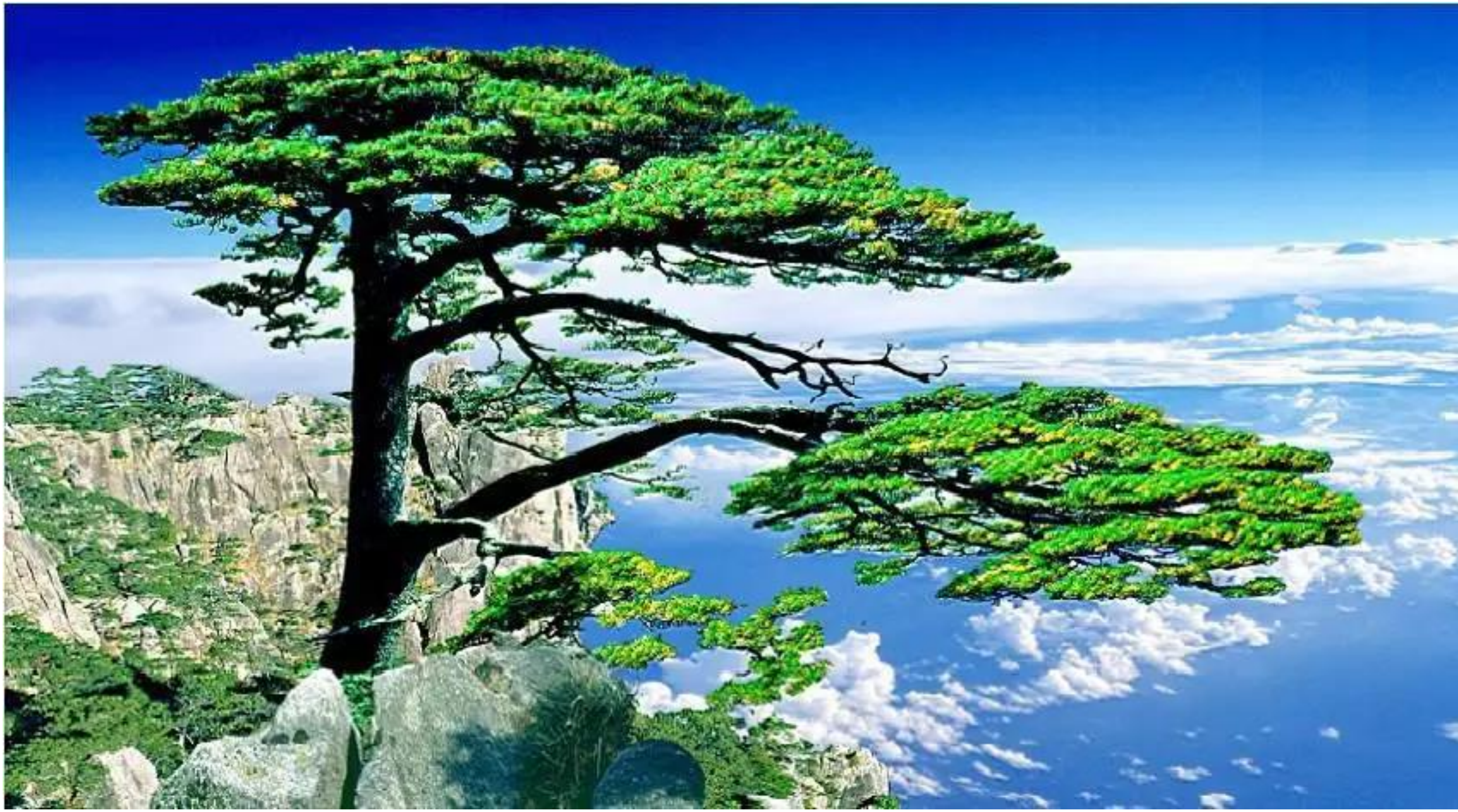
第6章 树和二叉树



第6章 树和二叉树

6.1 树的定义和基本术语

6.2 二叉树

6.3 遍历二叉树和线索二叉树

6.4 树和森林

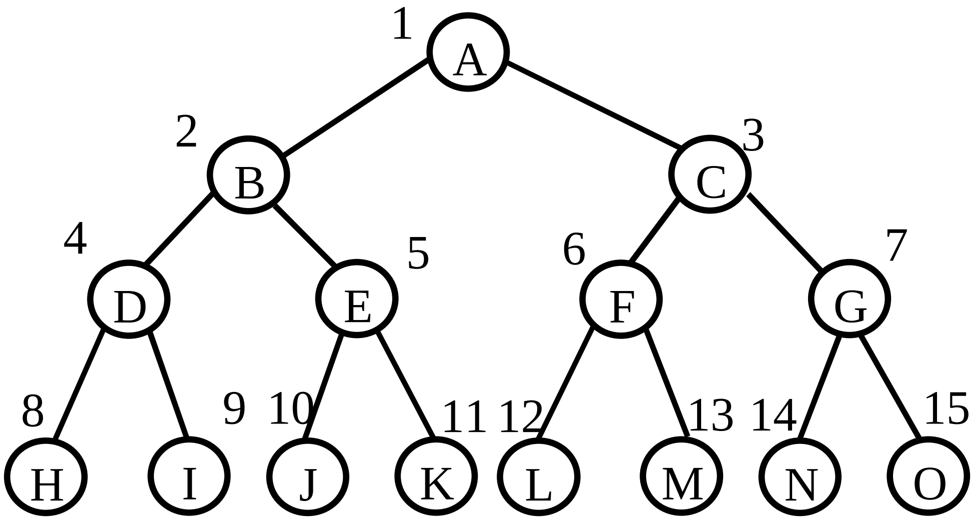
6.5 哈夫曼树及其应用

回顾

度≤2：每个结点最多只有两棵子树

有序树：子树有左右之分，其次序不能任意颠倒

满二叉树：深度为k且有2^k-1个结点，叶子节点在最底下一层，每一层节点数都最大



满二叉树：深度为k，有n个结点的二叉树，当且仅当其每一个结点的位置序号都与深度为k的满二叉树的结点编号一一对应时，成为完全二叉树

回顾

性质1： 在二叉树的第 i 层上至多有 2^{i-1} 个结点 $(i\geq 1)$

性质2： 深度为 k 的二叉树上至多含 2^k-1 个结点（k≥1）

性质3： 对任何一棵二叉树，若它含有 n_0 个叶子结点（0度节点）、n_2 个度为 2 的结点，则必存在关系式： $n_0 = n_2+1$

性质4： 具有n个(n > 0)结点的完全二叉树的s深度为 $\lfloor log_2 n \rfloor+1$

性质5： 若对含 n 个结点的完全二叉树从上到下且从左至右进行 1 至 n 的编号，则对完全二叉树中任意一个编号为 i 的结点：

(1) 若 $i=1$，则该结点是二叉树的根；否则，编号为 $\lfloor i/2 \rfloor$ 的结点为其**双亲**结点；

(2) 若 $2i>n$，则该结点无左孩子；否则，编号为 $2i$ 的结点为其**左孩子**结点；

(3) 若 $2i+1>n$，则该结点无右孩子；否则，编号为 $2i+1$ 的结点为其**右孩子**结点。

回顾

例：请计算完全二叉树双亲节点、孩子节点及所在层次

i=7，n=12

双亲节点：$\lfloor i/2 \rfloor=3$

2i>成立：无孩子节点

所在层次：$\lfloor \log_2 i \rfloor+1=3$

i=5，n=12

双亲节点：$\lfloor i/2 \rfloor$

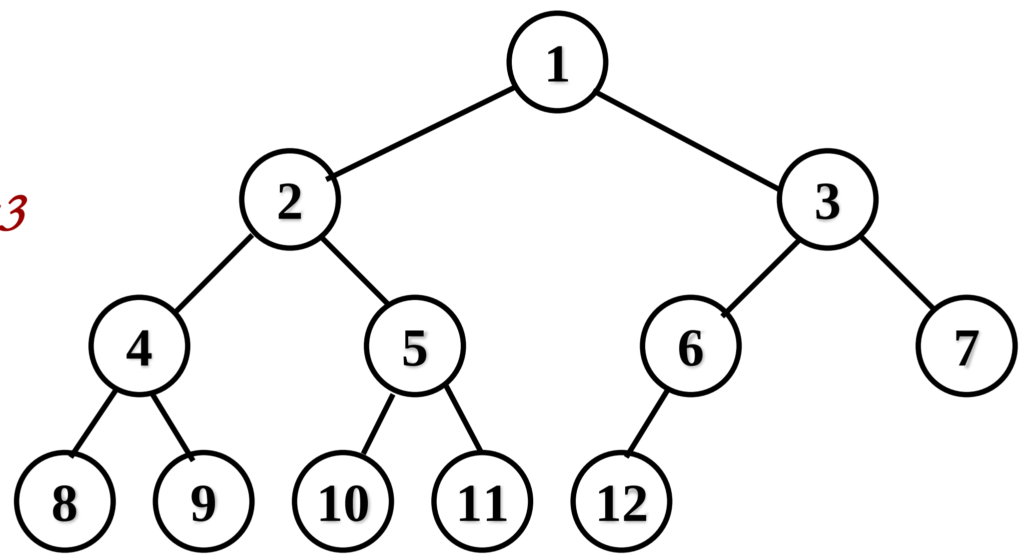
2i>n不成立： $=2$

左孩子：2i=10

2i+1>n不成立：

右孩子：2i+1=11

所在层次：$\lfloor \log_2 i \rfloor+1=3$



回顾

例：一棵完全二叉树有1000个结点，则它必有______个叶子结点，有______个度为2的结点，有______个结点只有非空左子树，有______个结点只有非空右子树。

分析题意：已知n=1000,求n_0和n_2,还要判断末叶子是挂在左边还是右边?

请注意：叶子结点总数≠末层叶子数！！！

深度$\lfloor \log_2 n \rfloor+1=10$

前9层$2^k-1=511$

第10层叶子节点1000-511=489

第9层叶子节点256-（244+1）=11

$n_0 = n_2+1$

正确答案：

全部叶子数=489+11=500个。

度为2的结点＝叶子总数－1=499个。

最后一结点为2i属于左叶子，右叶子是空的，所以有1个非空左子树。完全二叉树的特点决定不可能有左空右不空的情况，所以非空右子树数＝0。

回顾

思考

在k叉树中，每个节点最多有k个孩子。其子节点分别称为该节点的第一个，第二个…，第k个孩子。

① 包含n (n> 0)个元素的k叉树边数?

② 若k叉树的深度为h，h≥0，则该k叉树最少有? 个元素，最多有? 个元素。

③ 包含n个元素的k叉树的深度最大为? ，最小为?

6.2.3 二叉树的存储结构

二叉树的顺序存储结构

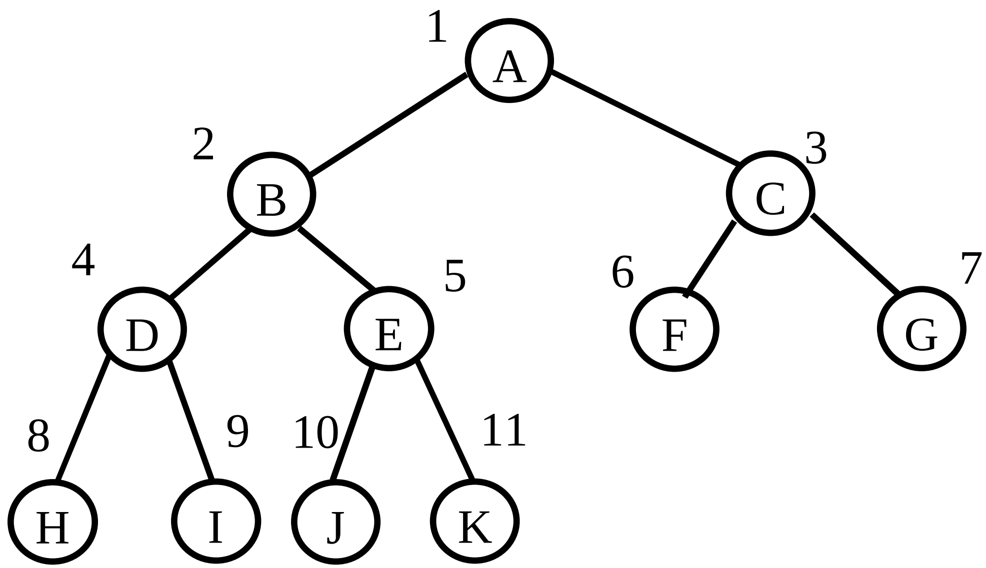
二叉树的链式存储结构

二叉树的顺序存储结构

- 用**数组**存储二叉树中的各个数据元素
- **存放次序：**对该树中每个结点进行**编号**，其编号从小到大的顺序就是结点存放在连续存储单元的先后次序。
- 若把二叉树存储到一维数组中，**编号就是下标值加1**(C/C++语言中数组的起始下标为0)。

二叉树的顺序存储结构

完全二叉树的顺序存储

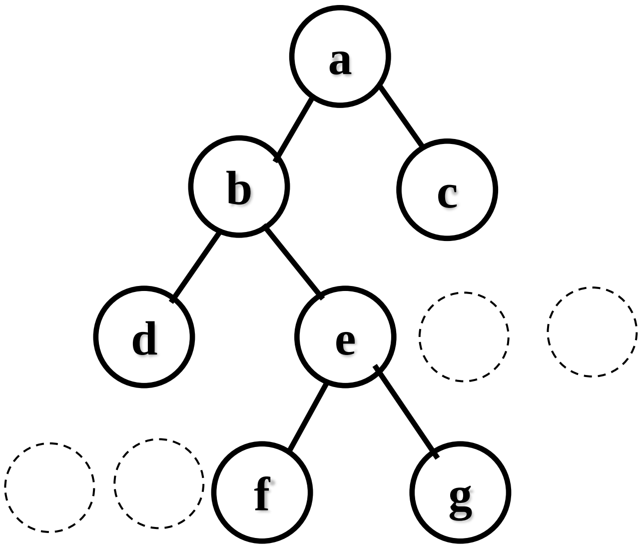


1	2	3	4	5	6	7	8	9	10	11
A	B	C	D	E	F	G	H	I	J	K

二叉树的顺序存储结构

非完全二叉树的顺序存储

- 对于一般的二叉树，可以参照完全二叉树的编码方法进行编码，位置空的结点置空



1	2	3	4	5	6	7	8	9	10	11
a	b	c	d	e	0	0	0	0	f	g

- 特点：
  - 结点间关系蕴含在其存储位置中
  - 浪费空间，适于存满二叉树和完全二叉树

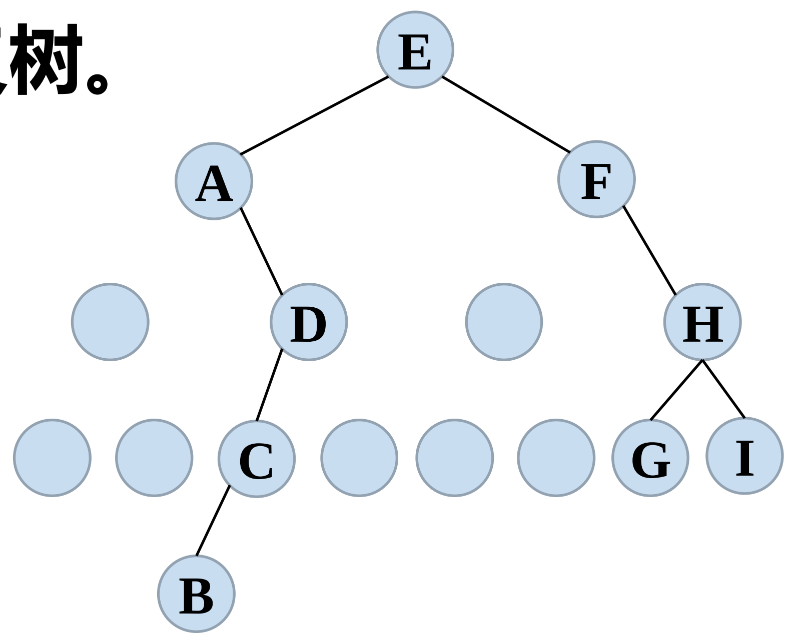
```
#define MAX_TREE_SIZE  100
typedef TElemType SqBiTree[MAX_TREE_SIZE];
SqBiTree   bt;
```

二叉树的顺序存储结构

- **某二叉树的结点数据采用顺序存储结构如下：**

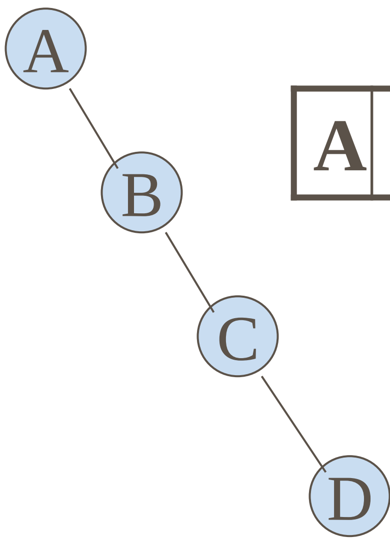
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
E	A	F		D		H			C				G	I					B

画出该二叉树。



二叉树的顺序存储结构

对于一般的二叉树，我们必须按照完全二叉树的形式来存储，就会造成空间的浪费。单支树就是一个极端情况。



单支树

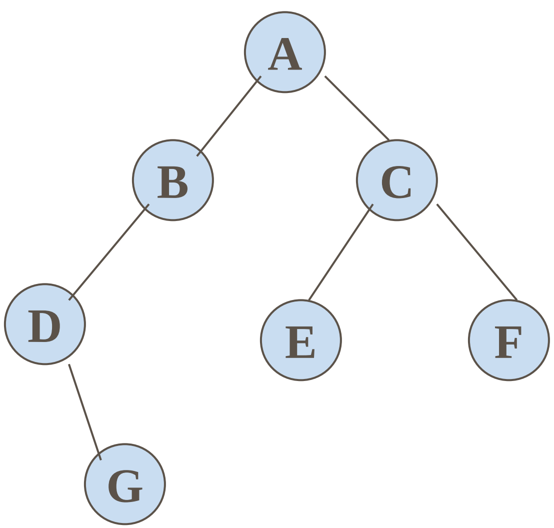
A	∧	**B**	∧	∧	∧	**C**	∧	∧	∧	∧	∧	∧	∧	**D**

二叉树的链式存储结构

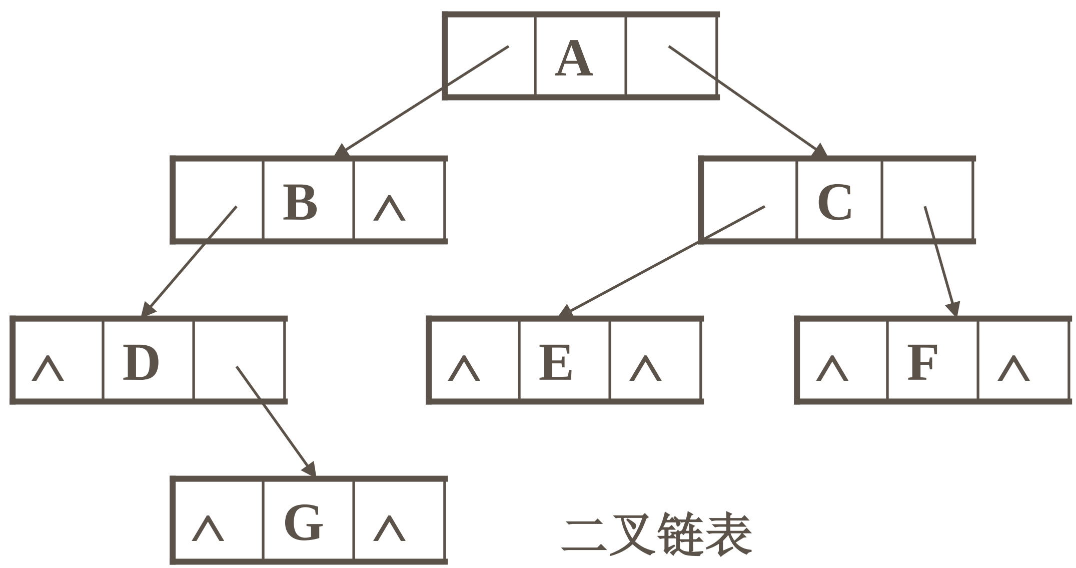
对于任意的二叉树来说，每个结点只有两个孩子，一个双亲结点。我们可以设计每个结点至少包括三个域：数据域、左孩子域和右孩子域：

二叉链表 →

LChild	Data	RChild



二叉树T



二叉链表

二叉树的链式存储结构

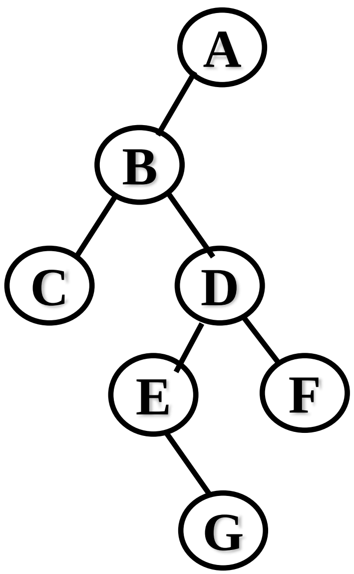
- **在二叉树的链接存储中，结点的结构如下:**

```
typedef struct node
{
    ElemType data;
    struct BiTNode *lchild,*rchild;
} BTNode，*BiTree ;
```

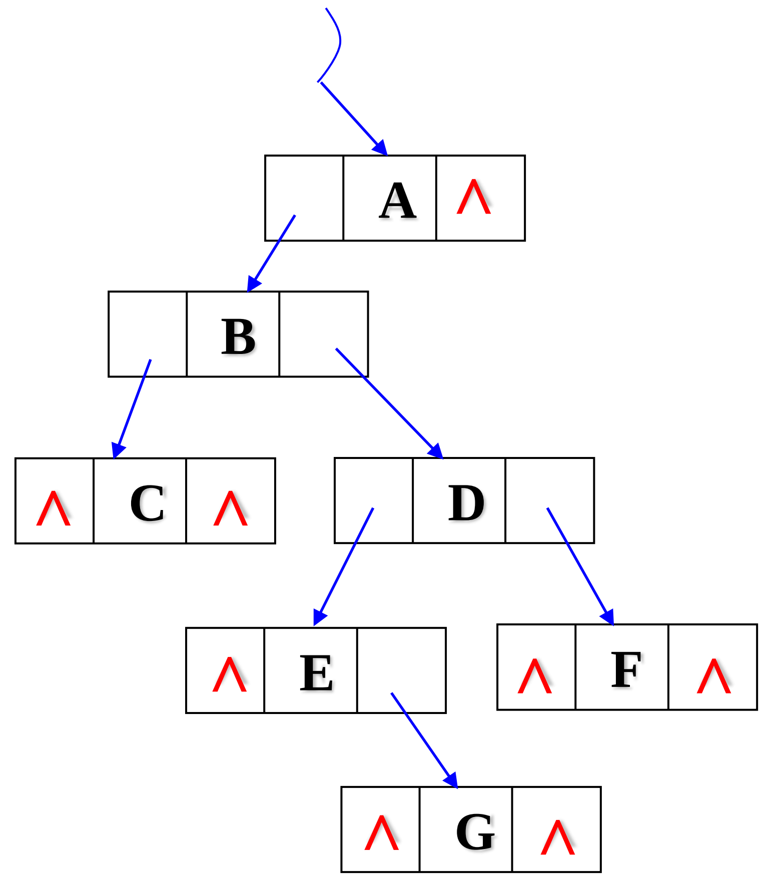
- **其中，data表示值域，用于存储对应的数据元素，lchild和rchild分别表示左指针域和右指针域，用于分别存储左孩子结点和右孩子结点的存储位置。**

lchild	data	rchild

二叉树的链式存储结构



二叉链表优缺点？？？



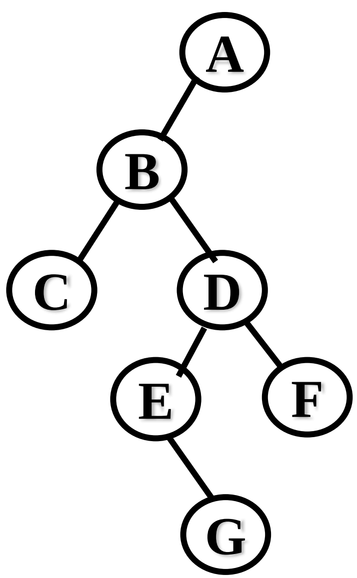
注意：在n个结点的二叉链表中，有n+1个空指针域: $2n-(n-1)=n+1$

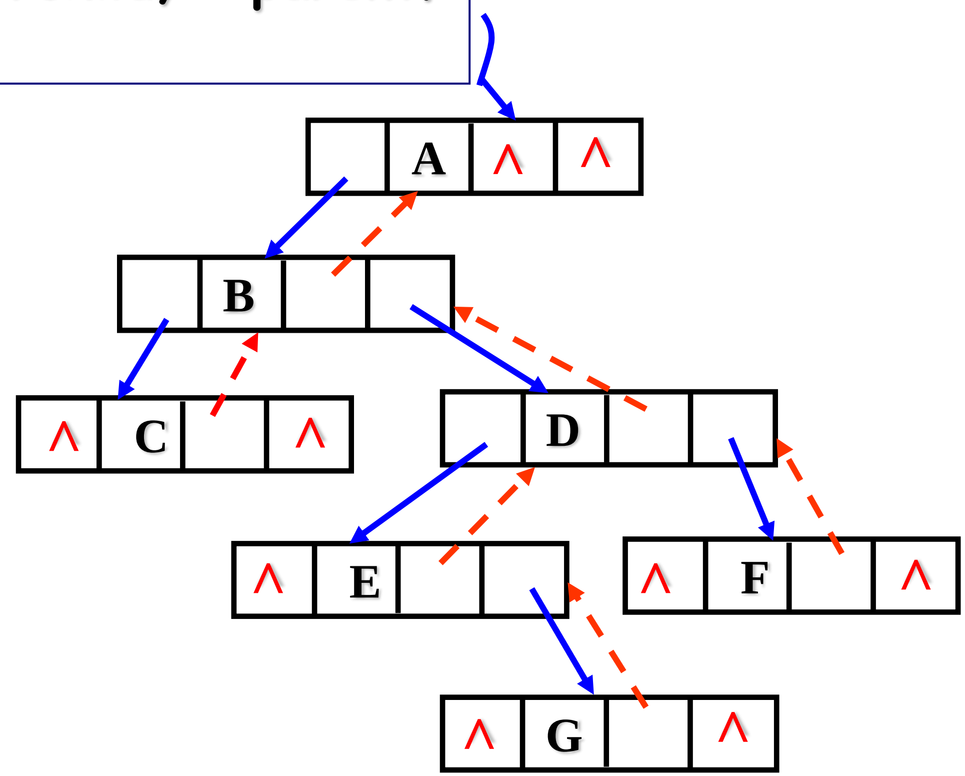
二叉树的链式存储结构

三叉链表

lchild	data	parent	rchild

```
typedef   struct   node
{   datatype   data;
    struct node   *lchild,   *rchild,   *parent;
}JD;
```

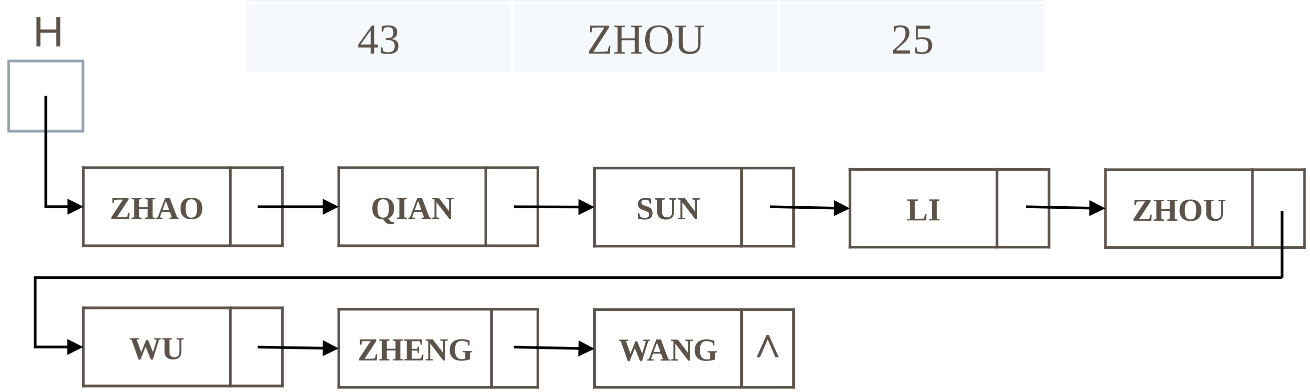




二叉树的链式存储结构

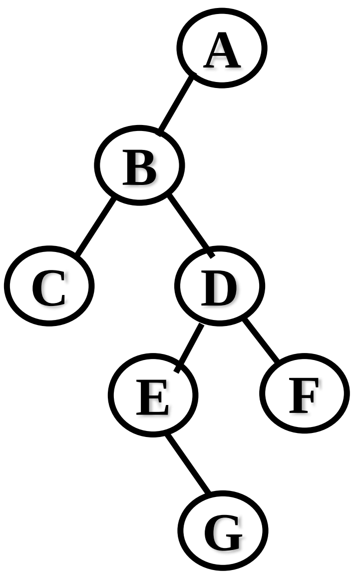
静态链表： (ZHAO, QIAN, SUN, LI, ZHOU, WU, ZHENG, WANG)

存储地址	数据域	指针域
1	LI	43
7	QIAN	13
13	SUN	1
19	WANG	NULL
25	WU	37
31	ZHAO	7
37	ZHENG	19
43	ZHOU	25



二叉树的链式存储结构

静态二叉链表和静态三叉链表



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

预先开辟空间，用数组表示

leftChild, rightChild——数组元素的下标

第6章 树和二叉树

6.1 树的定义和基本术语

6.2 二叉树

6.3 遍历二叉树和线索二叉树

6.4 树和森林

6.5 哈夫曼树及其应用

6.3.1 遍历二叉树

二叉树的遍历方法

二叉树的遍历是指从根结点出发，按照某种次序访问二叉树中的所有结点，使得每个结点被访问一次且仅被访问一次。

先序遍历
中序遍历
后序遍历
层序遍历

抽象操作，可以是对结点进行的各种处理，这里简化为输出结点的数据。

对于线性结构由于每个结点只有一个直接后继，遍历是很容易的事。

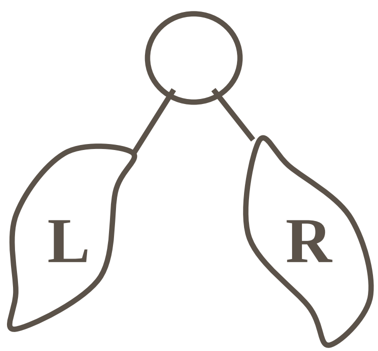
二叉树是非线性结构，每个结点可能有两个后继，如何访问二叉树的每个结点，而且每个结点仅被访问一次？

6.3.1 遍历二叉树

遍历

线性结构 —— 按照线性结构的逻辑顺序即可

非线性结构（比如二叉树） —— 寻找一个规律使二叉树上的结点能排列在一个线性队列上



一个二叉树由3部分组成：D（根），L(左子树)，R(右子树)

6.3.1 遍历二叉树

考虑二叉树的组成：

二叉树
- 根结点D
- 左子树L
- 右子树R

二叉树的遍历方式：

DLR、LDR、LRD、

DRL、RDL、RLD

如果限定先左后右，则二叉树遍历方式有三种：

前序：DLR

中序：LDR

后序：LRD

层序遍历：按二叉树的层序编号的次序访问各结点。

6.3.1 遍历二叉树

1、先序（根）遍历

若二叉树为空，则空操作返回；否则：

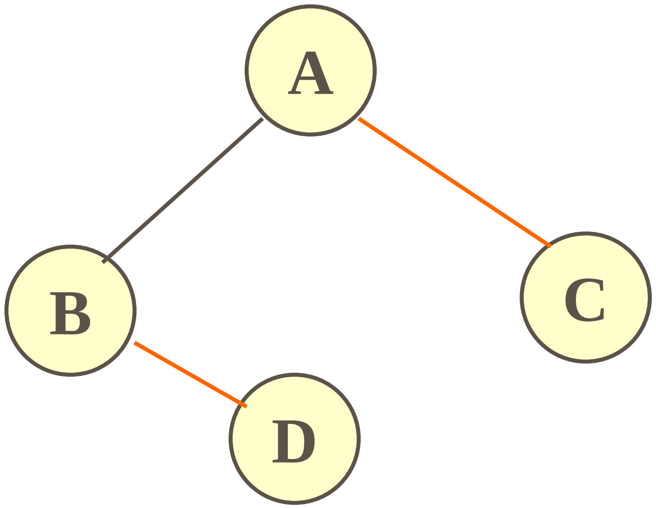
①访问根结点；

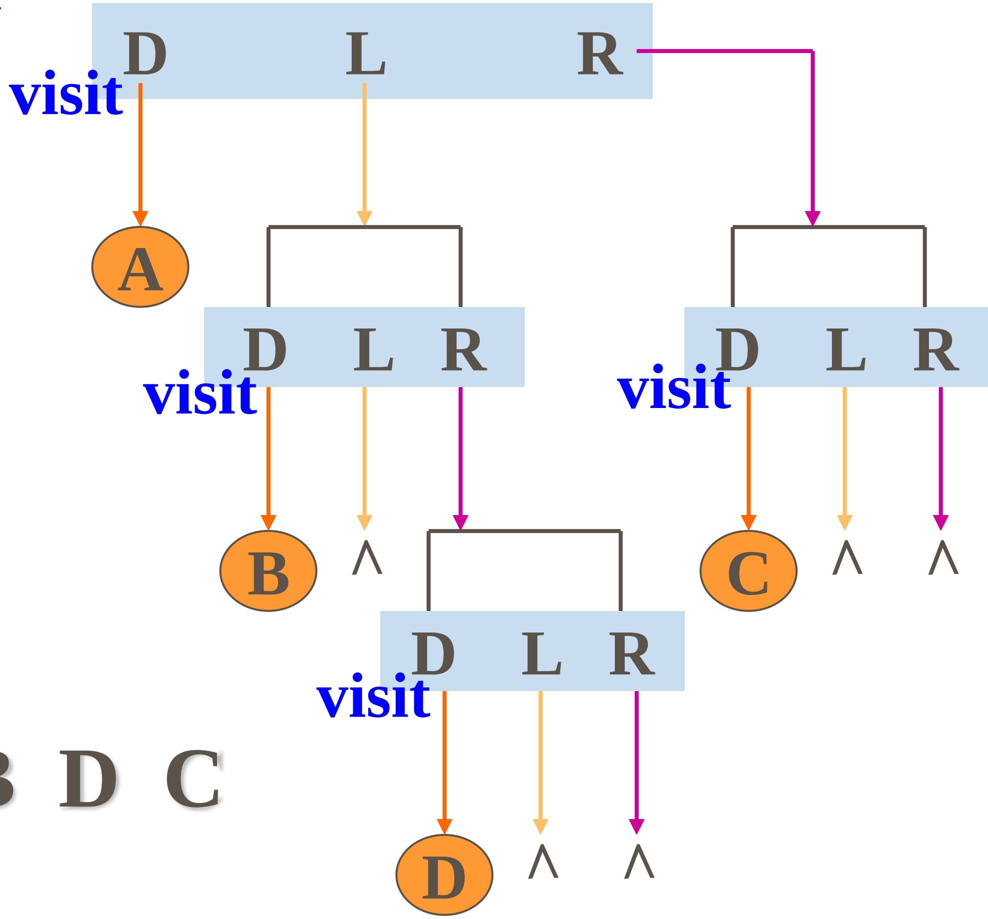
②先序遍历根结点的左子树；

③先序遍历根结点的右子树。

6.3.1 遍历二叉树

1、先序（根）遍历

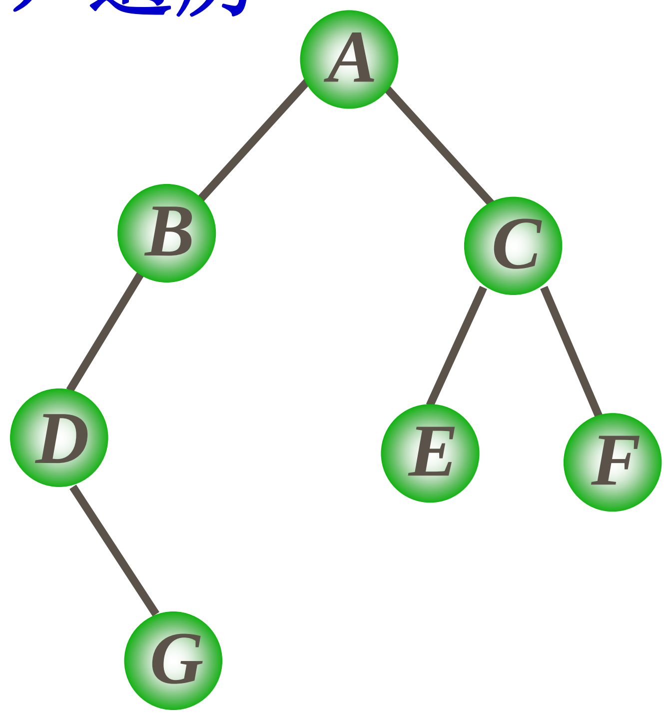




先序遍历序列：A B D C

6.3.1 遍历二叉树

1、先序（根）遍历



先序遍历序列：*A B D G C E F*

6.3.1 遍历二叉树

2、中序（根）遍历

若二叉树为空，则空操作返回；否则：

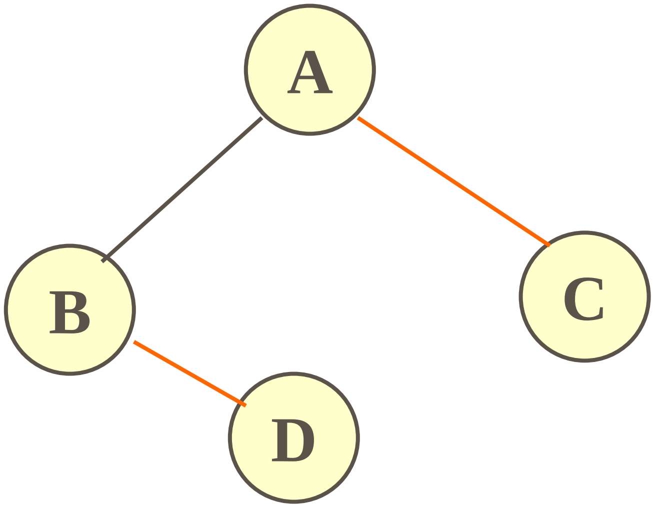
①中序遍历根结点的左子树；

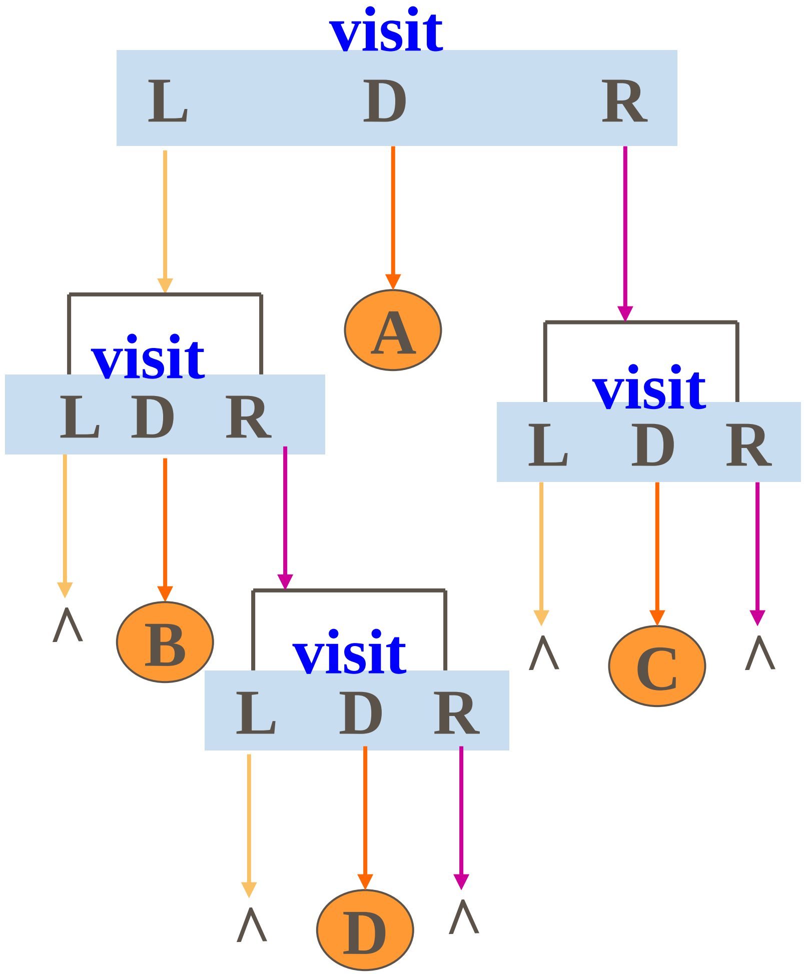
②访问根结点；

③中序遍历根结点的右子树。

6.3.1 遍历二叉树

2、中序（根）遍历

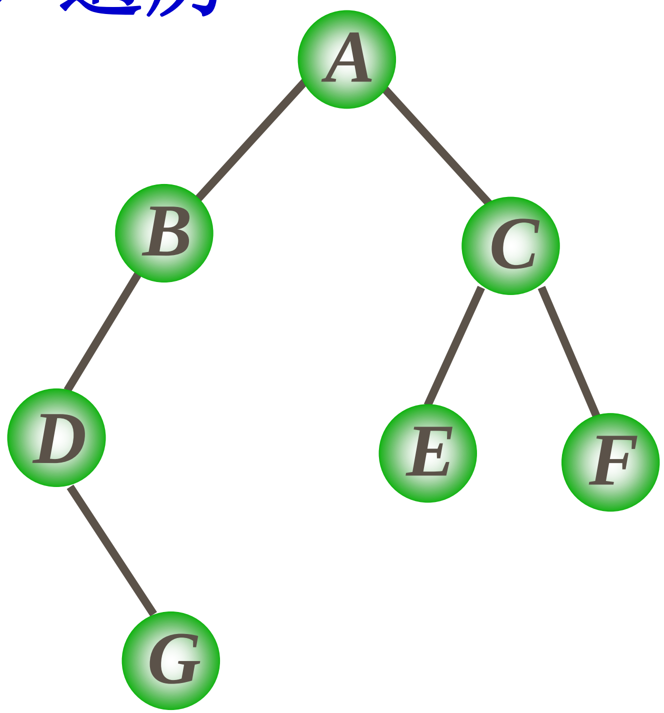




中序遍历序列：B D A C

6.3.1 遍历二叉树

2、中序（根）遍历



中序遍历序列：*D G B A E C F*

6.3.1 遍历二叉树

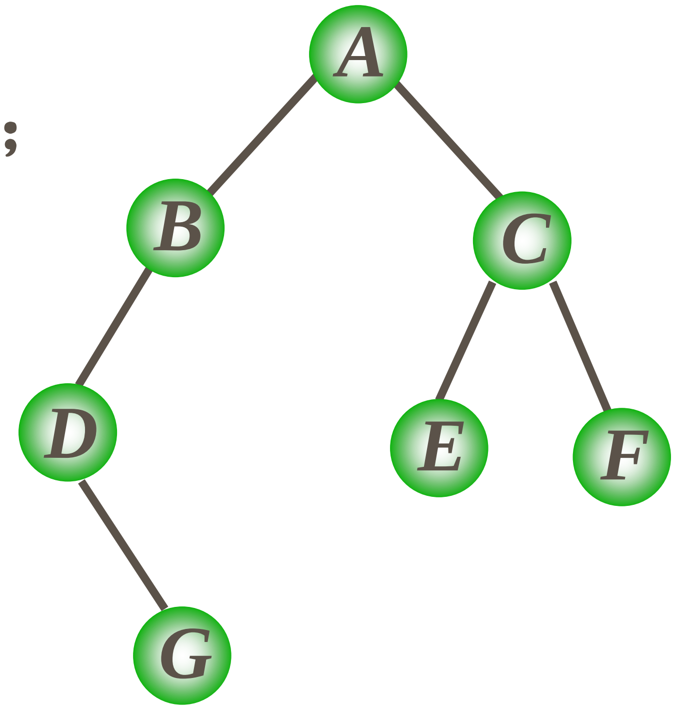
3、后序（根）遍历

若二叉树为空，则空操作返回；
否则：

①后序遍历根结点的左子树；

②后序遍历根结点的右子树；

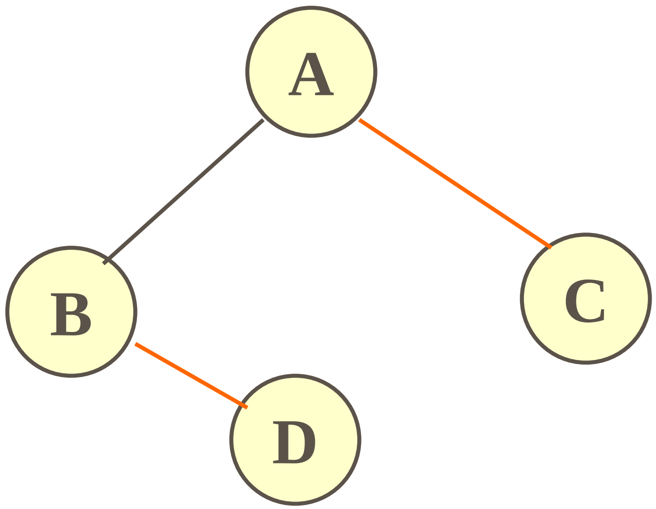
③访问根结点。

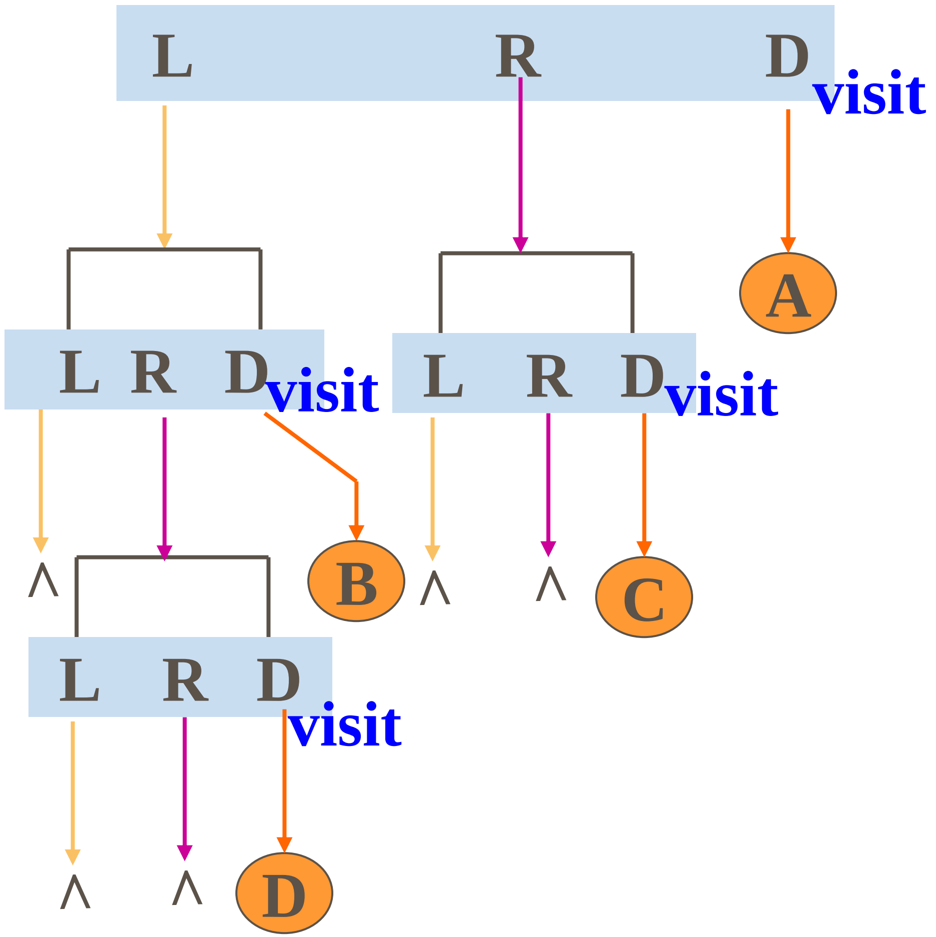


后序遍历序列：*G D B E F C A*

6.3.1 遍历二叉树

3、后序（根）遍历

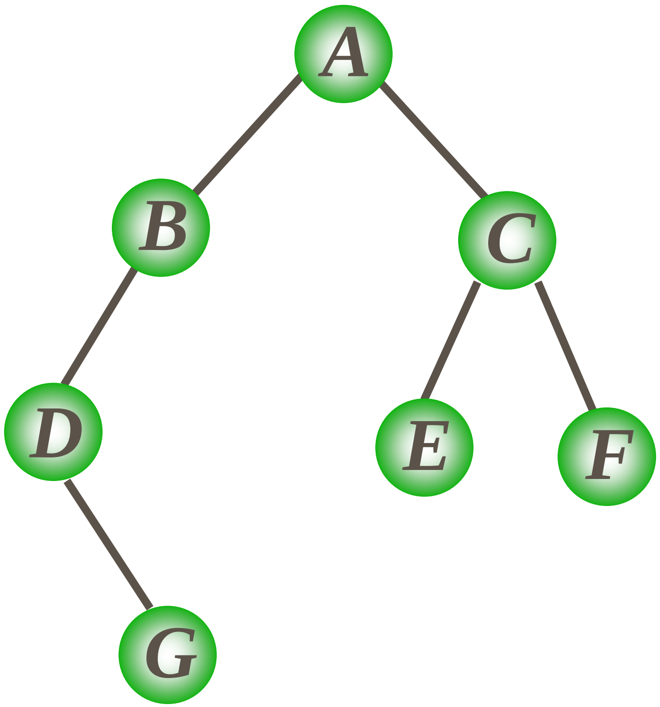




后序遍历序列： D B C A

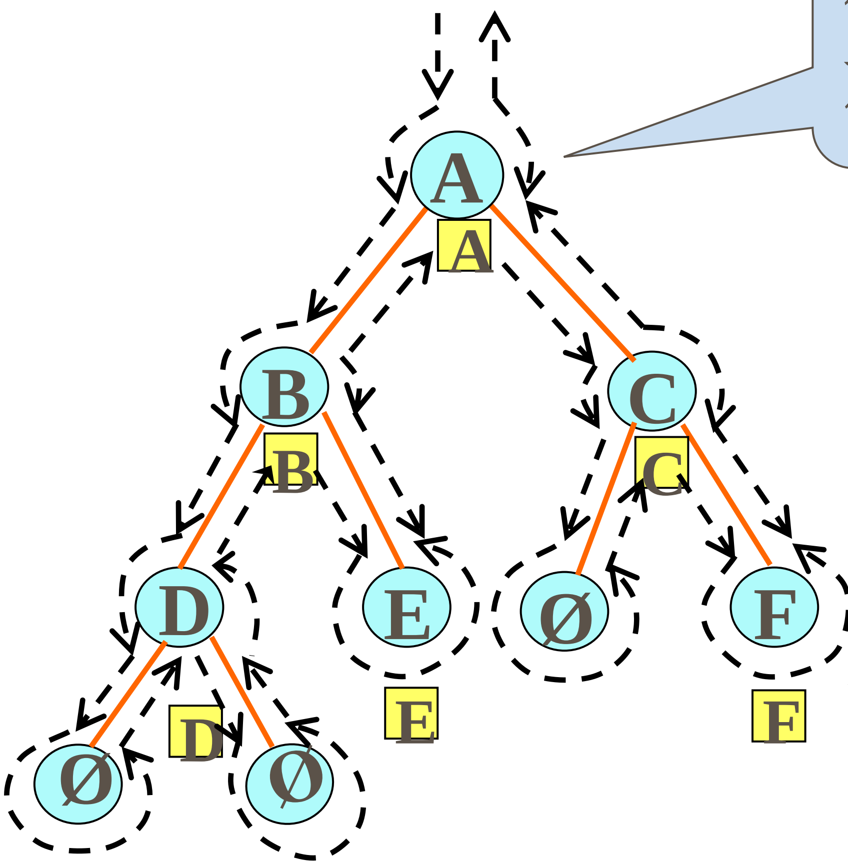
6.3.1 遍历二叉树

3、后序（根）遍历



后序遍历序列：*G D B E F C A*

6.3.1 遍历二叉树



对每个结点均途经了三次，去掉三种遍历中与递归无关的visit语句，则三种遍历算法完全相同。

从虚线的出发点到终点的路径上，每个结点经过3次。

第1次经过时访问，是先序遍历

第2次经过时访问，是中序遍历

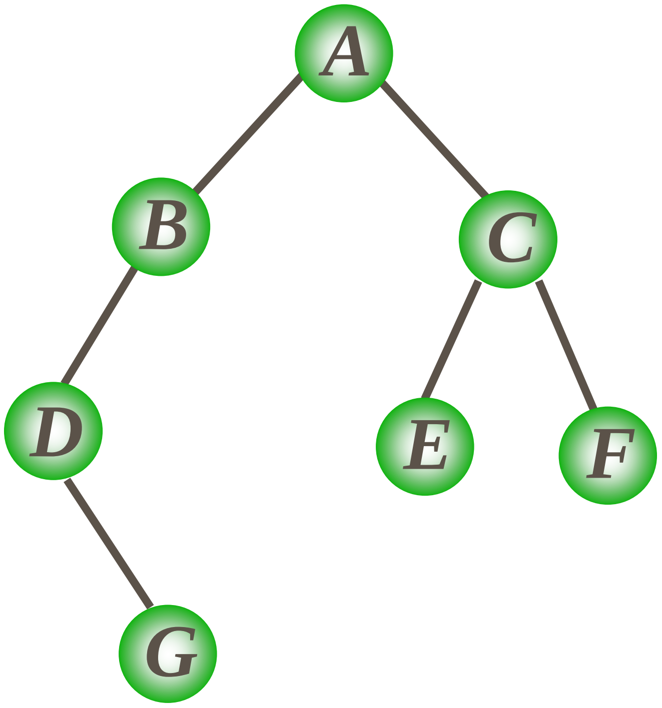
第3次经过时访问，是后序遍历

中序遍历结果为
DBEACF

6.3.1 遍历二叉树

4、层序遍历

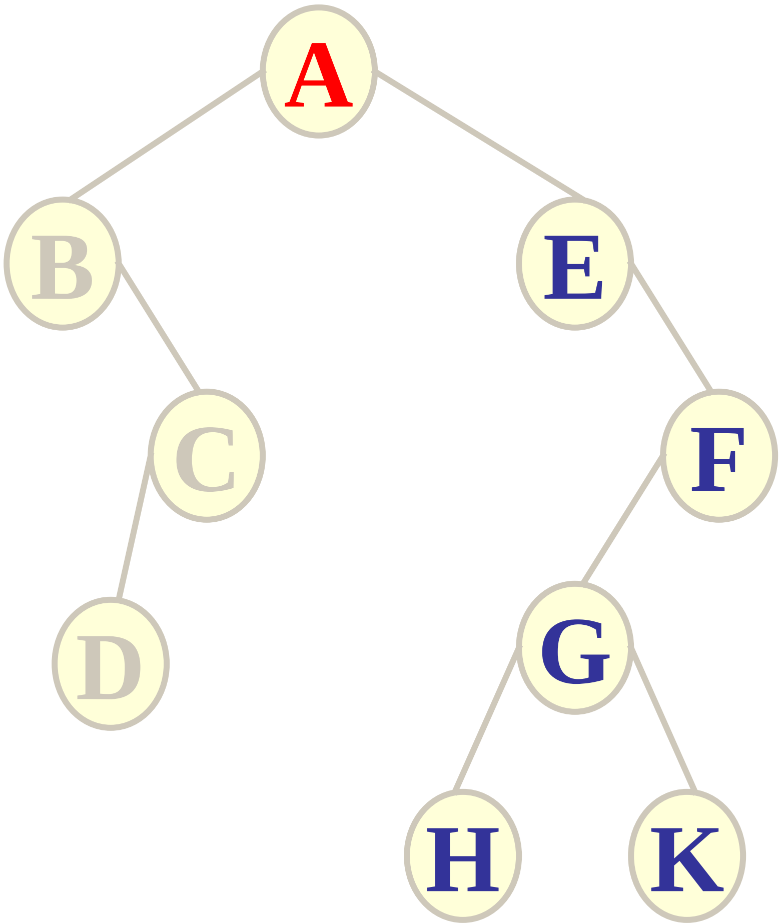
二叉树的层次遍历是指从二叉树的第一层（即根结点）开始，**从上至下**逐层遍历，在同一层中，则按**从左到右**的顺序对结点逐个访问。



层序遍历序列：*A B C D E F G*

6.3.1 遍历二叉树

例如：



先序序列：

A B C D E F G H K

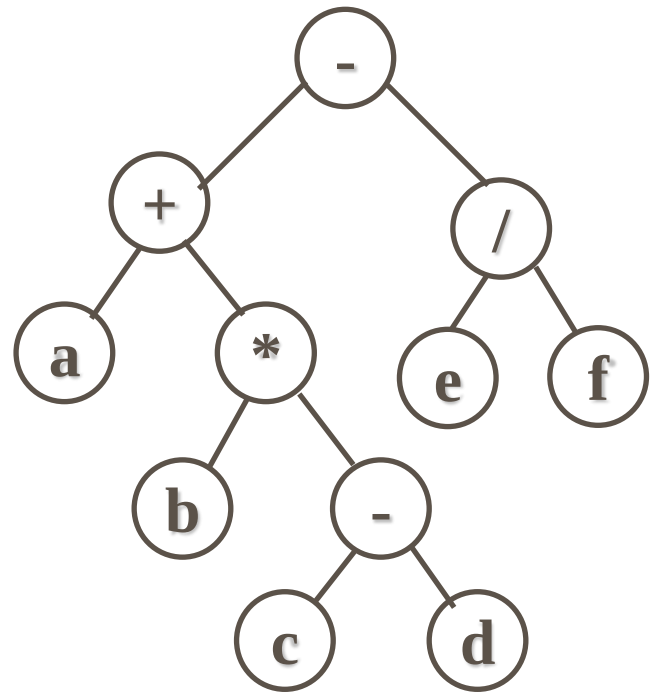
中序序列：

B D C A E H G K F

后序序列：

D C B H K G F E A

6.3.1 遍历二叉树



先序遍历顺序＝前缀表达式
中序遍历顺序＝中缀表达式
后序遍历顺序＝后缀表达式

先序遍历： - + a * b - c d / e f

中序遍历： a + b * c - d - e / f

后序遍历： a b c d - * + e f / -

6.3.1 遍历二叉树

遍历的递归算法

1　先序遍历递归算法

```
void Preorder (BiTree T, void( *visit)(TElemType& e))
{ // 先序遍历二叉树
  if (T!==NULL) {
    visit(T->data);           // 访问结点
    Preorder(T->lchild, visit); // 遍历左子树
    Preorder(T->rchild, visit);// 遍历右子树
  }
}
```

6.3.1 遍历二叉树

2 中序遍历递归算法

```
void Inorder (BiTree T, void( *visit)(TElemType& e))
{ // 中序遍历二叉树
  if (T!==NULL) {
    Preorder(T->lchild, visit); // 遍历左子树
    visit(T->data);            // 访问结点
    Preorder(T->rchild, visit);// 遍历右子树
  }
}
```

6.3.1 遍历二叉树

3 后序遍历递归算法

```
void Postorder (BiTree T, void( *visit)(TElemType& e))
{ // 后序遍历二叉树
   if (T!==NULL) {
      Preorder(T->lchild, visit); // 遍历左子树
      Preorder(T->rchild, visit);// 遍历右子树
      visit(T->data);            // 访问结点
   }
}
```

6.3.1 遍历二叉树

先序遍历的非递归实现

二叉树先序遍历的非递归算法的关键：在先序遍历过某结点的整个左子树后，如何找到该结点的右子树的根指针。

解决办法：在访问完该结点后，将该结点的指针保存在栈中，以便以后能通过它找到该结点的右子树。

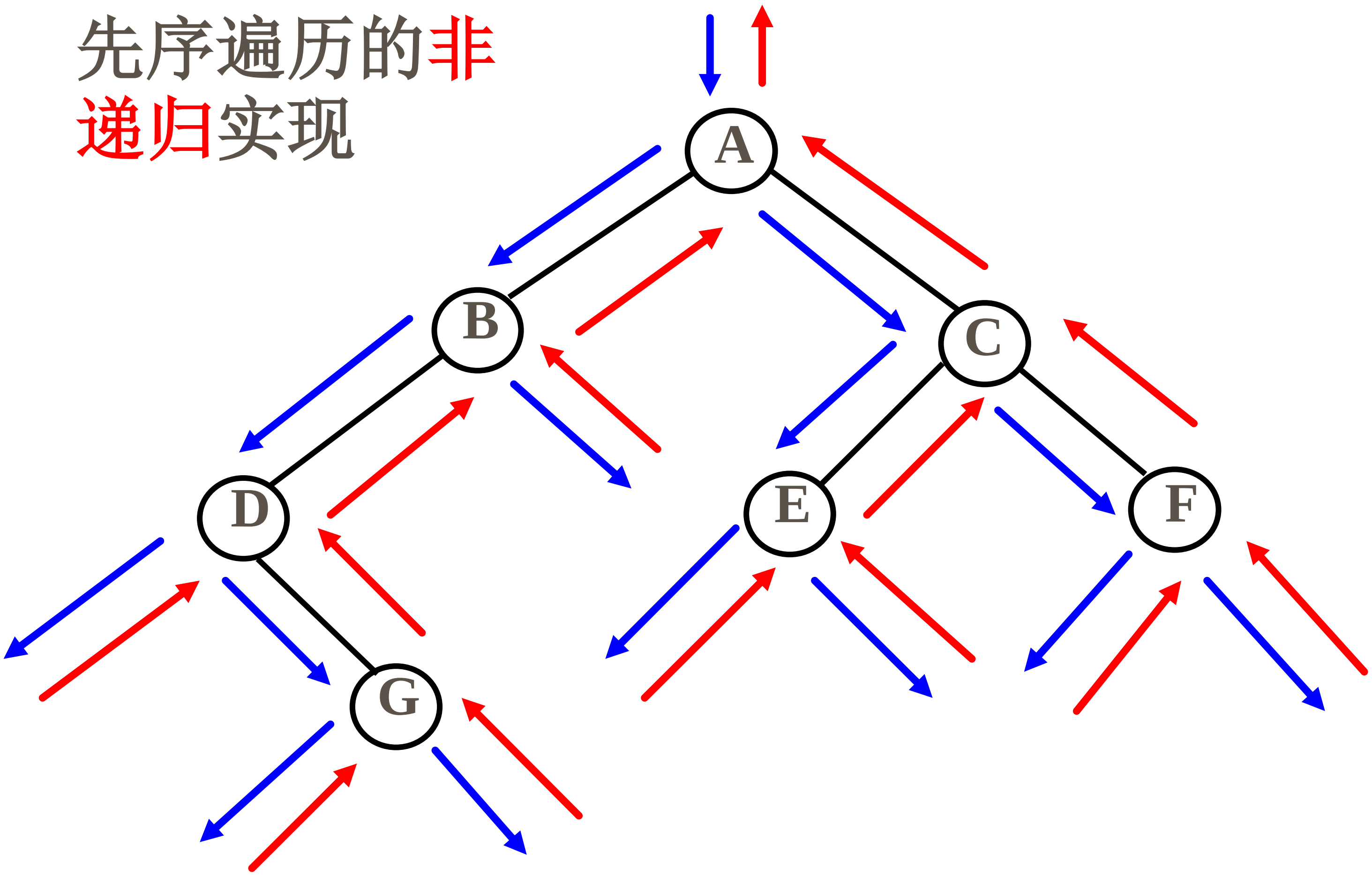
在先序遍历中，设要遍历二叉树的根指针为root，则有两种可能：

(1) 若root!=NULL，则表明？如何处理？

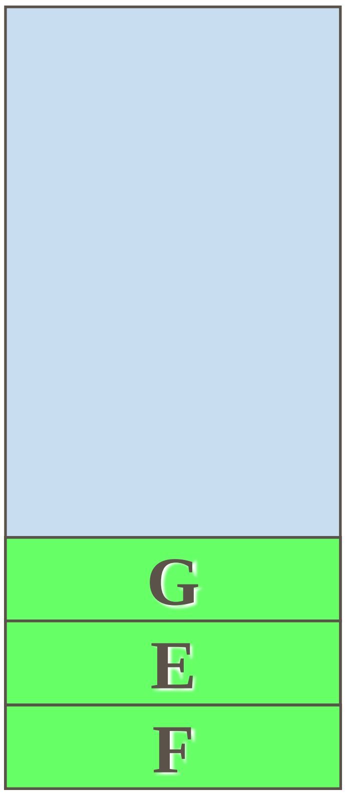
(2) 若root=NULL，则表明？如何处理？

6.3.1 遍历二叉树

先序遍历的非递归实现



栈内容



A B D G C E F

6.3.1 遍历二叉树

先序遍历——非递归算法（伪代码）

1.栈s初始化；

2.循环直到root为空且栈s为空

  2.1 当root不空时循环

    2.1.1 输出root->data;

    2.1.2 将指针root的值保存到栈中；

    2.1.3 继续遍历root的左子树

  2.2 如果栈s不空，则

    2.2.1 将栈顶元素弹出至root；

    2.2.2 准备遍历root的右子树；

6.3.1 遍历二叉树

需用到栈，顺序栈的定义如下：

```
typedef BiTNode* SElemType;
typedef struct{
   SElemType *base;
   SElemType *top;
   int stacksize;
}SqStack;
```

6.3.1 遍历二叉树

先序遍历的非递归算法1

```
void Preorder(BiTree T, void (*visit)(TElemType e)){
    SqStack S; InitStack(S); p = T;
    while (p != NULL || !StackEmpty(S)){
        if (p != NULL){
            visit(p->data); // 访问结点
            Push(S, p); p = p->lchild
        }
        else{
            Pop(S, p); p = p->rchild
        }
    }
}
```

6.3.1 遍历二叉树

先序遍历的非递归算法2

```
void Preorder(BiTree T, void (*visit)(TElemType e)){
    int top = 0; BiTree stack[20], p = T;
    do{
        while (p){
            visit(p->data);
            stack[top] = p; top++;
            p = p->lchild;
        }
        if (top){
            top--; p = stack[top];
            p = p->rchild;
        }
    } while (top || p);
}
```

6.3.1 遍历二叉树

中序遍历的非递归实现

设S为一个栈，p为指向根结点的指针，其处理过程为：

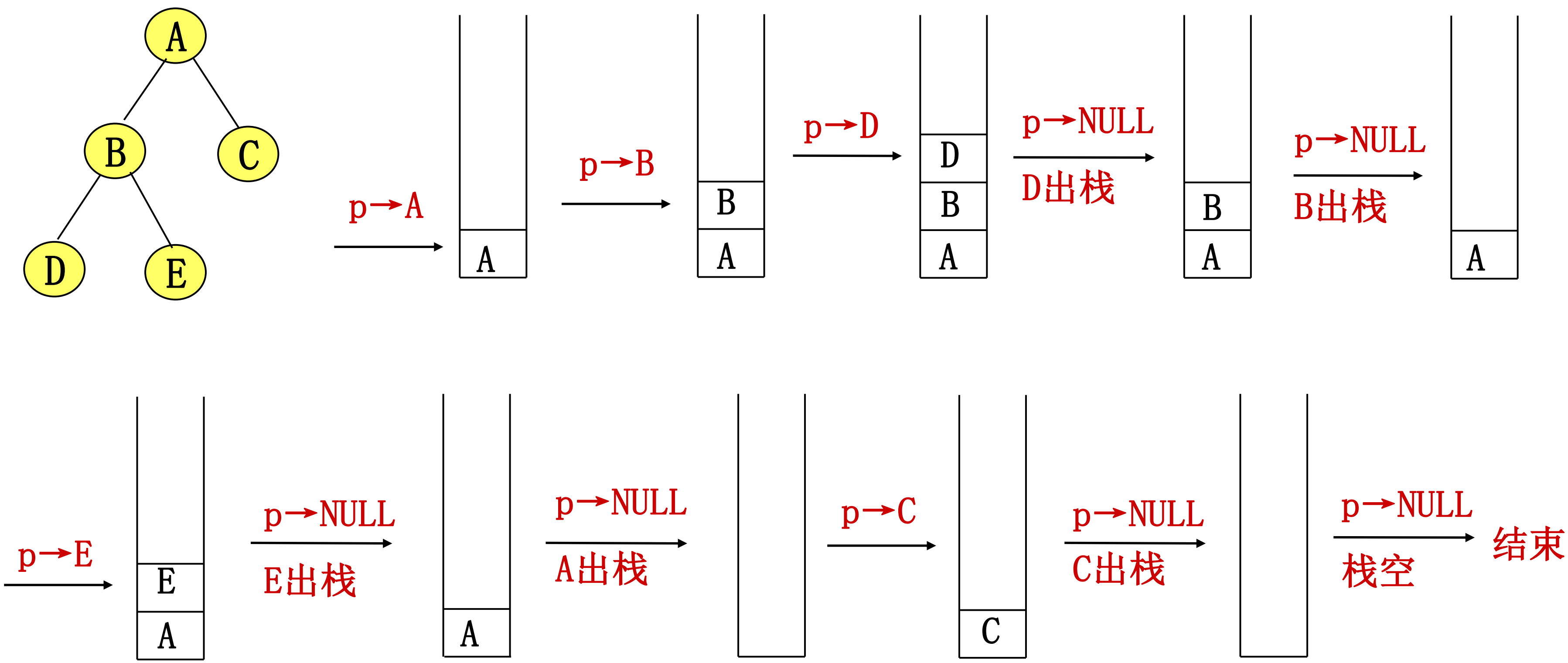
（1）当p非空时，将p所指结点的地址进栈，并将p指向该结点的左子树；

（2）当p为空时，弹出栈顶元素并访问之，同时将p指向该结点的右子树；

（3）重复（1）（2）步骤，直到栈空且p 也为空。

6.3.1 遍历二叉树

中序遍历的非递归实现



6.3.1 遍历二叉树

中序遍历的非递归算法1:算法6.3

```
void InOrderTraverse(BiTree T,void(*visit)(TElemType e)){
    StackInit(S); p = T;
    while (p || !StackEmpty(S)){
        while (p){ //树非空，根结点进栈
            Push(S, p); p = p->lchild; //指向P的左孩子
        }
        if (!StackEmpty(S)){
            Pop(S, p);
            visit(p->data); //访问结点
            p = p->rchild; //指向右孩子
        }
    }
}
```

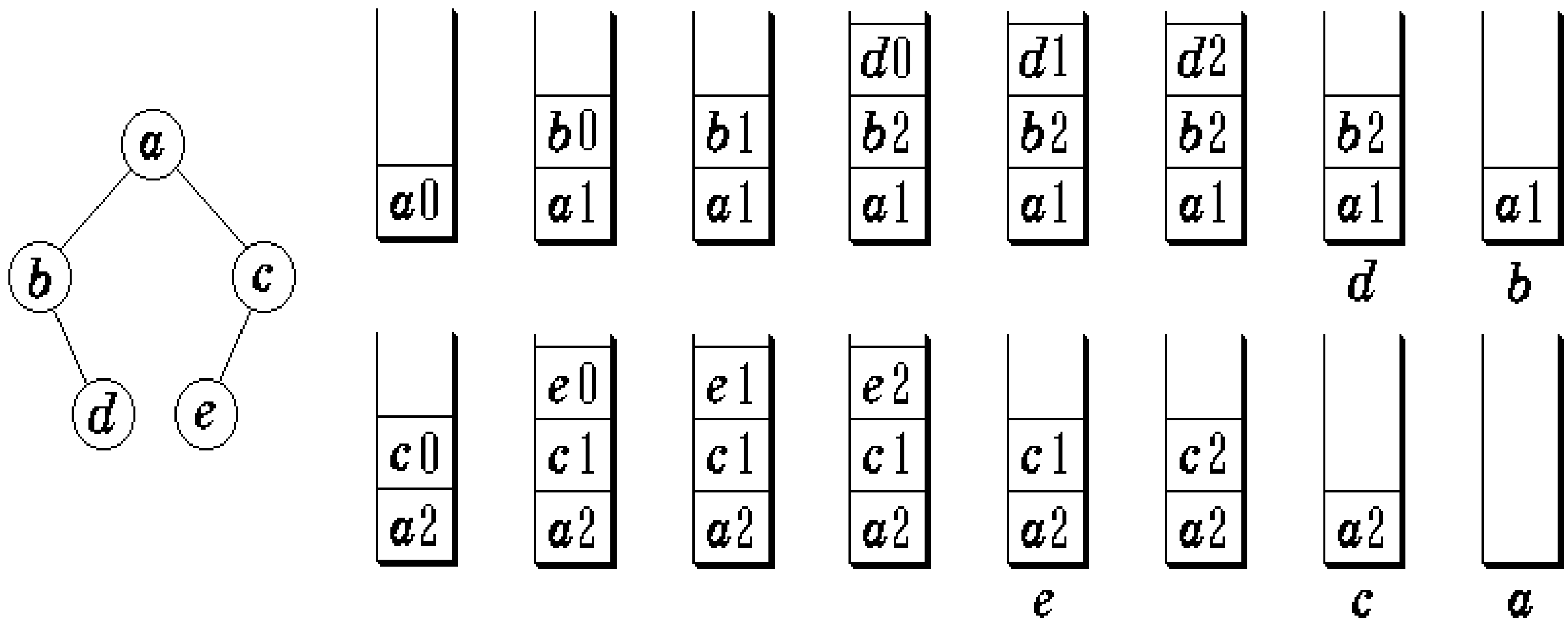
6.3.1 遍历二叉树

中序遍历的非递归算法2, 算法6. 2

```
void InOrderTraverse(BiTree T,void(*visit)(TElemType e)){
    InitStack(S); Push(S,T)
    while(!StackEmpty(S)){
        while(GetTop(S,p) && p) Push(S,p->lchild);
        // 向左走到尽头
        Pop(S,p);
        if((!StackEmpty(S)){
            Pop(S,p);
            visit(p->data);
            Push(S,p->rchild);
        }
    }
}
```

6.3.1 遍历二叉树

后序遍历时，每遇到一个结点，先把它推入栈中，让PopTim=0。在遍历其左子树前，改结点的PopTim=1，将其左孩子推入栈中。在遍历完左子树后，还不能访问该结点，必须继续遍历右子树，此时改结点的PopTim=2，并把其右孩子推入栈中。在遍历完右子树后，结点才退栈访问。



6.3.1 遍历二叉树

后序遍历的非递归算法1

```
void Postorder(BiTree T,void(*visit)(TElemType e)){
    BiTree p=T, q=NULL;
    SqStack S; InitStack(S); Push(S,p);
    while (!StackEmpty(S)){
        if(p && p!=q) { Push(S,p); p=p->lchild; }
        else {
            Pop(S,p);
            if (!StackEmpty(S))
                if (p->rchild && p->rchild!=q){
                    Push(S,p); p=p->rchild;
                } else { visit(p->data); q=p;}
        }
    }
}
```

6.3.1 遍历二叉树

<u>后序遍历的非递归算法2</u>

```
void postorder(BiTree T,void(*visit)(TElemType e)){
    BiTree p=T,q; int flag; SqStack S; InitStack(S);
    do {
        while (p){ S[top]=p; top++; p=p->lchild;}
        q=NULL;  flag=1;
        while (top) && flag){
            top--; p = S[top];
            if (p->rchild == q){
                visit(p->data); top--; q=p;
            } else { p=p->rchild;   flag=0; }
        }
    }while (top);
}
```

6.3.1 遍历二叉树

1）遍历的第一个和最后一个结点

第一个结点：

- 先序：根结点；
- 中序：沿着左链走，找到一个没有左孩子的点；
- 后序：从根结点出发，沿着左链走，找到一个既没有左孩子又没有右孩子的结点。

最后一个结点：

- 先序：从根结点出发，沿着右链走，找到一个没 有右孩子的结点；如果该结点有左孩子，再沿着其左孩子的右链走，以此类推，直到找到一个没有孩子的结点。
- 中序：从根结点出发，沿着右链走，找到一个没有右孩子的结点；
- 后序：根结点。

6.3.1 遍历二叉树

- 求中序的第一个结点的算法：

```
P=T;
while (P->lchild)  P=P->lchild;
printf(P->data);
```

- 求中序的最后一个结点的算法：

```
P=T;
while(P->rchild) P=P->rchild;
printf(P->data);
```

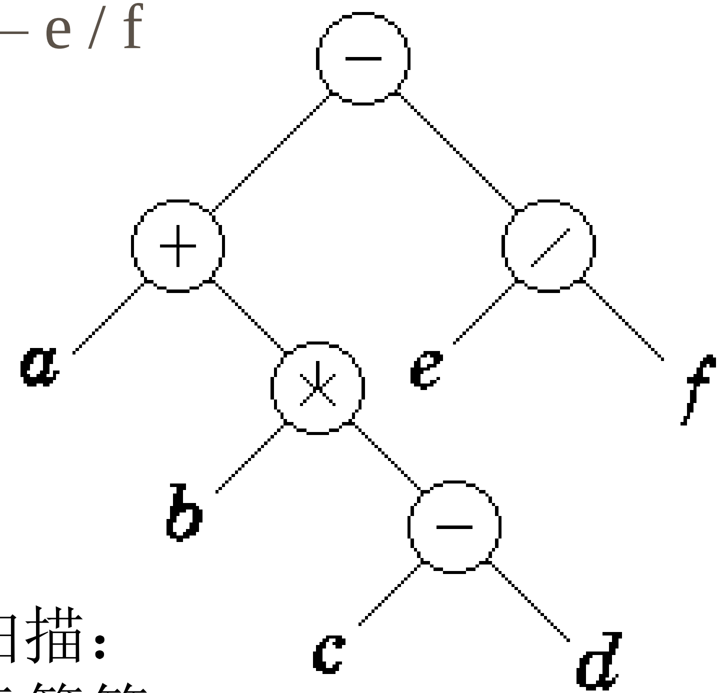
6.3.1 遍历二叉树

2）**先序+中序** **或中序+后序** 均可唯一地确定一棵二叉树

3）对于有n个节点的二叉树，其二叉链表存储结构中，有 _n+1_个指针域未利用，已经使用的有_n-1_个指针域，共有 _2n___个指针域

6.3.1 遍历二叉树

4）**运算表达式**　　a + b * (c - d) – e / f



先序：- + a * b - c d / e f

中序：a + b * c - d – e / f

后序：a b c d - * + e f /

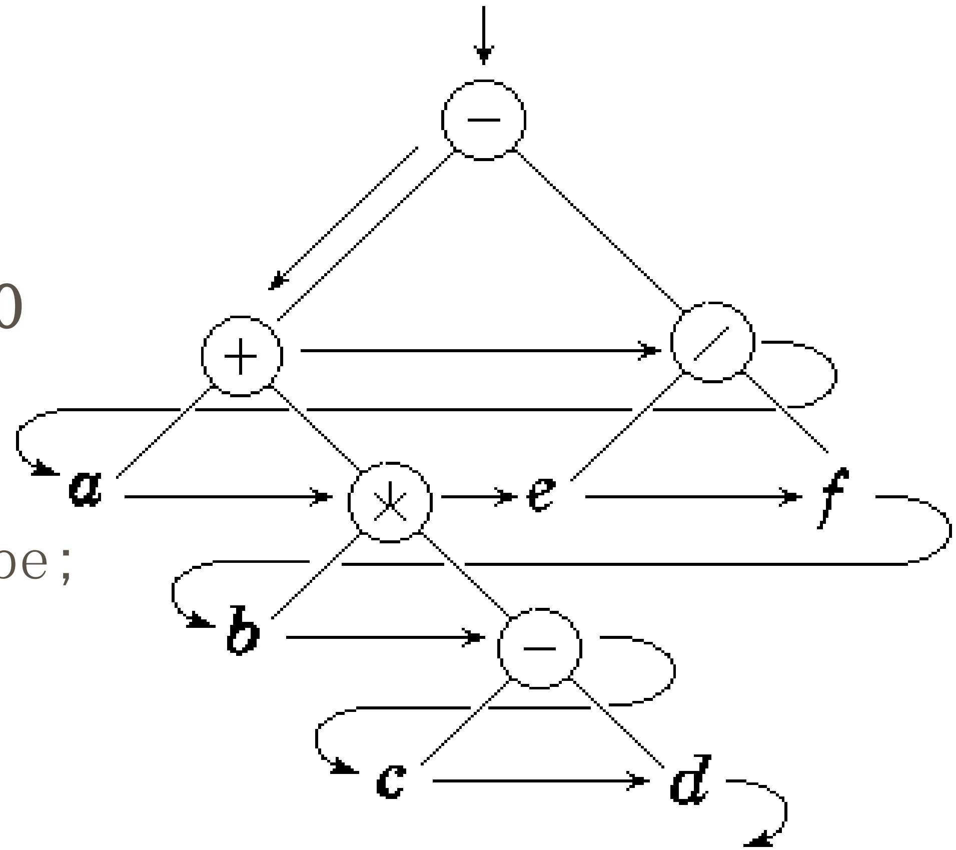
在逆波兰式中，自左到右依次扫描：是操作数，则依次进栈；遇到运算符。则退出两个操作数，对该两操作数进行该运算符的运算，运算的中间结果进栈；然后再继续重复上述的操作。

6.3.1 遍历二叉树

按层次遍历二叉树

从根开始逐层访问，用FIFO
队列实现。

```
typedef BiTNode* ElemType;
typedef struct{
      QElemType  *base;
      int front,rear;
}SqQueue;
```



6.3.1 遍历二叉树

按层次遍历二叉树

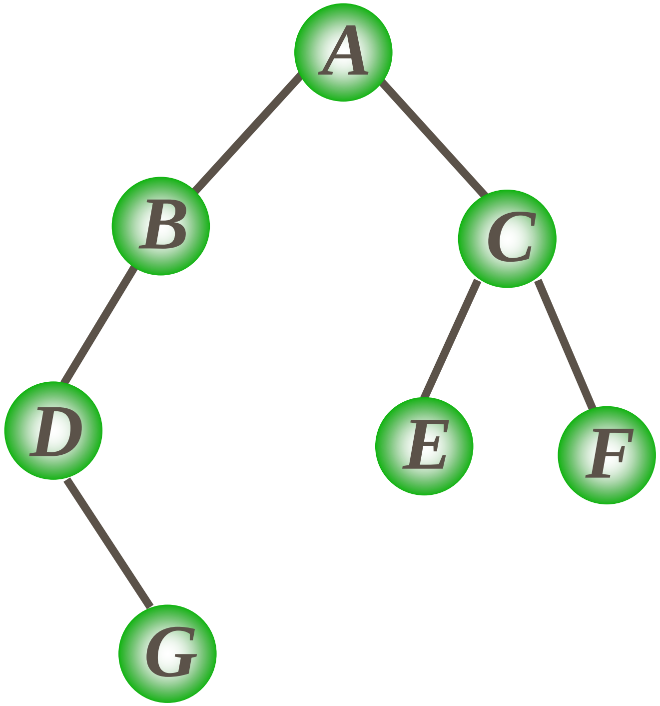
```
void LevelOrderTraverse(BiTree T,void(*visit)(TElemType e)){
    BiTree p; SqQueue Q; InitQueue(Q);
    if (T){
        Q.base[Q.rear] = T; Q.rear = (Q.rear + 1) % MAXQSIZE;
        while (Q.front != Q.rear) {
            p = Q.base[Q.front]; visit(p->data);
            Q.front = (Q.front + 1) % MAXQSIZE;
            if (p->lchild){
                Q.base[Q.rear] = p->lchild;
                Q.rear = (Q.rear + 1) % MAXQSIZE;
            }
            if (p->rchild){
                Q.base[Q.rear] = p->rchild;
                Q.rear = (Q.rear + 1) % MAXQSIZE;
            }
        }
    }
}
```

6.3.1 遍历二叉树

例：编写 求二叉树的叶子结点个数的算法
输入：二叉树的二叉链表
结果：二叉树的叶子结点个数

基本思想：遍历操作访问二叉树的每个结点，而且每个结点仅被访问一次。所以可在二叉树遍历的过程中，统计叶子结点的个数。

```
void leaf(BiTree T) {
//n计数二叉树的叶子结点的个数，初值n=0
if(T) {
    if(T->lchild==null&&T->rchild==null)
        n=n+1;
    leaf(T->lchild);
    leaf(T->rchild);
   }//if
}//leaf
```



6.3.1 遍历二叉树

例：利用二叉树后序遍历计算二叉树的深度

```
int Depth(BiTree T){
      int depl,depr;
      if (T){
            depl=Depth(T->lchild);
            depr=Depth(T->rchild);
            if (depl>=depr) return (depl+1);
            else return (depr+1);
      }
      return 0;
}
```

6.3.1 遍历二叉树

例：求二叉树结点个数

```
int Size(BiTree T)
{
   if (T==NULL)
     return 0;
   else
     return 1 + Size(T->lchild ) + Size(T->rchild);
}
```

6.3.1 遍历二叉树

例：左右子树互换

```
void Exchange(BiTree &T)
{
      BiTree S;
      if (T) {
            S=T->lchild;
            T->lchild=T->rchild;
            T->rchild=S;
            Exchange(T->lchild);
            Exchange(T->rchild);
      }
}
```

6.3.1 遍历二叉树

<u>例：复制二叉树</u>

```
void CopyTree(BiTree T,BiTree &T1){
  if(T){
    T1=(BiTree)malloc(sizeof(BiTNode));
    if (!T1){
      printf("Overflow\n");
      exit(1);
    }
    T1->data=T->data;
    T1->lchild=T1->rchild=NULL;
    CopyTree(T->lchild,T1->lchild);
    CopyTree(T->rchild,T1->rchild);
  }
}
```

正在答疑