第五章 树与森林

- 树的基本概念
- 二叉树
- 二叉树的存储表示
- 二叉树遍历及其应用
- 线索二叉树
- 树与森林
- 树与森林的遍历及其应用
- 堆
- Huffman树及其应用
- 本章小结

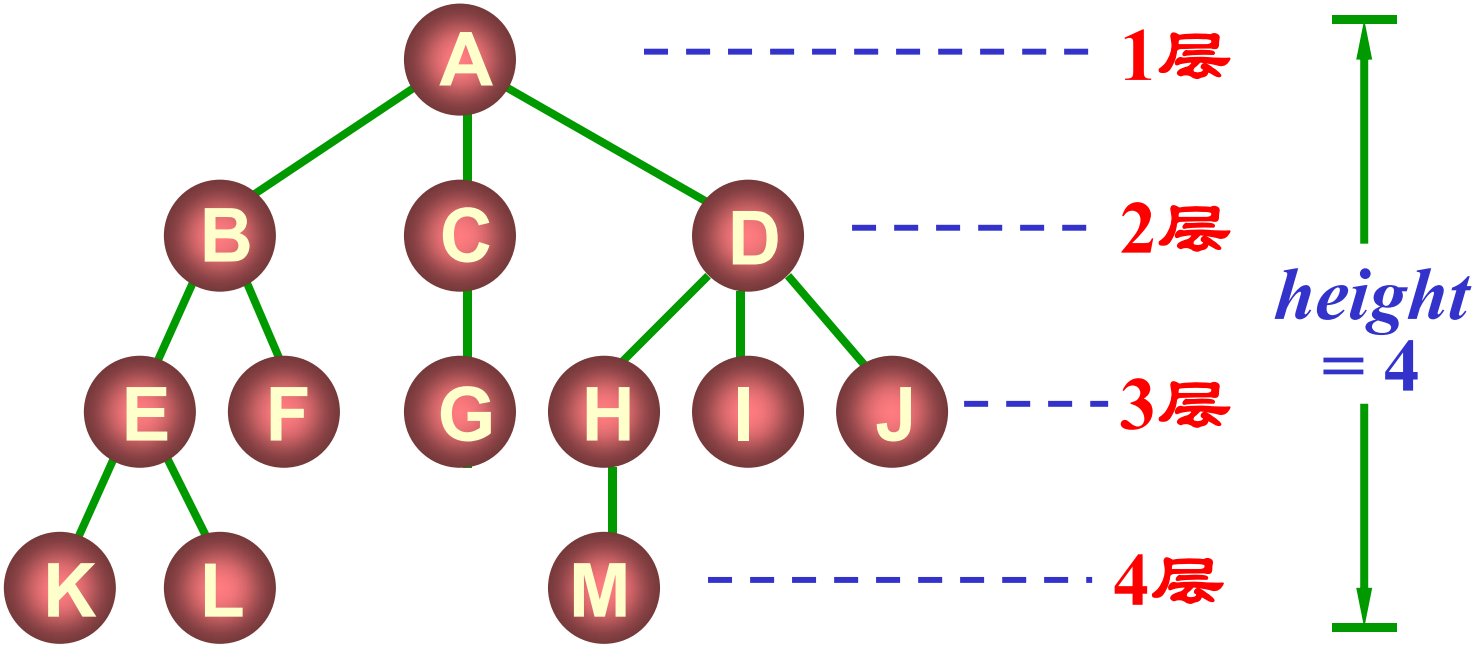
5.1 树的基本概念

树的定义

- 树是由 n ($n \geq 0$) 个结点组成的有限集合。如果 $n = 0$，称为空树；如果 $n > 0$，则
  - 有一个特定的称之为根(root)的结点，它只有直接后继，但没有直接前驱；
  - 除根以外的其它结点划分为 m ($m \geq 0$) 个互不相交的有限集合 $T_0, T_1, \cdots, T_{m-1}$，每个集合又是一棵树，并且称之为根的子树。

树的特点

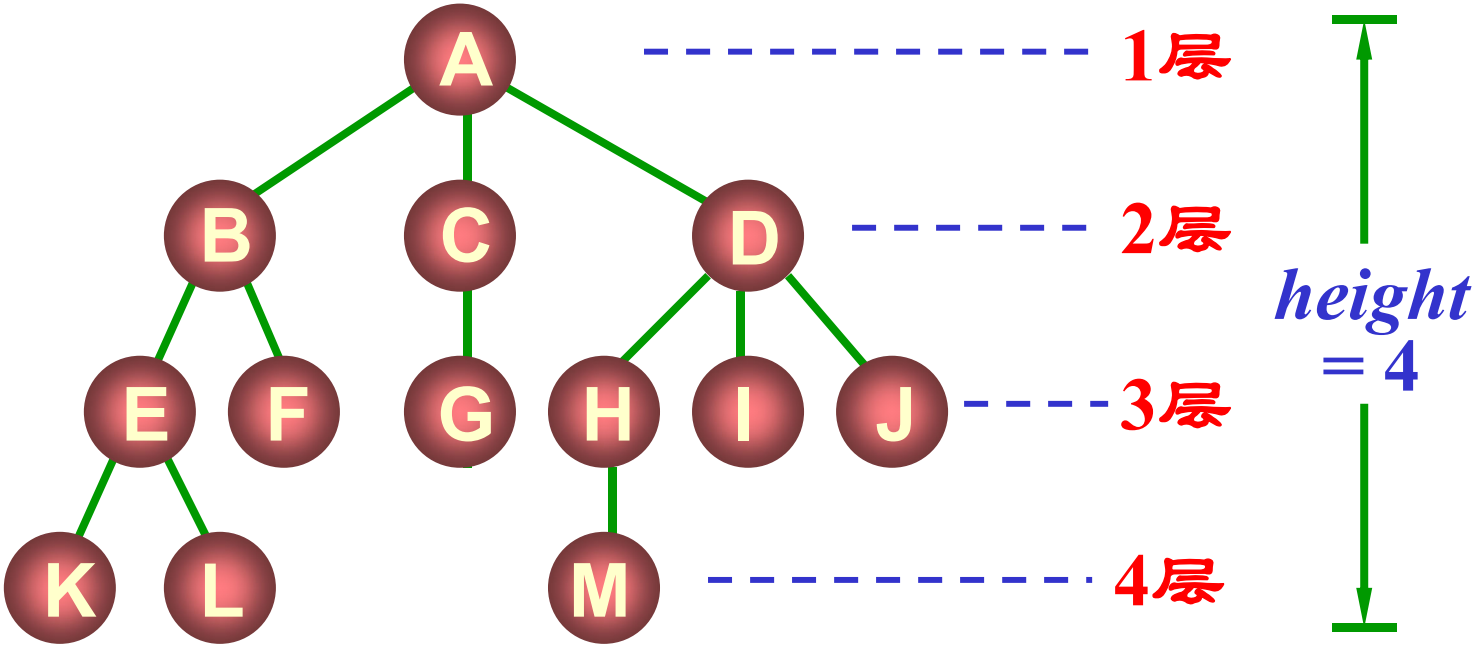
- 每棵子树的根结点有且仅有一个直接前驱，但可以有0个或多个直接后继。



- 结点(*node*)—包含数据项及指向其它结点的分支。
- 结点的度(*degree*)—结点所拥有的子树棵数。
- 叶(*leaf*) 结点—即度为0的结点，又称为终端结点。
- 分支(*branch*)结点—除叶结点之外的其它结点，又称为非终端结点。
- 子女(*child*)结点—若结点*x*有子树，则子树的根结点即为结点*x*的子女。
- 双亲(*parent*)结点—若结点*x*有子女，则它即为子女的双亲。
- 兄弟(*sibling*)结点—同一双亲的子女互称为兄弟。
- 祖先(*ancestor*)结点—从根结点到该结点所经分支上的所有结点。
- 子孙(*descendant*)结点—某一结点的子女，以及这些子女的子女都是该结点的子孙。

- 结点所处层次(*level*)—简称结点层次，即从根到该结点所经路径上的分支条数。（树中任一结点的层次为其双亲结点层次加1）
- 树的深度(*depth*)—树中结点所处的最大层次。（空树深度为0，只有一个根结点的树的深度为1）
- 树的高度(*height*)—自下向上定义高度。叶结点高度为1，非叶结点高度等于其子女结点高度最大值加1。高度与深度计算方向不同，但数值相等。
- 树的度(*degree*)—树中结点的度的最大值。
- 有序树—树中结点的各棵子树$T_0, T_1, \cdots$是有次序的，其中T_0为根的第1棵子树，T_1为根的第2棵子树。
- 无序树—树中结点的各棵子树之间的次序是不重要的，可以互相交换位置。
- 森林—为m棵树的集合。若删去一棵非空树的根结点，树就变成森林；若增加一个根结点，让森林中每一棵树的根结点都变成它的子女，则森林就变成一棵树。

- 结点
- 结点的度
- 分支结点
- 叶结点
- 子女
- 双亲
- 兄弟
- 祖先
- 子孙
- 结点层次
- 树的度
- 树的深度
- 树高度
- 森林



树类定义

```
template <class Type> class Tree {
//对象：树是由 n (>=0)个结点组成的有限集合
//在类界面中的 position 是树中结点的地址
//在顺序存储方式下是下标型，在链表存储方式下是指针型
// Type 是树结点中存放数据的类型
//要求所有结点的数据类型都是一致的
public:
    Tree ( );
    //构造函数，生成树的结构并初始化
    ~Tree ( );
    //析构函数，释放所占存储
    position Root ( );
    //返回树的根结点地址，若树为空，则返回0
```

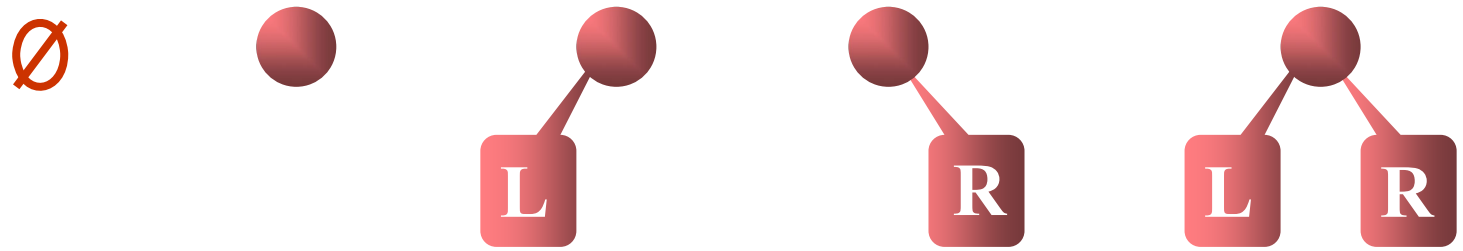
```
BuildRoot ( const Type &value );
//建立树的根结点
position FirstChild ( position p );
//返回结点 p 的第一个子女的地址，若无子女则返回 0
position NextSibling ( position p, position v );
//返回结点 p 的子女 v 的下一兄弟地址，若无兄弟则返回 0
position Parent ( position p );
//返回结点 p 的双亲结点地址，若 p 为根结点，则返回 0
Type GetData( position p );
//函数返回结点 p 中存放的值
Type Retrieve ( position p );
//函数返回结点 p 中存放的值
```

```
bool InsertChild ( const position p, const Type &value );
//在结点 p 下插入数据为 value 的新子女结点
//若结构中结点已满，则不能插入，函数返回 true
//若插入成功，则函数返回 false
bool DeleteChild ( position p, int i );
//删除结点 p 的第 i 个子女及其全部子孙结点
//若该结点的第 i 个子女结点不存在，则函数返回 false
//若删除成功，则函数返回 true
void DeleteSubTree ( position t );
//删除以 t 为根结点的子树
bool IsEmpty ( );
//判断树空否，若树结点数为0，则返回 true，否则返回 false
void Traversal ( void ( *visit ) ( position p ) );
//遍历，visit 是用户自编的访问结点 p 数据的函数
};
```

5.2 二叉树 (Binary Tree)

二叉树的定义一棵二叉树是结点的一个有限集合，该集合或者为空，或者是由一个根结点加上两棵分别称为左子树和右子树的、互不相交的二叉树组成。

二叉树的特点是每个结点最多有两个子女，即二叉树中不存在度大于2的结点，且二叉树的子树有左右之分，其子树的次序不能颠倒。



二叉树的五种不同形态

二叉树的性质

性质1 若二叉树的层次从1开始，则在二叉树的第i层最多有2^{i-1}个结点。$(i \geq 1)$

[证明用数学归纳法]

性质2 深度为k的二叉树最少有k个结点，最多有2^k-1个结点。$(k \geq 0)$

[证明用求等比级数前k项和的公式]

$$2^0 + 2^1 + 2^2 + \ldots + 2^{k-1} = 2^k - 1$$

性质3 对任何一棵二叉树，如果其叶结点有n_0个，度为2的非叶结点有n_2个，则有

$$n_0 = n_2 + 1$$

证明：若设度为1的结点有n_1个，总结点个数为n，总边数为e，则根据二叉树的定义：

$$n = n_0 + n_1 + n_2 \qquad e = 2n_2 + n_1 = n - 1$$

因此，有 $2n_2 + n_1 = n_0 + n_1 + n_2 - 1$

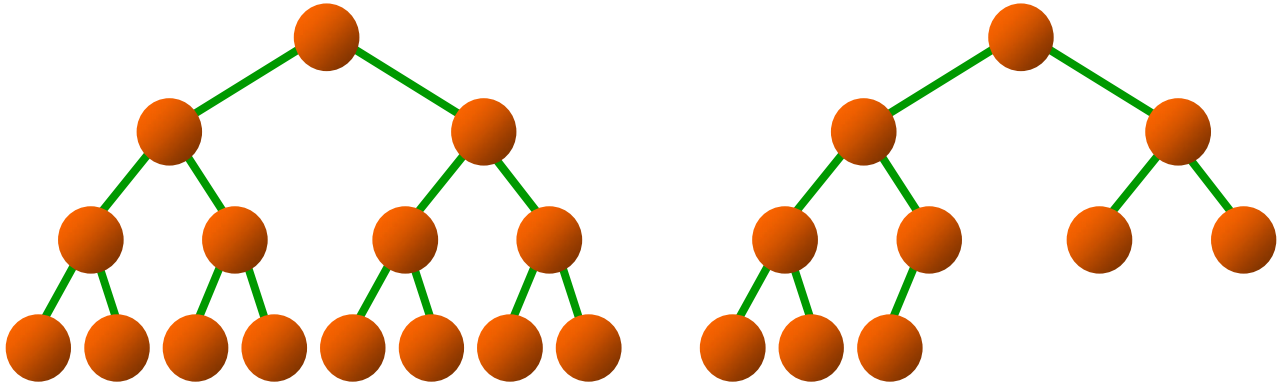
$$n_2 = n_0 - 1 \Rightarrow n_0 = n_2 + 1$$

定义1 满二叉树 (Full Binary Tree)

深度为k的满二叉树是有2^k-1个结点的二叉树，其中每一层结点都达到了最大个数。除最底层结点的度为0外，其它各层结点的度都为2。

定义2 完全二叉树 (Complete Binary Tree)

若设一棵具有n个结点的深度为k的二叉树，其每一个结点都与高度为k的满二叉树树中编号为1~n的结点一一对应。除第h层之外，其它各层(1~ k-1)的结点数都达到最大个数，第k层从右向左连续缺若干结点。



性质4 具有n $(n \geq 0)$个结点的完全二叉树的高度为$\lceil \log_2(n+1) \rceil$。

证明： 设完全二叉树的高度为k，则有

$$2^{k-1} - 1 < n \leq 2^k - 1$$

$2^{k-1} - 1$：上面$k-1$层结点数　　$2^k - 1$：包括第k层的最大结点数

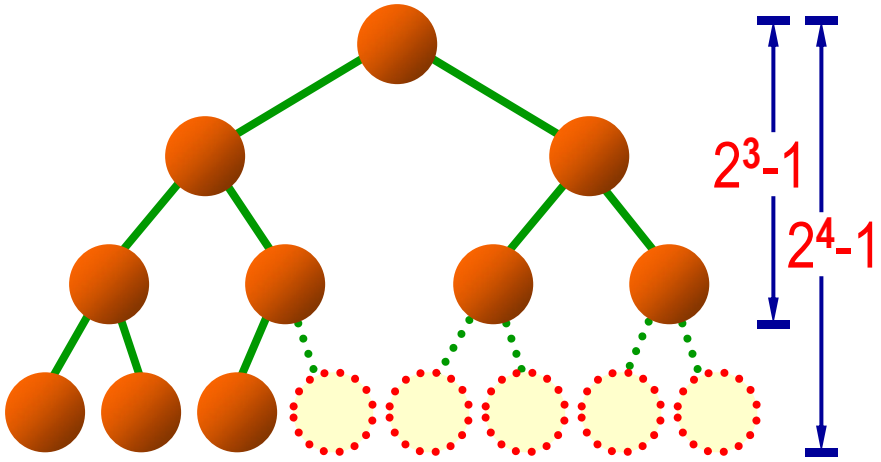
变形　$2^{k-1} < n+1 \leq 2^k$

取对数

$$k-1 < \log_2(n+1) \leq k$$

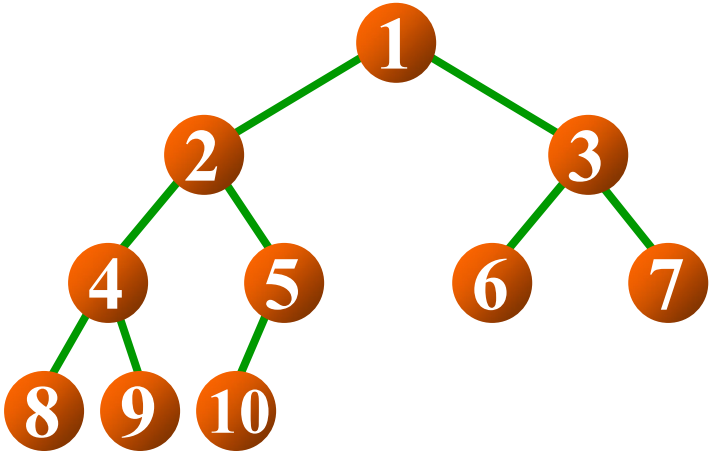
有$\lceil \log_2(n+1) \rceil = k$

$$k = \lceil \log_2(n+1) \rceil$$



性质5 如将一棵有n个结点的完全二叉树自顶向下，同一层自左向右连续给结点编号1, 2, …, n-1，则有以下关系：

- 若$i = 1$，则 i 无双亲；
  若$i > 1$，则 i 的双亲为$\lfloor i/2 \rfloor$。
- 若$2*i \le n$，则 i 的左子女为$2*i$；
  若$2*i+1 \le n$，则 i 的右子女为$2*i+1$。
- 若 i 为奇数，且i != 1，
  则其左兄弟为i-1；
  若i 为偶数，且i != n，
  则其右兄弟为i+1。
- i 所在层次为$\lfloor \log_2 i \rfloor + 1$。



二叉树类定义

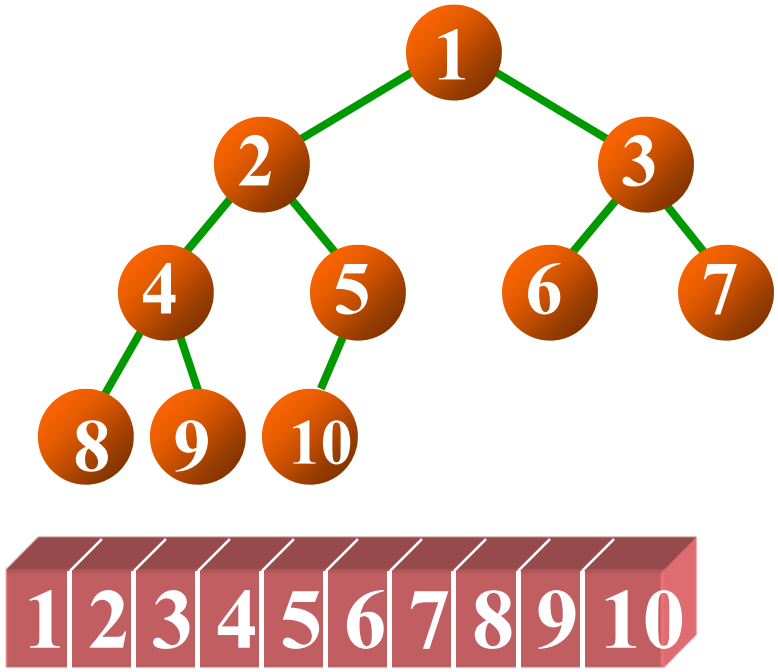
```
template <class Type> class BinaryTree {
public:
    BinaryTree ( );  //构造函数
    BinaryTree ( BinTreeNode <Type> *lch,
                 BinTreeNode <Type> *rch, Type item );
    //构造以 item 为根， lch 和 rch 为左右子树的二叉树
    int Height ( );  //返回树的深度或高度
    int Size ( );
    bool IsEmpty ( );  //判二叉树空否
    BinTreeNode <Type> * Parent
            ( BinTreeNode <Type> *current );
```

```
    BinTreeNode <Type> * LeftChild
                ( BinTreeNode <Type> *current );
    BinTreeNode <Type> * RightChild
                ( BinTreeNode <Type> *current );
    bool Insert ( Type item );
    bool Remove ( Type item );
    bool Find ( const Type &item ) const;
    bool GetData ( const Type &item ) const;
    BinTreeNode <Type> * GetRoot ( ) const;
    void PreOrdder ( void ( *visit ) ( BinTreeNode <Type> *p ) );
    void InOrdder ( void ( *visit ) ( BinTreeNode <Type> *p ) );
    void PostOrdder ( void ( *visit ) ( BinTreeNode <Type> *p ) );
    void LevelOrdder ( void ( *visit ) ( BinTreeNode <Type> *p ) );
};
```

5.3 二叉树的存储表示

二叉树的顺序表示

- **数组方式**
  - 当在数据处理过程中，二叉树的大小和形态不发生剧烈的动态变化时。
  - 用一组连续的存储单元存储二叉树的数据元素。
  - 用于表示完全二叉树的存储表示非常有效，表示一般二叉树不很理想。
  - 在树中进行插入和删除时，为反映结点层次变动，可能需要移动许多结点，降低算法效率。



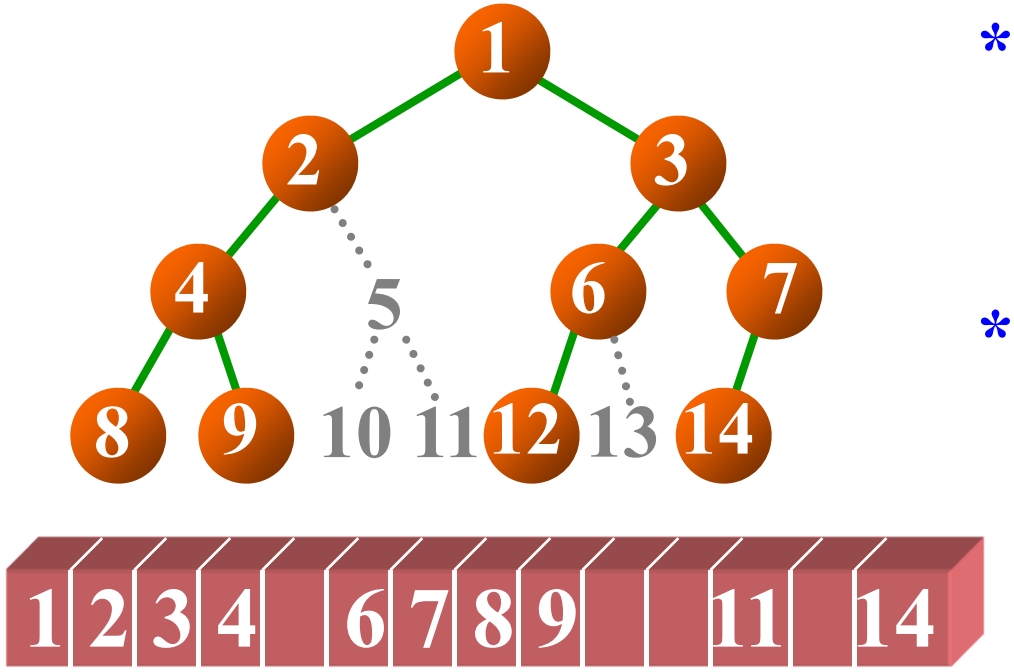
完全二叉树

顺序表示

（最简单、最省存储）

**对完全二叉树的所有结点按照层次次序自顶向下，得到一个结点的线性序列，按该序列将二叉树放在一维向量中。

**利用完全二叉树的性质5，从一个结点的编号推算出其双亲、子女、兄弟等结点的编号，找到这些结点。



一般二叉树

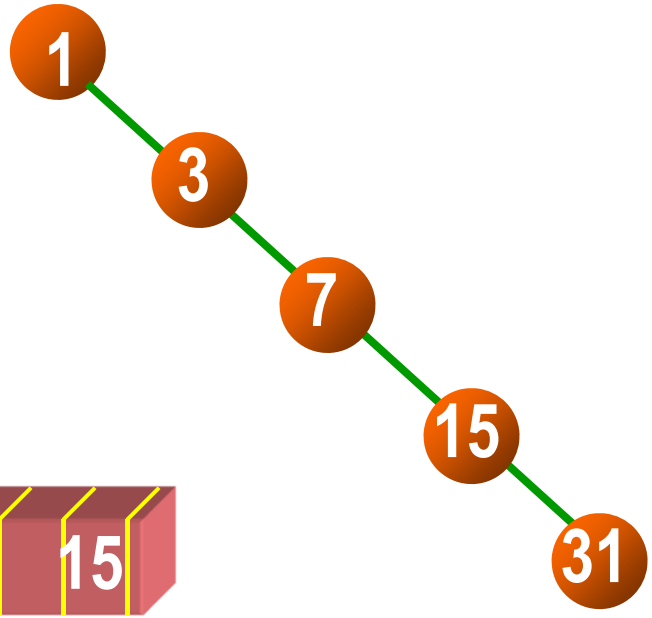
顺序表示

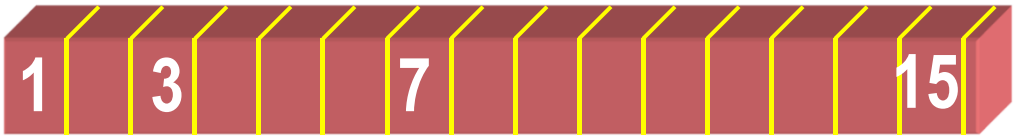
**对二叉树结点进行编号，然后按其编号将它放到一维数组中。

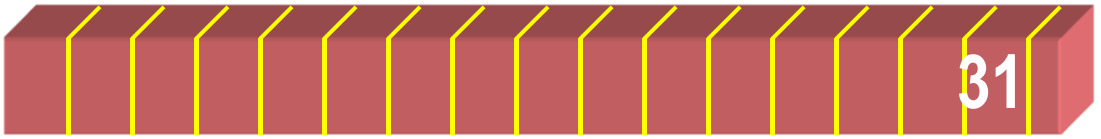
**在编号时，若遇到空子树，应在编号时假定有此子树进行编号，而在顺序存储时留出相应位置，由此找到其双亲、子女、兄弟结点的位置，但可能消耗大量空间。

极端情形：只有右单支的二叉树

由于一般二叉树必须仿照完全二叉树那样存储，可能会浪费很多存储空间，单支树就是一个极端情况。



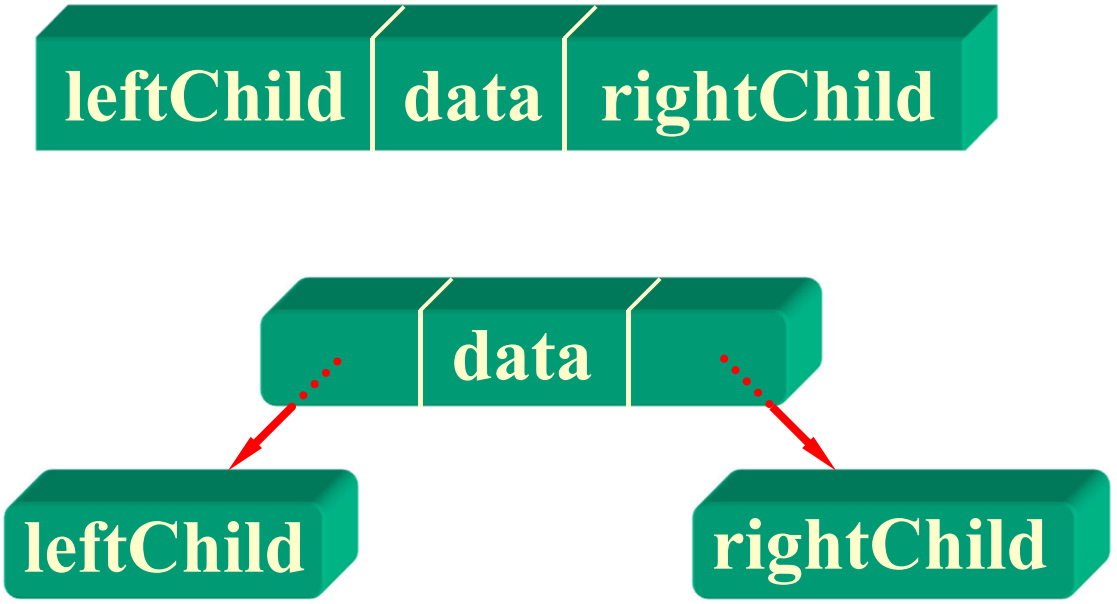




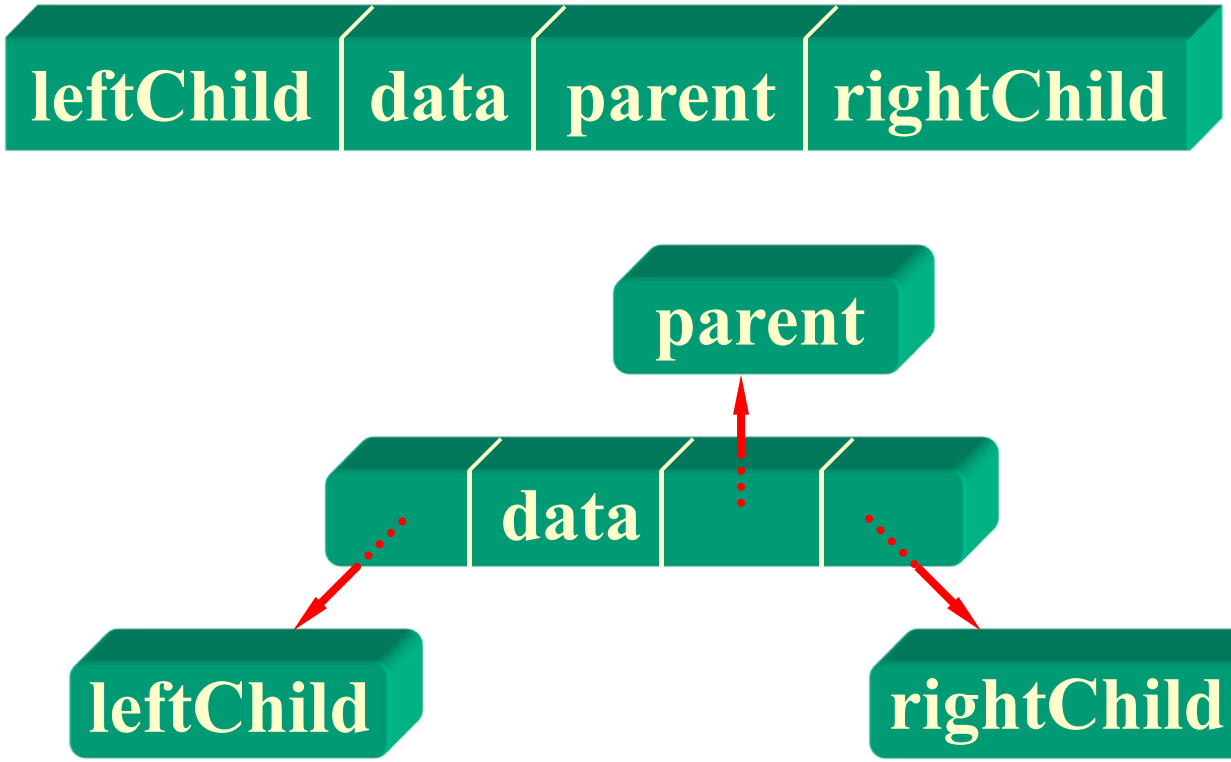
**要求一个可存储31个结点的一维数组，但只在其中几个位置放有结点数据，其它大多数结点空间都未利用，又不能压缩到一起，造成很大空间浪费。

二叉树的链表表示

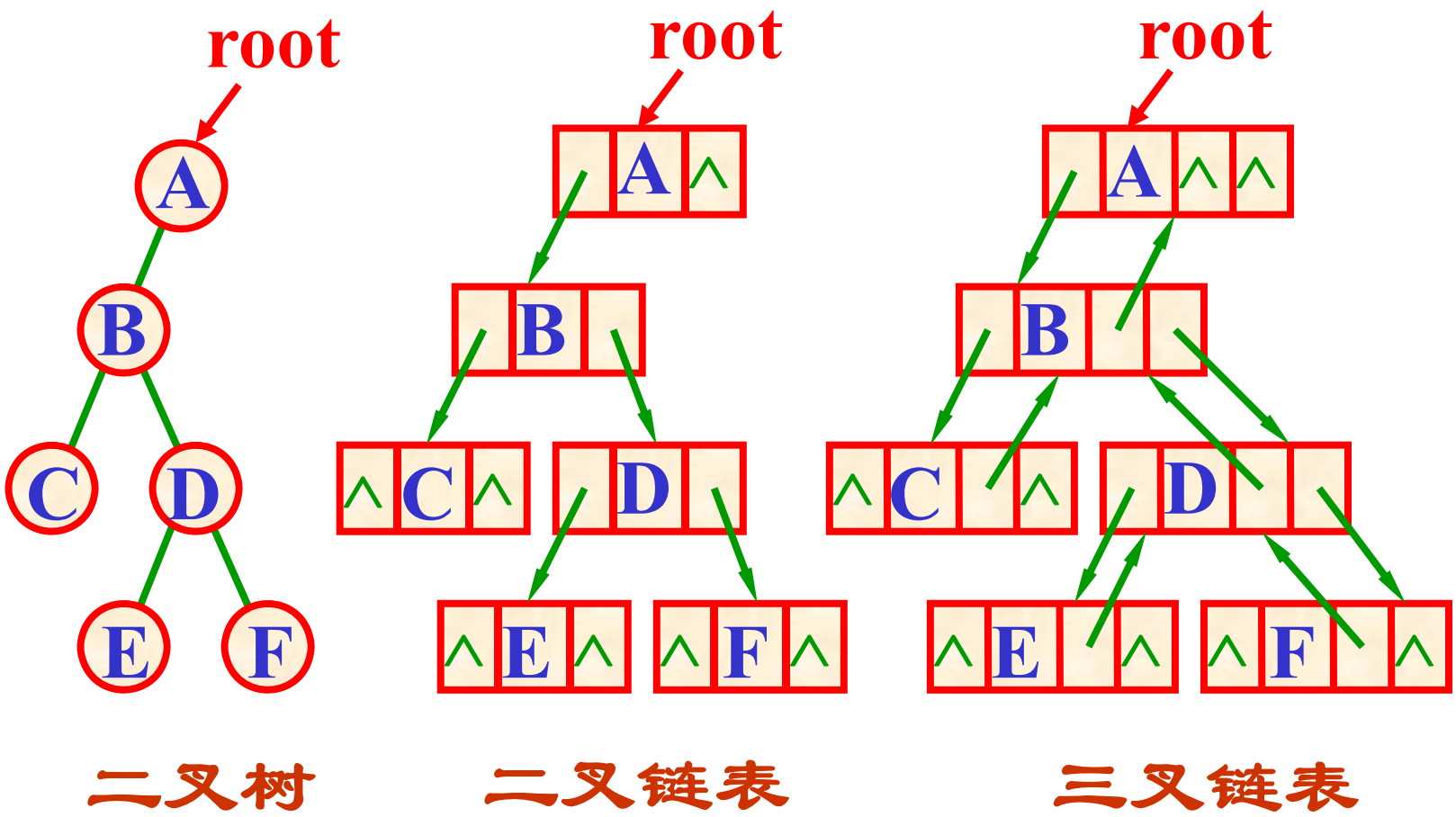
- **链表方式**
  - 便于插入、删除和修改；
  - 数据不需移动；
  - 只需修改指针；
  - 可提高效率。



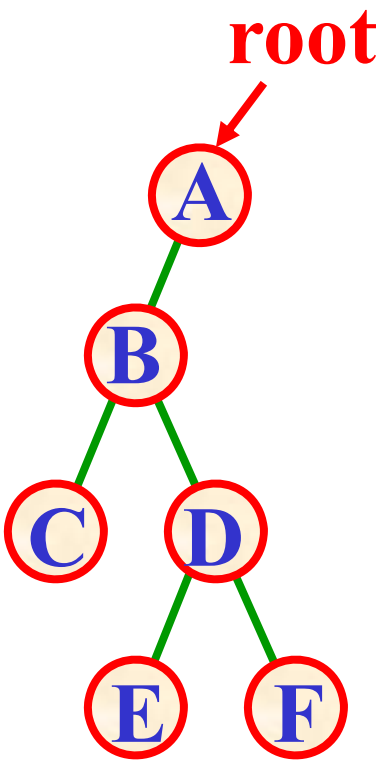
二叉链表


leftChild
data
parent
rightChild
parent
data
leftChild
rightChild

三叉链表



二叉树链表表示的示例



	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	-1
5	F	3	-1	-1

二叉树的的静态链表结构

二叉树类定义

```
template <class Type> class BinaryTree;

template <class Type> class BinTreeNode {
friend class BinaryTree <Type>;
private:
    Type data;
    BinTreeNode <Type> *leftChild;
    BinTreeNode <Type> *rightChild;
public:
    BinTreeNode ( ) : leftChild (NULL), rightChild (NULL) { }
    BinTreeNode ( Type item,
        BinTreeNode <Type> *left = NULL,
          BinTreeNode <Type> *right = NULL ) :
              data (item), leftChild (left), rightChild (right) { }
```

```
    Type GetData ( ) const { return data; }
    BinTreeNode <Type> * GetLeft ( ) const
                { return leftChild; }
    BinTreeNode <Type> * GetRight ( ) const
                { return rightChild; }
    void SetData ( const Type &item )
                { data = item; }
    void SetLeft ( BinTreeNode <Type> *L )
                { leftChild = L; }
    void SetRight ( BinTreeNode <Type> *R )
                { rightChild = R; }
};
```

```
template <class Type> class BinaryTree {
protected:
    BinTreeNode <Type> *root;
    Type RefValue;
    void CreateBinTree ( istream &in,
                         BinTreeNode <Type> *&subtree );
    bool Insert ( BinTreeNode <Type> *&subtree, const Type &x );
    void Destroy ( BinTreeNode <Type> *&subtree );
    bool Find ( BinTreeNode <Type> *&SubTree,
                const Type &x ) const;
    BinTreeNode <Type> * Copy ( BinTreeNode <Type> *orignode );
    int Height ( BinTreeNode <Type> *subtree );
    int Size ( BinTreeNode <Type> *subtree );
    BinTreeNode <Type> * Parent ( BinTreeNode <Type> *subtree,
                                  BinTreeNode <Type> *Parent );
```

```
BinTreeNode * Find ( BinTreeNode <Type> *&subtree,
                                    const Type &x ) const;
void Traverse ( BinTreeNode <Type> *&subtree,
                                    ostream &out );
void PreOrder ( BinTreeNode <Type> *&SubTree,
                          void ( *visit ) ( BinTreeNode <Type> *p ) );
void InOrder ( BinTreeNode <Type> *&SubTree,
                          void ( *visit ) ( BinTreeNode <Type> *p ) );
void PostOrder ( BinTreeNode <Type> *&SubTree,
                          void ( *visit ) ( BinTreeNode <Type> *p ) );
friend istream & operator >> ( istream &in,
                                    BinaryTree <Type> &Tree );
friend ostream & operator << ( ostream &out,
                                    BinaryTree <Type> &Tree );
```

```
public:
    BinayTree ( ) : root ( NULL ) ;
    BinayTree ( Type value ), RefValue(value), root ( NULL) { }
    BinaryTree ( BinaryTree <Type> &s);
    ~BinaryTree ( ) { Dstroy(root); }
    bool IsEmpty ( ) { return ( root == NULL ) ? true : false; }
    BinTreeNode <Type> * Parent ( BinTreeNode <Type> *current )
       { return ( root = = NULL || root = = current ) ?
                                  NULL : Parent ( root, current ); }
    BinTreeNode <Type> * LeftChild
                                   ( BinTreeNode <Type> *current )
       { return ( current != NULL) ? current->leftChild : NULL; }
    BinTreeNode <Type> * RightChild
                                   ( BinTreeNode <Type> *current )
       { return ( current != NULL) ? current->rightChild : NULL; }
```

```
    int Height ( ) { return Height (root); }
    int Size ( ) { return Size (root); }
    BinTreeNode <Type> *GetRoot ( ) const { return root; }
    void PreOrder ( void ( *visit ) ( BinTreeNode <Type> *p ) )
       { PreOrder ( root, visit ); }
    void InOrder ( void ( *visit ) ( BinTreeNode <Type> *p ) )
       { InOrder ( root, visit ); }
    void PostOrder ( void ( *visit ) ( BinTreeNode <Type> *p ) )
       { PostOrder ( root, visit ); }
    void LevelOrder ( void ( *visit ) ( BinTreeNode <Type> *p ) );
    int Insert ( ) ( const Type item );
    BinTreeNode <Type> * Find ( Type item ) const;
};
```

二叉树部分成员函数的实现

```
template <class Type> void BinaryTree <Type> ::
Destroy ( BinTreeNode <Type> *subTree ) {
    if ( subTree != NULL ) {
        Destroy ( subTree->leftChild );
        Destroy ( subTree->rightChild );
        delete subTree;
    }
}
```

```
template <class Type> BinTreeNode <Type>
        * BinaryTree <Type> :: Parent
          ( BinTreeNode <Type> *subTree,
                  BinTreeNode <Type> *current ) {
    if ( subTree == NULL ) return NULL;
    if ( subTree->leftChild == current ||
            subTree->rightChild == current )
        return subTree;
    BinTreeNode <Type> *p;
    if (( p = Parent ( subTree->leftChild, current ) )
                != NULL ) return p;
    else return Parent ( subTree->rightChild, current );
}
```

```
template <class Type> void BinaryTree <Type> ::
Traverse ( BinTreeNode <Type> *subTree,
                        ostream &out ) {
//私有函数：搜索并输出根为 subTree 的二叉树
    if ( subTree != NULL ) {
        out << subTree->data << ' ';
        Traverse ( subTree->leftChild, out );
        Traverse ( subTree->rightChild, out );
    }
}
```

```
istream & operator >>
        ( istream &in, BinaryTree <Type> &Tree ) {
//重载操作：输入并建立一棵二叉树 Tree
// in 是输入流对象
    Type item;
    cout << "构造二叉树：\n "; //打印标题：构造二叉树
    cout << "输入数据（用" << Tree.RefValue << "结束）：";
    //提示：输入数据
    in >> item; //输入，RefValue 是输入结束标志
    while ( item != Tree.RefValue ) { //判断是否结束输入
        Tree.Insert ( item ); //插入到树中
         cout << "输入数据（用" << Tree.RefValue << "结束）：";
         in >> item; //输入
    }
    return in;
}
```

```
ostream & operator <<
    ( ostream &out, BinaryTree <Type> &Tree ) {
    out << "二叉树的前序遍历。 \n";
    Tree.Traverse ( Tree.root, out );
    out << endl;
    return out;
}
```

采用广义表建立二叉树

- 广义表的表名放在表前，表示树的根结点，括号中是根的左、右子树。
- 每个结点的左、右子树用逗号隔开。若仅有右子树没有左子树，逗号不能省略。
- 在整个广义表表示输入的结尾加上一个特殊符号（如“#”，存于RefValue中），表示输入结束。

A(B(D, E(G,)), C(, F))#

算法基本思想

- 依次从保存广义表的字符串*ls*中输入字符。
  - 若是字母则表示结点值，建立新结点，作为左子女(k=1)或右子女(k=2)链接到其父结点上。
  - 若是左括号“(”，则表明子表开始，将k置为1；若遇到的是右括号“)”，则表明子表结束。
  - 若遇到逗号“,”，则表示以左子女为根的子树处理完毕，应接着处理以右子女为根的子树，将k置为2。如此处理，直至读入结束符“#”为止。
  - 使用一个栈，在进入子表之前将根结点指针进栈，以便括号内的子女链接之用，在子表处理结束时退栈。

```
void CreateBinTree ( istream &in, BinTreeNode <char> *&BT )
{ //从输入流in输入二叉树的广义表表示建立对应的二叉链表
    Stack < BinTreeNode <char> * > s;  BT = NULL;
    BinTreeNode <char> *p, *t;  int k;  char ch;
    in >> ch;
    while ( ch != RefValue ) {
        switch ( ch ) {
          case '(' : s.Push(p);  k = 1;  break;
          case ')' : s.Pop(t);  break;
          case ',' : k = 2;  break;
          default : p = new BinTreeNode (ch);
                    if ( BT == NULL ) BT = p;
                    else if ( k == 1 ) { S.GetTop(t);  t->leftChild = p; }
                    else { S.GetTop(t);  t->rightChild = p; }
        }
        in >> ch;
    }
}
```

5.4 二叉树遍历及其应用

树的遍历就是按某种次序访问树中的结点，要求每个结点访问一次且仅访问一次。

设访问根结点记作 V，
遍历根的左子树记作 L，
遍历根的右子树记作 R，

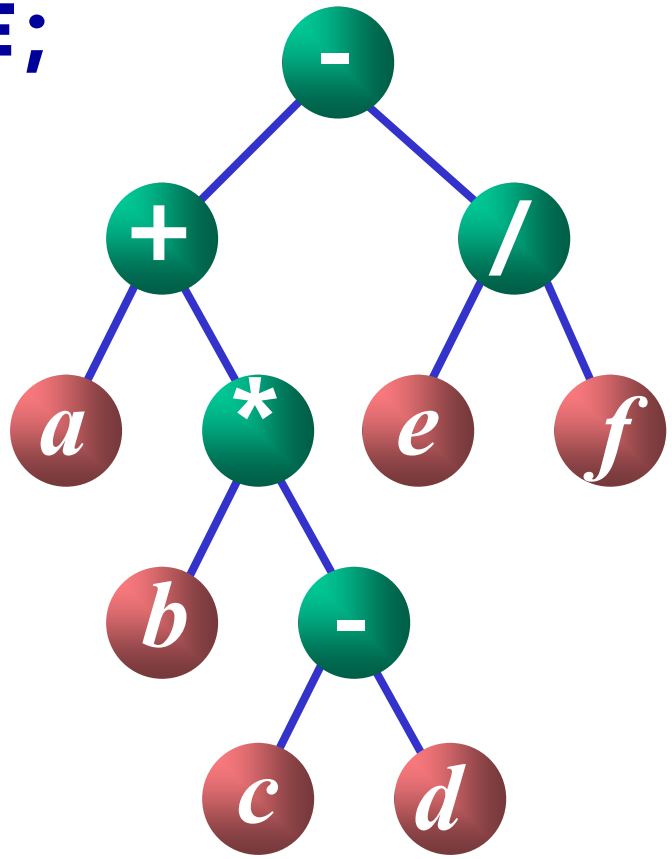
则可能的遍历次序有

前序	VLR	镜像	VRL
中序	LVR	镜像	RVL
后序	LRV	镜像	RLV

中序遍历 (Inorder Traversal)

中序遍历二叉树算法的框架是：

- 若二叉树为空，则空操作；
- 否则
  - 中序遍历左子树 (L)；
  - 访问根结点 (V)；
  - 中序遍历右子树 (R)。



遍历结果

a + *b* * *c* - *d* - *e* / *f*

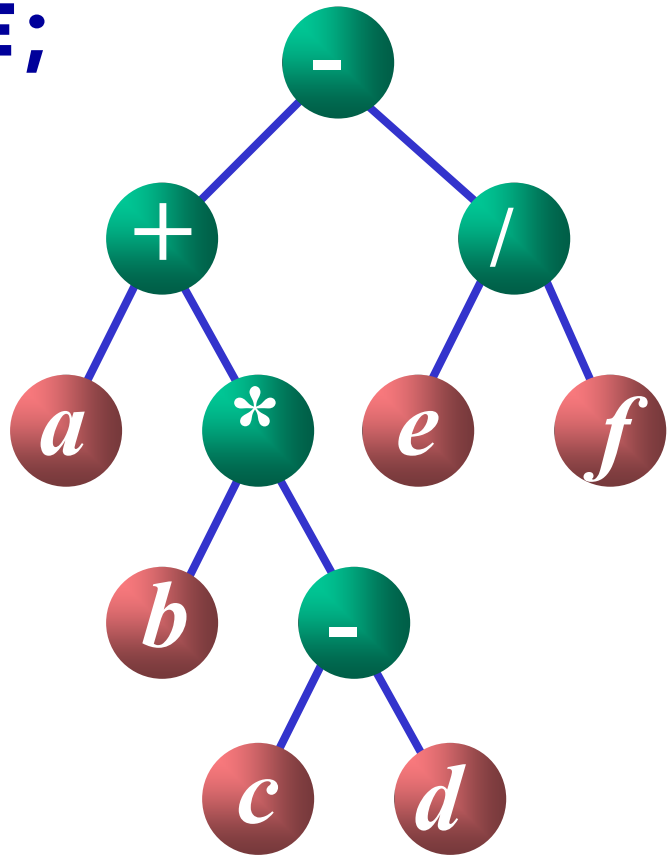
二叉树递归的中序遍历算法

```
template <class Type> void BinaryTree <Type> ::
InOrder ( BinTreeNode <Type> *subTree,
                void ( *visit ) ( BinTreeNode <Type> *p ) ) {
    if ( subTree != NULL ) {
        InOrder ( subTree->leftChild );
        visit ( subTree->data );
        InOrder ( subTree->rightChild );
    }
}
```

前序遍历 (Preorder Traversal)

前序遍历二叉树算法的框架是：

- **若二叉树为空，则空操作；**
- **否则**
  - **访问根结点 (V)；**
  - **前序遍历左子树 (L)；**
  - **前序遍历右子树 (R)。**



遍历结果

*- + a * b - c d / e f*

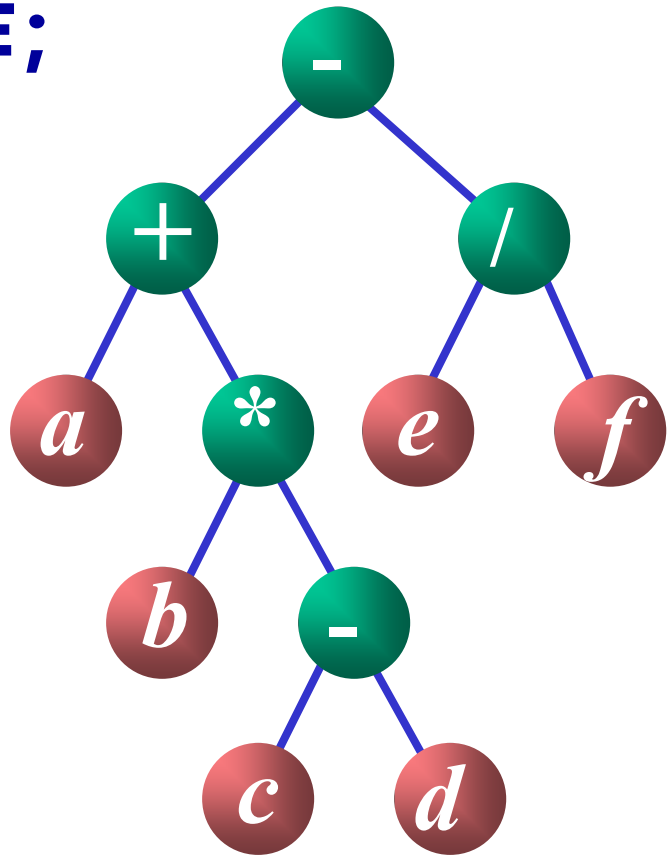
二叉树递归的前序遍历算法

```
template <class Type> void BinaryTree <Type> ::
PreOrder ( BinTreeNode <Type> *subTree,
                  void ( *visit ) ( BinTreeNode <Type> *p ) ) {
    if ( subTree != NULL ) {
        visit ( subTree->data );
        PreOrder ( subTree->leftChild );
        PreOrder ( subTree->rightChild );
    }
}
```

后序遍历 (Postorder Traversal)

后序遍历二叉树算法的框架是：

- 若二叉树为空，则空操作；
- 否则
  - 后序遍历左子树 (L)；
  - 后序遍历右子树 (R)；
  - 访问根结点 (V)。



遍历结果

*a b c d - * + e f / -*

二叉树递归的后序遍历算法

```
template <class Type> void BinaryTree <Type> ::
PostOrder ( BinTreeNode <Type> *subTree,
                    void ( *visit ) ( BinTreeNode <Type> *p ) ) {
    if ( subTree != NULL ) {
        PostOrder ( subTree->leftChild );
        PostOrder ( subTree->rightChild );
        visit ( subTree->data );
    }
}
```

应用二叉树遍历的事例

利用二叉树后序遍历计算二叉树结点个数

为计算二叉树的结点个数，可以遍历根结点的左右子树，分别计算左右子树结点个数，然后再将访问根结点的语句改为相加语句。

二叉树结点个数=左右子树结点个数+根结点个数

```
template <class Type> int BinaryTree <Type> ::
Size ( BinTreeNode <Type> *subTree ) const {
    if ( subTree == NULL ) return 0;
    else return 1 + Size ( subTree->leftChild )
                  + Size ( subTree->rightChild );
}
```

利用二叉树后序遍历计算二叉树的高度

如果二叉树为空，空树高度为0；否则先递归计算根结点左右子树的高度，再求出两者中最大者，并加1（增加根结点时高度加1），得到整个高度。

```
template <class Type> int BinaryTree <Type> ::
Height ( BinTreeNode <Type> *subTree ) const {
    if ( subTree == NULL ) return 0;
    else
      return 1 + Max ( Height ( subTree->leftChild ),
                       Height ( subTree->rightChild ) );
}
```

利用二叉树前序遍历实现复制构造函数

如果二叉树 s 非空，则首先复制根结点，相当于前序遍历算法中的访问根结点语句；然后分别复制根结点的左右子树，相当于前序遍历左右子树。

```
template <class Type> int BinaryTree <Type> ::
BinaryTree ( const BinaryTree <Type> &s ) {
//公共函数：复制构造函数
   root = Copy ( s.root );
}
```

```
BinTreeNode <Type> * BinaryTree <Type> ::
Copy ( BinTreeNode <Type> *orignode ) {
//私有函数：该函数返回一个指针
//给出一个以 orignode 为根的二叉树的副本
  if ( orignode == NULL ) return NULL;
  //根为空，返回空指针
  BinTreeNode <Type> * temp =
              new BinTreeNode <Type>; //创建根结点
  temp->data = orignode->data; //传送数据
  temp->leftChild = Copy ( orignode->leftChild );
  //复制左子树
  temp->rightChild = Copy ( orignode->rightChild );
  //复制右子树
  return temp; //返回根指针
}
```

利用二叉树前序遍历判断两棵二叉树是否相等

```
int operator == ( const BinaryTree <Type> &s,
                  const BinaryTree <Type> &t )
{  //判断两棵二叉树的等价
   //假定它是 BinaryTree 类的友元
   return ( Equal ( s.root, t.root ) ) ? true : false;
}
```

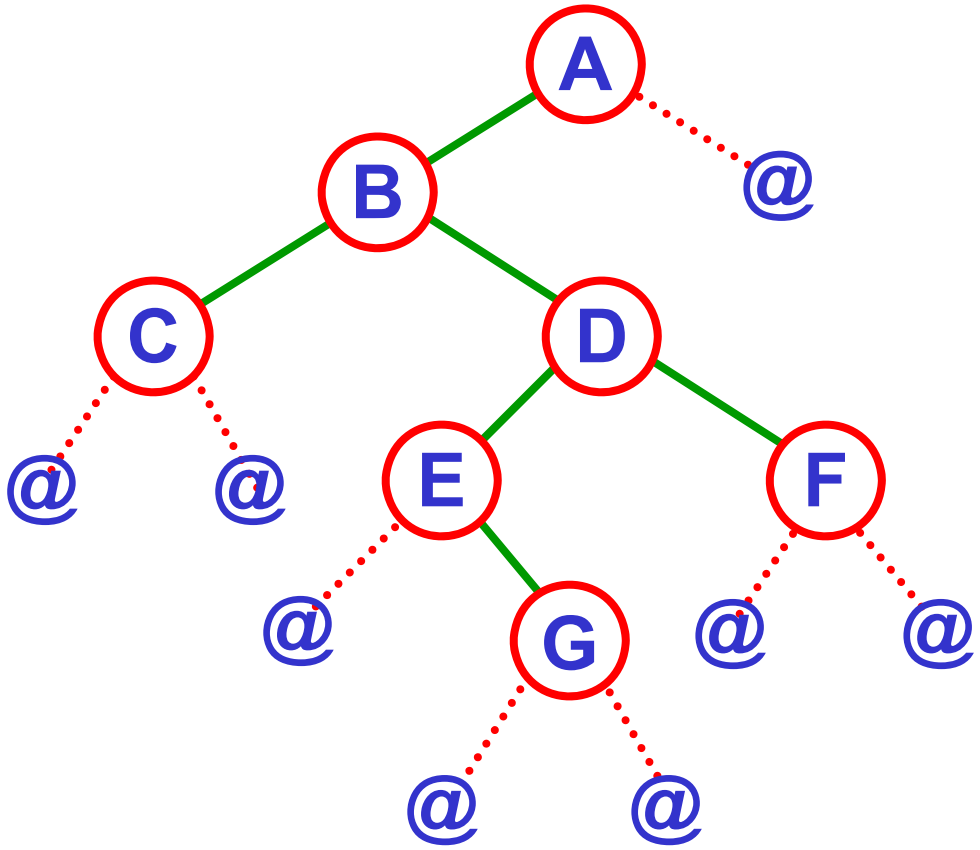
```
bool Equal ( BinTreeNode <Type> *a,
                    BinTreeNode <Type> *b ) {
//如果 a 和 b 的子树不等同，则函数返回 false
//否则函数返回 true
//假定它是 BinTreeNode 类的友元
  if ( a == NULL && b == NULL ) return true;
  //两者都是 NULL
  if ( a != NULL && b != NULL &&
       a->data == b->data &&  //两者根结点的数据相等
       Equal ( a->leftChild, b->leftChild ) &&  //且左子树相同
        Equal ( a->rightChild, b->rightChild ) )  //且右子树相同
       return true;
  return false;
}
```

利用前序遍历建立二叉树

- 以递归方式建立二叉树。
- 输入结点值的顺序必须对应二叉树结点前序遍历的顺序，并约定以输入序列中不可能出现的值作为空结点的值以结束递归，此值在RefValue中。例如，用“@”或用“-1”表示字符序列或正整数序列空结点。

如图所示的二叉树的前序遍历顺序为

A B C @ @ D E @ G @ @ F @ @ @



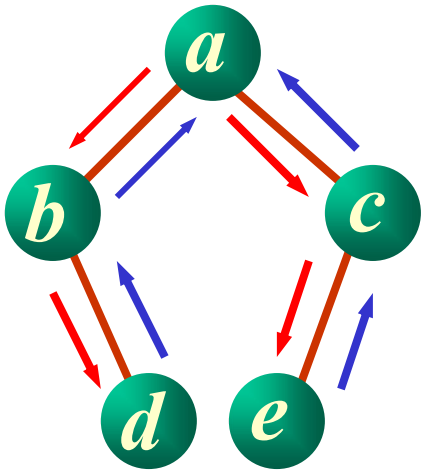
```
template <class Type> void BinaryTree <Type> ::
CreateBinTree ( ifstream &in,
                BinTreeNode <Type> *&subTree ) {
//私有函数：以递归方式建立二叉树
   Type item;
   if ( !in.eof ( ) ) {
      in >> item;  //读入根结点的值
      if ( item != RefValue ) {
         subTree = new BinTreeNode <Type> ( item );
         //建立根结点
         if ( subTree == NULL )
           { cerr << "存储分配错!" << endl; exit (1); }
         CreateBinTree ( in, subTree->leftChild );
         CreateBinTree ( in, subTree->rightChild );    }
      else subTree = NULL;  //封闭叶结点
   }
}
```

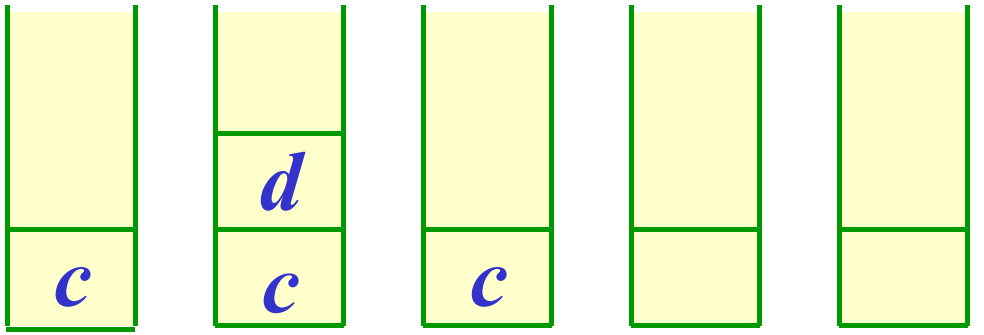
利用前序遍历输出二叉树

- 二叉树的二叉链表以广义表形式打印出来。
  - 根结点作为广义表表名放在由左右子树组成的表前；
  - 表用一对圆括号括起来。
- 首先输出根结点，然后再依次输出其左右子树。
  - 输出左子树之前要打印出左括号；
  - 在输出右子树之后要打印出右括号；
  - 依次输出的左右子树要求至少有一个非空，若都为空就无需输出。

```
template <class Type> void BinaryTree <Type> ::
PrintTree ( BinTreeNode <Type> *BT ) {
    if ( BT != NULL ) {  //树为空时结束递归
        cout << BT->data;  //输出根结点的值
        if ( BT->leftChild != NULL || BT->rightChild != NULL ) {
            cout << '(';  //输出左括号
            PrintBTree ( BT->leftChild );  //输出左子树
            cout << ',';  //输出逗号分隔符
            if ( BT->rightChild != NULL )  //若右子树不为空
                PrintBTree ( BT->rightChild );  //输出右子树
            cout << ')';  //输出右括号
        }
    }
}
```

利用栈的前序遍历非递归算法



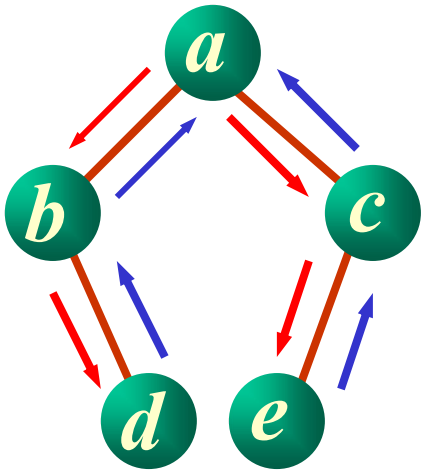


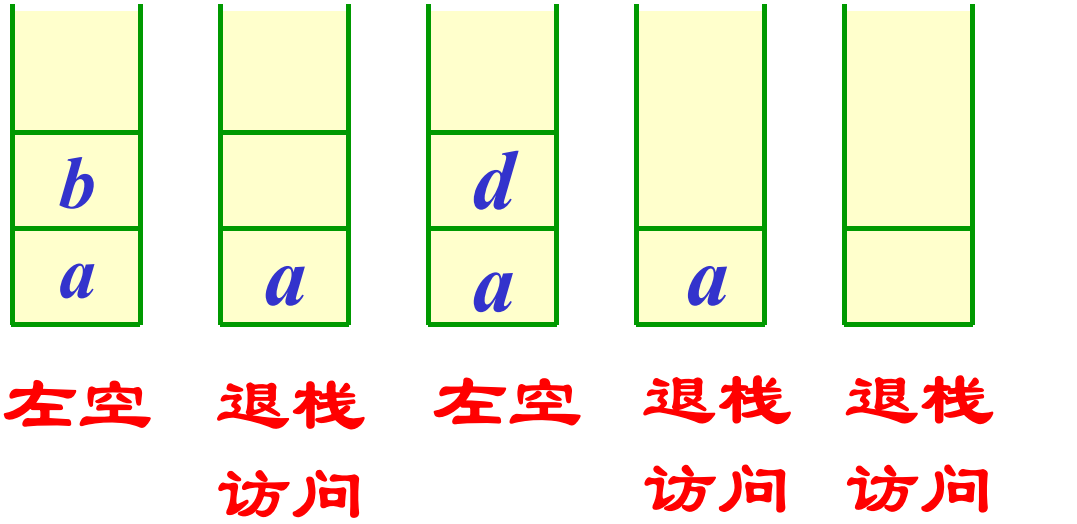
访问	访问	退栈	退栈	访问
a	b	d	c	e
进栈	进栈	访问	访问	左进
c	d	d	c	空
左进	左进	左进	左进	栈空
b	空	空	e	结束

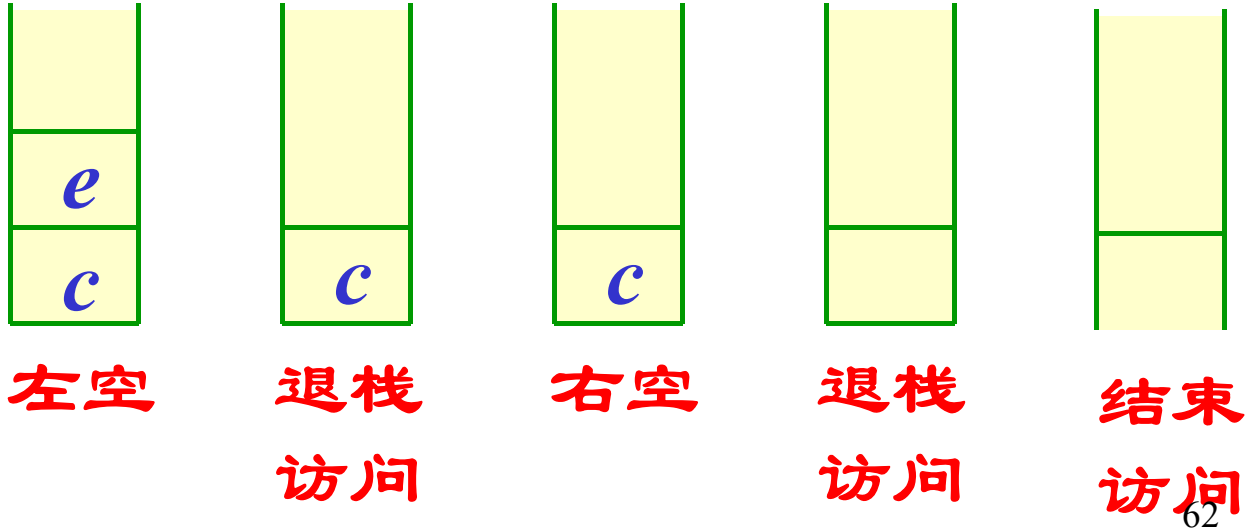
```
template <class Type> void BinaryTree <Type> ::
PreOrder ( void ( *visit ) BinTreeNode <Type> *p ) {
    Stack < BinTreeNode <Type> * > S;
    BinTreeNode <Type> *p = root;  //初始化
    S.Push ( NULL );
    while ( p != NULL ) {
        visit ( p );
        if ( p->rightChild != NULL )
            S.Push ( p->rightChild );
            //预留右子树指针在栈中
        if  ( p->leftChild != NULL )
            p = p->leftChild;  //进左子树
        else { p = S.GetTop ( );  S.Pop ( ); }
        //左子树为空，进右子树    }
}
```

```
template <class Type> void BinaryTree <Type> ::
PreOrder ( void ( *visit ) BinTreeNode <Type> *p ) {
    Stack < BinTreeNode <Type> * > S;
    BinTreeNode <Type> *p;  //初始化
    S.Push ( root );
    while ( !S.IsEmpty ( ) ) {
        S.Pop ( p );  visit ( p );
        if ( p->rightChild != NULL )
            S.Push ( p->rightChild );
            //预留右子树指针在栈中
        if  ( p->leftChild != NULL )
            S.Push ( p->leftChild );
    }
}
```

利用栈的中序遍历非递归算法







```
template <class Type> void BinaryTree <Type> ::
InOrder ( void ( *visit ) BinTreeNode <Type> *p ) {
    Stack < BinTreeNode <Type>* > S;
    BinTreeNode <Type> *p = root;  //初始化
    do {
        while ( p != NULL )
        { S.Push(p);  p = p->leftChild; }
        if ( !S.IsEmpty ( ) ) {  //栈非空
            p = S.GetTop ( );  S.Pop ( );  //退栈
            visit (p);  //访问根
            p = p->rightChild;  //向右链走
        }
    } while ( p != NULL || !S.IsEmpty ( ) );
}
```

利用栈的后序遍历非递归算法

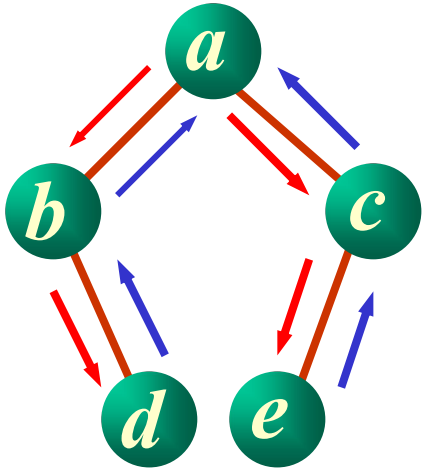
后序遍历时使用的栈的结点定义

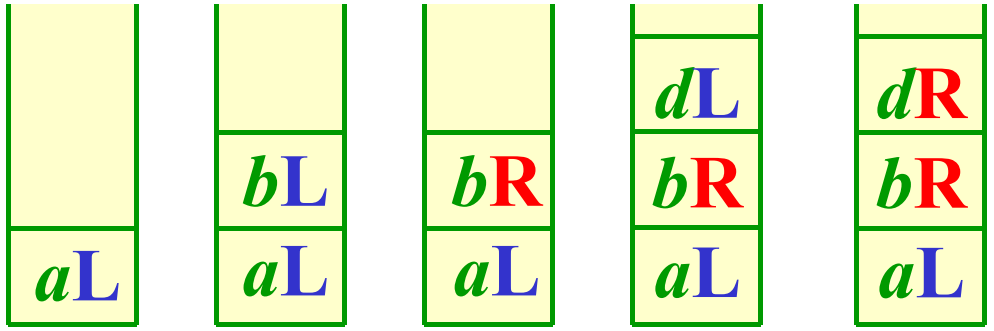
```
template <class Type> struct StkNode {
    BinTreeNode <Type> *ptr;
    enum tag { L, R };
    //该结点退栈标记
    StkNode ( BinTreeNode <Type> *N = NULL )
      : ptr (N),  tag (L) { }  //构造函数
};
```

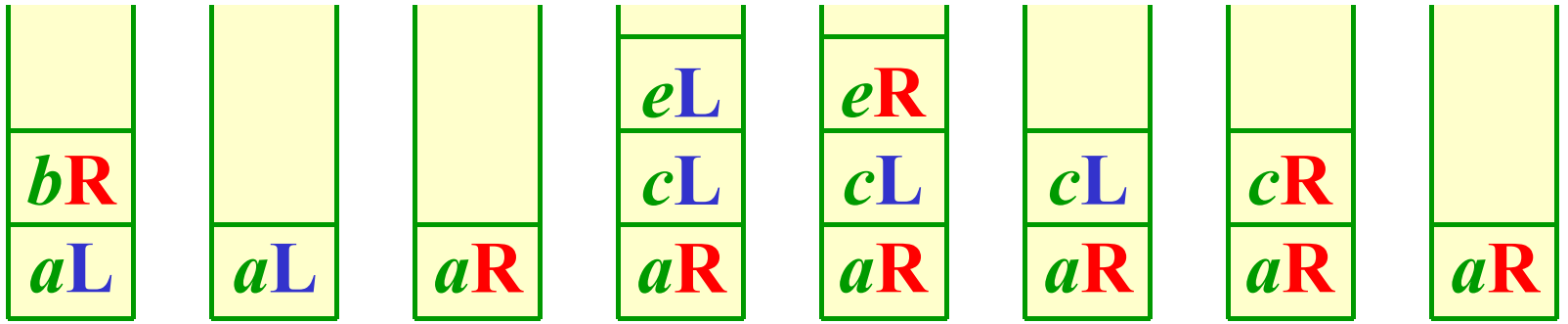
ptr	tag{L, R}

根结点的 tag = L，表示从左子树退出，访问右子树；

tag = R，表示从右子树退出，访问根。


a
b
c
d
e


aL
bL
aL
bR
aL
dL
bR
aL
dR
bR
aL


bR
aL
aL
aR
eL
cL
aR
eR
cL
aR
cL
aR
cR
aR
aR

```
template <class Type> void BinaryTree <Type> ::
PostOrder ( void ( *visit ) BinTreeNode <Type> *p ) {
    Stack < StkNode <Type> > S;
    stkNode <Type> w;
    BinTreeNode <Type> *p = root;
    do {
        while ( p != NULL ) {  //向左子树走
            w.ptr = p;  w.tag = L;  S.Push ( w );
            p = p->leftChild;
        }
        int continue = 1;  //继续循环标记
```

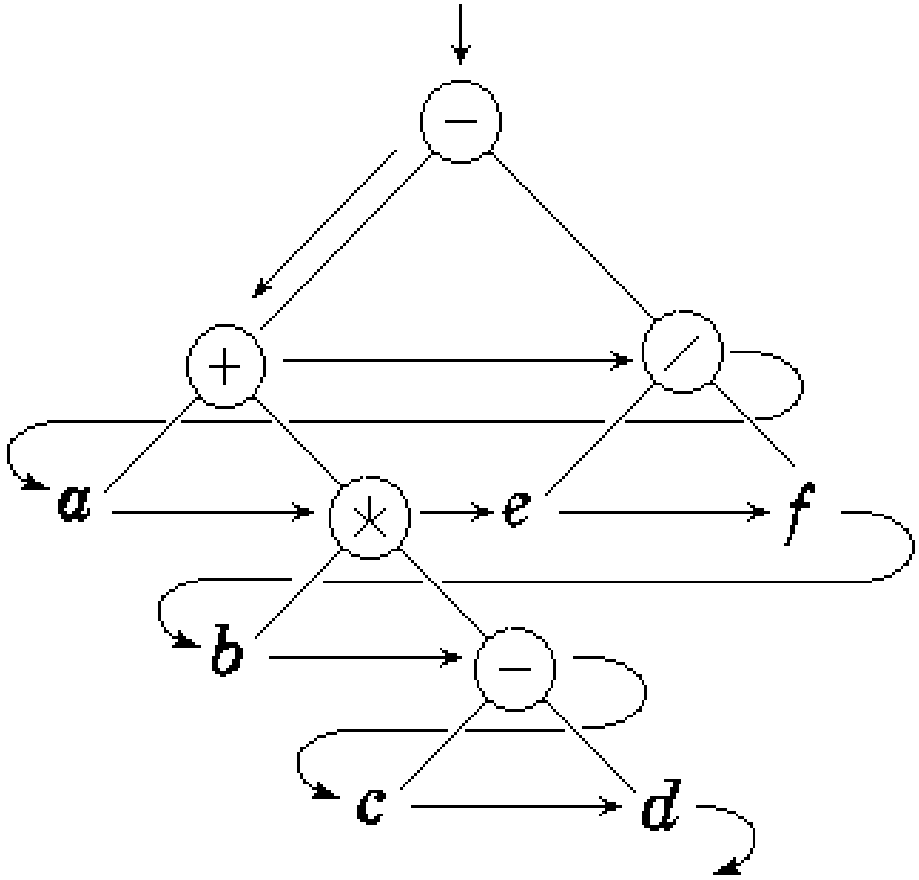
```
while ( continue && !S.IsEmpty ( ) ) {
    w = S.GetTop ( );  S.Pop ( );  p = w.ptr;
    switch ( w.tag ) {  //判断栈顶 tag 标记
      case L :  w.tag = R;  S.Push (w);
                continue = 0;
                p = p->rightChild;  break;
      case R :  visit ( p );  break;
    }
  }
} while ( p != NULL || !S.IsEmpty ( ) );
cout << endl;
}
```

- 由于对每个结点仅访问一次，无论是否用递归实现，前序、中序、后序遍历二叉树的**时间复杂度都是O(n)，空间复杂度也是O(n)**。

- 对前序、中序、后序来说，**每一层都只有一个结点时，需要的栈空间最大**。因为在二叉树遍历过程中，栈的最大深度等于二叉树的高度。在同样多的结点个数下，每一层只有一个结点时，二叉树的高度为最大。

层次序遍历算法

从二叉树根结点开始，自上向下，自左向右逐层访问树中各个结点。



遍历结果

- + / *a* * *e f b* - *c d*

层次序遍历使用FIFO队列，而不是使用栈来实现遍历。

当访问一个结点时，将其子女依次加到队列的队尾，然后再退出并访问已在队列队头的结点，这样可以实现树结点的按层访问。

```
template <class Type> void BinaryTree <Type> ::
LevelOrder ( void ( *visit ) BinTreeNode <Type> *p ) {
    Queue < BinTreeNode <Type>* > Q;
    BinTreeNode <Type> *p = root;
    Q.EnQueue ( p );
    while ( !Q.IsEmpty ( ) ) {
        Q.DeQueue ( p );
        visit (p);
        if ( p->leftChild != NULL )
            Q.EnQueue ( p->leftChild );
        if ( p->rightChild != NULL )
            Q.EnQueue ( p->rightChild );
    }
}
```

二叉树遍历的游标类 (Tree Iterator)

- 游标类负责二叉树的非递归遍历工作。
  - 每个游标类都提供四种功能：
    - 让控制定位到第一个结点；
    - 定位到下一个结点；
    - 测试是否到达二叉树最后一个结点；
    - 访问当前结点。
  - 结点访问顺序取决于遍历方式。

二叉树游标抽象基类

```
template <class Type> class TreeIterator {
public:
    TreeIterator ( const BinaryTree <Type> &BT )
            : T (BT), current (NULL) { }
    virtual ~TreeIterator ( ) { }
    virtual void First ( ) = 0;
    //虚函数：置第一个位置为当前位置
    virtual void operator ++ ( ) = 0;
    //虚函数：当前位置进一
    int operator + ( ) const { return current != NULL; }
    //判断是否树中有效位置
```

```
    const Type & operator ( ) ( ) const;
    //返回当前位置中的数据值
protected:
    const BinaryTree <Type> &T;
    BinTreeNode <Type> *current;
private:
    TreeIterator ( const TreeIterator & ) { }
    //构造函数
    TreeIterator & operator =
                ( const TreeIterator & ) const;
    //赋值
};
```

```
template <class Type>
const Type & TreeIterator <Type> ::
operator ( ) ( ) const {
    if ( current == NULL )
      { cout << "非法访问" << endl;  exit (1); }
    return current->GetData ( );
    //返回当前位置中的数据值
}
```

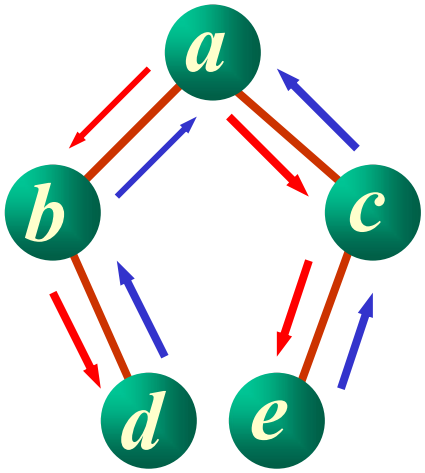
后序游标类

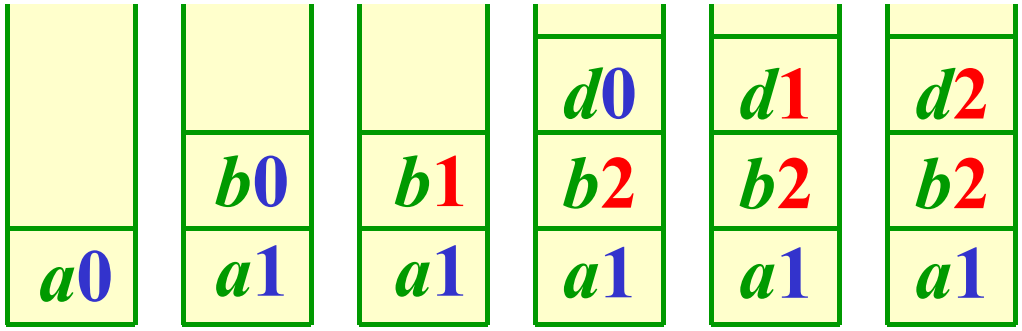
- 为记忆走过的结点，设置一个工作栈：

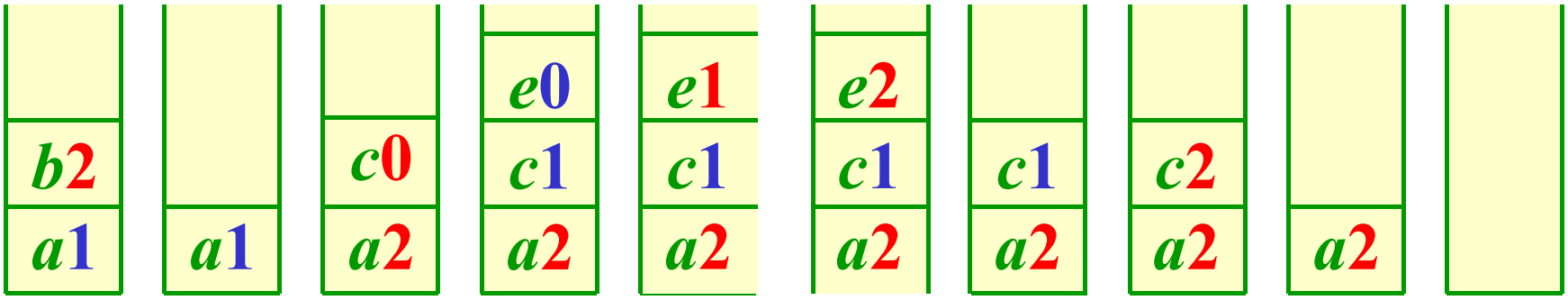
结点地址 *Node	退栈次数 S.PopTim

- S.PopTim = 0, 1或2，表示第几次退栈，用以判断下一步向哪一个方向走。
- 后序遍历时，每遇到一个结点，先把它推入栈中，让S.PopTim = 0。

- 在遍历其左子树前，改结点的S.PopTim = 1，将其左子女推入栈中。在遍历完左子树后，还不能访问该结点，要继续遍历右子树，此时改结点的S.PopTim = 2，并把其右子女推入栈中。在遍历完右子树后，结点才退栈访问。


a
b
c
d
e


a0
b0
a1
b1
a1
d0
b2
a1
d1
b2
a1
d2
b2
a1


b2
a1
a1
c0
a2
e0
c1
a2
e1
c1
a2
e2
c1
a2
c1
a2
c2
a2
a2

后序遍历游标类定义

```
template <class Type> struct StkNode {
//后序遍历使用的递归工作栈结点定义
    const BinTreeNode <Type> *Node;
    //结点指针
    int PopTim;  //退栈次数
    StkNode ( BinTreeNode <Type> *N = NULL ) :
        Node (N), PopTim (0) { }
};
```

```
template <class Type> class PostOrder :
    public TreeIterator <Type> {
public:
    PostOrder ( const BinaryTree <Type> &BT );
    ~PostOrder ( ) { }
    void First ( );
    //定位到后序次序下第一个结点
    void operator ++ ( );
    //定位到后序次序下的下一个结点
protected:
    Stack < StkNode <Type> > st;  //工作栈
};
```

- **构造函数*PostOrder***
  - 初始化工作栈；
  - 把根结点进栈。
- ***First***
  - 清除栈中元素；
  - 让根结点进栈。
- **operator ++**
  - 测试栈是否为空，如果栈空，则遍历完成，设置当前指针 *current* 为空并返回。
  - 否则，反复执行退栈和进栈操作，直到结点第3次从栈中退出，才能访问该结点。

后序游标类成员函数的实现

```
template <class Type> PostOrder <Type> ::
PostOrder ( const BinaryTree <Type> &BT ) :
    TreeIterator <Type> ( BT ) {  //构造函数
//对栈 st 进行初始化，并把根结点进栈
        st.Push ( StkNode <Type> ( T.GetRoot( ) ) );
}

template <class Type> void PostOrder <Type> ::
First ( ) {
    st.MakeEmpty ( );
    if ( T.GetRoot ( ) != NULL )
      st.Push ( StkNode <Type> ( T.GetRoot ( ) ) );
    operator ++ ( );  //进到后序下一个结点
}
```

```
template <class Type> void PostOrder <Type> ::
operator ++ ( ) {
    if ( st.IsEmpty ( ) ) {
      if ( current == NULL ) {
        cout <<"已经遍历结束" << endl;  exit (1);
      }
      current = NULL;  return;  //遍历完成
    }
    StkNode <Type> Cnode;
    for ( ; ; ) {  //循环，找后序下的下一个结点
      Cnode = st.Pop ( );
      if ( ++Cnode.PopTim == 3 )  //从右子树退出
      { current = Cnode.Node;  return; }
```

```
        st.Push ( Cnode );  //否则加一进栈
        if ( Cnode.PopTim == 1 ) {  //左子女进栈
          if ( Cnode.Node->GetLeft ( ) != NULL )
             st.Push ( StkNode <Type>
                        ( Cnode.Node->GetLeft ( ) ) );
        }
        else {  // =2，右子女进栈
           if ( Cnode.Node->GetRight ( ) != NULL )
              st.Push ( StkNode <Type>
                         ( Cnode.Node->GetRight ( ) ) );
        }
    }
}
```

中序游标类

最先访问的是最左下的结点，在走向该结点的过程中，利用栈记下沿途经过的结点。在访问完该结点之后，位于栈顶的正好是其双亲结点，将该双亲结点从栈中退出立即访问它，再向右子树方向走。如果右子树非空，又可重复上述过程。当栈中无任何元素，则表明所有元素已访问完，此时退出循环，结束遍历过程。

中序遍历与后序遍历走过的路线一样，只是访问结点在第2次退栈，即从左子树退出时访问结点，再遍历右子树。

在函数执行return之前应将该结点的非空右子女压入栈中，在下一次调用函数operator++时可从遍历右子女开始继续下去。

中序遍历也要使用递归工作栈 *st*。

中序游标类定义

```
template <class Type> class InOrder :
    public PostOrder <Type> {
public:
    InOrder ( BinaryTree <Type> &BT ) :
                        PostOrder <Type> ( BT ) ( ) { }
    ~InOrder ( ) { }
    void First ( );
    //定位到中序次序的第一个结点
    void operator ++ ( );
    //定位到中序次序的下一个结点
};
```

```
template <class Type> InOrder <Type> ::
InOrder ( const BinaryTree <Type> &BT ) :
TreeIterator <Type> ( BT ) {  //构造函数
//对栈 st 进行初始化，并把根结点进栈
      st.Push ( BT.GetRoot ( ) ); }

template <class Type> InOrder <Type> :: First ( ) {
   st.MakeEmpty ( );  //置空栈
   if ( BT.GetRoot ( ) ) st.Push ( BT.GetRoot ( ) );
   //根结点进栈
   operator ++ ( );  //进到中序下一个结点
}
```

中序游标类operator ++( )操作的实现

```
template <class Type> void InOrder <Type> ::
operator ++ ( ) {
    if ( st.IsEmpty ( ) ) {
    //当栈空且 current 也为空，则遍历完成
        if ( current == NULL )
        { cout << "已经遍历完成" << endl;  exit (1); }
        current = NULL;  return;
    }
    StkNode <Type> Cnode;
```

```
for ( ; ; ) { //循环，找中序下的下一个结点
  Cnode = st.Pop ( ); //退栈
  if ( ++Cnode.PopTim == 2 ) {
  //从左子树退出，退栈计数为2，无左子女结点
    current = Cnode.Node; //成为当前结点
    if ( Cnode.Node->GetRight ( ) != NULL )
      st.Push ( StkNode <Type>
                ( Cnode.Node->GetRight ( ) ) );
     return;
  }
```

```
        st.Push ( Cnode );  //否则加一进栈
        if ( Cnode.Node->GetLeft ( ) != NULL )
            st.Push ( StkNode <Type>
                        (Cnode.Node->GetLeft ( ) ) );
            //左子女进栈
    }
}
```

前序游标类

前序遍历与中序遍历走过的路线一样，只是访问结点是在第1次退栈之后进行。

在把当前指针指向该结点后，执行return前应先将该结点的右子女，再将该结点的左子女压入栈中，以使将来退栈顺序正好相反，让左子女先于右子女访问。

前序游标类定义

```
template <class Type> class PreOrder :
    public TreeIterator <Type> {
public:
    PreOrder ( BinaryTree <Type> &BT ) ;
    ~PreOrder ( ) { }
    void First ( );
    //定位到前序次序的第一个结点
    void operator ++ ( );
    //定位到前序次序的下一个结点
protected:
    Stack < StkNode <Type> > st;  //工作栈
};
```

```
template <class Type> PreOrder <Type> ::
PreOrder ( const BinaryTree <Type> &BT ) :
TreeIterator <Type>  ( BT ) {  //构造函数
//对栈 st 进行初始化，并把根结点进栈
      st.Push ( BT.GetRoot ( ) );
}

template <class Type> PreOrder <Type> :: First ( ) {
   st.MakeEmpty ( );  //置空栈
   if ( BT.GetRoot ( ) ) st.Push ( BT.GetRoot ( ) );
   //根结点进栈
   operator ++ ( );  //进到前序下一个结点
}
```

前序游标类operator ++( )操作的实现

```
template <class Type> void PreOrder <Type> ::
operator ++ ( ) {
if ( st.IsEmpty ( ) ) {
//当栈空且 current 也为空，则遍历完成
     if ( current == NULL ) {
          cout << "已经遍历完成" << endl;  exit (1);  }
          current = NULL;  return;  }
    current = st.Pop ( );
     if ( current->GetRight ( ) != NULL )
          st.Push ( current->GetRight ( ) );
     if ( current->GetLeft ( ) != NULL ) {
          st.Push ( current->GetLeft ( ) );
}
```

层次序游标类定义

```
template <class Type> class LevelOrder :
    public TreeIterator <Type> {
public:
    LevelOrder ( const BinaryTree <Type> &BT );
    ~LevelOrder ( ) { }
    void First ( );  //定位到层次次序的第一个结点
    void operator ++ ( );
    //定位到层次次序的下一个结点
protected:
    Queue < const BinTreeNode <Type> * > qu;
};
```

层次序游标类的operator ++ ( )操作

```
template <class Type> LevelOrder <Type> ::
    LevelOrder ( const BinaryTree <Type> &BT ) :
        TreeIterator <Type> ( BT ) //构造函数
//对队列 qu 进行初始化，并把根结点进队列
{ qu.EnQueue ( T.GetRoot ( ) ); }

template <class Type> viod LevelOrder <Type> ::
First ( ) {
//初始化：只有根结点在队列中
    qu.MakeEmpty ( );
    if ( T.GetRoot ( ) ) qu.EnQueue ( T.GetRoot ( ) );
    //根结点进队列
    operator ++ ( );
}
```

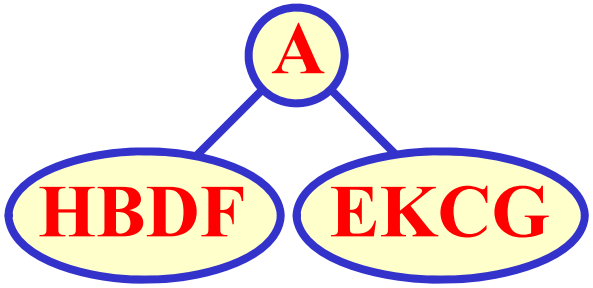
```
template <class Type> void LevelOrder <Type> ::
operator ++ ( ) {
   if ( qu.IsEmpty ( ) ) {  //当队空则遍历完成
     if ( current == NULL )
     {  cout << "已经遍历完成" << endl;  exit (1);  }
     current = NULL;  return;
   }
   current = qu.DeQueue ( );  //退队
   if ( current->GetLeft ( ) != NULL )  //左子女
      qu.EnQueue ( current->GetLeft ( ) );  //进队列
   if ( current->GetRight ( ) != NULL )  //右子女
      qu.EnQueue ( current->GetRight ( ) );  //进队列
}
```

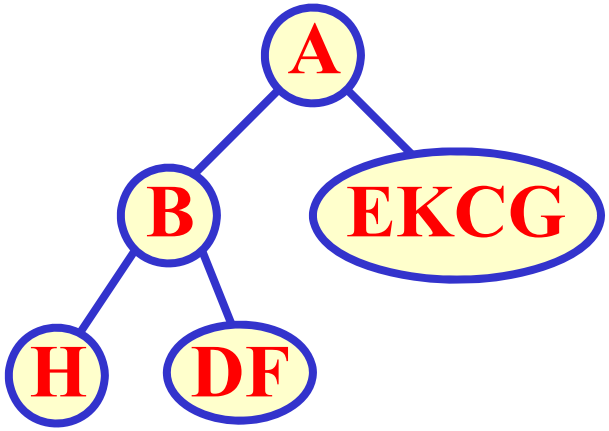
二叉树的计数

- 确定有n个结点的不同二叉树有多少种；
- 确定可以用栈得出的从1到n的数字有多少种不同的排列。

由二叉树的前序序列和中序序列可唯一地确定一棵二叉树。

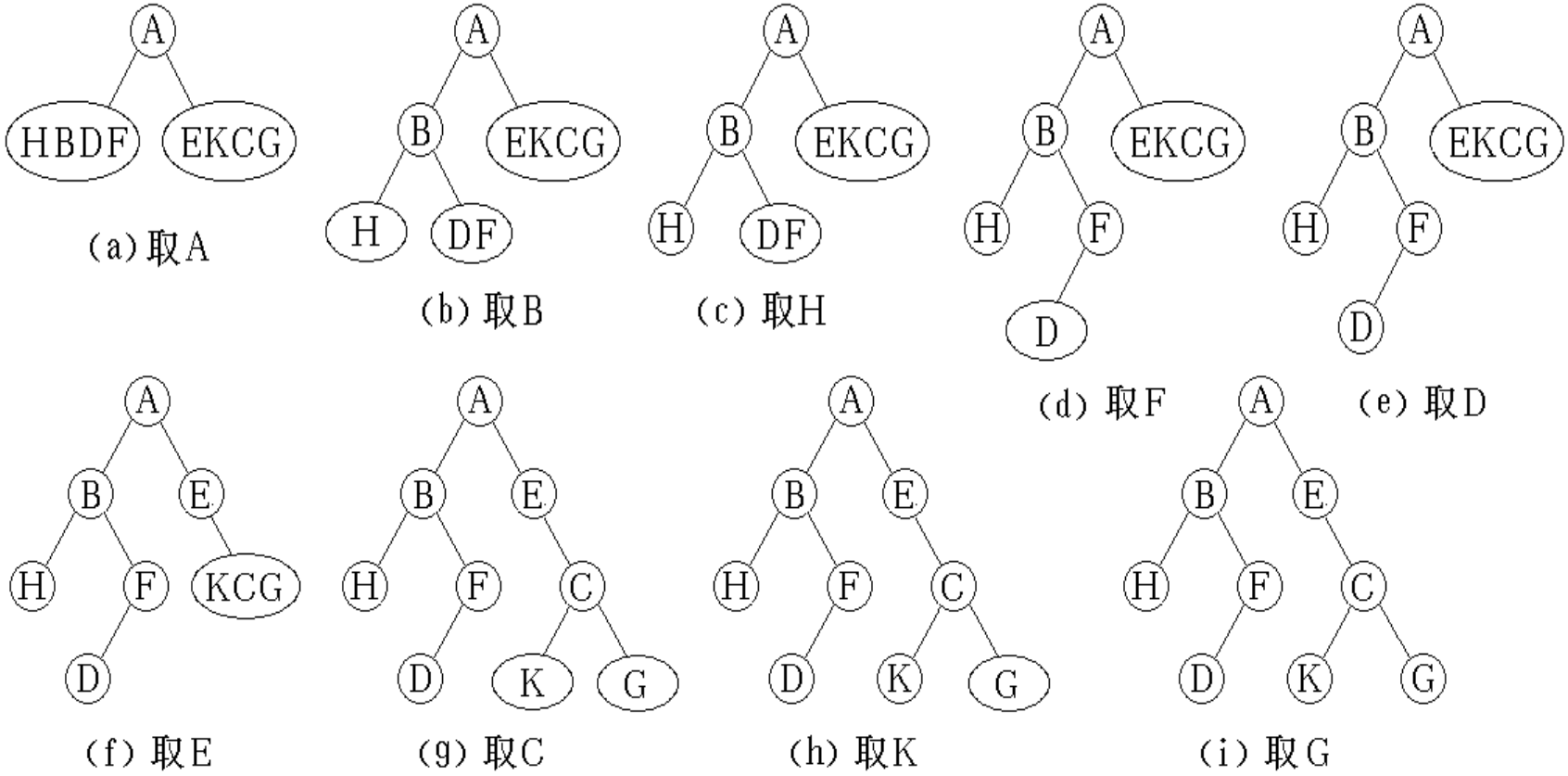
例，前序序列 { ABHFDECKG } 和中序序列 { HBDFAEKCG }，构造二叉树过程如下：





前序{ABHFDECKG}

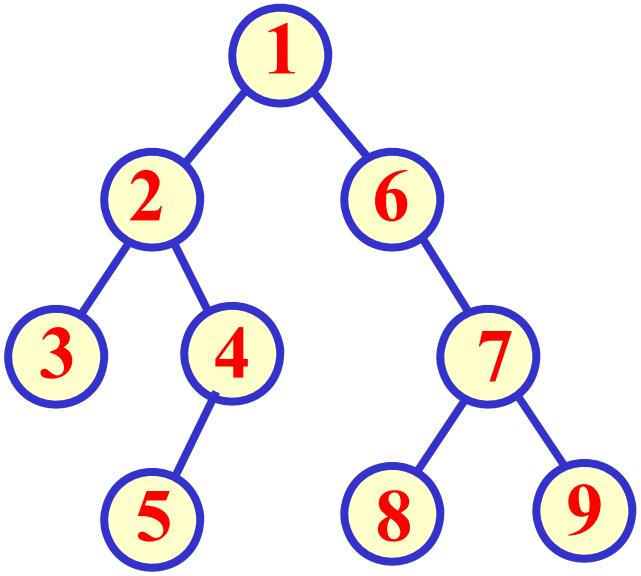
中序{HBDFAEKCG}

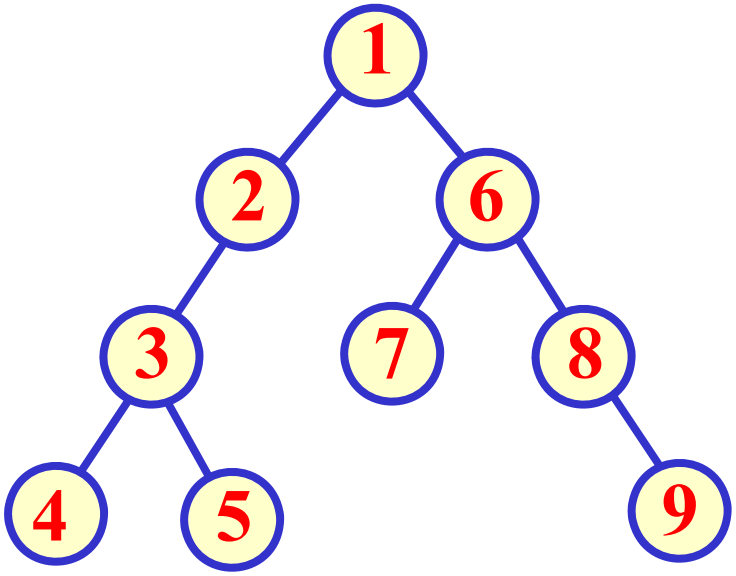


利用前序序列和中序序列构造二叉树

```
template <class Type> BinaryTree <Type> * BinaryTree <Type> ::
CreateBinaryTree ( Type *VLR, Type *LVR, int n ) {
    if ( n == 0 ) return NULL;
    int k = 0;
    while ( VLR[0] != LVR[k] ) k++;  //在中序序列中寻找根
    BinTreeNode <Type> *t =
                    new BinTreeNode <Type> ( VLR[0] );  //创建根结点
    t->leftChild = CreateBinaryTree ( VLR+1, LVR, k );
    //从前序 VLR+1 开始对中序的 0~k-1 左子序列
    //的 k 个元素递归建立左子树
    t->rightChild = CreateBinaryTree ( VLR+k+1, LVR+k+1, n-k-1 );
    //从前序 VLR+k+1 开始对中序的 k+1~n-1 右子序列
    //的 n-k-1 个元素递归建立右子树
    return t;
}
```

如果前序序列固定不变，给出不同的中序序列，可得到不同的二叉树。



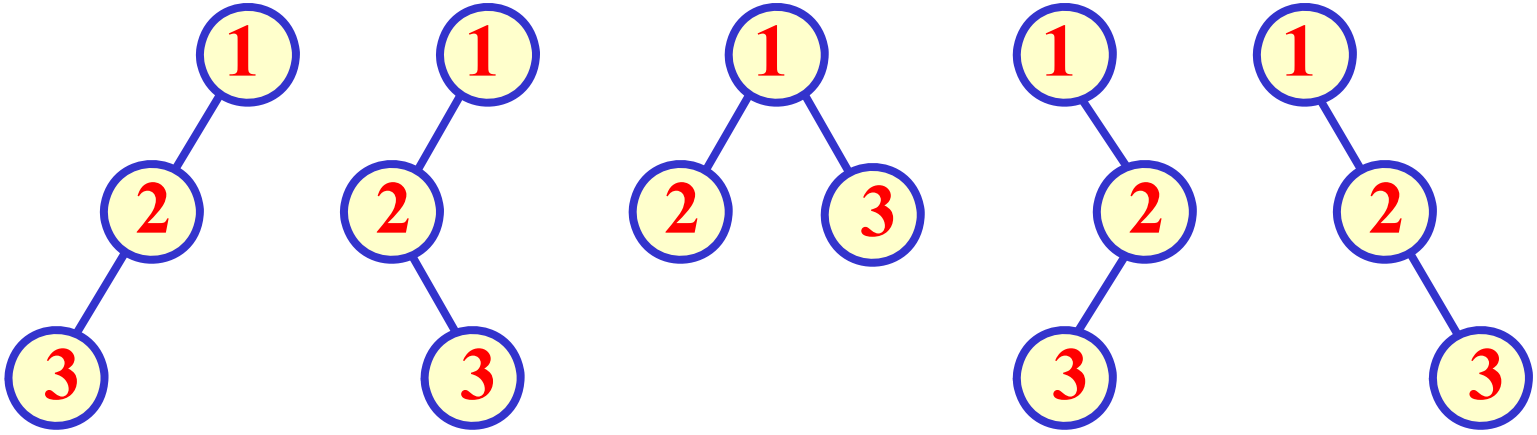


前序 { 123456789 }

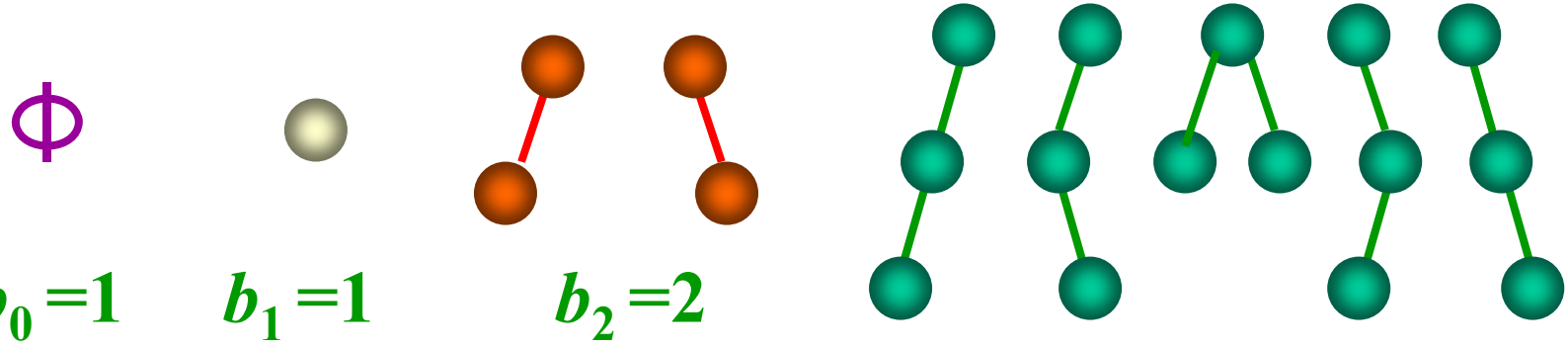
中序 { 325416879 }　　中序 { 435217689 }

问题是有 n 个数据值，可能构造多少种不同的二叉树？可以固定前序排列，选择所有可能的中序排列。

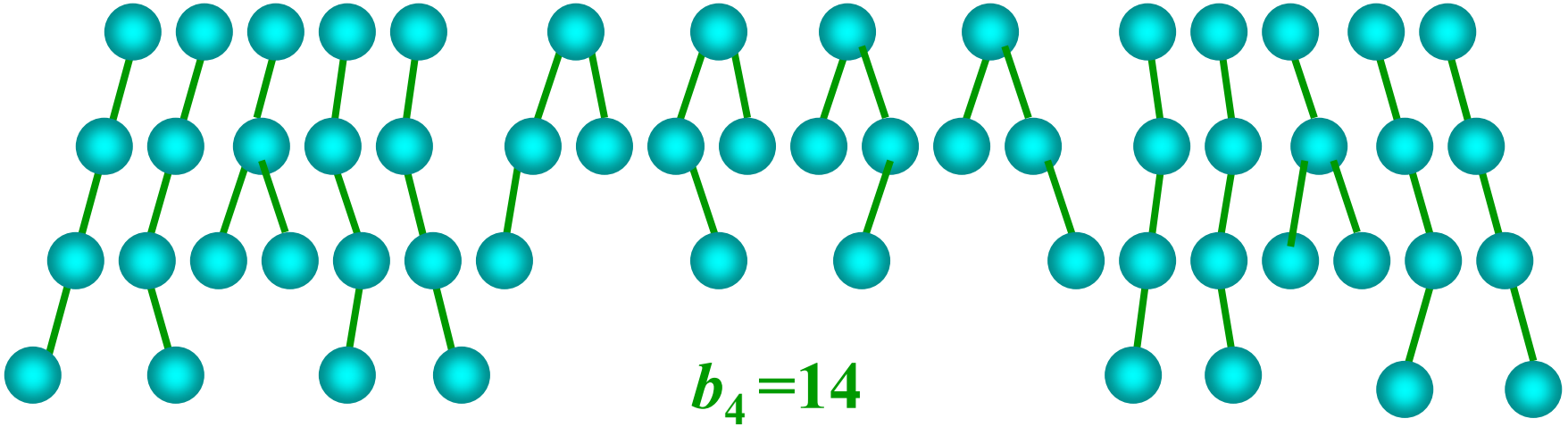
例如，有 3 个数据 { 1, 2, 3 }，可得 5 种不同的二叉树。它们的前序排列均为 123，中序序列可能是 123, 132, 213, 231, 321。



有0, 1, 2, 3, 4个结点的不同二叉树如下：



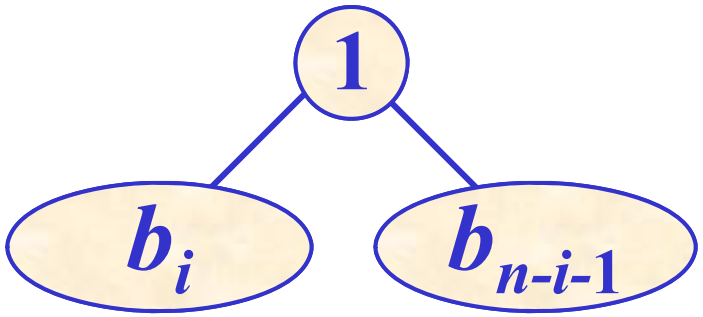
$b_0=1$ $b_1=1$ $b_2=2$ $b_3=5$



$b_4=14$

计算具有 n 个结点的不同二叉树的棵数

$$b_n = \sum_{i=0}^{n-1} b_i \cdot b_{n-i-1}$$



- b_i表示二叉树中由i个结点的左子树；
- b_{n-i-1}表示二叉树中由$n-i-1$个结点的右子树；
- $b_i * b_{n-i-1}$表示由根结点、b_i左子树和b_{n-i-1}右子树组成的一棵二叉树。
  - 不同二叉树棵数等于左子树上可能的不同二叉树棵数与右子树上可能的不同二叉树棵数的乘积。

例如，具有$n=4$个结点的不同二叉树

- 当$i=0$时，$b_0*b_3=1*5=5$；
- 当$i=1$时，$b_1*b_2=1*2=2$；
- 当$i=2$时，$b_2*b_1=2*1=2$；
- 当$i=3$时，$b_3*b_0=5*1=5$；
- $b_4=5+2+2+5=14$

*Catalan*函数

$$b_n = \frac{1}{n+1} C_{2n}^{n} = \frac{1}{n+1} \frac{(2n)!}{n! \cdot n!}$$

$$b_3 = \frac{1}{3+1} C_6^3 = \frac{1}{4} \frac{6 \cdot 5 \cdot 4}{3 \cdot 2 \cdot 1} = 5$$

$$b_4 = \frac{1}{4+1} C_8^4 = \frac{1}{5} \frac{8 \cdot 7 \cdot 6 \cdot 5}{4 \cdot 3 \cdot 2 \cdot 1} = 14$$

3-1 铁路进行列车调度时，常把站台设计成栈式结构的站台。

(1) 设有编号为1, 2, 3, 4, 5, 6的六辆列车，顺序入栈式构的站台，则可能的出栈序列有多少种?

(2) 若进站的六辆列车顺序如上所述，那么是否能够得到435612、325641、154623和135426的出栈序列，如果不能，说明为什么不能；如果能，说明如何得到（即写出“进栈”或“出栈”的序列）。

基本思路：

6辆火车顺序进栈，1号火车最先压进栈中。接下来，有可能1号火车直接出栈，2号以及其它火车再顺序进栈，在这些火车进栈的过程中，可能有的火车在其后边火车进栈之前已出栈。也有可能，1号火车仍然保留在栈底，2号以及其它火车再顺序进栈，那么在这些火车进栈的过程中，可能有的火车在其后边火车进栈之前已出栈，而且也有可能1号火车在其它火车进栈的过程中也要出栈。问题所要求解的就是有多少种这样的出栈序列。

解法：

因为1号火车最先压入栈底，将1号火车作为一个基点，1号火车可作为栈顶存在，也可作为非栈顶存在。如果1号火车作为栈顶，也就是1号火车上面未压入任何火车，1号肯定最先出栈，那么如果想求解出栈序列种数，就只需考虑其它5辆火车的进栈出栈组合数就可以了。如果1号火车上面已经压入若干火车，而且除这些火车以及1号火车之外，剩余火车仍然在栈外，这时不仅需要考虑1号火车上面已压入栈中的若干火车的进栈出栈组合数，而且也要考虑栈外剩余火车的进栈出栈组合数，将这两个组合数相乘，才是总的组合数。

解法（续）：

从另一个角度考虑，将第一列火车作为二叉树的根结点，将1号火车出栈之前进行进栈出栈操作的火车数目记为i，i个火车序列相当于根结点的左子树，剩余$n-i-1$个火车进栈出栈序列相当于根结点的右子树。这样，归结为给定123456顺序排列的6辆火车，相当于给定前序序列123456，求解存在多少种中序序列，即确定有多少棵不同的二叉树。

随堂练习

例1：一棵完全二叉树上由1001个结点，其中叶子结点的个数为_____。一棵有124个叶结点的完全二叉树，最多有_____个结点。

例2：设深度为k (k>=0)的二叉树上只有度为0和度为2的结点，则这类二叉树上所含的结点总数最少为_____，最多为_____。设n0为哈夫曼树的叶子结点数目，则该哈夫曼树共有____个结点。

例3：在一棵二叉树的前序序列、中序序列和后序序列中，任意两种序列的组合可以唯一地确定这棵二叉树吗？若可以，有哪些组合，证明之；若不可以，有哪些组合，证明之。

例1：一棵完全二叉树上由1001个结点，其中叶子结点的个数为 501 。一棵有124个叶结点的完全二叉树，最多有 248 个结点。

(1) 由二叉树性质知：n0=n2+1，且完全二叉树的n1=0或1；已知二叉树的总结点数n=n0+n1+n2，即有n=2n0+n1-1；将总结点数1001代入得：1001=2n0+n1-1，因1001为奇数，故n1=0，得n0=501。

(2) 由二叉树性质知：n0=n2+1，n2=123。

例2：设深度为k (k>=0)的二叉树上只有度为0和度为2的结点，则这类二叉树上所含的结点总数最少为 2k+1 ，最多为 $2^{k+1}-1$ 。设n0为哈夫曼树的叶子结点数目，则该哈夫曼树共有 2n0-1 个结点。

由哈夫曼树构造方法知，对n0个叶子结点构造哈夫曼树需要进行n0-1次构造新结点操作，最终形成共有2n0-1个结点的哈夫曼树，即2n0-1。

例3：在一棵二叉树的前序序列、中序序列和后序序列中，任意两种序列的组合可以唯一地确定这棵二叉树吗？若可以，有哪些组合，证明之；若不可以，有哪些组合，证明之。

前序序列和中序序列、后序序列和中序序列可以唯一确定一棵二叉树，而前序序列和后序序列不能。

例4：假设二叉树用左右链表示，试编写一算法，判别给定二叉树是否为完全二叉树。

例5：设计算法：统计一棵二叉树中所有叶结点的数目。

例6：假设二叉树用左右链表示，试编写一算法，求任意二叉树中第一条最长的路径，并输出此路径上各结点的值。

例4：假设二叉树用左右链表示，试编写一算法，判别给定二叉树是否为完全二叉树。

根据完全二叉树定义可知，对完全二叉树按照从上到下，从左到右的次序遍历应满足：

(1) 若某结点没有左孩子，则一定无右孩子；

(2) 若某结点缺（左或右）孩子，则其所有后继一定无孩子。

反之，可采用按层次序遍历二叉树的方法依次对每个结点进行判断。为此增加一个标志以表示所有已扫描过的结点均有左、右孩子，将局部判断结果存入CM中，CM表示整个二叉树是否是完全二叉树，B为1表示到目前为止所有结点均有左右孩子。

例5：设计算法：统计一棵二叉树中所有叶结点的数目。

可将此问题视为一种特殊的遍历问题，这种遍历中“访问一个结点”的内容为判断该结点是否为树叶，若是树叶则叶子数加1。显然，可以采用任何遍历方法，在此采用前序遍历方法。下面算法中记录叶子数的count初值假定为0。

例6：假设二叉树用左右链表示，试编写一算法，求任意二叉树中第一条最长的路径，并输出此路径上各结点的值。

采用非递归后序遍历二叉树，当后序遍历访问到由p所指的树叶结点时，此时stack中所有结点均为p所指结点的祖先，由这些祖先便构成了一条从根结点到此树叶结点的路径。此外，另设一longestpath数组来保存二叉树中最长的路径结点值，m为最长的路径长度。

不用栈的二叉树中序遍历算法

- **使用栈进行二叉树遍历**
  - 最大问题—栈需要附加存储空间。
    - 在遍历一棵二叉树的过程中，栈的大小动态变化，与二叉树高度有关。
- **不使用栈的二叉树非递归遍历**
  - 在二叉树的每个结点中增加一个双亲域。
    - 沿着走过的路径退回到任何一个子树的根结点，再向另一个方向走。
  - 建立线索化二叉树
    - 要求在每一结点中增加两个标志以区分子女指针和线索。

5.5 线索化叉树 (Threaded Binary Tree)

动机

如何有效查找结点前序、中序或后序结点。

如何充分利用二叉树中为*NULL*的指针域。

n 个结点的二叉树共有指针域 2*n* 个，

其中，不为 *NULL* 的有 *n*-1 个；

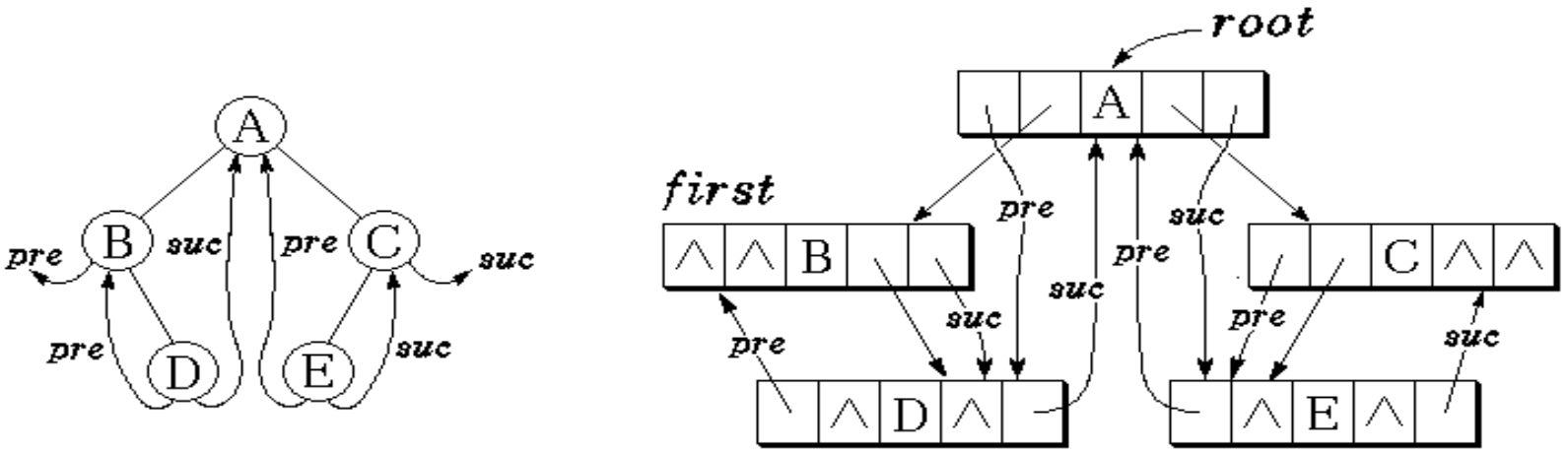
为 *NULL* 的有 *n*+1 个。

性质3 对任何一棵二叉树, 如果其叶结点个数为 n_0，度为2的非叶结点个数为 n_2，则有$n_0 = n_2 + 1$。

线索 (Thread)

希望找到二叉树中某个结点的前驱或后继，而不需要每次都对二叉树进行遍历一遍。可在二叉链表中增加一个前驱指针域和一个后继指针域，分别指向该结点在某种次序下的前驱和后继结点。

Pred	*leftChild*	*data*	*rightChild*	*Succ*



增加 *Pred* 指针和 *Succ* 指针的二叉树

(以中序遍历为例)

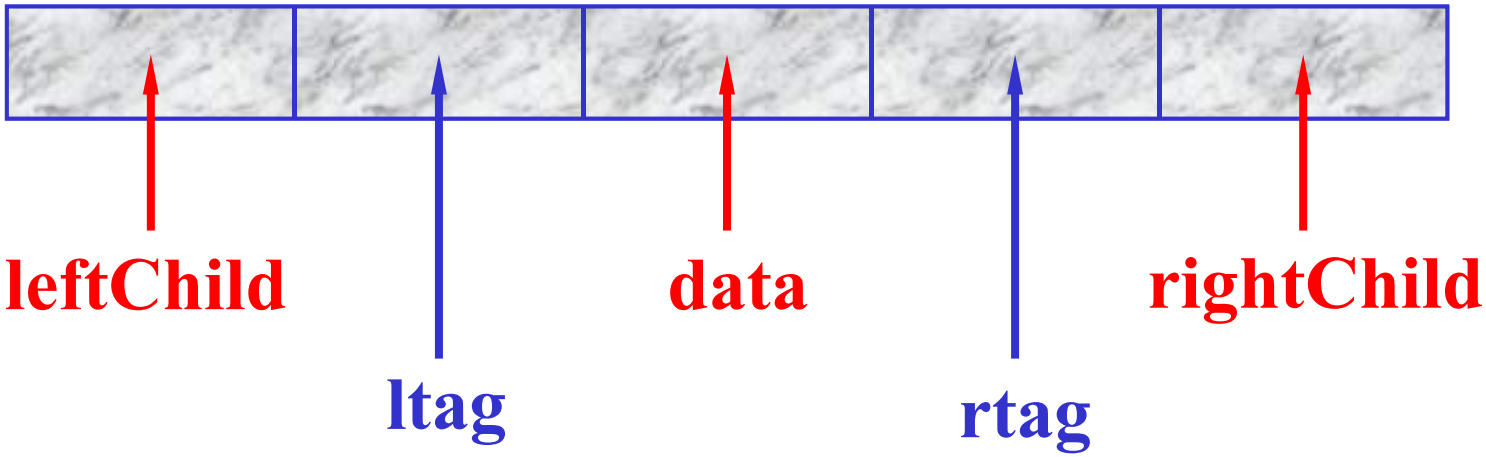
- **浪费存储空间**
  - 每个结点增加两个指针域；
  - 而原来的*leftChild*与*rightChild*指针域中存在许多空指针而未加利用。
- **为避免存储空间浪费**
  - 利用原来空指针域存放结点的前驱和后继指针。
    - 利用空的*leftChild*域存放结点的前驱结点指针；
    - 利用空的*rightChild*域存放结点的后继结点指针。

- **线索**—指示前驱与后继的指针。

- **线索二叉树**—加上线索的二叉树。
  - 由于存在线索，无需遍历就可得到任一结点的前驱与后继结点的指针。

- **线索二叉链表**—对应的二叉链表。

线索二叉树及其二叉链表表示

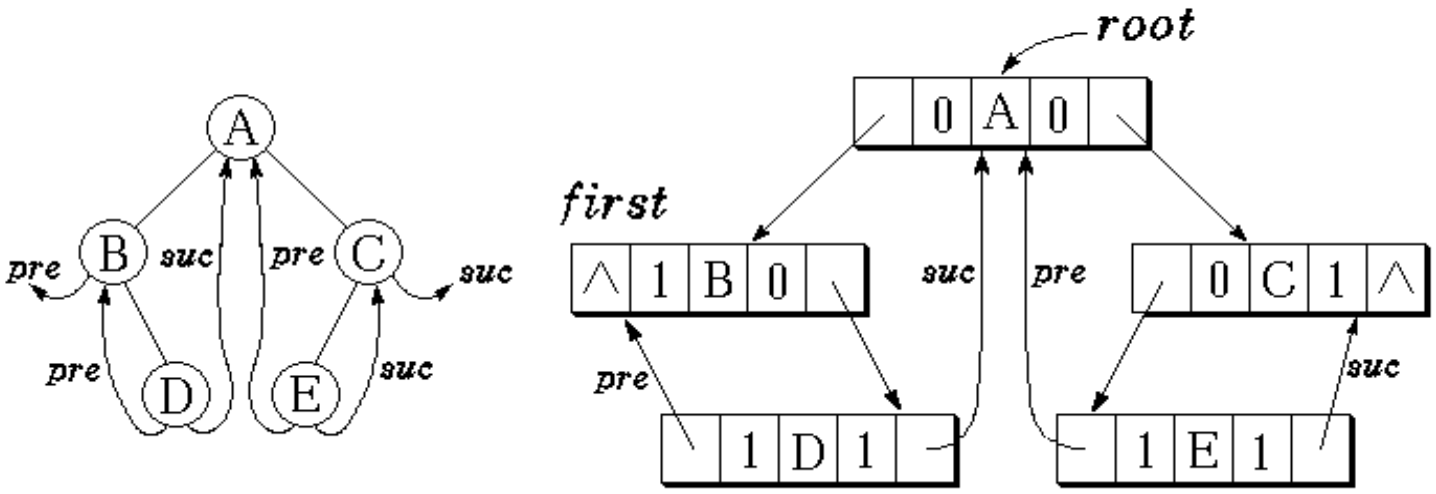


ltag=0，leftChild为左子女指针

ltag=1，leftChild为前驱线索

rtag=0，rightChild为右子女指针

rtag=1，rightChild为后继指针



有两个线索处于悬空状态：

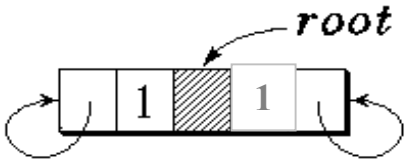
**树中中序下的第一个结点*B*的前驱线索；

**树中中序下的最后一个结点*C*的后继线索。

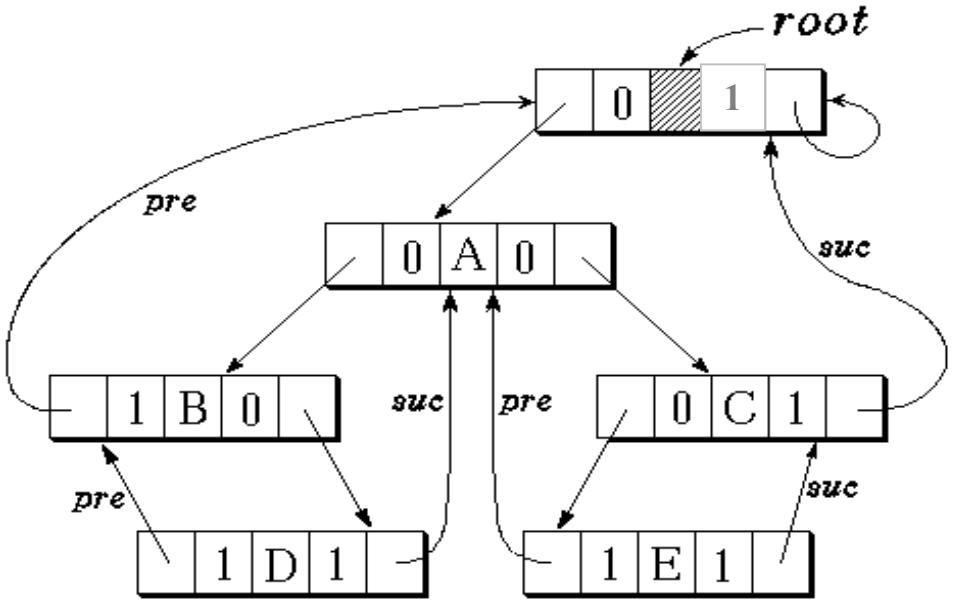
带表头结点的中序穿线链表

为避免线索悬空，引入表头结点至线索二叉树：

线索二叉树 ⇒ 表头结点的左子树



（a）空二叉链表



（b）非空二叉链表

中序线索二叉树类定义

```
template <class Type> class ThreadTree;
template <class Type> class ThreadNode {
friend class ThreadTree <Type>;
private:
    int ltag, rtag;
    ThreadNode <Type> *leftChild, *rightChild;
    Type data;
public:
    ThreadNode ( const Type item )
       : data (item), leftChild (NULL),
         rightChild (NULL), ltag (0), rtag (0) { }
};
```

```
template <class Type> class ThreadTree {
protected:
   ThreadNode <Type> *root;  //根
   void CreateInThread ( ThreadNode <Type> *current,
                         ThreadNode <Type> *&pre );  //建树
   ThreadNode <Type> * Parent ( ThreadNode <Type> *t );
public:
   ThreadTree ( ) : root (NULL) { };  //构造函数
   void CreateInThread ( );
   ThreadNode <Type> * First ( ThreadNode <Type> *current );
   ThreadNode <Type> * Last  ( ThreadNode <Type> *current );
   ThreadNode <Type> * Next ( ThreadNode <Type> *current );
   ThreadNode <Type> * Prior ( ThreadNode <Type> *current );
   void InOrder ( void ( *visit ) ( ThreadNode <Type> *p ) );
   void PreOrder ( void ( *visit ) ( ThreadNode <Type> *p ) );
   void PostOrder ( void ( *visit ) ( ThreadNode <Type> *p ) );
   ……
};
```

在中序线索化二叉树中部分成员函数的实现

```
template <class Type>
ThreadNode <Type> * ThreadTree <Type> ::
First ( ThreadNode <Type> *current ) {
//函数返回以 current 为根的线索化二叉树
//在中序序列下的第一个结点
    ThreadNode <Type> *p = current;
    while ( p->ltag == 0 )
        p = p->leftChild;  //最左下的结点
    return p;
}
```

```
template <class Type>
ThreadNode <Type> * ThreadTree <Type> ::
Next ( ThreadNode <Type> *current ) {
//函数返回在线索化二叉树中结点 current
//在中序下的后继结点
   ThreadNode <Type> *p = current->rightChild;
   if ( current->rtag == 0 )
      return First (p);
   // rtag == 0 ， 表示有右子女
   else  return p;
   // rtag == 1 ， 直接返回后继线索
}
```

```
template <class Type>
ThreadNode <Type> * ThreadTree <Type> ::
Last ( ThreadNode <Type> *current ) {
    ThreadNode <Type> *p = current;
    while ( p->rtag == 0 )  p = p->rightChild;
    return p;
}

template <class Type>
ThreadNode <Type> * ThreadTree <Type> ::
Prior ( ThreadNode <Type> *current ) {
    ThreadNode <Type> *p = current->leftChild;
    if ( current->ltag == 0 )  return Last (p);
    else return p;
}
```

- **利用线索遍历二叉树**
  - 首先，利用First ( )运算找到二叉树在中序序列下的第一个结点，将其作为当前结点；
  - 然后，利用求后继结点的运算Next ( )按中序次序逐个访问，直到二叉树的最后一个结点。
  - 在遍历过程中，可以不用栈。

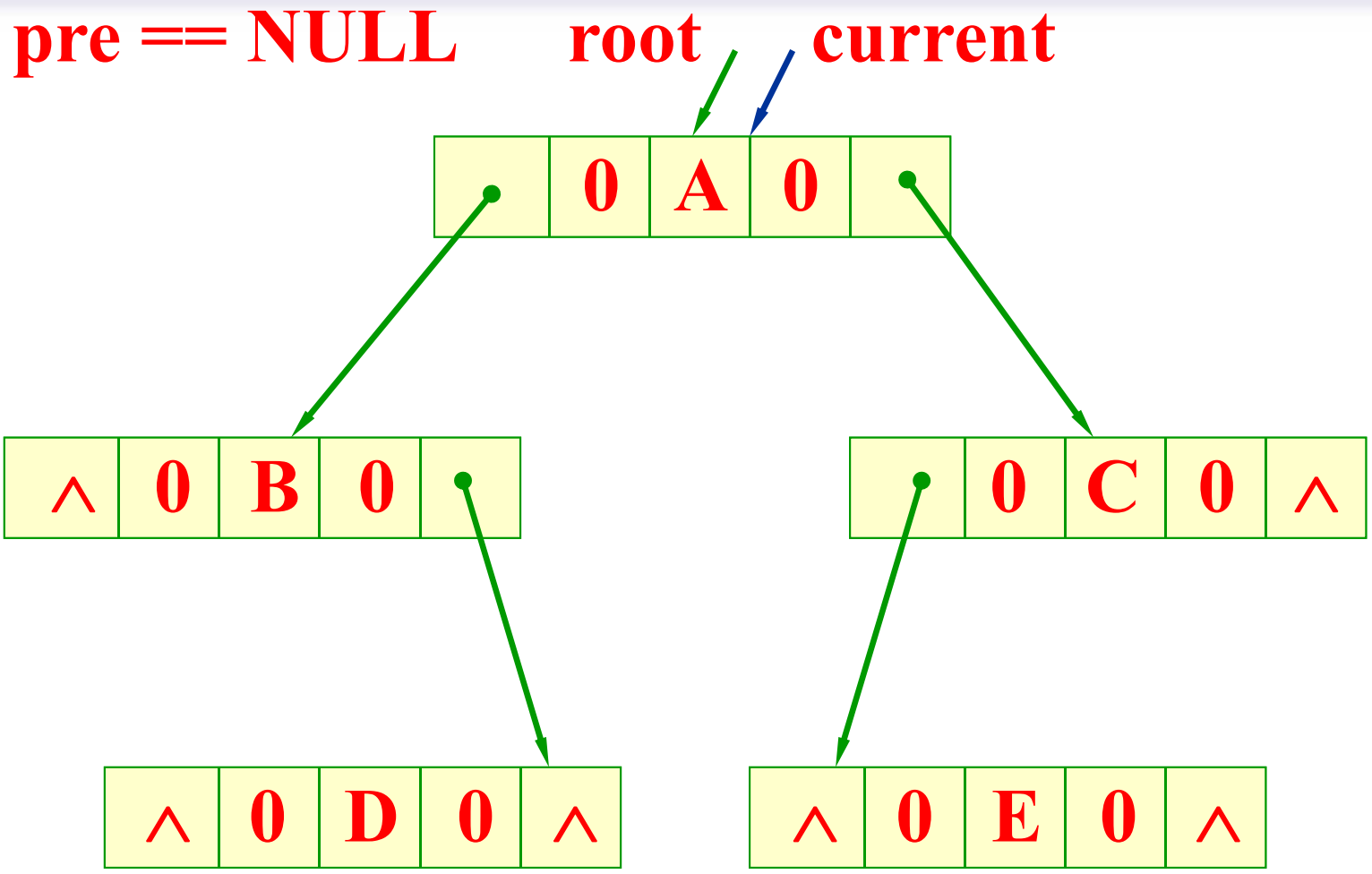
```
template <class Type> void ThreadTree <Type> ::
Inorder ( void ( *visit ) ( ThreadNode <Type> *p ) ) {
//线索化二叉树的中序遍历
    ThreadNode <Type> *p;
    for ( p = First (root); p != NULL; p = Next (p) )
        visit (p);
}
```

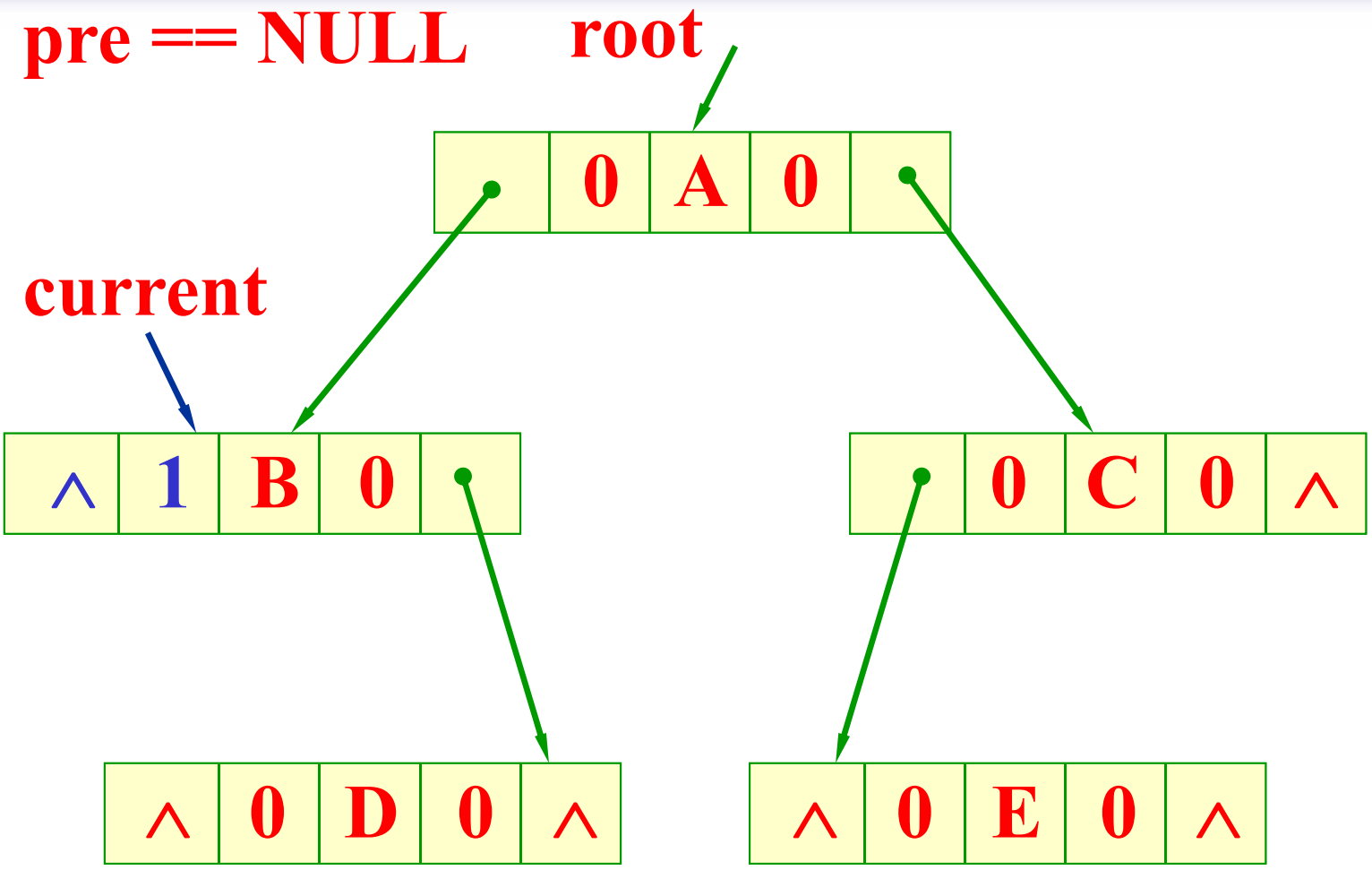
通过中序遍历建立中序线索化二叉树

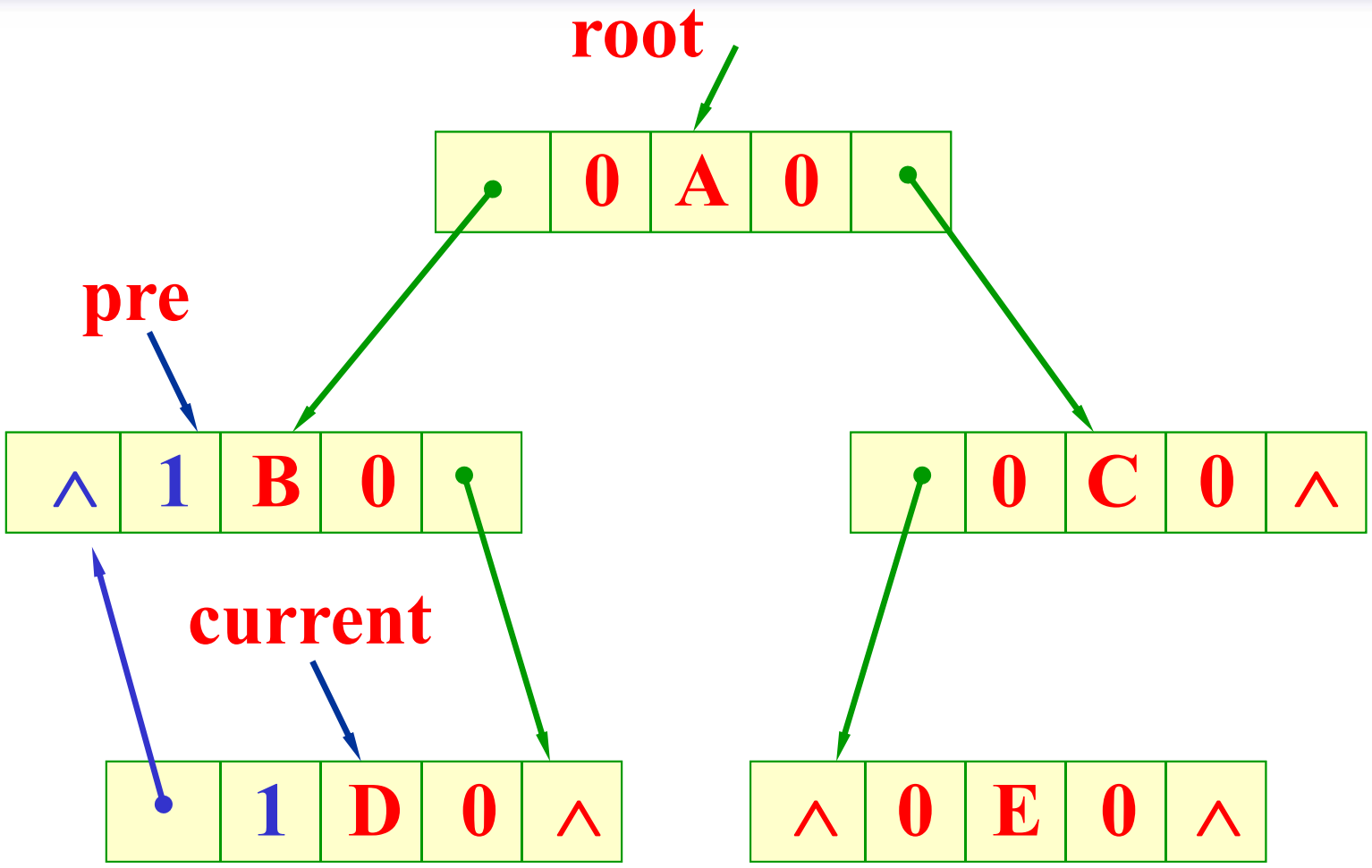
```
template <class Type> void ThreadTree <Type> ::
CreateInThread ( ThreadNode <Type> *current,
                 ThreadNode <Type> *&pre ) {
//通过中序遍历进行线索化
// pre 在遍历过程中总是指向遍历指针 p 在中序下的前驱结点
//即在中序遍历过程中刚刚访问过的结点
//在中序遍历时，每遇到空指针域，立即填入前驱或后继线索
    if ( current == NULL ) return;
    CreateInThread ( current->leftChild, pre ); //左子树线索化
    if ( current->leftChild == NULL ) {
        current->leftChild = pre;
        current->ltag = 1;
    } //建立当前结点的前驱线索
```

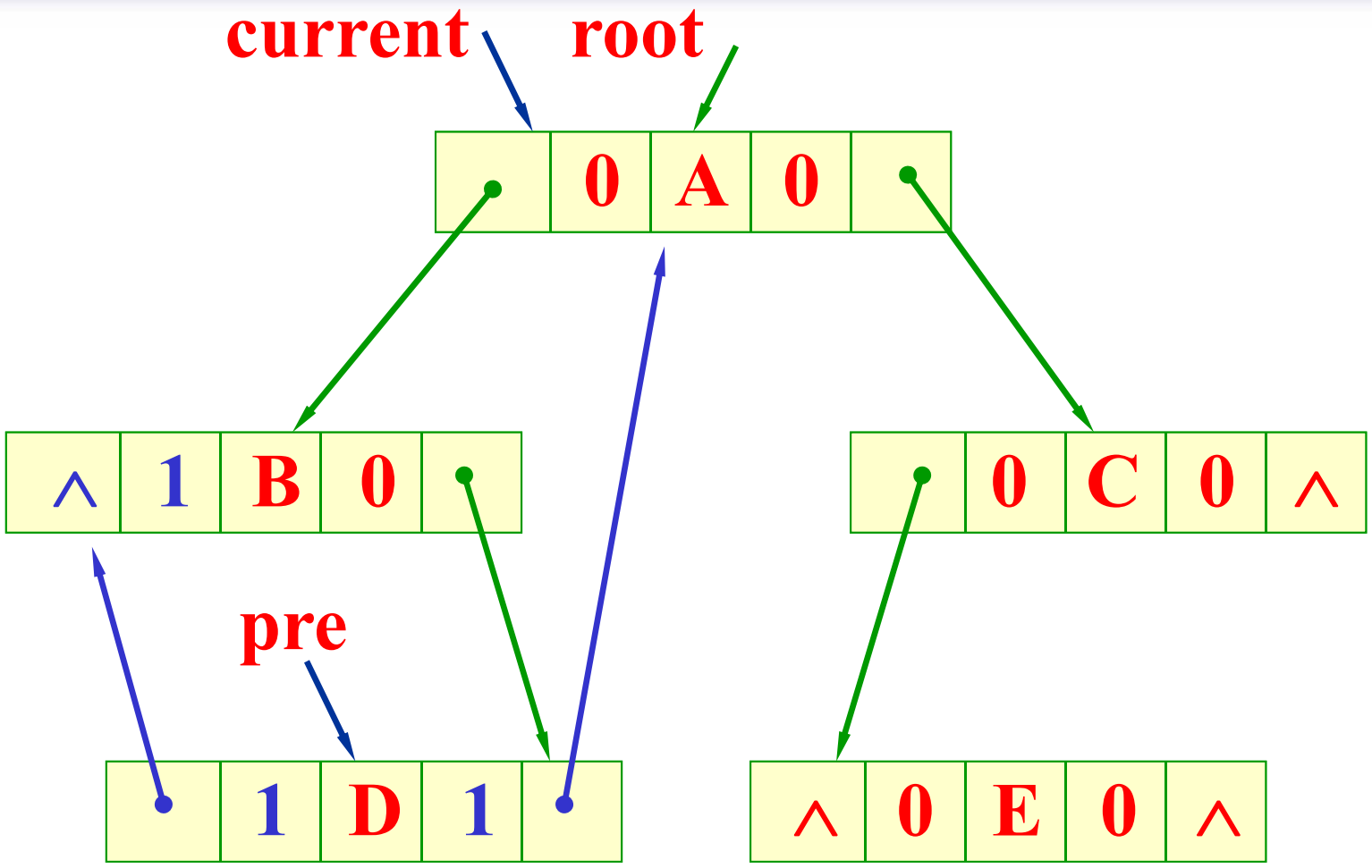
```
        if ( pre != NULL && pre->rightChild == NULL )
        {
            pre->rightChild = current;
            pre->rtag = 1;
        } //建立前驱结点的后继线索
        pre = current; //前驱跟上当前指针
        CreateInThread ( current->rightChild, pre );
        //递归，右子树线索化
    }
}
```

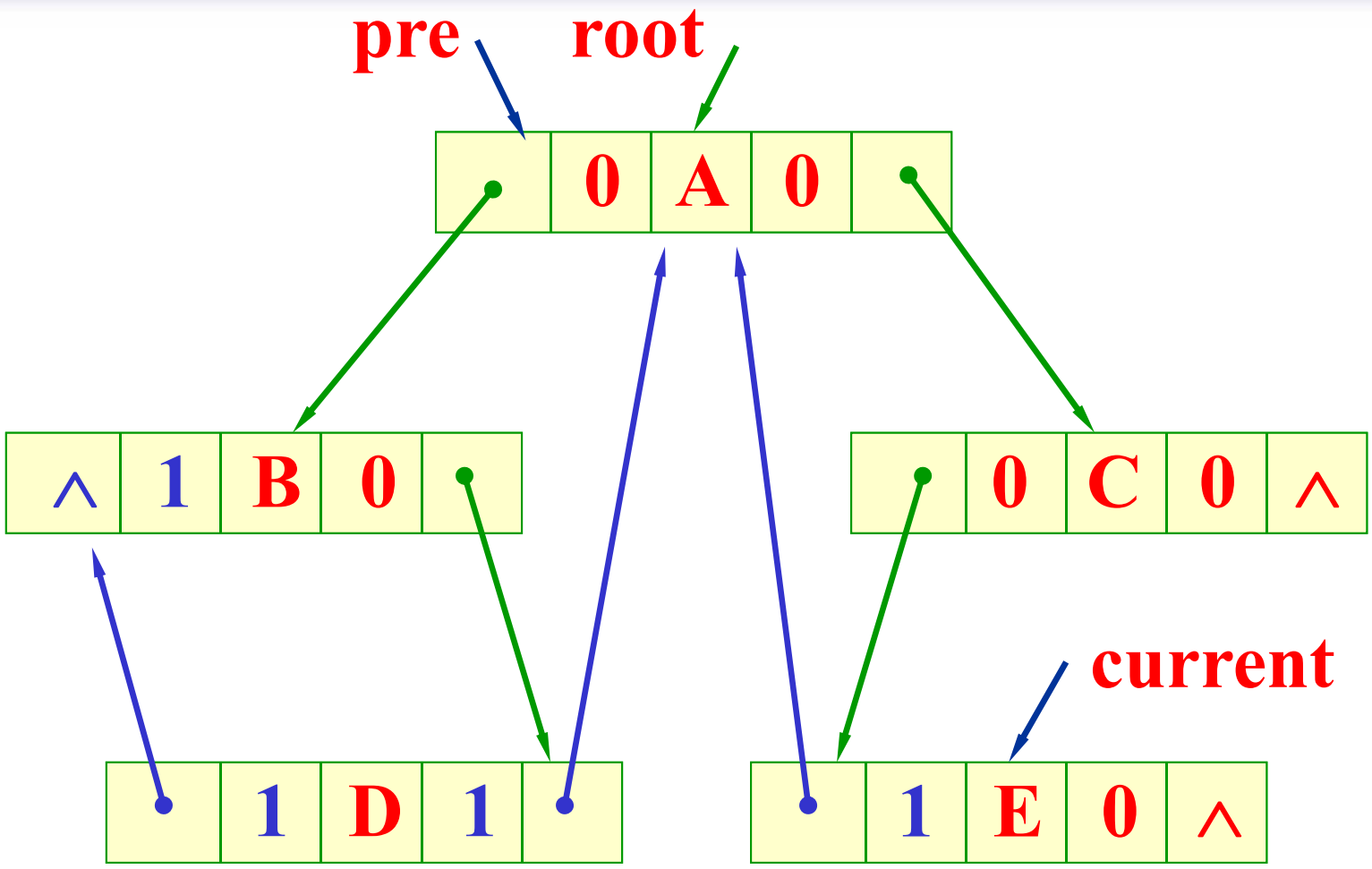
```
template <class Type> void ThreadTree <Type> ::
CreateInThread ( )
{
    ThreadNode <Type> *pre = NULL;
    //前驱指针
     if ( root != NULL ) {  //非空二叉树，线索化
        CreateInThread ( root, pre );
        //中序遍历线索化二叉树
        pre->rightChild = NULL;
        pre->rtag = 1;
        //后处理，中序最后一个结点
    }
}
```

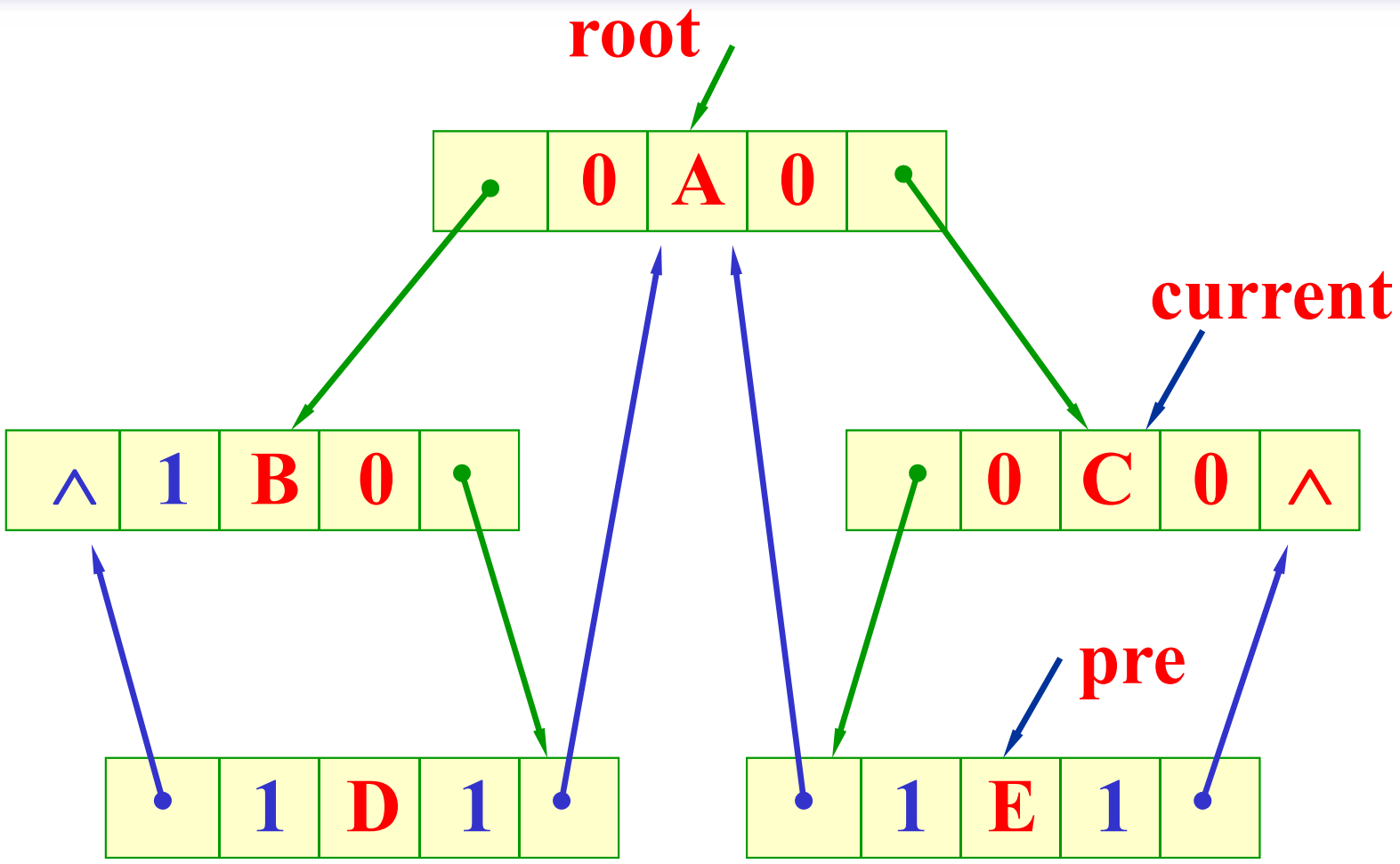

pre == NULL
root
current
0
A
0
∧
0
B
0
0
C
0
∧
∧
0
D
0
∧
∧
0
E
0
∧

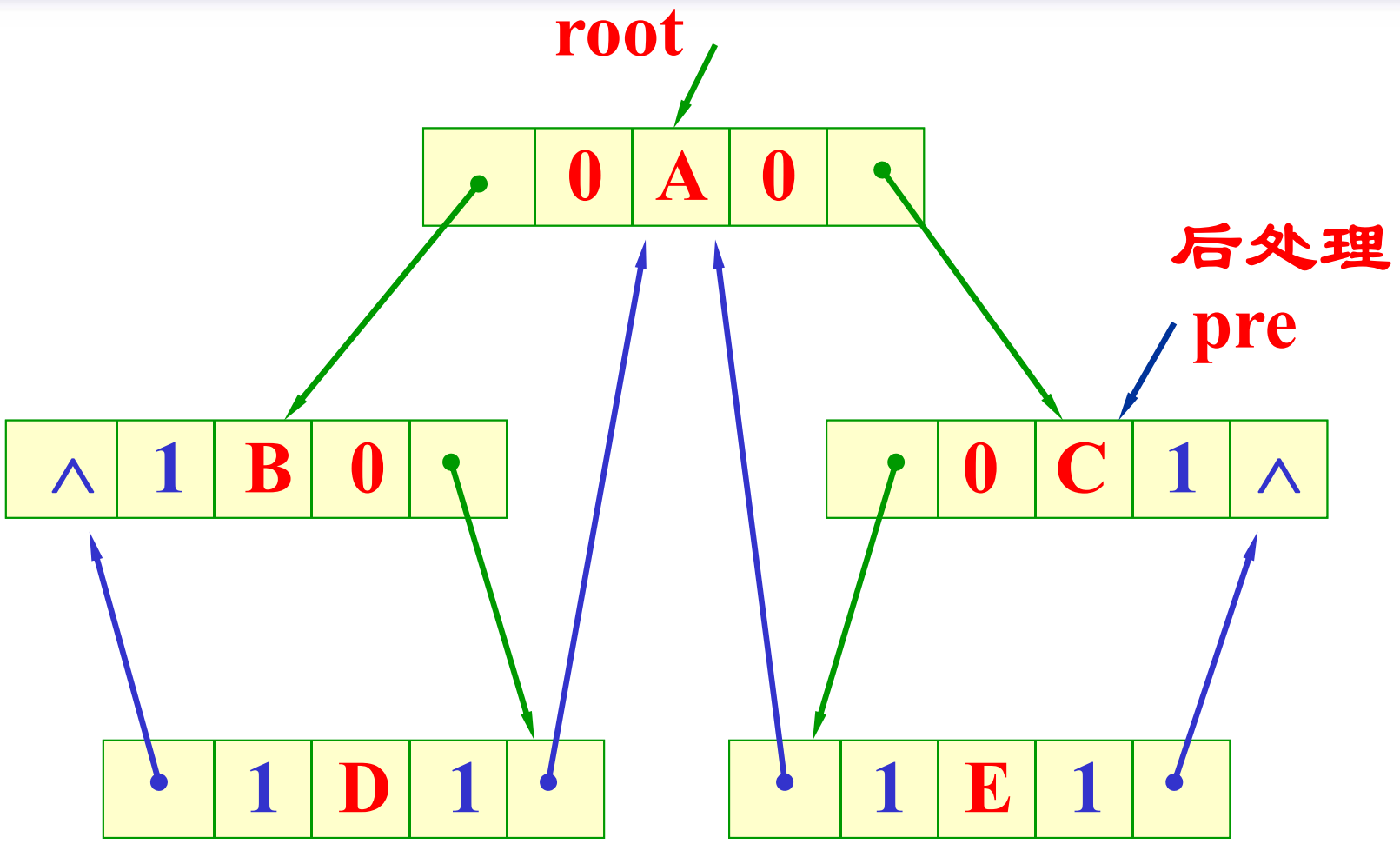

pre == NULL
root
current
0 A 0
∧ 1 B 0
0 C 0 ∧
∧ 0 D 0 ∧
∧ 0 E 0 ∧


root
0 A 0
pre
∧ 1 B 0
0 C 0 ∧
current
1 D 0 ∧
∧ 0 E 0 ∧

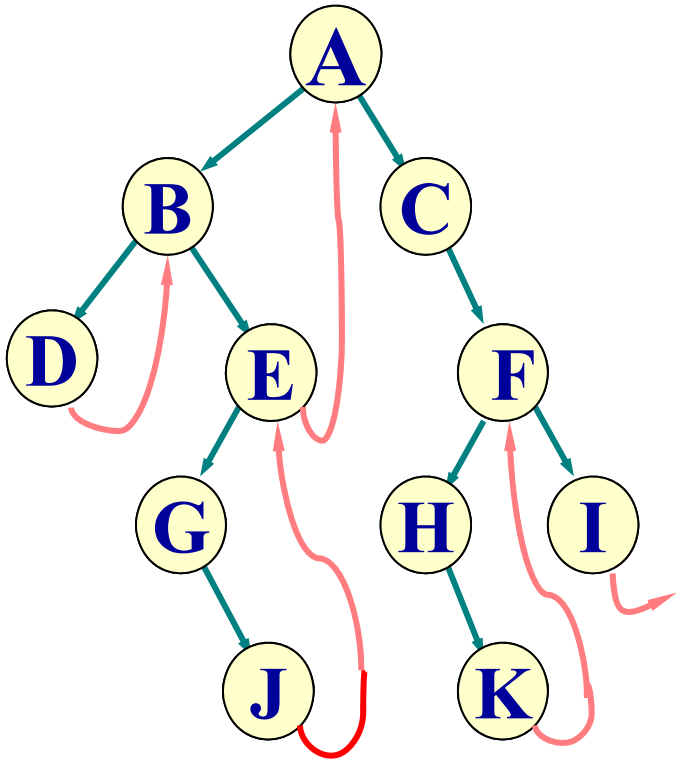

current
root
0 A 0
∧ 1 B 0
0 C 0 ∧
pre
1 D 1
∧ 0 E 0 ∧


pre
root
0 A 0
∧ 1 B 0
0 C 0 ∧
current
1 D 1
1 E 0 ∧


root
0 A 0
current
∧ 1 B 0
0 C 0 ∧
pre
1 D 1
1 E 1

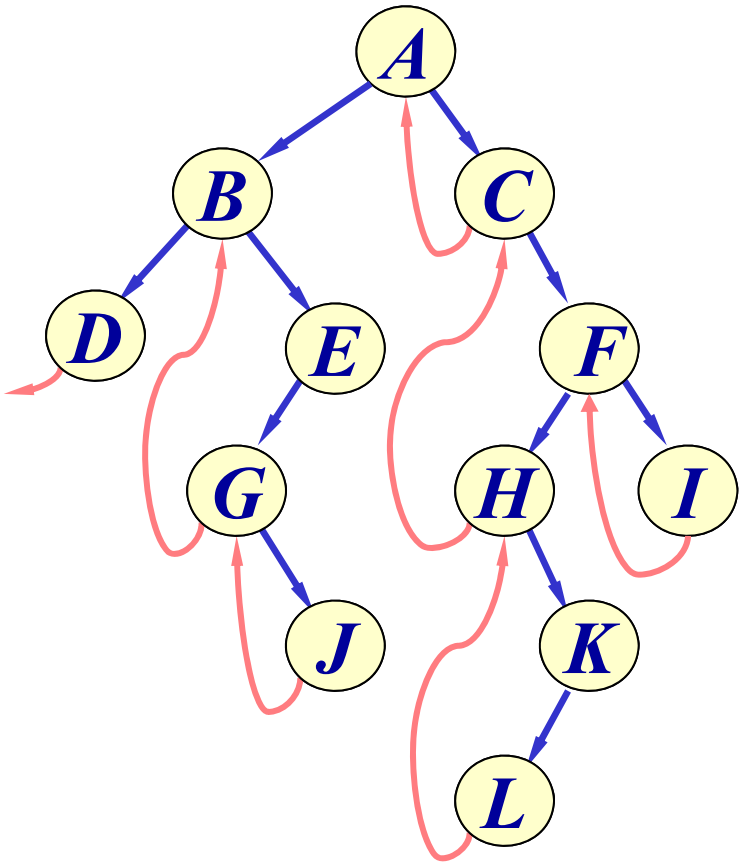

root
0 A 0
后处理
pre
∧ 1 B 0
0 C 1 ∧
1 D 1
1 E 1

寻找当前结点在中序下的后继




```
if ( current->rtag == 1 )
   if ( current->rightChild != T.root )
      后继为 current->rightChild
   else 无后继
else  // current->rtag != 1
   if ( current->rightChild != T.root )
      后继为当前结点右子树
    的中序下的第一个结点
   else 出错情况
```

寻找当前结点在中序下的前驱




```
if ( current->ltag == 1 )
   if ( current->leftChild !=T.root )
      前驱为current->leftChild
   else 无前驱
else  // current->ltag == 0
   if ( current->leftChild !=T.root )
      前驱为当前结点左子树的
      中序下的最后一个结点
   else 出错情况
```

在中序线索二叉树上实现前序遍历

- 若当前结点有左子女，则前序下的后继结点即为左子女结点；否则，如当前结点有右子女，则前序后继即为右子女结点。
- 对于叶结点，则沿着中序后继线索走到一个有右子女结点的结点，这个右子女结点就是当前结点的前序后继结点。

```
template <class Type> void ThreadTree <Type> ::
PreOrder ( void ( *visit ) ( ThreadNode <Type> *p ) ) {
    ThreadNode <Type> *p = root;
    while ( p != NULL ) {
        visit ( p );
        if ( p->ltag == 0 ) p = p->leftChild;
        else if ( p->rtag == 0 ) p = p->rightChild;
            else {
                while ( p != NULL && p->rtag == 1 )
                    p = p->rightChild;
                if ( p != NULL ) p = p->rightChild;
            }
    }
}
```

在中序线索二叉树上实现后序遍历

- 从根出发，沿着左子女链一直找下去，找到左子女不再是左子女指针的结点，再找到该结点的右子女，在以此结点为根的子树上再重复上述过程，直到叶结点为止。
- 从此结点开始后序遍历父结点的右子女，在遍历过程中，每次都先找到当前结点的父结点，如果当前结点是父结点的右子女，或者虽然当前结点是父结点的左子女，但这个父结点没有右子女，则后序下的后继即为该父结点；否则，在当前结点的右子树（如果存在）上重复执行上面的操作。

```
template <class Type> void ThreadTree <Type> ::
PostOrder ( void ( * visit ) ( ThreadNode <Type> *p ) ) {
    ThreadNode <Type> *t = root;
    while ( t->ltag ==0 || t->rtag == 0 )
        if ( t->ltag == 0 ) t = t->leftChild;
        else if ( t->rtag == 0 ) t = t->rightChild;
    visit (t);
    ThreadNode <Type> *p;
    while ( ( p = Parent (t) != NULL ) {
        if ( p->rightChild == t || p->rtag == 1 ) t = p;
        else {
                t = p->rightChild;
                while ( t->ltag == 0 || t->rtag ==0 )
                    if ( t->ltag == 0) t = t->leftChild;
                    else if ( t->rtag == 0 ) t = t->rightChild;  }
        visit (t);    }
}
```

在中序线索二叉树中求父结点

- 从当前结点走到树上层的一个**中序前驱**（不一定是直接前驱），然后向下找**父结点**。
- 从当前结点走到树上层的一个**中序后继**（不一定是直接后继），然后向左下找**父结点**。

```
template <class Type> ThredNode <Type> * ThreadTree <Type> ::
Parent ( ThreadNode <Type> *t ) {
    ThreadNode <Type> *p;
    if ( t == NULL ) return NULL;
    for ( p = t; p->ltag == 0; p = p->leftChild );
    if ( p->leftChild != NULL )
        for ( p = p->leftChild; p->leftChild != t &&
                                p->rightChild != t; p = p->rightChild );
    else {
        for ( p = t; p->rtag == 0; p = p->rightChild );
        for ( p = p->rightChild; p->rightChild != t &&
                                 p->leftChild != t; p = p->leftChild );
     }
     return p;
}
```

中序线索化二叉树的插入操作

- 在一棵中序线索化二叉树中插入一个新结点时，必须修改插入位置的前驱、后继线索。
  - 保留整个原有中序序列；
  - 将所插入新结点正确排入中序序列中。

- 当插入一个新结点r作为结点s的右子女时：
  - 如果s的rightChild是线索，则结点r直接成为s的右子女。
    - 原来s的后继线索送到r的rightChild域；
    - r的leftChild域存放指向s的前驱线索。
  - 如果s的rightChild是右子女指针，则以该右子女为根的子树成为r的右子树。
    - 该子树中中序下的第一个结点的前驱线索指向r；
    - r的前驱线索指向s。

在中序线索化二叉树中插入结点

```
template <class Type> void ThreadTree <Type> ::
InsertRight ( ThreadNode <Type> *s,
                        ThreadNode <Type> *r ) {
//将 r 当做 s 的右子女插入
    r->rightChild = s->rightChild;
    // s 的右子女指针或后继线索传给 r
    r->rtag = s->rtag;
    //标志同一传送
    r->leftChild = s;  r ->ltag = 1;
    // r 的 leftchild 成为指向 s 的前驱线索
    s->rightChild = r;  s->rtag = 0;
    // r 成为 s 的右子女
```

```
if ( r->rtag = = 0 ) {
// s 原来有子女
        ThreadNode <Type> *temp;
        temp = First ( r->rightChild );
        //在 s 原来的右子树中找中序下第一个结点
        temp->leftChild = r;
        //建立指向 r 的前驱线索
    }
}
```

中序线索化二叉树的删除操作 (I)

- 设结点r是结点s的右子女，分4种情况删除。
  - 如果结点r是叶结点，无左子树和右子树。
    - 只需修改一个指针就可将结点r从二叉树中摘下；
    - 保证前驱和后继线索有效。

```
s->rightChild = r->rightChild;
//重新链接后继线索
s->rtag = 1;
```

中序线索化二叉树的删除操作 (II)

- 设结点r是结点s的右子女，分4种情况删除。
  - 如果结点r还有右子树，在摘下结点r时必须重新链接r的右子树。
    - 首先在r的右子树中查找在中序下的第一个结点，设由指针fr指向它，然后执行：

```
ThreadNode <Type> *fr = First ( r->rightChild );
fr->leftChild = r->leftChild;
//重新链接 r 的中序后继结点的前驱线索
s->rightChild = r->rightChild;
//重新链接子女指针，将结点 r 从链中摘下
```

中序线索化二叉树的删除操作 (III)

- 设结点r是结点s的右子女，分4种情况删除。
  - 被删结点有左子树，无右子树。
    - 首先在r的左子树中查找中序下的最后一个结点，设由la指示它，然后执行：

```
ThreadNode <Type> *la = Last ( r->leftChild );
la->rightChild = r->rightChild;
//重新链接 r 的中序前驱结点的后继线索
s->rightChild = r->leftChild;
//重新链接子女指针， 将结点 r 从链中摘下
```

中序线索化二叉树的删除操作 (IV)

- 设结点r是结点s的右子女，分4种情况删除。
  - 被删结点既有左子树又有右子树。
    - 按照中序结点顺序的的要求：
      - 在摘下r时，把r的左子女上升到s的右子女位置；
      - r的右子树链接到左子树中序下的最后一个结点下面。

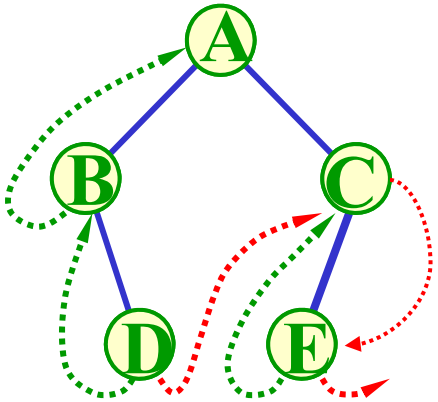
- **首先在r的左子树中查找中序下的最后一个结点，设由la指示它，然后执行：**

```
ThreadNode <Type> *la = Last ( r->leftChild );
la->rightChild = r->rightChild;
//将 r 的右子树链接到 temp 的右链上
la->rtag = r->rtag;  s->rightChild = r->leftChild;
//重新链接子女指针，将结点 r 从链中摘下
```

- **然后，在r的右子树中寻找中序下的第一个结点，用指针fr指示：**

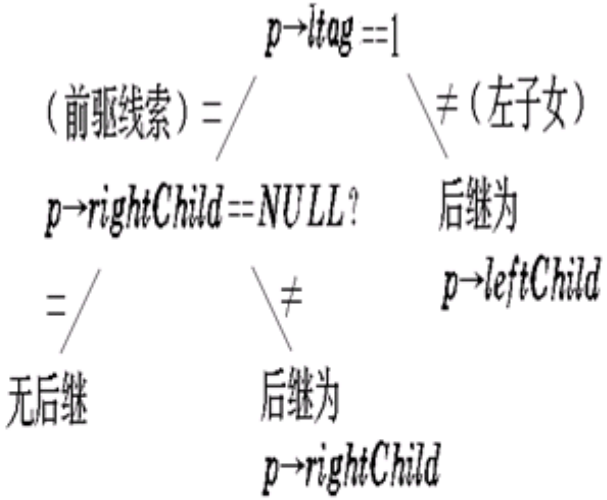
```
ThreadNode <Type> *fr = First ( r->rightChild );
fr->leftChild = la;
//重新链接 r 的中序后继结点的前驱线索
```

前序线索二叉树

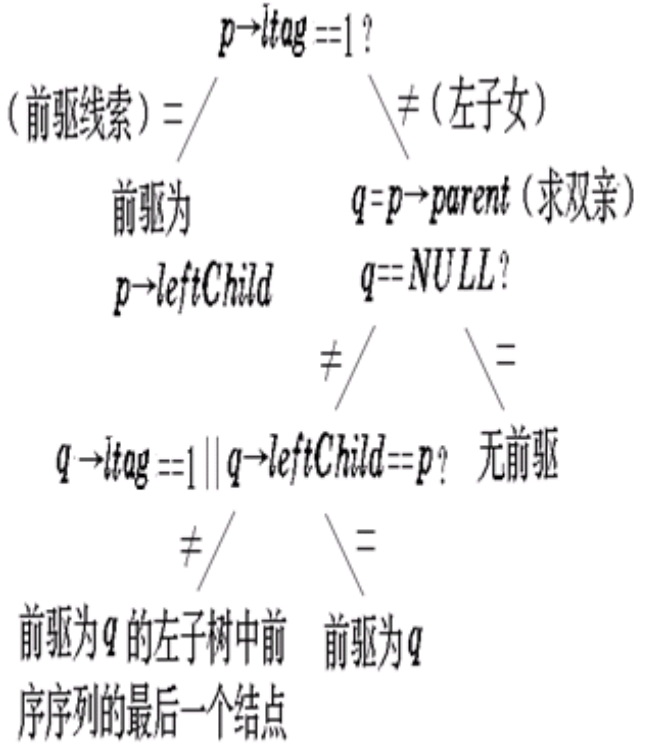


前序序列

A B D C E

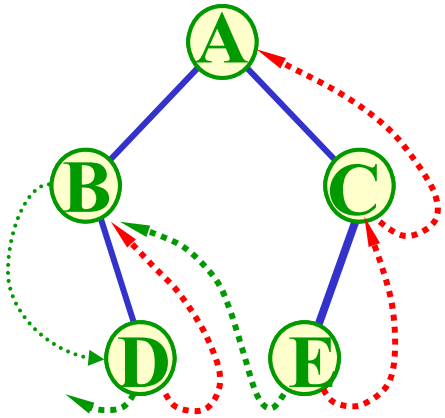


(a) 求结点 p 的后继



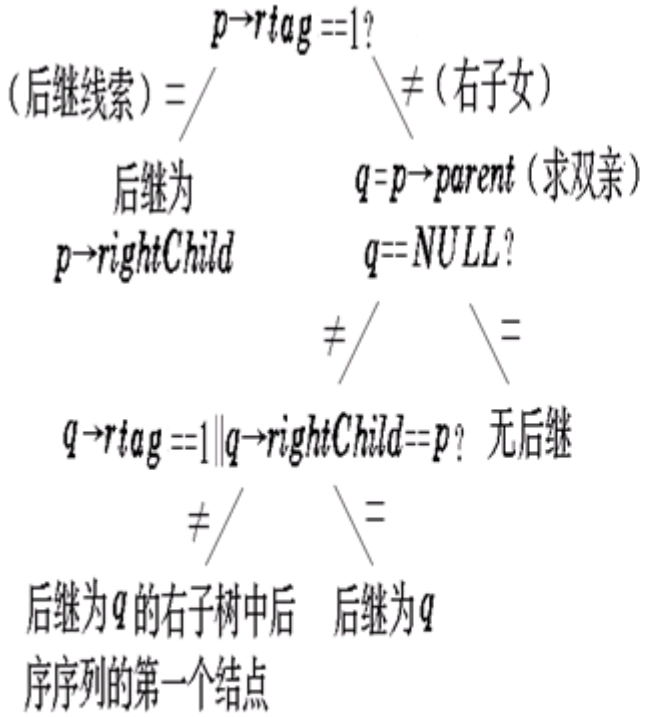
(b) 求结点 p 的前驱

后序线索二叉树

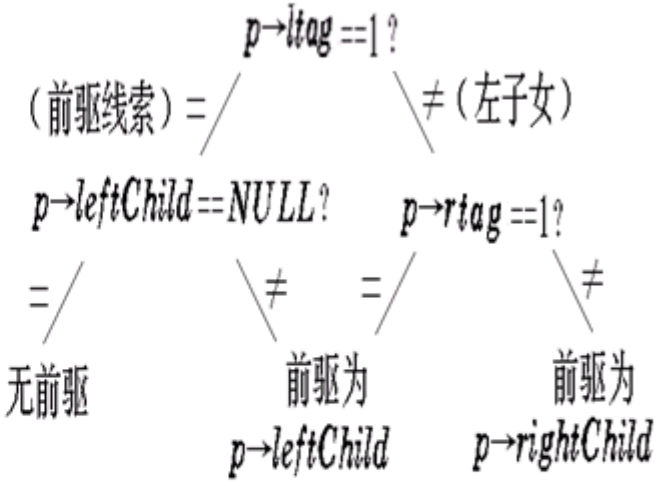


后序序列

D B E C A



(a) 求结点 p 的后继

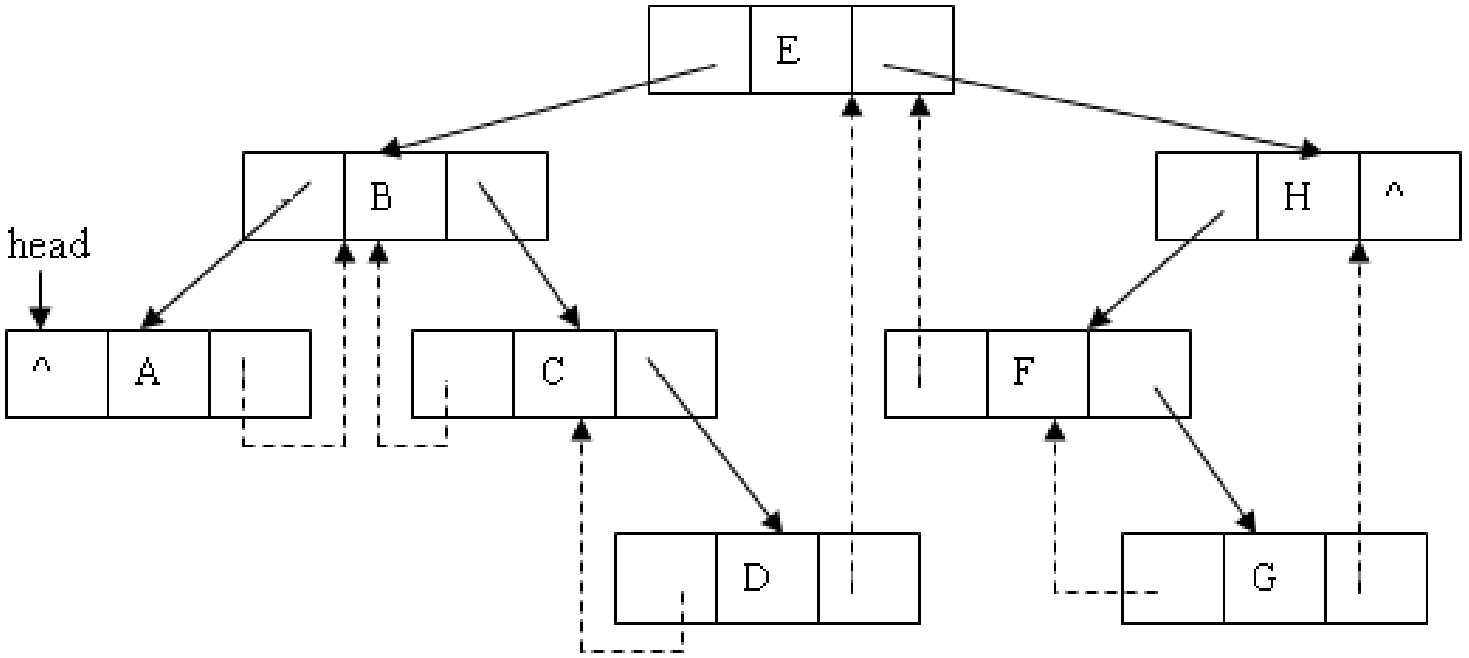


(b) 求结点 p 的前驱

线索排序

- 首先，用给定的**n**个结点的序列建造一棵线索树，使得树中每个结点的值大于该结点的非空左子树中所有结点的值，而都小于该结点的非空右子树中所有结点的值，称这样的树为线索排序树。
- 然后，按中序遍历线索排序树，这时得到**n**个结点的新序列，它是由小到大排好序的。这种利用线索树进行排序的方法称之为线索排序。

线索排序树



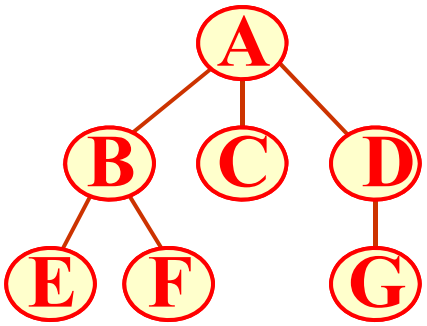
建立线索树

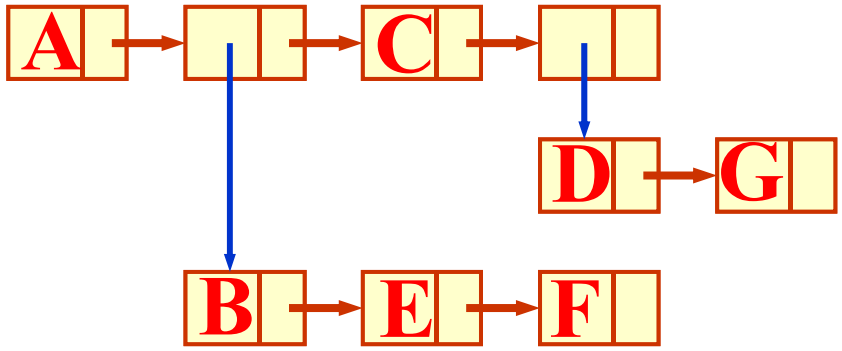
```
NODE *thread_sort_tree(char a[ ], int n) {  //n>=0
    NODE *root, *head, *p, *r;  int i;  if (n==0)  return NULL;
    root=new NODE( );  root->data=a[0];  root->leftchild==NULL;
    root->ltag=0;  root->rtag=0;  head=root;
    for (i=1; i<n; i++) {
        r=new NODE( );  r->data=a[i];  p=root;
        while (1) {
            if (r->data<=p->data)
                if (p->ltag==0 && p->leftchild!=NULL)  p=p->leftchild;
                else break;
            else if (p->rtag==0 && p->rightchild!=NULL)  p=p->rightchild;
            else break;
        }
        if (r->data<p->data) {
            r->leftchild=p->leftchild;  r->ltag=p->ltag;  r->rightchild=p;
            r->rtag=1;  p->leftchild=r;  p->ltag=0;
            if (r->leftchild==NULL)  head=r;
        }
        else if (r->data>p->data) {
            r->rightchild=p->rightchild;  r->rtag=p->rtag;  r->leftchild=p;
            r->ltag=1;  p->rightchild=r;  p->rtag=0;
        }
    }
    return head;
}
```

5.6 树与森林

树的存储表示

- 广义表表示



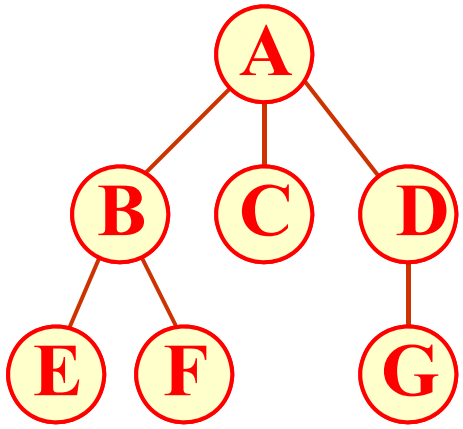


树的广义表表示（结点的utype域没有画出）

叶结点 && 根结点 && 分支结点

原子结点 && 表头结点 && 子表结点

❒ 双亲表示



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

利用一组连续的存储单元来存放树中的结点。每个结点有两个域，一个是data域，用来存放数据元素，一个是parent域，用来存放指示其双亲结点位置的指针。

❒ 左子女-右兄弟表示法

一棵树中每个结点具有的子树棵数可能不尽相同，因此如果用链接指针指示子树根结点的地址，每个结点所需的链接指针各不相同。采用变长结点的方式，为各个结点设置不同数目的指针域，将给存储管理带来很多麻烦。

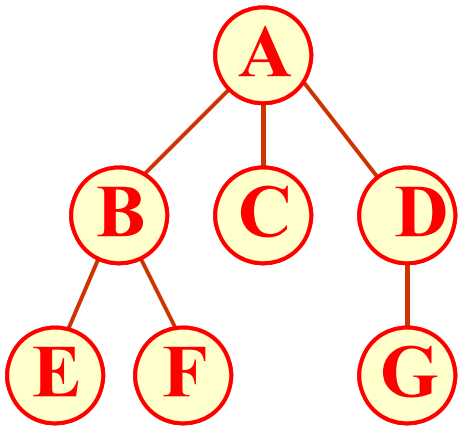
第一种解决方案　　等数量的链域

根据树的度d为每个结点设置d个指针域。

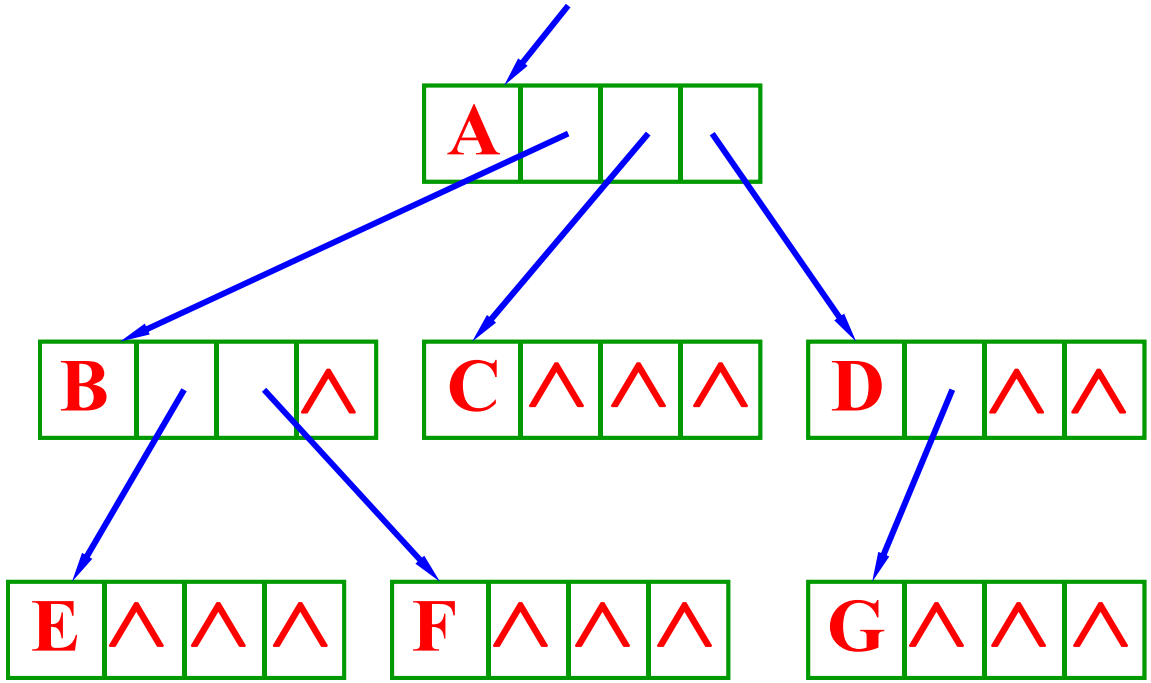
data	$child_1$	$child_2$	$child_3$	·····	$child_d$

采用多重链表作为固定长结点的链表形式。由于树中有许多结点的度小于d，造成许多空指针域，空间浪费较大。d越大，空间浪费越多。

优点在于：管理容易。



空链域$2n+1$个



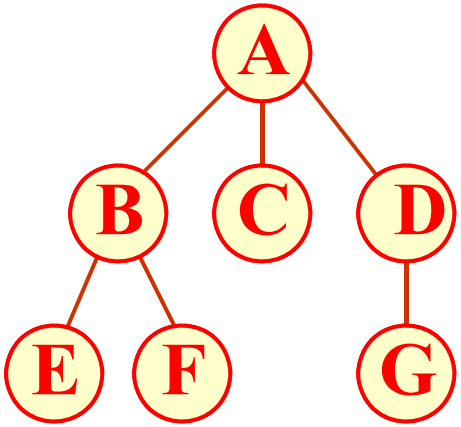
第二种解决方案

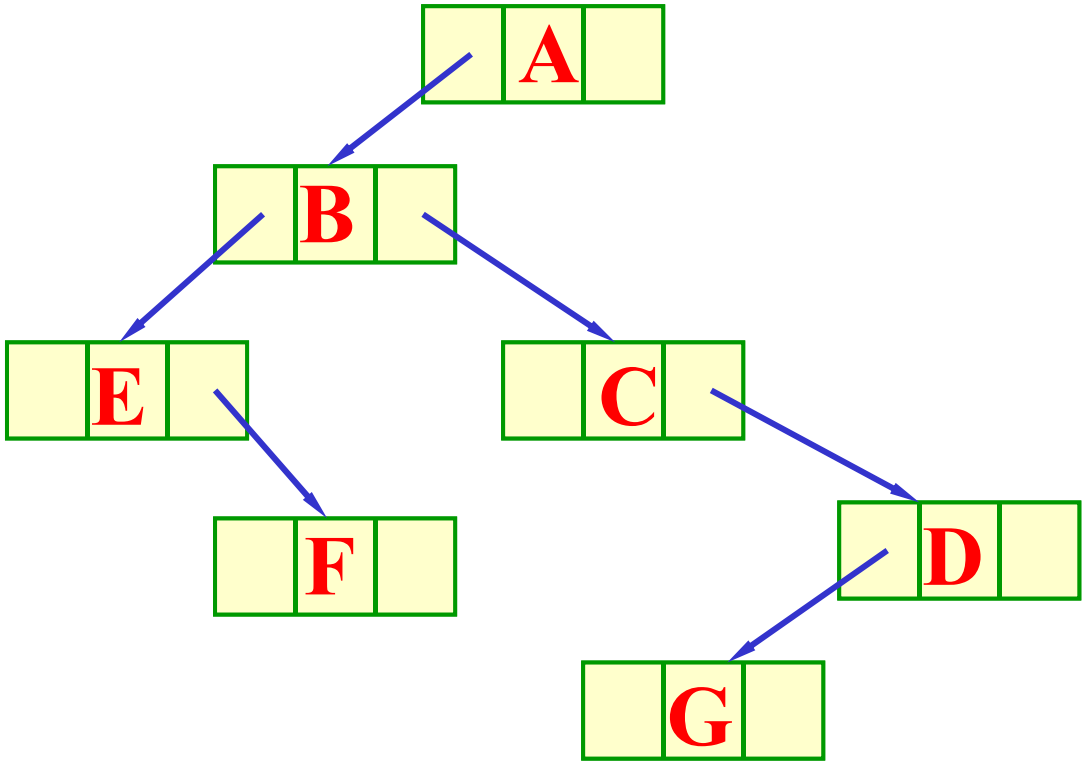
左子女--右兄弟表示法，为一种二叉树表示法。由于$d=2$，则为最节省空间的树的存储表示，每个结点由3个域组成。

data	firstChild	nextSibling

若要检测某一结点的所有子女，只需先通过结点的*firstChild*指针，找到其第一个子女，再通过该子女结点的*nextSibling*指针找到其下一兄弟，依此类推，找到其它兄弟，直到*nextSibling*指针为空，检测结束。

树的左子女-右兄弟表示





用左子女-右兄弟表示实现的树的类定义

```
template <class Type> class Tree;
template <class Type> class TreeNode {
friend class <Type> Tree;
private:
    Type data;
    TreeNode <Type> *firstChild, *nextSibling;
public:
    TreeNode ( Type value=0,
        TreeNode <Type> *fc=NULL,
        TreeNode <Type> *ns=NULL ) :
            data(value), firstChild(fc), nextSibling(ns) { }
};
```

```
template <class Type> class Tree {
private:
   TreeNode <Type> *root, *current;
   //根指针及当前指针
   bool Find ( TreeNode <Type> *p, Type value );
   void RemovesubTree ( TreeNode <Type> *p );
   //删除以 p 为根的子树
   int FindParent ( TreeNode <Type> *t,
                    TreeNode <Type> *p );
   //在树 t 中搜索结点 p 的双亲
```

```
public:
    Tree ( ) { root = current = NULL; }
    bool Root ( );
    bool IsEmpty ( ) { return root == NULL; }
    bool FirstChild ( );
    bool NextSibling ( );
    bool Parent ( );
    bool Find ( Type target );
    //树的其它公共操作
    ……
};
```

用左子女-右兄弟表示实现的树的部分操作

```
template <class Type>
bool Tree <Type> :: Root ( ) {
//让树的根结点成为树的当前结点
    if ( root == NULL )
        { current = NULL;  return false; }
    else { current = root;  return true; }
}
```

```
template <class Type>
bool Tree <Type> :: Parent ( ) {
//在树中找当前结点的双亲，成为当前结点
    TreeNode <Type> *p = current;
    if ( current == NULL || current == root )
      { current = NULL;  return false; }
    //空树或根为当前结点，返回 false
    return FindParent ( root, p );  //从 t 找 p 双亲
}
```

```
template <class Type> bool Tree <Type> ::
FindParent ( TreeNode <Type> *t,
                        TreeNode <Type> *p ) {
//在根为 t 的树中找 p 的双亲，成为当前结点
   TreeNode <Type> *q = t->firstChild;  bool succ;
   while ( q != NULL && q != p ) {
   //循根的长子的兄弟链，递归在子树中搜索
      if ( ( int succ = FindParent (q, p) ) == true )
         return succ;
      q = q->nextSibling;    }
   if ( q != NULL && q == p )
     { current = t;  return true; }
   else return false;  //未找到双亲，当前结点不变
}
```

```
template <class Type>
bool Tree <Type> :: FirstChild ( ) {
//在树中找当前结点的长子，成为当前结点
   if ( current != NULL &&
            current->firstChild != NULL )
   { current = current->firstChild;  return true; }
   current = NULL;  return false;
}
```

```
template <class Type>
bool Tree <Type> :: NextSibling ( ) {
//在树中找当前结点的兄弟，成为当前结点
   if ( current != NULL &&
            current->nextSibling != NULL )
   { current = current->nextSibling;  return true; }
   current = NULL;  return false;
}

template <class Type>
bool Tree <Type> :: Find ( Type value ) {
   if ( IsEmpty ( ) ) return 0;
   return Find ( root, value );
}
```

```
template <class Type> bool Tree <Type> ::
Find ( TreeNode <Type> *p, Type value ) {
//在根为 p 的树中找值为 target 的结点
//找到后该结点成为当前结点，否则当前结点不变
//函数返回成功标志：＝true，成功；＝false，失败
    bool result = false;
    if ( p->data == value )
      { result = true;  current = p; }  //搜索成功
    else {  //搜索失败
       TreeNode <Type> *q = p->firstChild;
       while ( q != NULL && ! ( result = Find ( q, value ) ) )
              q = q->nextSibling;    }
    return result;
}
```

子女结点的插入操作

如果在某一个指定的结点current下插入一个子女时，需要先判断该结点是否有子女，即检测结点的firstChild域是否为NULL。

如果空，则表明无子女，直接将firstChild指针指向该子女结点即可。否则，需要按照该结点的第一个子女结点的nextSibling链走到链尾，把新子女作为最后一个子女链入。

```
template <class Type> void Tree <Type> ::
InsertChild ( Type  value ) {
//在当前结点下插入一个包含有值 value 的新结点
//若当前结点原来没有子女
//则新结点成为当前结点的第一个子女
//否则成为当前结点的最右子女
  TreeNode <Type> *newNode =
                        new TreeNode <Type> (value);
  //创建新结点
  if ( current->firstChild = = NULL )
     current->firstChild = newNode;
  //当前结点无子女时，新结点成为其第一个子女
```

```
    else { //当前结点有子女时
      TreeNode <Type> *p = current->firstChild;
      //找当前结点第一个子女
      while ( p->nextSibling != NULL )
      //扫描兄弟链，找最后子女
          p = p->nextSibling;
      p->nextSibling = newNode;
      //新结点成为其最后子女
}
```

子女结点的删除操作

递归实现过程

删除某一个结点*current*的子女时先要搜索到该子女，将其从在左子女--右兄弟链中摘下，再一棵子树一棵子树地删除它的所有子树，最后再把该子女结点释放掉。

```
template <class Type> int Tree <Type> ::
RemoveChild ( int  i ) {
//删除树中当前结点的第 i 个子女及其全部子孙结点
//若该结点的第 i 个子女结点不存在，则函数返回0
//若删除成功，则函数返回1
   TreeNode <Type> *s;
   if ( i == 1 ) {  //要删的是第一个子女
      s = current->firstChild;  //要删的结点是 *s
      if ( s != NULL )
          current->firstChild = s->nextSibling;
      //将 s 从链中摘下
      else return 0;  //无子女时退出
   }
```

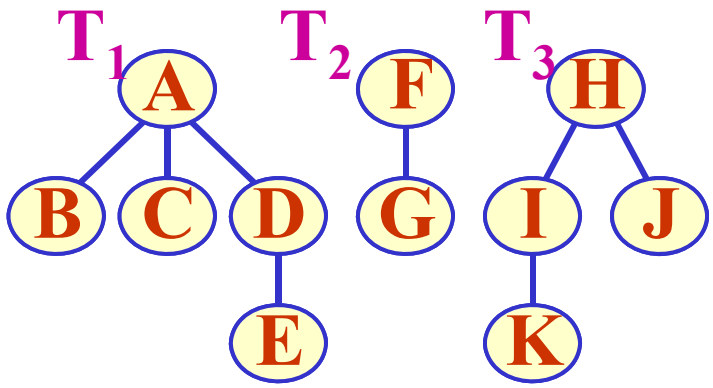
```
    else { //要删的不是第一个子女
        TreeNode <Type> *q = current->firstChild;
        int k = 1;
        while ( q != NULL && k < i-1 )
            { q = q->nextSibling;  k++; }
        //寻找第 i-1 个子女结点
        if ( q != NULL ) { //找到第 i-1 个结点
            s = q->nextSibling; //要删的结点是 *s
            if ( s != NULL )
                q->nextSibling = s->nextSibling;
            //将s从链中摘下
            else return 0; //无第 i 个子女时退出 }
        else return 0; //链中结点个数少于 i 时退出 }
     RemovesubTree (s); return 1; //删除以 s 为根的子树
}
```

```
template <class Type> void Tree <Type> ::
RemovesubTree ( TreeNode <Type> *p ) {
//私有函数：删除以 p 为根结点的子树
  TreeNode <Type> *q = p->firstChild, *next;
  //找 p 的第一个子女 q
  while ( q != NULL ) {
          next = q->nextSibling;
          // next 记下下一个子女
          RemovesubTree (q);  q = next;
          //删除以 q 为根的子树
  }
  delete p;
}
```

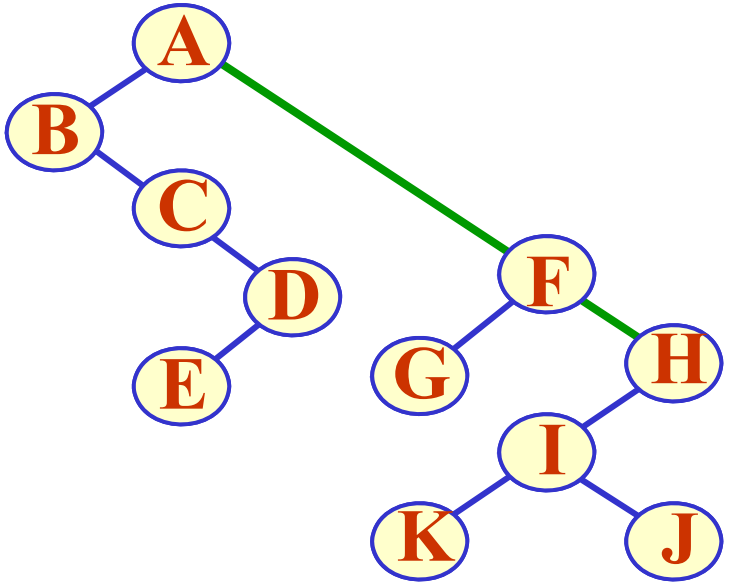
```
template <class Type> void Tree <Type> ::
RemovesubTree ( ) {
//公共函数：删除以当前结点 current 为根结点的子树
//若当前结点是根结点，则整棵树都删去
  if ( current ! = NULL ) {
      if ( current == root ) root = NULL ;
      RemovesubTree (current);  current = NULL;
  }
}
```

森林与二叉树的转换

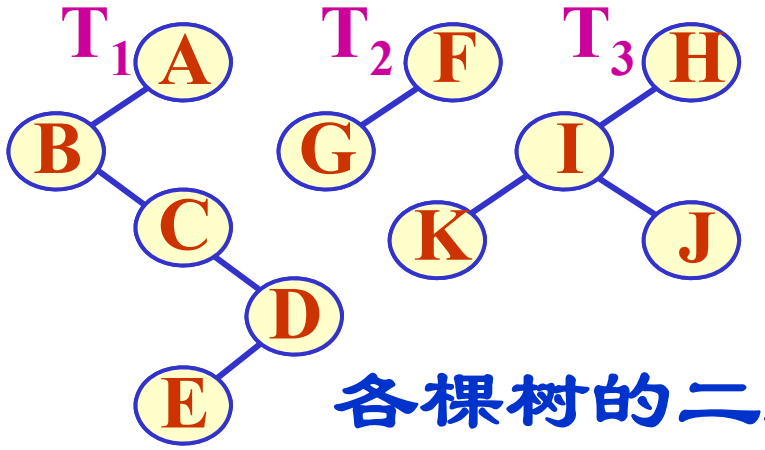
森林与二叉树的对应关系



3 棵树的森林



森林的二叉树表示



各棵树的二叉树表示

(1) 森林转化成二叉树的规则

① 若 F 为空，即 $n = 0$，则

对应的二叉树 B 为空二叉树。

② 若 F 不空，则

对应二叉树 B 的根 root (B) 是 F 中第一棵树 T_1 的根 root (T_1)；

其左子树为 B (T_{11}, T_{12}, …, T_{1m})，其中，T_{11}, T_{12}, …, T_{1m} 是 root (T_1) 的子树；

其右子树为 B (T_2, T_3, …, T_n)，其中，T_2, T_3, …, T_n 是除 T_1 外其它树构成的森林。

(2) 二叉树转换为森林的规则

① 如果 B 为空，则对应的森林 F 也为空。

② 如果 B 非空，则

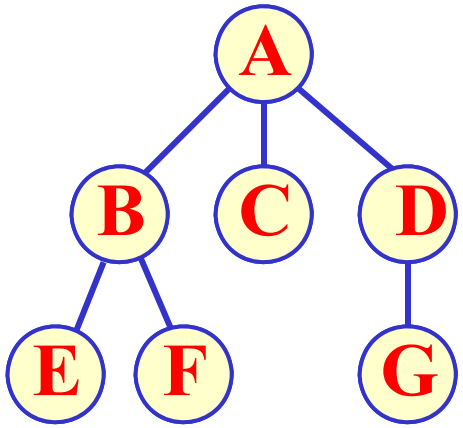
F 中第一棵树 T_1 的根为 root；

T_1 的根的子树森林 $\{T_{11}, T_{12}, \cdots, T_{1m}\}$ 是由 root 的左子树 LB 转换而来，F 中除了 T_1 之外其余的树组成的森林 $\{T_2, T_3, \cdots, T_n\}$ 是由 root 的右子树 RB 转换而成的森林。

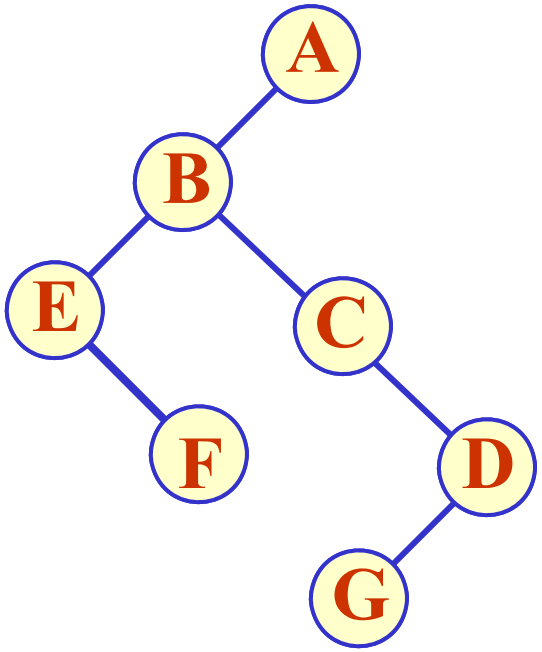
5.7 树与森林的遍历及其应用

树的遍历

- 深度优先遍历
  - 先根次序遍历
  - 后根次序遍历
- 广度优先遍历

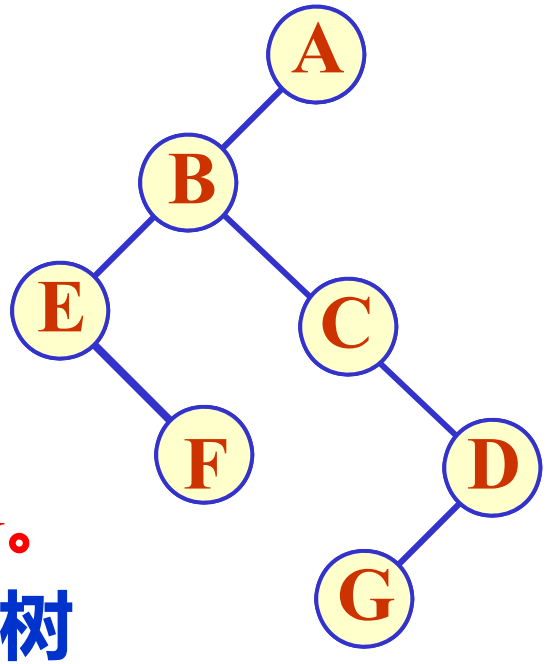


树的二叉树表示



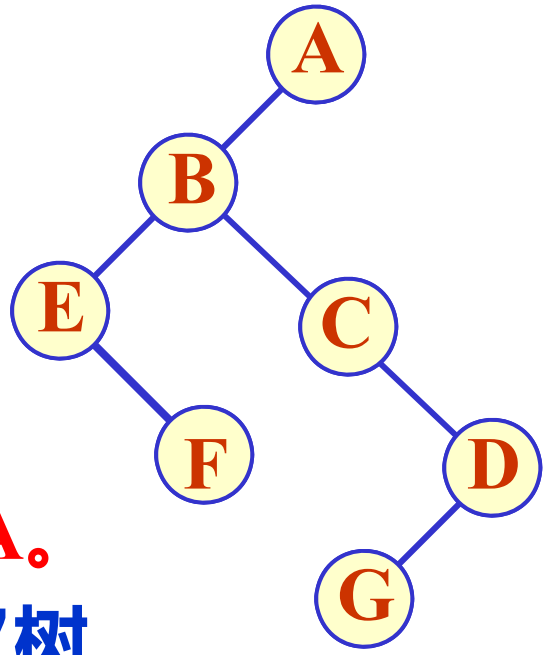
树的先根次序遍历

- 当树非空时
  - 访问根结点；
  - 依次先根遍历根的各棵子树。
- 树先根遍历 ABEFCDG。
- 对应二叉树前序遍历 ABEFCDG。
- 树的先根遍历结果与其对应二叉树表示的前序遍历结果相同。
- 树的先根遍历可以借助对应二叉树的前序遍历算法实现。



树的后根次序遍历

- 当树非空时
  - 依次后根遍历根的各棵子树；
  - 访问根结点。
- 树后根遍历 EFBCGDA。
- 对应二叉树中序遍历 EFBCGDA。
- 树的后根遍历结果与其对应二叉树表示的中序遍历结果相同。
- 树的后根遍历可以借助对应二叉树的中序遍历算法实现。



(1) 树的先根次序遍历的递归算法

```
template <class Type> void Tree <Type> ::
PreOrder ( ostream &out, TreeNode <Type> *p ) {
//以当前指针 p 为根，先根次序遍历
   if ( p != NULL ) {
      out << p->data;  //访问根结点
      for ( p = p->firstChild; p != NULL;
                        p = p->nextSibling )
           PreOrder ( out, p );  //递归先根遍历子树
   }
}
```

(2) 树的后根次序遍历的递归算法

```
template <class Type> void Tree <Type> ::
PostOrder ( ostream &out, TreeNode <Type> *p ) {
//以当前指针 p 为根，后根次序遍历
    if ( p != NULL ) {
        for ( p = p->firstChild; p != NULL;
                        p = p->nextSibling )
            PostOrder ( out, p );  //递归先根遍历子树
        out << p->data;  //访问根结点
    }
}
```

(3) 按先根次序遍历树的迭代算法

```
template <class Type> void Tree <Type> ::
NorecPreOrder ( ) {
    Stack < TreeNode <Type> * > st (DefaultSize);
    TreeNode <Type> *p = current;  // p 保存当前结点
    do {
        while ( current != NULL ) {
        //当前指针不空，从根沿长子链向下
            visit ( );  st.Push (current);
            //访问该结点，进栈
            FirstChild ( );
            //当前指针指到第一棵子树根结点
        } //走到无左子女结点跳出循环
```

```
        while (current == NULL && !st.IsEmpty ( ) )
        { //无子女或无兄弟
            current = st.Pop ( );  NextSibling ( );
            //退栈，转向下一兄弟
        } //栈空，退出循环
    } while ( current != NULL );
    current = p;  //恢复最初的当前指针
}
```

(4) 按后根次序遍历树的迭代算法

```
template <class Type> void Tree <Type> ::
NorecPostOrder ( ) {
    Stack < TreeNode <Type> * > st (DefaultSize);
    TreeNode <Type> *p = current;  // p 保存当前结点
    do {
        while ( current != NULL ) {
        //当前指针不空，从根沿长子链向下
            st.Push (current );  //进栈
            FirstChild ( );  //找第一棵子树根结点
        }
```

```
        while ( current == NULL && !st.IsEmpty ( ) )
        { //向左无子女或者向右无兄弟
            current = st.Pop ( );  visit ( );
            //退栈并访问
            NextSibling ( );
            //向右到下一棵子树（兄弟） }
    } while ( current != NULL );
    current = p;  //恢复最初的当前指针
}
```

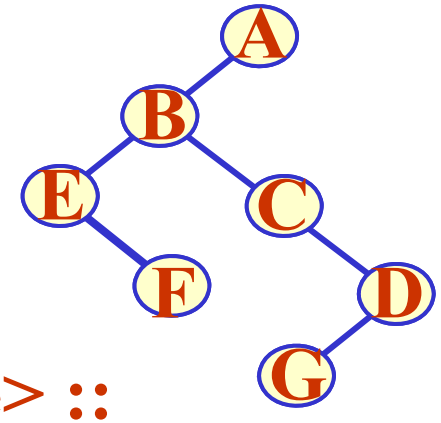
(5) 借助二叉树实现树的先根遍历算法

```
template <class Type> void Tree <Type> ::
PreOrder1 ( TreeNode <Type> *t,
                    void ( *visit ) ( TreeNode <Type> *p ) {
    if ( t == NULL )  return;
    visit ( t );
    PreOrder1 ( t->firstChild, visit );
    PreOrder1 ( t->nextSibling, visit );
}
```

(6) 借助二叉树实现树的后根遍历算法

```
template <class Type> void Tree <Type> ::
PostOrder1 ( TreeNode <Type> *t,
                    void ( *visit ) ( TreeNode <Type> *p ) {
    if ( t == NULL )  return;
    PostOrder1 ( t->firstChild, visit );
    visit ( t );
    PostOrder1 ( t->nextSibling, visit );
}
```

广度优先（层次次序）遍历



按广度优先次序遍历树的结果

ABCDEFG

```
template <class Type> void Tree <Type> ::
LevelOrder ( ostream &out, TreeNode <Type> *p ) {
//按广度优先次序分层遍历树
//树的根结点是当前指针 p
    Queue < TreeNode <Type> * > Qu(DefaultSize);
    TreeNode <Type> *p;
    if ( p != NULL ) {
        Qu.EnQueue (p);
```

```
        while ( Qu.IsEmpty ( ) ) {
            Qu.DeQueue ( );
            out << p->data;  //队列中取一个并访问之
            for ( p = p->firstChild; p!=NULL;
                                p = p->nextSibling )
                Qu.EnQueue (p);
        }

}
```

应用树遍历的事例

利用先根遍历计算树的结点总数

```
template <class Type> int Tree <Type> ::
Count_Node ( TreeNode <Type> *t ) const {
    if ( t == NULL ) return 0;
    int count = 1;
    count += Count_Node ( t->firstChild );
    count += Count_Node ( t->nextSibling );
    return count;
}
```

利用后根遍历计算树的深度

```
template <class Type> int Tree <Type> ::
Find_Depth ( BinTreeNode <Type> *t ) const {
    if ( t == NULL ) return 0;
    int fc_depth = Find_Depth ( t->firstChild ) + 1;
    int ns_depth = Find_Depth ( t->nextSibling );
    return ( fc_depth > ns_depth ) ?
                        fc_depth : ns_depth;
}
```

利用广义表构造树

```
template <class Type> TreeNode <Type> *
Tree <Type> :: Create_Tree ( T *&GL ) {
    if ( *GL == '\0' ) return NULL;
    if ( *( GL++ ) == ')' ) return NULL;
    if ( *GL == '(' ) GL++;
    TreeNode <Type> *t =
                    new TreeNode <Type> ( *( GL++ ) );
     t->firstChild = Create_Tree ( GL );
     t->nextSibling = Create_Tree ( GL );
     return;
}
```

树的输出——输出广义表

```
template <class Type> void Tree <Type> ::
Show_Tree ( TreeNode <Type> *t ) {
    if ( t == NULL ) return;
    cout << '(';
    cout << t->data;
    for ( TreeNode <Type> *p = t->firstChild;
                    p != NULL; p = p->nextSibling )
        Show_Tree ( p );
    cout << ')';
}
```

森林的遍历

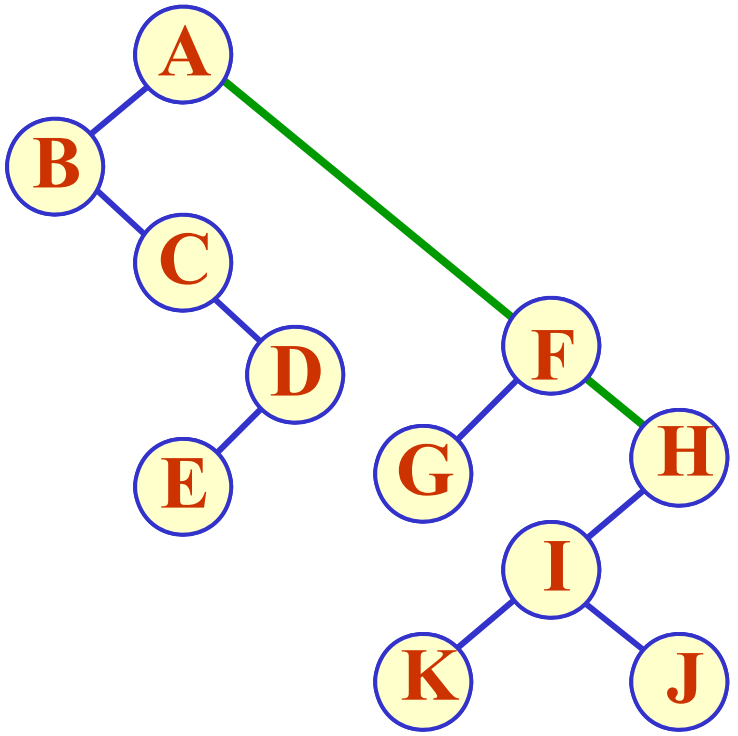
(1) 先根次序遍历的规则

❒ 若森林 F 为空，返回；否则

☞ 访问 F 的第一棵树的根结点；

☞ 先根次序遍历第一棵树的子树森林；

☞ 先根次序遍历其它树组成的森林。



森林的二叉树表示

ABCDEFGHIKJ

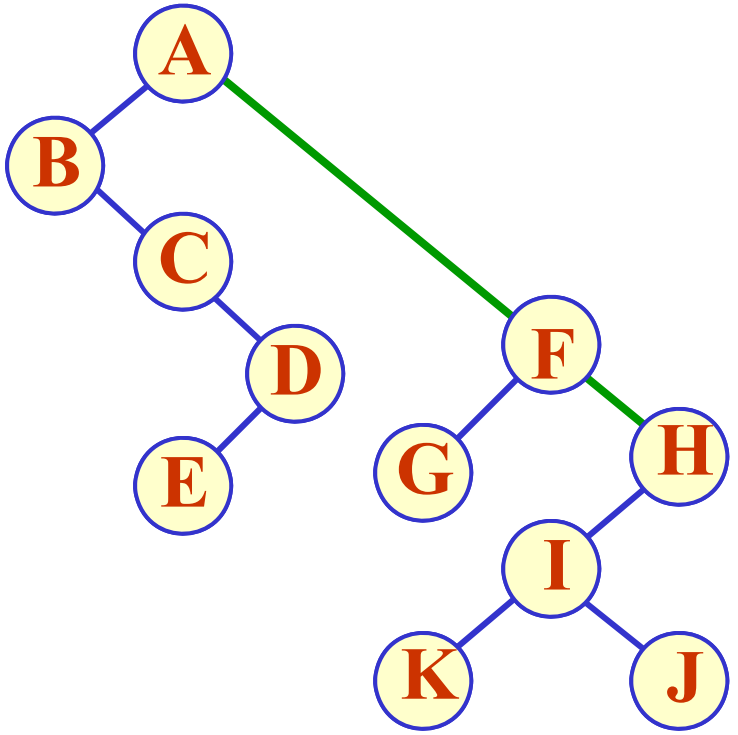
(2) 后根次序遍历的规则

❒ 若森林 F 为空，返回；否则

☞后根次序遍历第一棵树的子树森林；

☞访问 F 的第一棵树的根结点；

☞ 后根次序遍历其它树组成的森林。



森林的二叉树表示

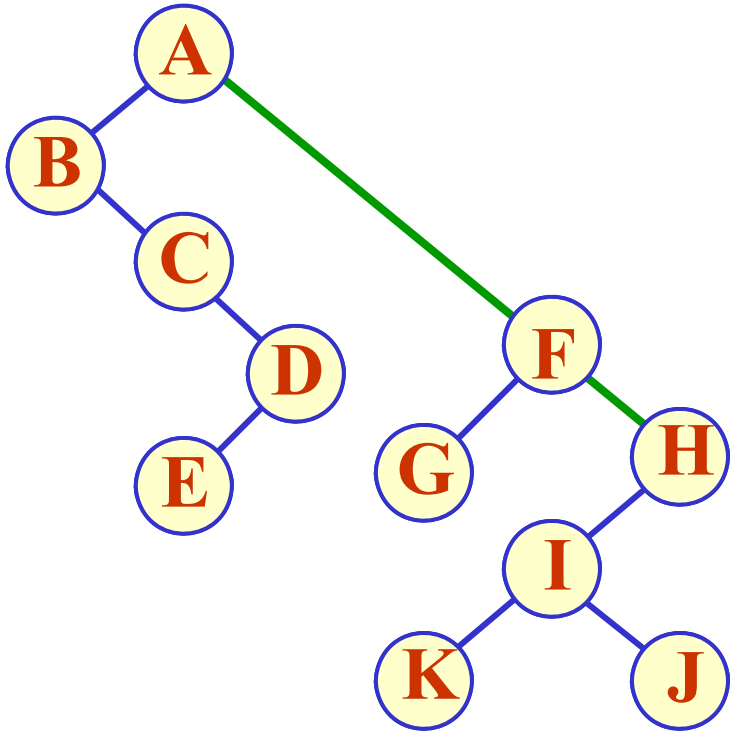
BCEDAGFKIJH

(3) 广度优先遍历（层次序遍历）

- 若森林 F 为空，返回；

否则

- 依次遍历各棵树的根结点；
- 依次遍历各棵树根结点的所有子女；
- 依次遍历这些子女结点的子女结点，……。



森林的二叉树表示

AFHBCDGIJEK

5.8 堆 (Heap)

优先级队列

每次出队列的是优先权最高的元素。

```
template <class Type> class MinPQ {
public:
   Virtual void Insert ( const Type & ) = 0;
   Virtual Type * Remove ( Type & ) = 0;
};
```

最小优先级队列类的定义

堆的定义

如果有一个关键码集合$K=\{k_0, k_1, k_2, \ldots, k_{n-1}\}$，把其中所有元素按完全二叉树的顺序（数组）存储方式存放在一个一维数组中，并且满足：

$$K_i \le K_{2i+1} \ \&\& \ K_i \le K_{2i+2}$$

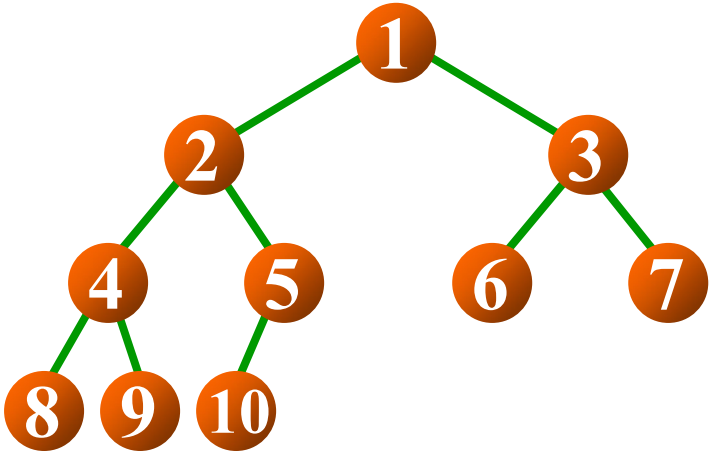
$$K_i \ge K_{2i+1} \ \&\& \ K_i \ge K_{2i+2}$$

$$(i=0, 1, \ldots, \lfloor (n-2)/2 \rfloor)$$

则称该集合为最小堆（或者最大堆）。

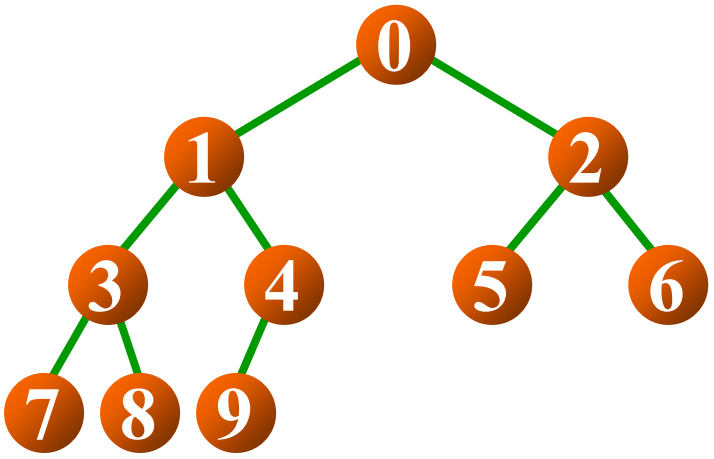
性质5 如将一棵有n个结点的完全二叉树自顶向下，同一层自左向右连续给结点编号1, 2, …, n-1，则有以下关系：

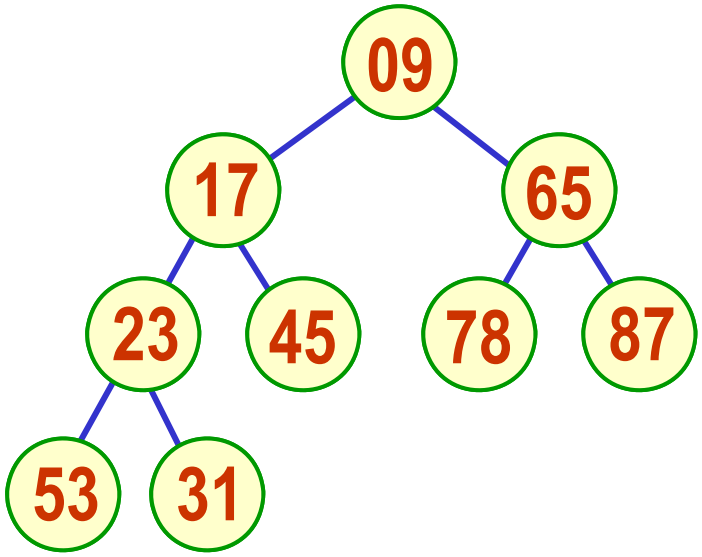
- 若$i = 1$，则 i 无双亲；
  若$i > 1$，则 i 的双亲为$\lfloor i/2 \rfloor$。
- 若$2*i \le n$，则 i 的左子女为$2*i$；
  若$2*i+1 \le n$，则 i 的右子女为$2*i+1$。
- 若 i 为奇数，且i != 1，
  则其左兄弟为i-1；
  若i 为偶数，且i != n，
  则其右兄弟为i+1。
- i 所在层次为 $\lfloor \log_2 i \rfloor + 1$。

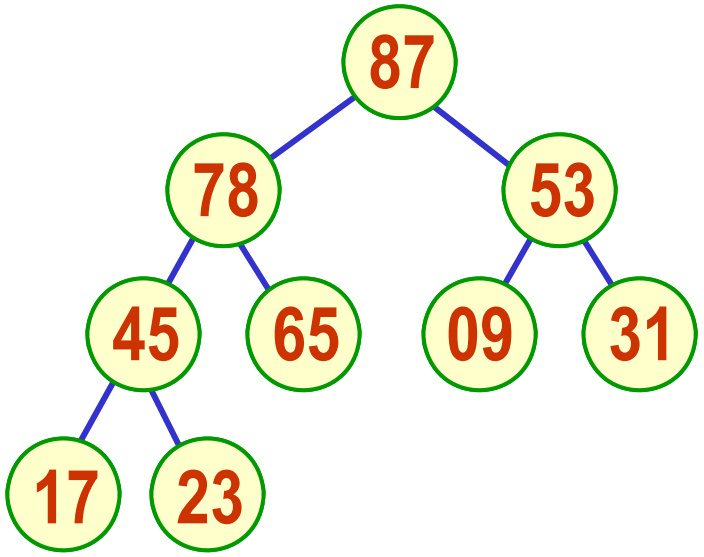


性质5 如将一棵有n个结点的完全二叉树自顶向下，同一层自左向右连续给结点编号0, 1, 2, …, n-1，则有以下关系：

- 若$i = 0$，则 i 无双亲；
  若$i > 0$，则 i 的双亲为$\lfloor (i-1)/2 \rfloor$。
- 若$2*i+1 < n$，则 i 的左子女为 $2*i+1$；
  若$2*i+2 < n$，则 i 的右子女为$2*i+2$。
- 若 i 为偶数，且i != 0，
  则其左兄弟为i-1；
  若i 为奇数，且i != n-1，
  则其右兄弟为i+1。
- i 所在层次为$\lfloor \log_2(i+1) \rfloor$。







最小堆 **最大堆**

**最小堆中任一结点的关键码均小于或等于其左、右子女关键码，位于堆顶（即完全二叉树的根结点位置）的结点的关键码为整个集合最小值。

**最大堆中任一结点的关键码均大于或等于其左、右子女关键码，位于堆顶的结点的关键码为整个集合最大值。

最小堆的类定义

```
#define DefaultSize  10;
template <class Type>
class MinHeap : public MinPQ <Type> {
private:
    Type *heap;  //存放最小堆元素的数组
    int currentSize;  //最小堆当前元素个数
    int maxHeapSize;  //最多允许元素个数
    void SiftDown ( int start, int m );
    //从 start 到 m 自顶向下进行调整成为最小堆
    void SiftUp ( int start );
    //从 start 到 0 自底向上进行调整成为最小堆
```

```
public:
    MinHeap ( int sz );  //构造函数：建立空堆
    MinHeap ( Type arr[ ], int n );  //构造函数
    MinHeap ( const MinHeap &R );
    ~MinHeap ( ) { delete [ ] heap; }
    bool Insert ( const Type &x );  //插入
    bool Remove ( Type &x );  //删除
    bool IsEmpty ( ) const  //判堆空否
      { return ( CurrentSize == 0 ) ? true : false; }
    bool IsFull ( ) const  //判堆满否
      { return ( CurrentSize == MaxHeapSize )
                                  ? true : false; }
    void MakeEmpty ( ) { CurrentSize = 0; }
};
```

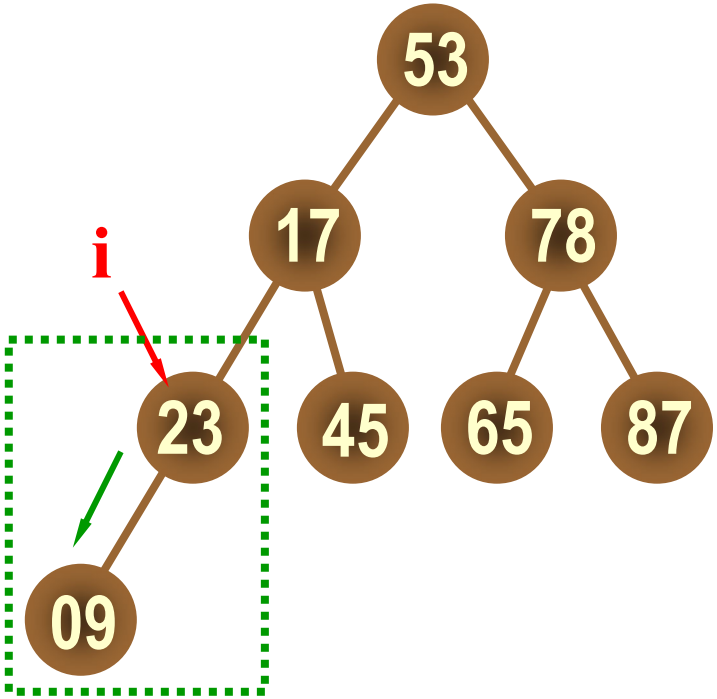
堆的建立

```
template <class Type> MinHeap <Type> ::
MinHeap ( int sz ) {
//根据给定大小 sz，建立堆对象
    maxHeapSize = ( DefaultSize < sz ) ? sz : DefaultSize;
    //确定堆的大小
    heap = new Type [maxHeapSize];
    if ( heap == NULL ) {
       cerr << "存储分配错！" << endl;  exit(1);
    }
    currentSize = 0;
}
```

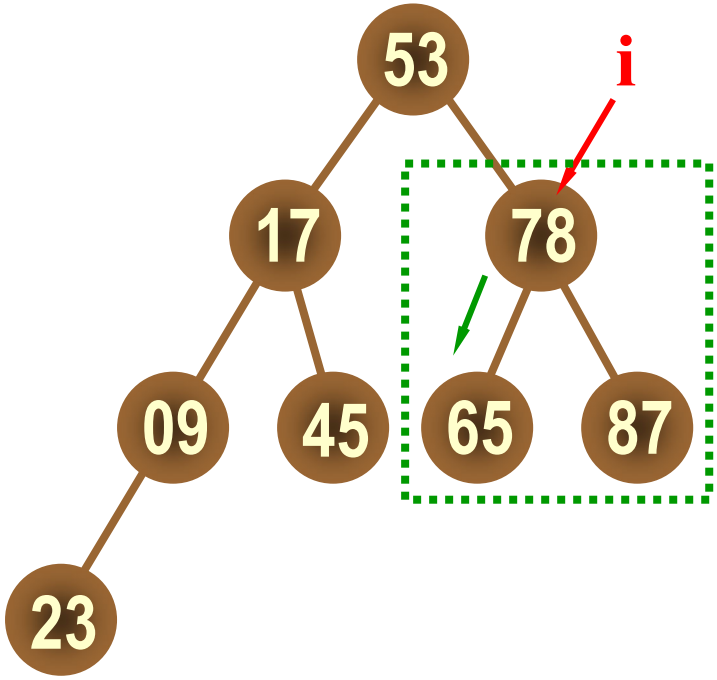
```
template <class Type> MinHeap <Type> ::
MinHeap ( Type arr[ ], int n ) {
//根据给定数组中的数据和大小，建立堆对象
    maxHeapSize = DefaultSize < n ? n : DefaultSize;
    heap = new Type [maxHeapSize];
    if ( heap == NULL ) {
       cerr << "存储分配错！" << endl;  exit(1);
    }
    for ( int i = 0; i < n; i++ )  //数组传送
        heap[i] = arr[i];
```

```
    currentSize = n;  //当前堆大小
    int currentPos = ( currentSize-2 )/2;
    //找最初调整位置：最后的分支结点号
    while ( currentPos >= 0 ) {
    //从下到上逐步扩大，形成堆
        SiftDown ( currentPos, currentSize-1 );
        //从 currentPos 开始，到 CurrentSize 止调整
        currentPos--;
    }
}
```

将一组用数组存放的任意数据调整成堆

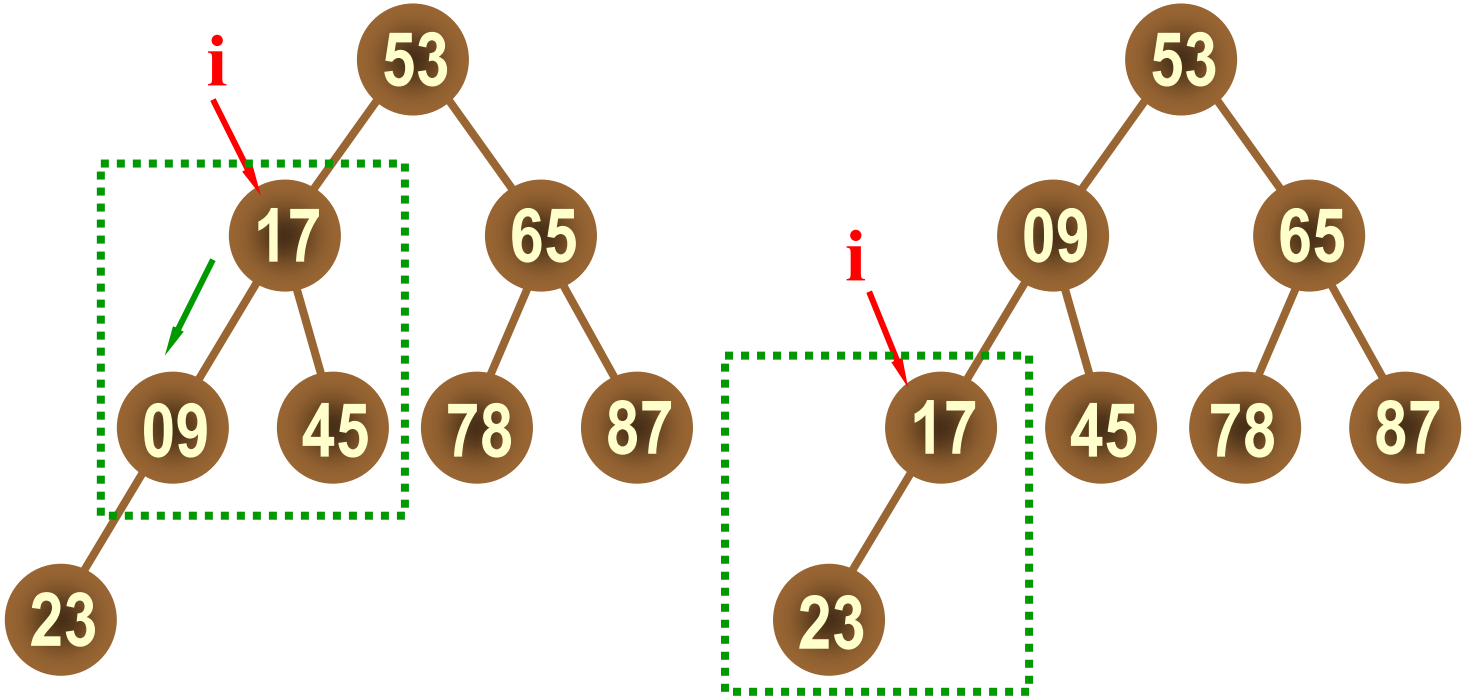


currentPos = i = 3

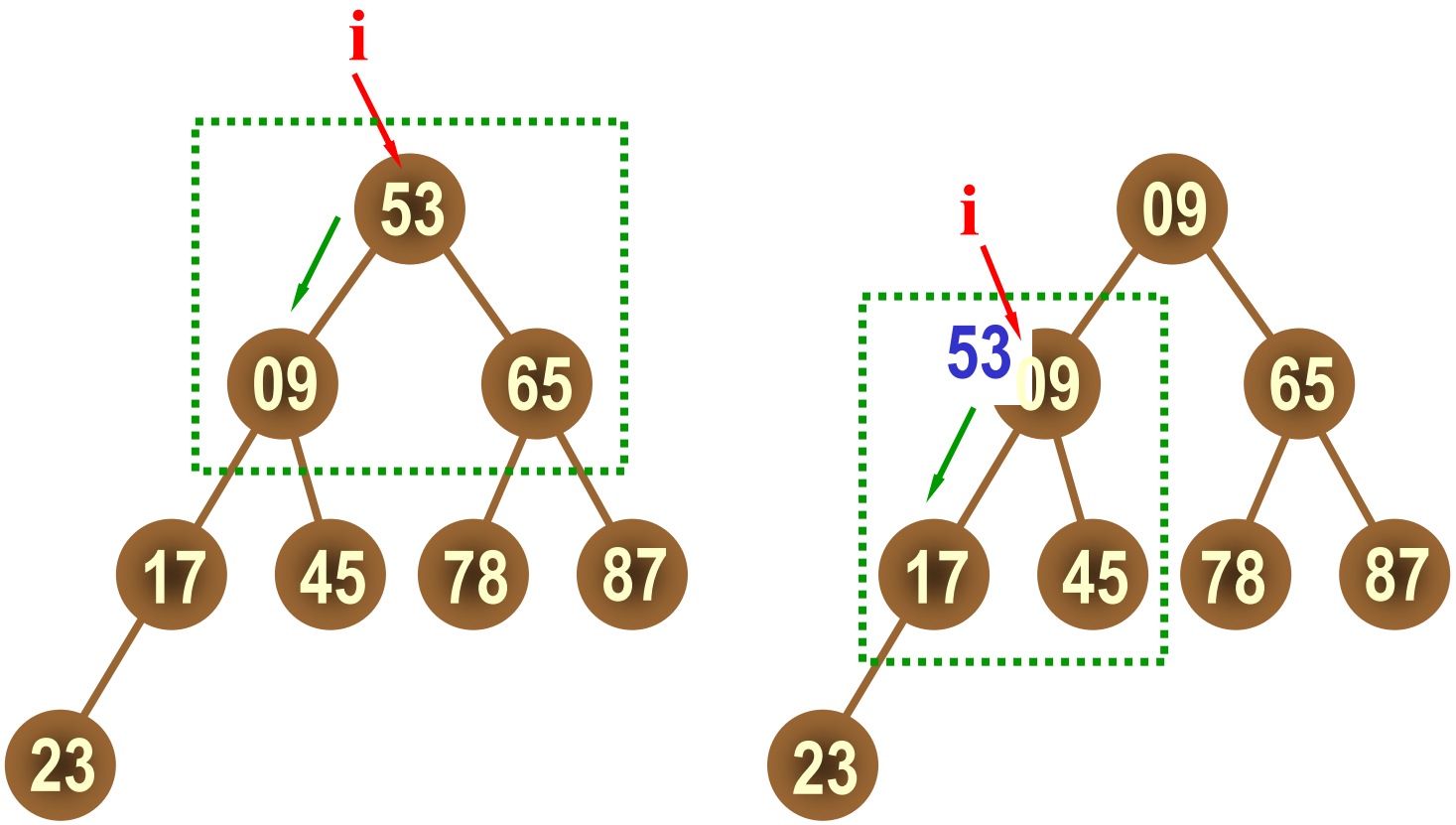


currentPos = i = 2

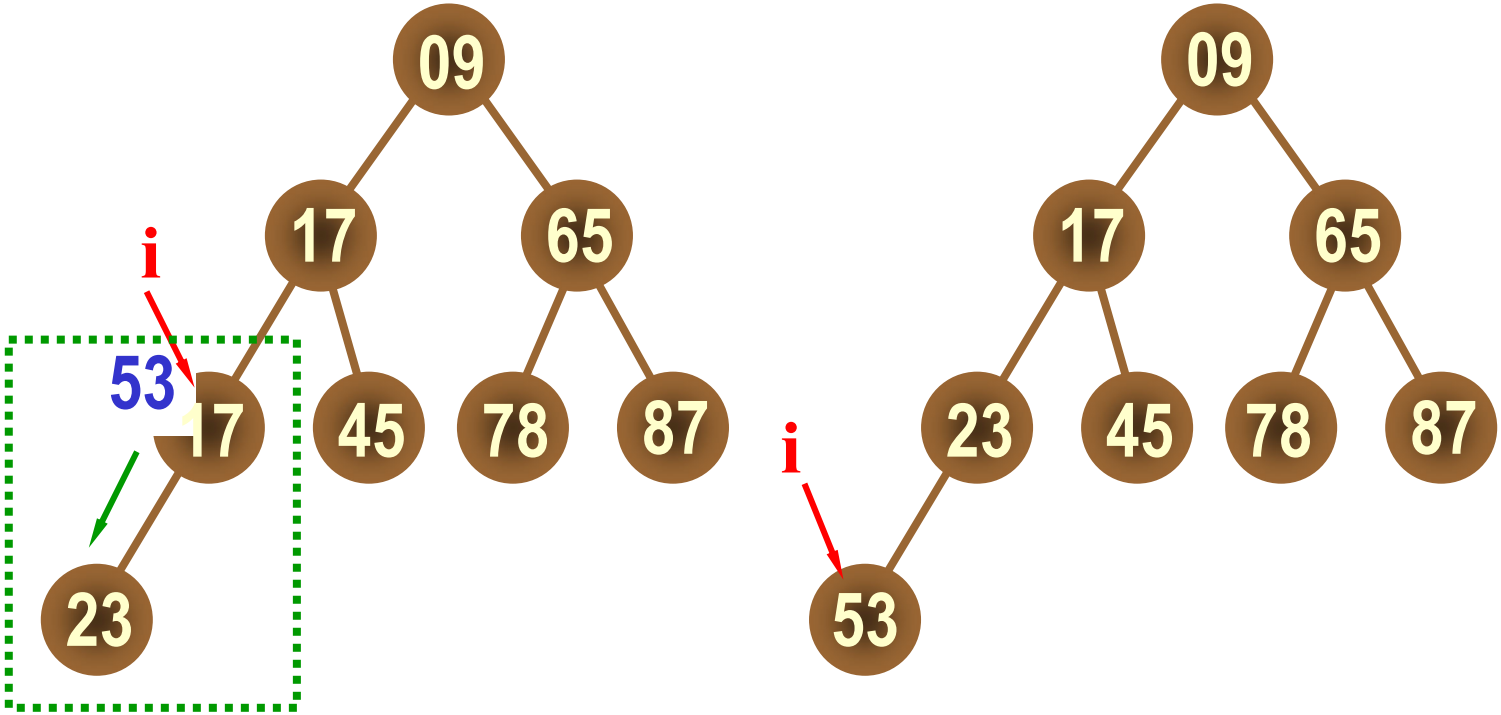
自下向上逐步调整为最小堆



currentPos = i = 1



currentPos = i = 0


09
17
65
i
53
17
45
78
87
23
09
17
65
23
45
78
87
i
53

最小堆的向下调整算法

- 对有m个记录的集合R，将其置为完全二叉树的顺序存储。
  - 从某个记录结点i开始向下调整
    - 如果结点i左子女的关键码小于右子女关键码：R[j].key < R[j+1].key (j=2i+1)，则沿结点i的左分支进行调整。
    - 否则，沿结点i的右分支进行调整，让j指示参加调整的子女。

- **调整方法**
  - **以R[i]与R[j]进行关键码比较**
    - **如果R[i].key > R[j].key，则两结点对调位置，把关键码小的结点上浮，然后令i=j, j=2i+1，继续进行下一层比较。**
    - **如果R[i].key <= R[j].key，则不对调，也不再向下一层继续比较，算法终止。**
    - **最后结果是关键码最小的结点上浮到了堆顶，最小堆形成。**

```
template <class Type> void MinHeap <Type> ::
SiftDown ( int start, int m ) {
    int i = start, j = 2*i+1;  // j 是 i 的左子女
    Type temp = heap[i];
    while ( j <= m ) {
        if ( j < m && heap[j] > heap[j+1] )
            j++;  //两子女中选小者
        if ( temp <= heap[j] ) break;
        else {
                heap[i] = heap[j];  //下面的上浮
                i = j;  j = 2*j+1;  //向下滑动   }
    }
    heap[i] = temp;
}
```

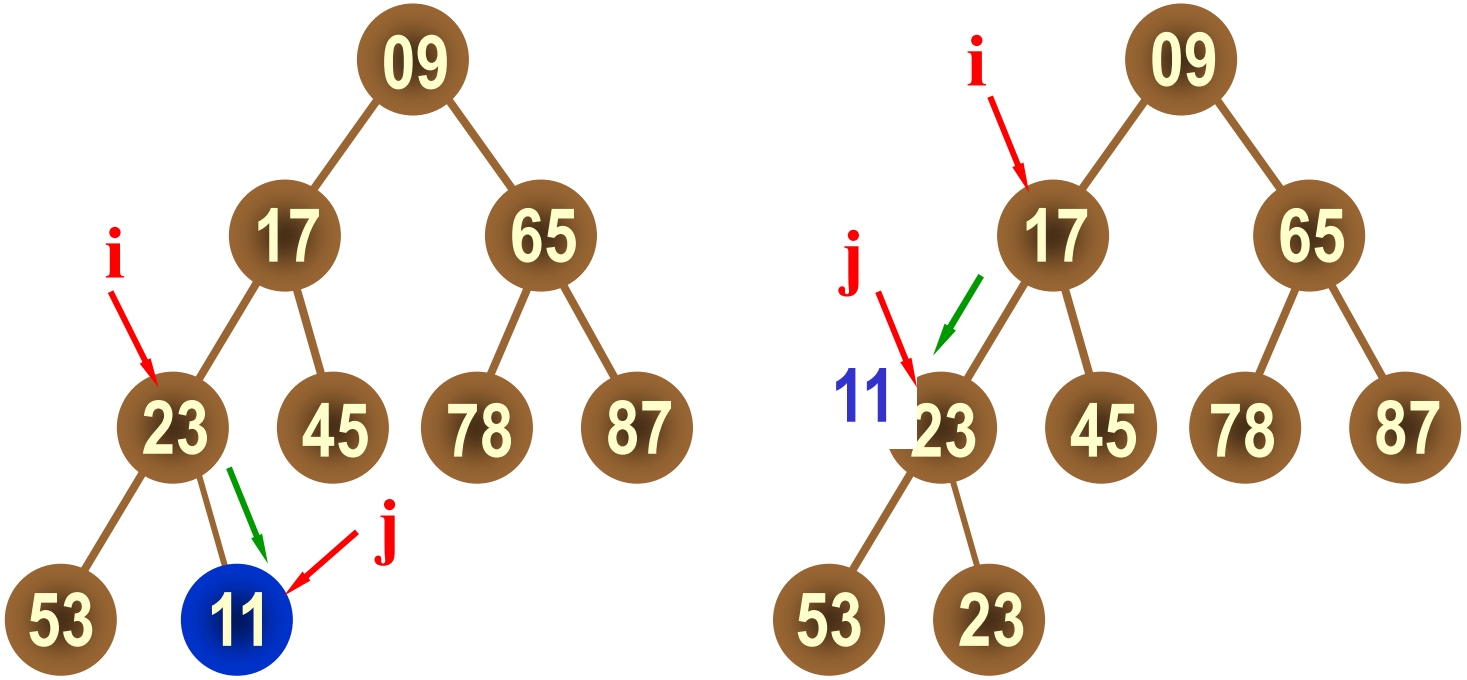
堆的插入

```
template <class Type> bool MinHeap <Type> ::
Insert ( const Type &x ) {
//在堆中插入新元素 x
   if ( currentSize == maxHeapSize )  //堆满
     { cerr << "堆已满" << endl;  return false; }
   heap[currentSize] = x;  //插在表尾
   SiftUp (currentSize);  //向上调整为堆
   currentSize++;  //堆元素增一
   return true;
}
```

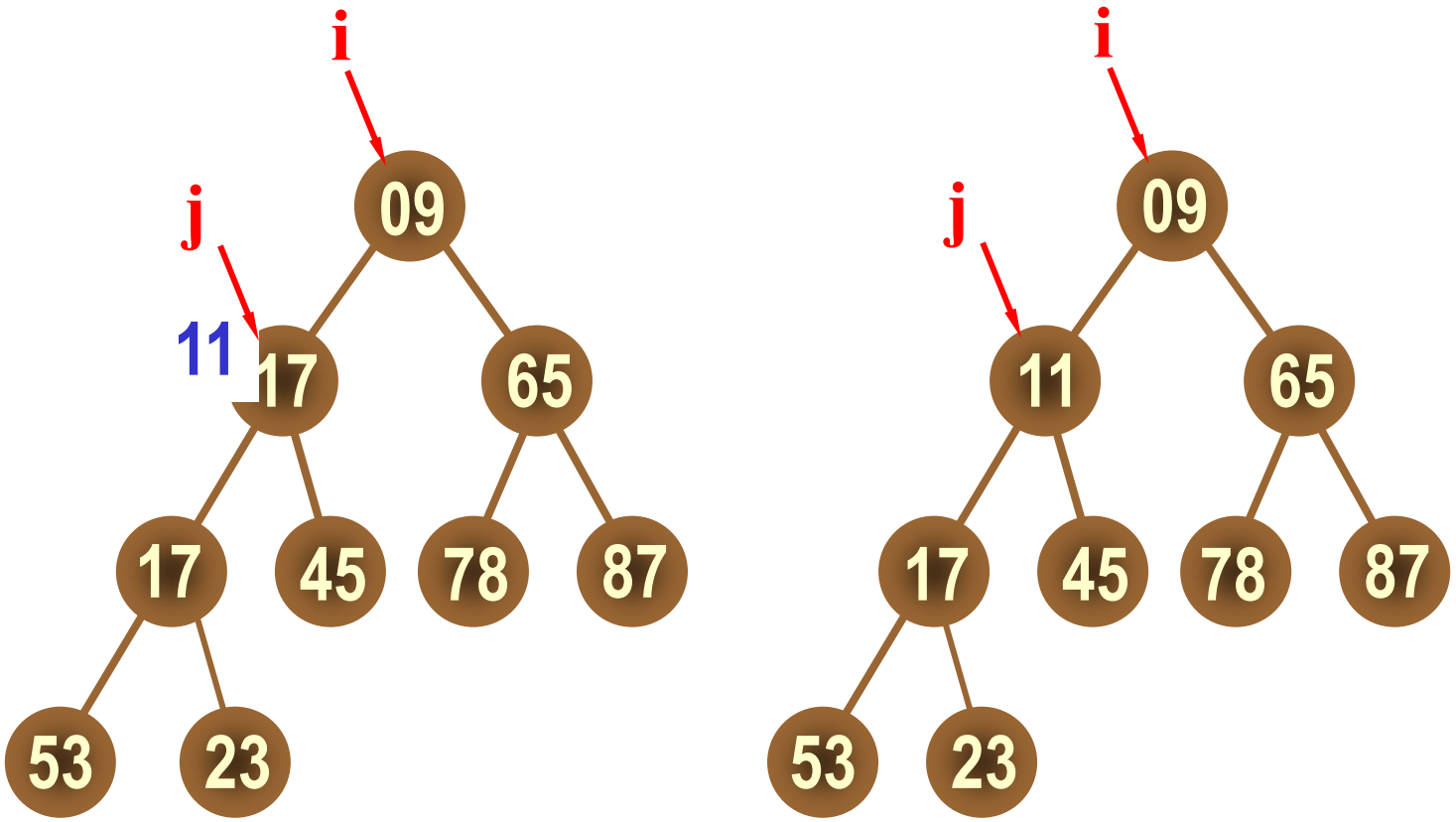
最小堆的向上调整算法

```
template <class Type> void MinHeap <Type> ::
SiftUp ( int start ) {
//从 start 开始，向上直到 0，调整堆
    int j = start,  i = (j-1)/2;  // i 是 j 的双亲
    Type temp = heap[j];
    while ( j > 0 ) {
        if ( heap[i] <= temp ) break;
        else { heap[j] = heap[i];  j = i;  i = (i-1)/2; }
    }
    heap[j] = temp;
}
```

最小堆的向上调整



在堆中插入新元素11


i
j
11
09
17
65
17
45
78
87
53
23
i
j
09
11
65
17
45
78
87
53
23

最小堆的删除算法

从最小堆删除具有最小关键码记录的操作，是将最小堆的堆顶元素（即完全二叉树的顺序表示的第0号元素）删除。

在将该元素取走后，一般以堆的最后一个结点填补取走的堆顶元素，并将堆的实际元素个数减1。

用最后一个元素取代堆顶元素将破坏堆，需要从堆顶向下进行调整。

```
template <class Type> bool MinHeap <Type> ::
Remove ( Type &x ) {
    if ( !CurrentSize )
      { cout << "堆已空" << endl;  return false; }
    x = heap[0];  //最小元素出队列
    heap[0] = heap[currentSize-1];
    currentSize--;  //用最小元素填补
    SiftDown ( 0, currentSize-1 );  //调整
    return true;
}
```

5.9 Huffman树及其应用

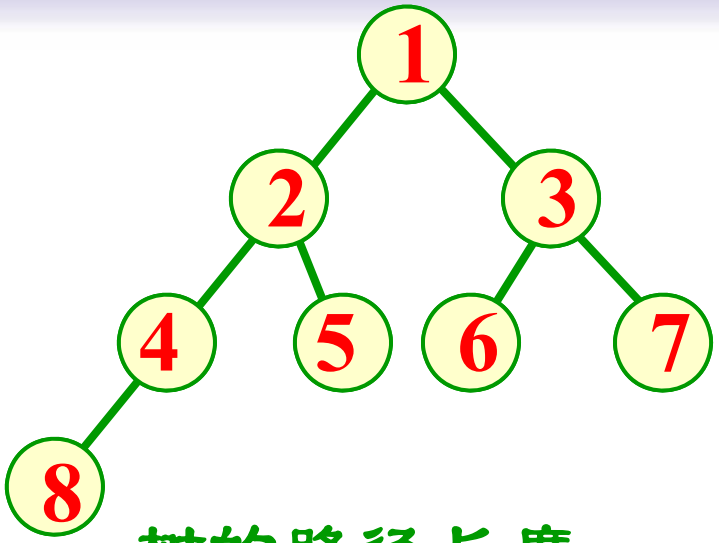
路径长度 (Path Length)

两个结点之间的路径长度 PL 是连接两结点的路径上的分支数。树的路径长度是各结点到根结点的路径长度之和。

树的外部路径长度是各叶结点（外结点）到根结点的路径长度之和 EPL。

树的内部路径长度是各非叶结点（内结点）到根结点的路径长度之和 IPL。

树的路径长度 PL = EPL + IPL



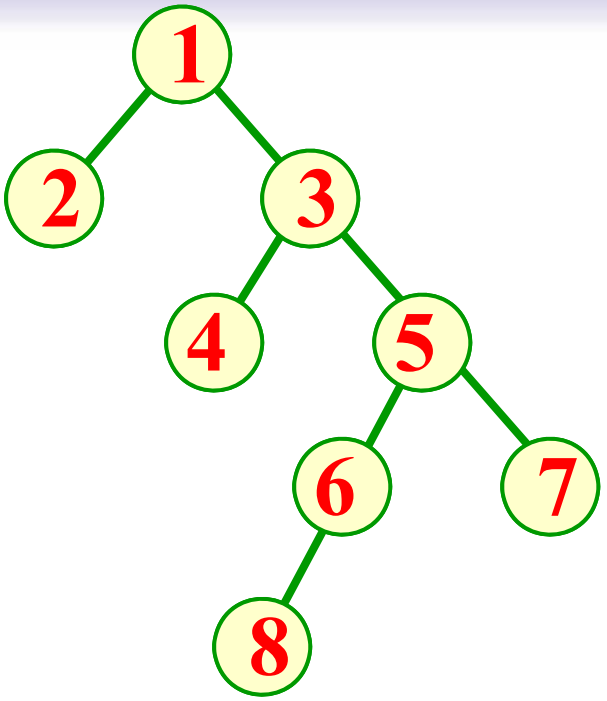
树的路径长度

PL = 0+1*2+

2*4+3*1

= 13

完全二叉树



树的路径长度

PL = 0+1*2+2*2+

3*2+4*1

= 16

一般二叉树

n 个结点的二叉树的路径长度不小于下述数列前 n 项的和，即

$$PL = \sum_{i=0}^{n-1} \lfloor \log_2(i+1) \rfloor$$

$$= 0+1+1+2+2+2+2+3+3+\cdots$$

其路径长度最小者为 $PL = \sum_{i=0}^{n-1} \lfloor \log_2(i+1) \rfloor$

带权路径长度

(Weighted Path Length, WPL)

假设给定一个有n个权值的集合$\{w_0, w_1, w_2, \cdots, w_{n-1}\}$，其中$w_i \geq 0$ $(0 \leq i \leq n-1)$。若T是一棵有n个叶结点的二叉树，而且将权值$w_0, w_1, w_2, \cdots, w_{n-1}$分别赋给$T$的$n$个叶结点，则称$T$是权值为$w_0, w_1, w_2, \cdots, w_{n-1}$的扩充二叉树。

带有权值的叶结点称为扩充二叉树的外结点。

其余不带权值的分支结点称为内结点。

外结点的带权路径长度为树T的根到该结点的路径长度与该结点上权值的乘积。

具有n个外结点的扩充二叉树的带权路径长度为树的各外结点（叶结点）所带的权值与该结点到根的路径长度的乘积的和。

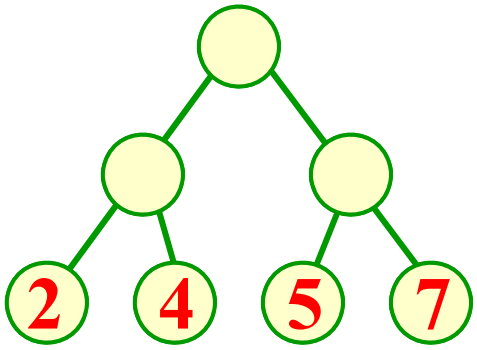
$$WPL = \sum_{i=0}^{n-1} w_i * l_i$$

**w_i为外结点i所带的权值；

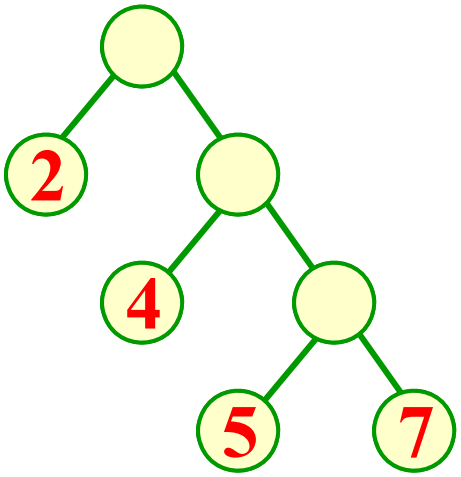
**l_i为外结点i到根结点的路径长度。

在权值为$w_0, w_1, w_2, \cdots, w_{n-1}$的扩充二叉树中，其$WPL$最小的扩充二叉树称为最优二叉树。

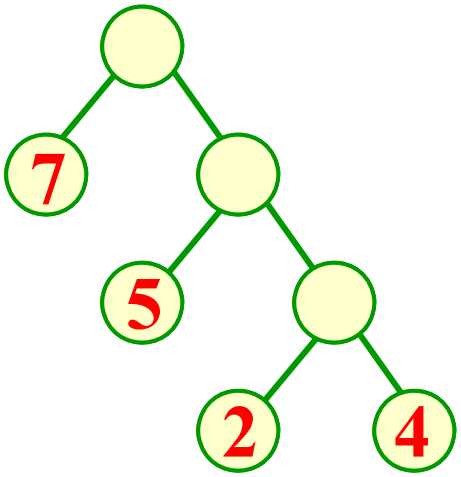
具有不同带权路径长度的扩充二叉树



WPL = 2*2+4*2
+5*2+7*2
= 36



WPL = 2*1+4*2
+5*3+7*3
= 46



WPL = 7*1+5*2
+2*3+4*3
= 35

带权路径长度达到最小

带权路径最小的扩充二叉树不一定是完全二叉树

Huffman树

- 带权路径长度达到最小的扩充二叉树即为Huffman树。
- 在Huffman树中，权值大的结点离根最近。

Huffman算法

(1) 由给定的 n 个权值 $\{w_0, w_1, w_2, \cdots, w_{n-1}\}$，构造具有 n 棵扩充二叉树的森林 $F = \{T_0, T_1, T_2, \cdots, T_{n-1}\}$，其中每棵扩充二叉树 T_i 只有一个带权值 w_i 的根结点，其左、右子树均为空。

(2) 重复以下步骤，直到 F 中仅剩下一棵树为止：

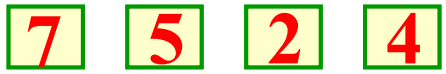
① 在 F 中选取两棵根结点的权值最小的扩充二叉树，做为左、右子树构造一棵新的二叉树。置新的二叉树的根结点的权值为其左、右子树上根结点的权值之和。

② 在 F 中删去这两棵二叉树。

③ 把新的二叉树加入 F。

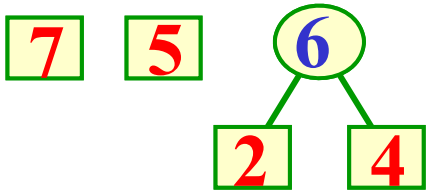
Huffman树的构造过程

F : {7} {5} {2} {4}



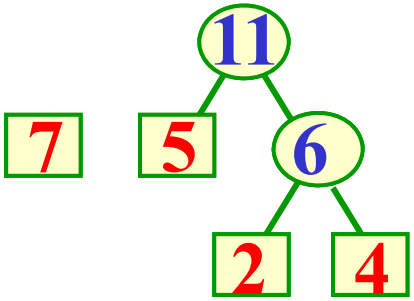
初始

F : {7} {5} {6}



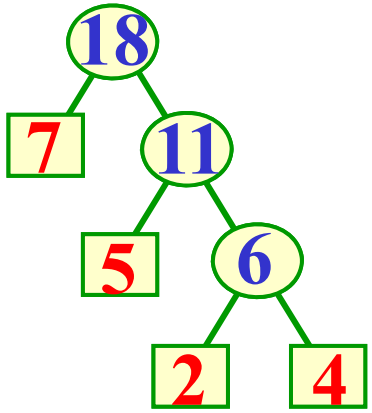
合并{2}与{4}

F : {7} {11}



合并{5}与{6}

F : {18}



合并{7}与{11}

Huffman树类定义

```
const  int DefaultSize = 20;
template <class Type> class HuffmanTree;
template <class Type> class HuffmanNode {
friend class HuffmanTree;
private:
    Type data;
    HuffmanNode <Type> *leftChild, *rightChild, *parent;
public:
     HuffmanNode ( Type elem,
                          HuffmanNode <Type> *left = NULL,
                          HuffmanNode <Type> *right = NULL,
                          HuffmanNode <Type> *pr = NULL ) :
          data(elem), leftChild(left), rightChild(right), parent(pr) { }
};
```

```
template <class Type> class HuffmanTree {
public:
    HuffmanTree ( Type w[ ], int n );
    ~HuffmanTree ( ) { delete Tree ( root ); }
protected:
    HuffmanNode <Type> *root;
    void DeleteTree ( HuffmanNode <Type> *t );
    void MergeTree ( HuffmanNode <Type> &ht1,
                     HuffmanNode <Type> &ht2,
                     HuffmanNode <Type> *&parent );
};
```

建立Huffman树的算法

```
template <class Type> HuffmanTree <Type> ::
HuffmanTree ( Type w[ ],  int n )
{
   HuffmanNode <Type> *parent, &first, &second;
   HuffmanNode <Type> *NodeList =
                                   new HuffmanNode <Type> [n];
   MinHeap < Type > hp;
   for ( int i = 0; i < n; i++ ) {
         NodeList[i].data = w[i+1];
         NodeList[i].leftChild = NULL;
         NodeList[i].rightChild = NULL;
         NodeList[i].parent = NULL;
         hp.Insert ( NodeList[i] );
    }
```

```
    for ( int i = 0; i < n-1; i++) {
        hp.RemoveMin (first);
        hp.RemoveMin (second);
        Merge ( first, second, parent );
        hp.Insert (*parent);
        root = parent;
    }
}
```

```
template <class Type> void HuffmanTree <Type> ::
MergeTree ( HuffmanNode  <Type> &bt1,
                    HuffmanNode <Type> &bt2,
                        HuffmanNode <Type> *&parent )
{
   parent->leftChild = &bt1;
   parent->rightChild = &bt2;
   parent->data.key =
            bt1.root->data.key + bt2.root->data.key;
   bt1.root->parent = bt2.root->parent = parent;
 }
```

采用静态链表的Huffman树

可以采用静态链表方式存储Huffman树。为查找在构造过程中森林里的树根，为每个结点设置双亲指针。

Huffman树的定义

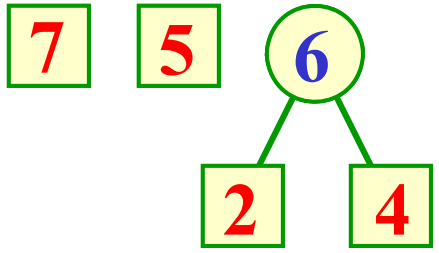
```
const  int n = 20;
const int m = 2*n-1;
typedef struct {
    float weight;
    int parent, lchild, rchild;
} HTNode;
typedef HTNode HuffmanTree[m];
```

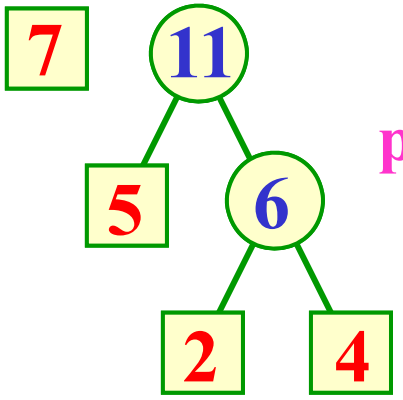
建立Huffman树的过程如图所示：

7 5 2 4

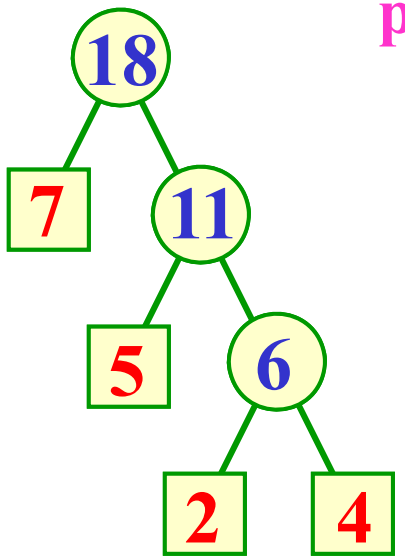
	weight	parent	leftChild	rightChild
0	7	-1	-1	-1
1	5	-1	-1	-1
2	2	-1	-1	-1
3	4	-1	-1	-1
4		-1	-1	-1
5		-1	-1	-1
6		-1	-1	-1

	weight	parent	leftChild	rightChild
0	7	-1	-1	-1
1	5	-1	-1	-1
p1→ 2	2	4 ~~-1~~	-1	-1
p2→ 3	4	4 ~~-1~~	-1	-1
i→ 4	6	-1	2 ~~-1~~	3 ~~-1~~
5		-1	-1	-1
6		-1	-1	-1





	weight	parent	leftChild	rightChild
0	7	-1	-1	-1
p1 → 1	5	5 ~~-1~~	-1	-1
2	2	4	-1	-1
3	4	4	-1	-1
p2 → 4	6	5 ~~-1~~	2	3
i → 5	11	-1	1 ~~-1~~	4 ~~-1~~
6		-1	-1	-1



		weight	parent	leftChild	rightChild
p1 →	0	7	6 ~~-1~~	-1	-1
	1	5	5	-1	-1
	2	2	4	-1	-1
	3	4	4	-1	-1
	4	6	5	2	3
p2 →	5	11	6 ~~-1~~	1	4
i →	6	18	-1	0 ~~-1~~	5 ~~-1~~

建立Huffman树的算法

```
void CreateHuffmanTree ( HuffmanTree T,
                                    float fr[ ], int n ) {
    for ( int i = 0; i < n; i++ )
        T[i].weight = fr[i];
    for ( i = 0; i < m; i++ ) {
        T[i].parent = -1;
        T[i].leftChild = -1;
        T[i].rightChild = -1; }
    for ( i = n; i < m; i++ ) {
        int min1 = min2 = MaxNum;
        int pos1, pos2;
```

```
        for ( int j = 0; j < i; j++ )
          if ( T[j].parent == -1 )
              if ( T[j].weight < min1 )
                { pos2 = pos1;  min2 = min1;
                  pos1 = j;  min1 = T[j].weight;  }
              else if ( T[j].weight < min2 )
                        { pos2 = j;  min2 = T[j].weight;  }
          T[i].leftChild = pos1;
          T[i].rightChild = pos2;
          T[i].weight = T[pos1].weight + T[pos2].weight;
          T[pos1].parent = T[pos2].parent = i;
    }
}
```

Huffman编码

主要用途是实现数据压缩。

设给出一段报文：

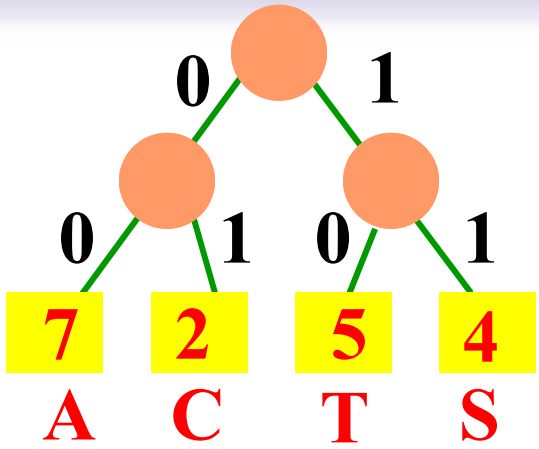
CAST CAST SAT AT A TASA

字符集合是 { C, A, S, T }，各个字符出现的频度（次数）是 W = { 2, 7, 4, 5 }。

若给每个字符以等长编码

A : 00　T : 10　C : 01　S : 11

则总编码长度为 (2+7+4+5) * 2 = 36。



若按各个字符出现的概率不同而给予不等长编码，可望减少总编码长度。

各字符出现概率为{ 2/18, 7/18, 4/18, 5/18 }，化整为 { 2, 7, 4, 5 }。以它们为各叶结点上的权值，建立Huffman树。左分支赋 0，右分支赋 1，得Huffman编码（变长编码）。

A : 0　T : 10　C : 110　S : 111

它的总编码长度：7*1+5*2+(2+4)*3 = 35，比等长编码的情形要短。

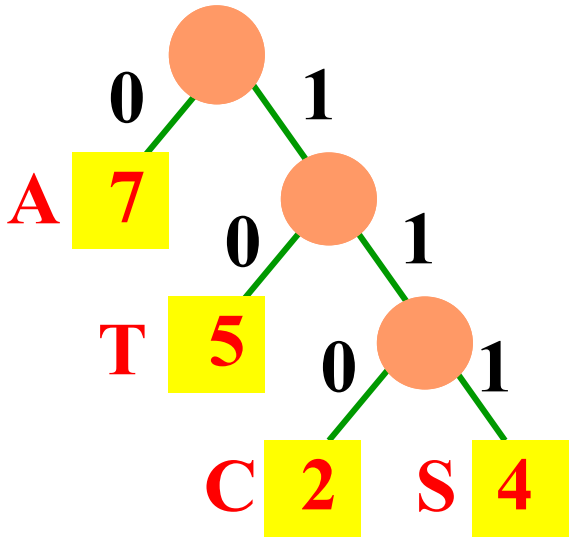
总编码长度正好等于Huffman树的带权路径长度WPL。

Huffman编码是一种无前缀编码，解码时不会混淆。

√CAST: 110011110

√给出11001010111对应的字符串？

C AT T S

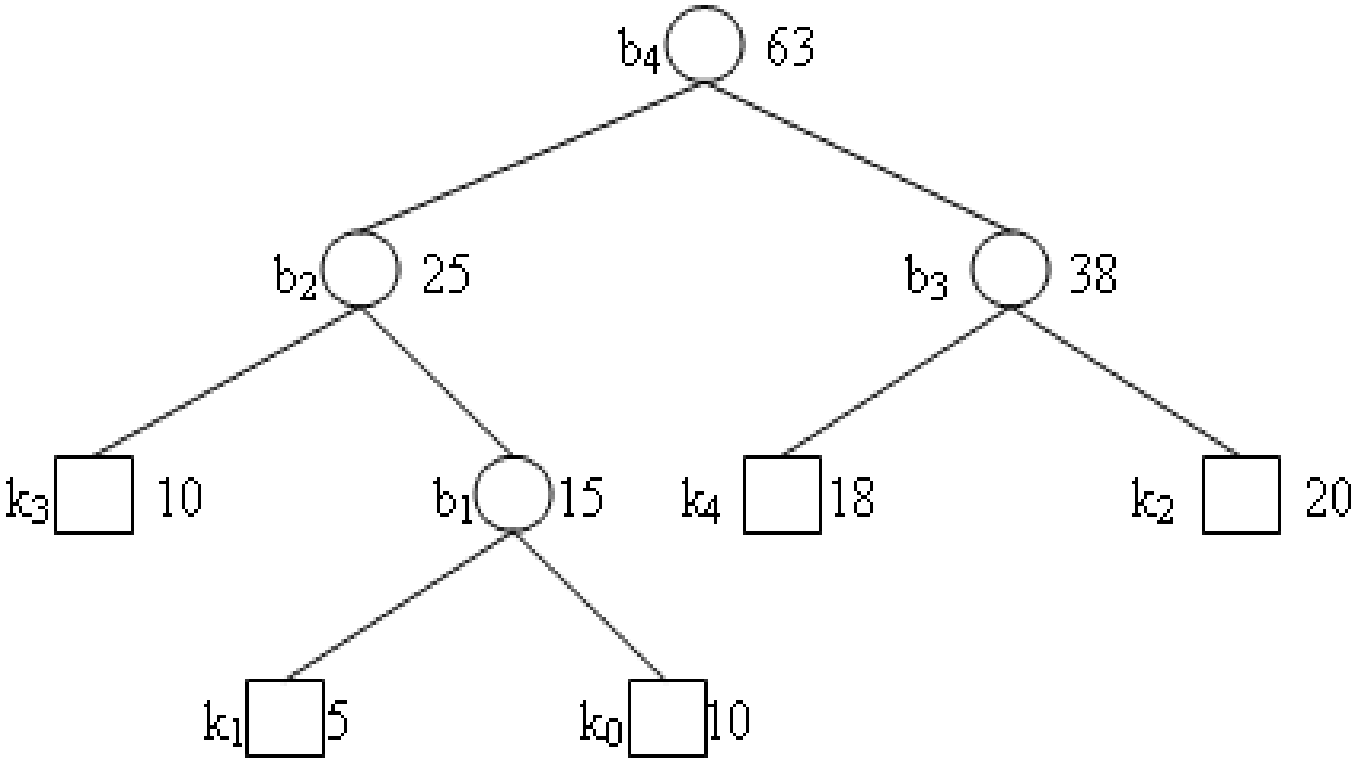


Huffman编码树

Huffman编码举例

- 例如，给定一些仅由五个字母（信息）a，b，c，d，e组成的单词，在这些字母中，它们的使用频率依次为10, 5, 20, 10, 18。
- 可以在图6-21中由左到右依次用d，b，a，e，c作为外部结点，这样一来，字母的使用频率就和树中的外部结点的权相一致了，并且把向左分支记为0，把向右分支记为1。
- 于是就得到一种编码：d是00、b是010、a是011、e是10、c是11。

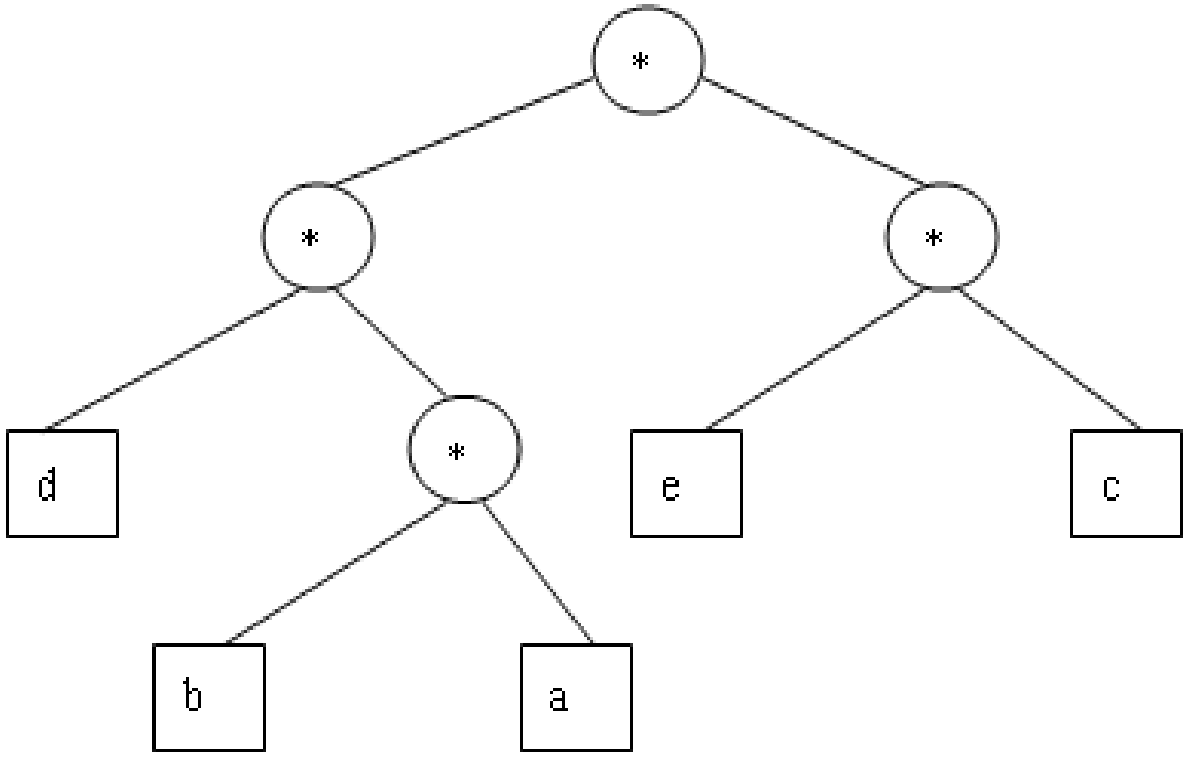
图6-21



用Huffman编码解码

- 图6-22给出相应的译码树。每种合格的编码都能用图6-22的译码树容易地得到相应的、唯一的译码。
  - 例如，0100110010的相应译码是bade。然而，不合格编码001110111就得不到译码。

图6-22



Huffman编码的特点

- Huffman编码是一种无前缀编码。解码时不会混淆。
- 同样的元素集合，产生的Huffman树及编码都可能是不一样的。

- 若假设A、B、C、D的编码分别为01、0、00、1，则电文"BCBDDAB"的编码序列便为"000011010"（共9位）。但此编码存在多义性，即该编码序列可以译为"CCDDAB"、"BCADBDB"，也可译为"BBBBDDBDB"等。
- 一种正确的编码方法要求任一字符的编码都不能是另一字符编码的前缀，这种编码是非前缀编码。非前缀编码可确保译码的唯一性。
- 哈夫曼编码就能实现这样的要求，并可使电文编码序列更短。

本章小结

- 知识点
  - 二叉树
  - 遍历方法
  - 线索二叉树
  - 二叉树计数
  - 堆-完全二叉树的结构
  - **Huffman树**

常见题型

- 树一般是数据结构中重点考察的内容之一
- 树的基本性质
  - 结点的数量、度、和树的高度的关系；
  - 完全二叉树的性质
    - 深度k (k>=0)的二叉树只有度为0和2的结点，则结点数至少为2k+1
    - 如果是完全二叉树，则结点数至少为2^k【注意k>0还是k>=0】
- 二叉树的存储
  - 顺序存储表示与链表存储表示之间的转换、计算结点的存储地址；查找两个指定结点u和v的共同祖先；左右子树交换；二叉树的计数；……。
  - 遍历：根据某两种遍历序列构造二叉树；遍历也可以反方向进行，例如从右到左输出二叉树的叶子结点（从右到左层次遍历）；二叉树遍历与表达式的前中后缀表示的关系。
- 树和森林
  - 与二叉树的相互转换
- 堆的构造和应用
- Huffman树
  - 编码、译码、 Huffman树的性质（有多少个结点、多少个叶子结点等）

- **课程习题**

  - 笔做题——5.20, 5.23, 5.29, 5.33
    （以作业形式提交）

  - 上机题——5.24, 5.26, 5.27, 5.28,

  - 思考题——5.3, 5.4, 5.6, 5.12, 5.13, 5.14,
    5.15, 5.16, 5.18, 5.19, 5.31