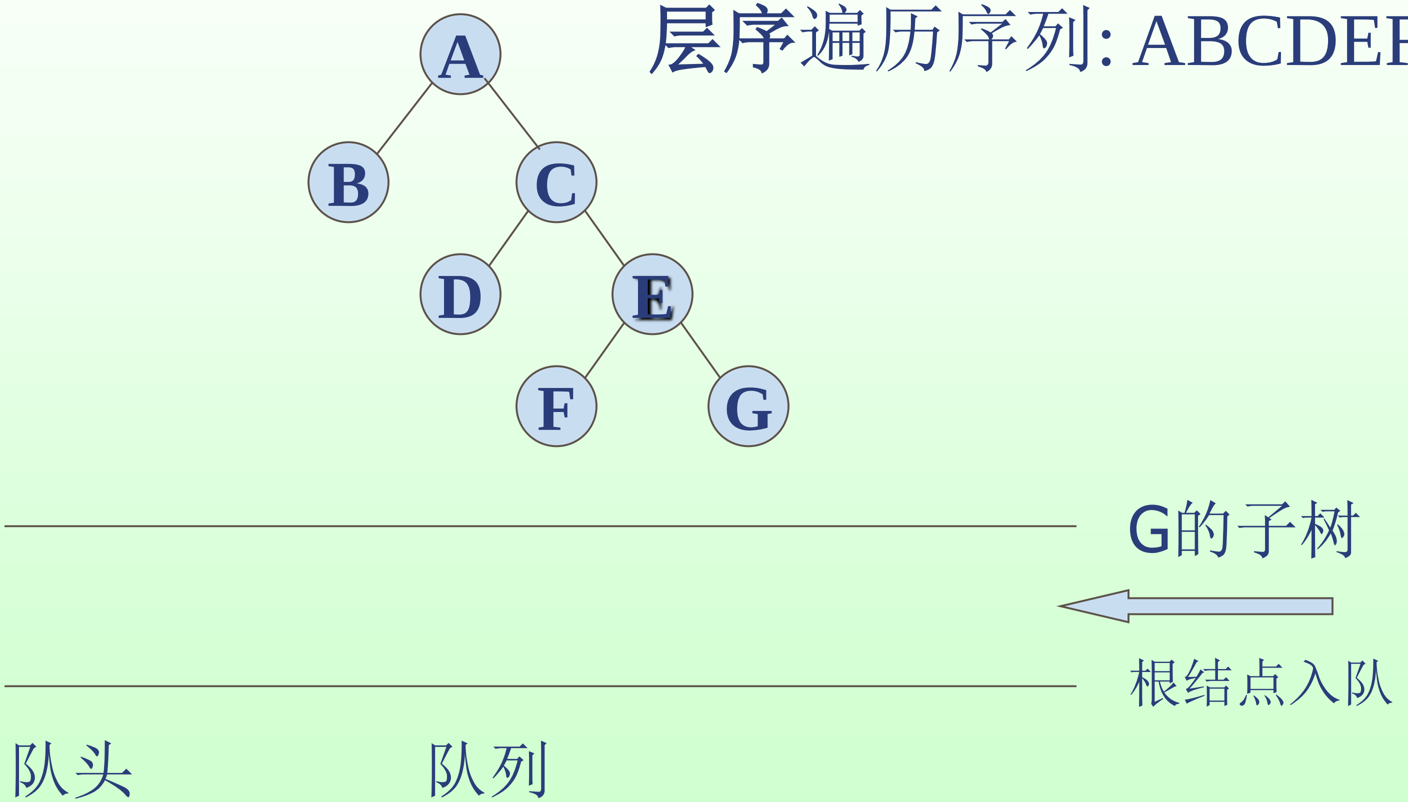
层序遍历序列: ABCDEF



层次遍历

❿ 先根，后子树；先左子树，后右子树

层序遍历序列: ABCDEFG



层次遍历二叉树

```
void LevelOrder（BiTree  bt）
 { BiTree  Queue[MAXNODE];   //一维数组用以实现队列
     int front,rear;       //队首元素和队尾元素
     if (bt = = NULL) return;
     front= -1;   rear=0;
     queue[rear]= bt;
     while(front!=rear)
        {front++;
         Visit ( queue[front]->data );      /*访问队首结点的数据域*/
         if (queue[front]->lchild!=NULL)     /*左孩子结点入队*/
          { rear++;
            queue[rear]= queue[front]->lchild;          }
         if (queue[front]->rchild!=NULL)   /*右孩子结点入队列*/
          { rear++;
            queue[rear]=queue[front]->rchild;          }
        }
}
```

7. 建立二叉树

由结点序列恢复二叉树

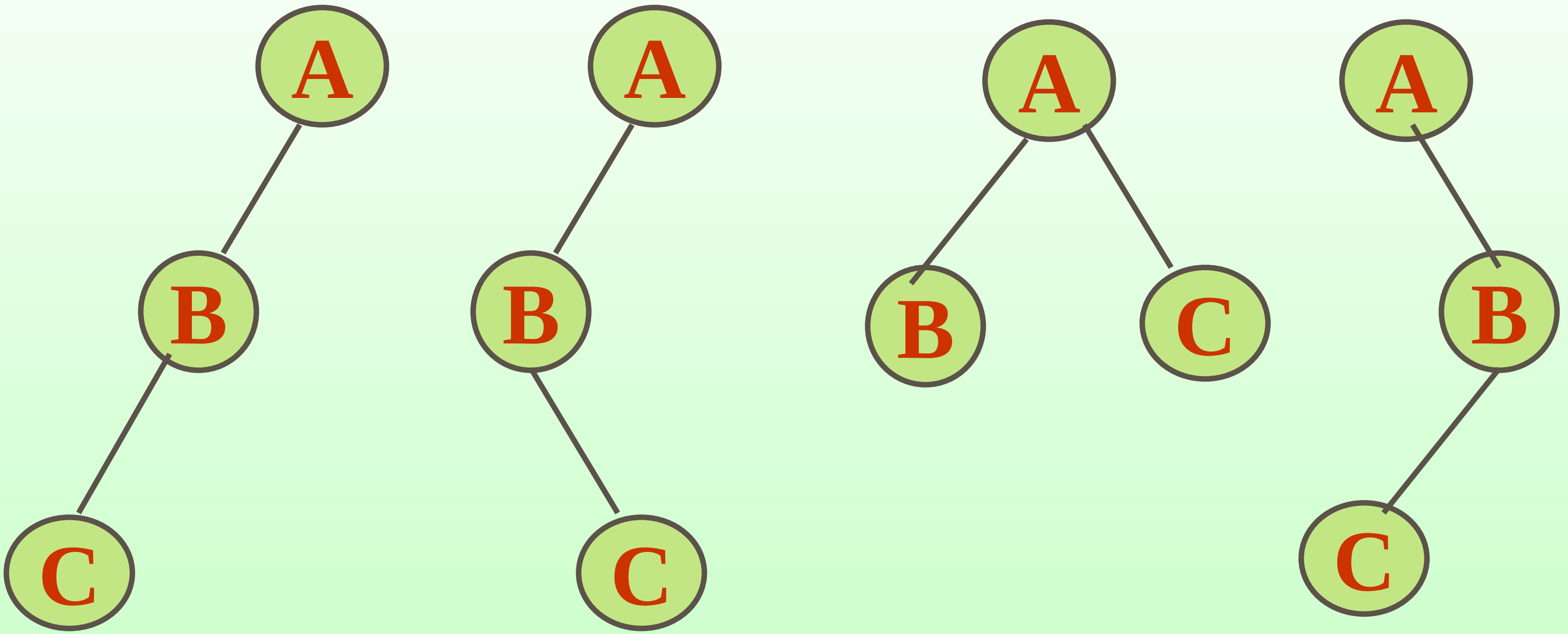
在对一棵二叉树进行遍历，只要遍历的策略已确定，就可以得到一个唯一的结点序列。

那么，给定一个遍历的结点序列，能否唯一的确定一棵二叉树？

答案：不能！

结点序列确定二叉树的不唯一性

先序序列：A B C



关于二叉树的先序、中序和后序遍历序列确定二叉树的问题。

● 任何n(n≥0)个不同结点的二叉树,都可由它的中序序列和先序序列唯一地确定。

证明:

先序序列是$a_1a_2\ldots a_n$
中序序列是$b_1b_2\ldots b_n$
根结点:a_1。
在中序序列中与a1相同的结点为:b_j。

$$\{ b_1\ldots b_{j-1}\}b_j\{b_{j+1}\ldots b_n\} \longleftrightarrow a_1\{a_2\ldots a_k\}\{a_{k+1}\ldots a_n\}$$

●任何n(n＞0)个不同结点的二叉树,都可由它的中序序列和后序序列唯一地确定。

证明：

后序序列是$a_1a_2\ldots a_n$
中序序列是$b_1b_2\ldots b_n$
根结点：a_n。
在中序序列中与a_n相同的结点为：b_j。

$$\{ b_1\ldots b_{j-1}\}b_j\{b_{j+1}\ldots b_n\} \longleftrightarrow \{a_1a_2\ldots a_k\}\{a_{k+1}\ldots a_{n-1}\}a_n$$

由结点序列恢复二叉树

- 可恢复二叉树的结点序列组合：

  （1）先序序列和中序序列

  （2）中序序列和后序序列

- 不可恢复二叉树的结点序列组合：

  先序序列和后序序列

由先序和中序序列恢复二叉树

二叉树的先序序列 根 左子树 右子树

二叉树的中序序列 左子树 根 右子树

由先序和中序序列恢复二叉树举例

先序序列： A B C D E F G

中序序列： C B D A E G F

左子树 右子树