第六章 树与森林

- 树和森林的概念
- 二叉树 (Binary Tree)
- 二叉树遍历 (Binary Tree Traversal)
- 线索化二叉树 (Threaded Binary Tree)
- 堆 (Heap)
- 树与森林 (Tree & Forest)
- 二叉树的计数
- 霍夫曼树 (Huffman Tree)
- 小结

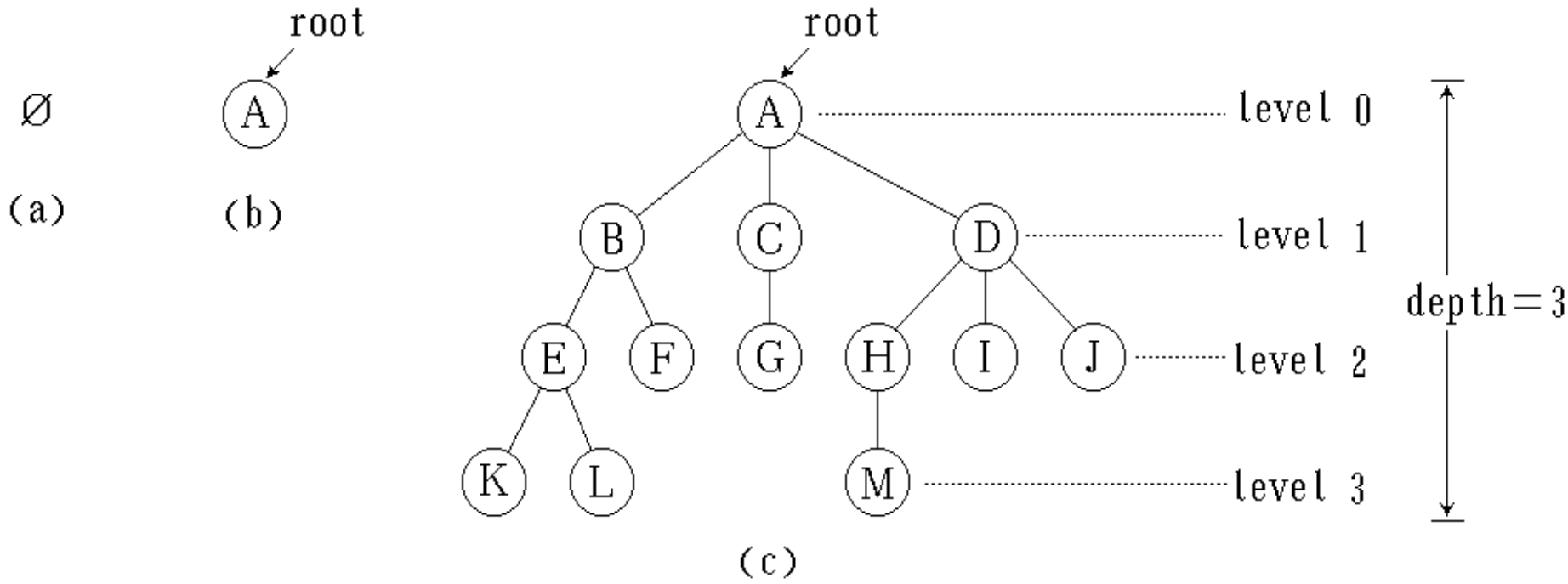
树和森林的概念

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(subTree)。

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)
- 结点的度(degree)
- 分支(branch)结点
- 叶(leaf)结点
- 子女(child)结点
- 双亲(parent)结点
- 兄弟(sibling)结点
- 祖先(ancestor)结点
- 子孙(descendant)结点
- 结点所处层次(level)
- 树的高度(depth)
- 树的度(degree)
- 有序树
- 无序树
- 森林

树的抽象数据类型

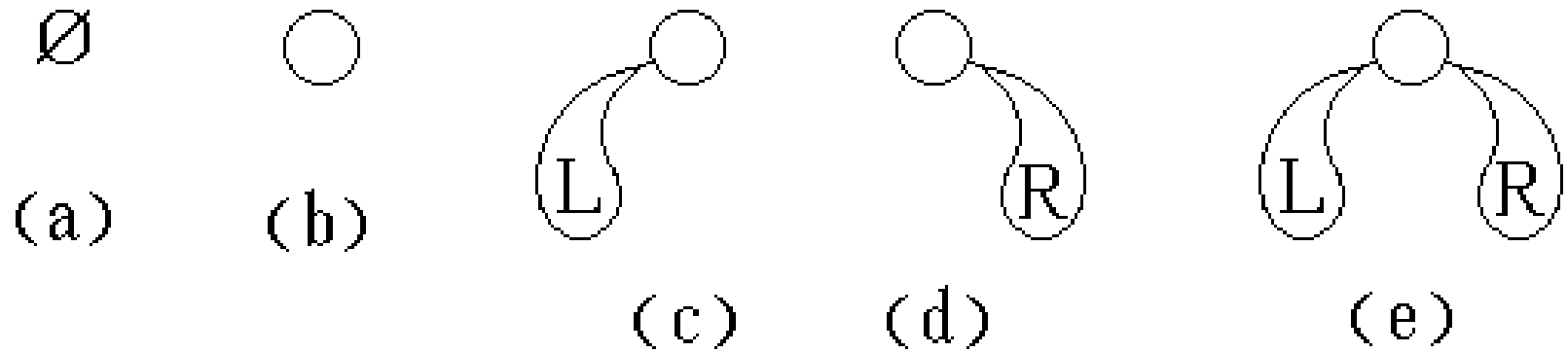
```
template <class Type> class Tree {
public:
    Tree ( );
    ~Tree ( );
    position Root ( );
    BuildRoot ( const Type& value );
    position FirstChild ( position p );
    position NextSibling ( position p, position v );
    position Parent ( position p );
    Type Retrieve ( position p );
```

```
    int InsertChild ( const position p,
                                  const Type &value );
    int DeleteChild ( position p, int i );
    void DeleteSubTree ( position t );
    int IsEmpty ( );
}
```

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。



二叉树的五种不同形态

二叉树的性质

性质1 若二叉树的层次从0开始，则在二叉树的第 i 层最多有 2^i 个结点。$(i \geq 0)$

[证明用数学归纳法]

性质2 高度为 k 的二叉树最多有 $2^{k+1}-1$ 个结点。$(k \geq -1)$

[证明用求等比级数前k项和的公式]

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

$$n_0 = n_2 + 1$$

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，

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

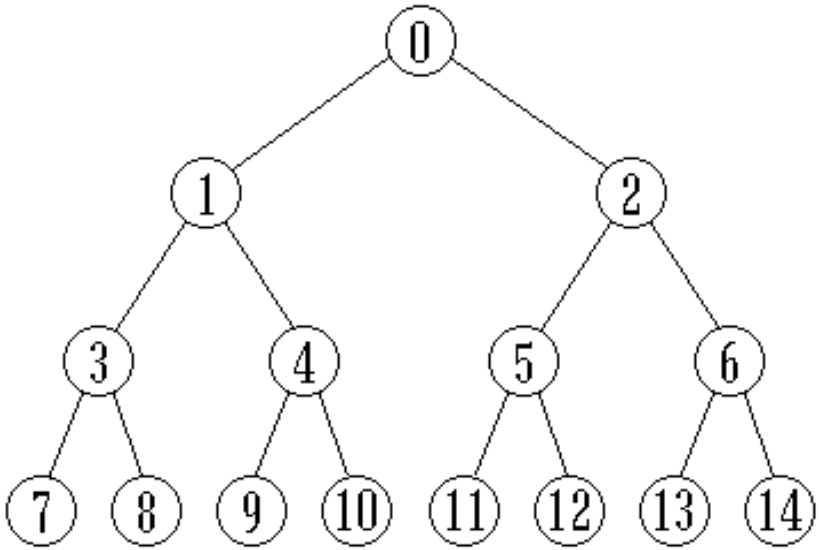
因此，有 $2n_2 + n_1 = n_0 + n_1 + n_2 - 1$

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

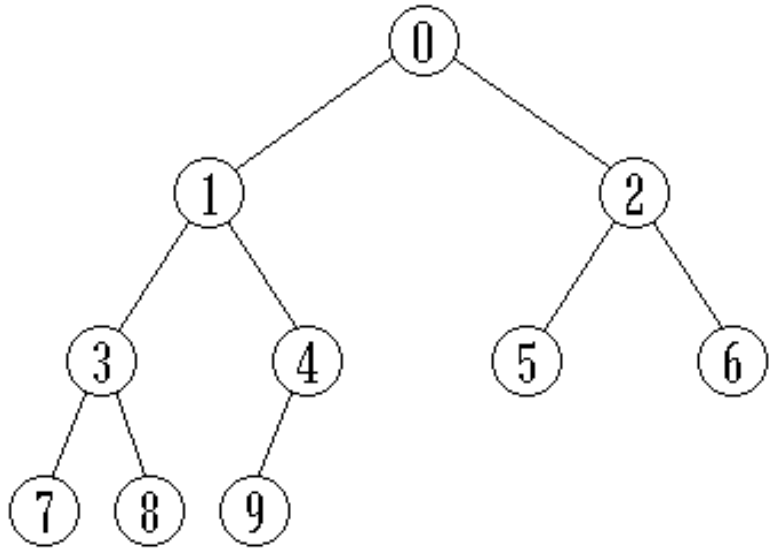
定义1 满二叉树(Full Binary Tree)

定义2 完全二叉树(Complete Binary Tree)

若设二叉树的高度为h，则共有h+1层。除第 h 层外，其它各层 (0 ~ h-1) 的结点数都达到最大个数，第 h 层从右向左连续缺若干结点，这就是完全二叉树。



（a） 满二叉树



（b）完全二叉树

性质4 具有 n 个结点的完全二叉树的高度为

$$\lceil \log_2(n+1) \rceil - 1$$

证明：设完全二叉树的高度为h，则有

$$2^h - 1 < n \le 2^{h+1} - 1 \Rightarrow 2^h < n+1 \le 2^{h+1} \Rightarrow$$

取对数 $h < \log_2(n+1) \le h+1$

性质5 如果将一棵有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，然后按此结点编号将树中各结点顺序地存放于一个一维数组中，并简称编号为i的结点为结点i ($0 \le i \le n-1$。则有以下关系：

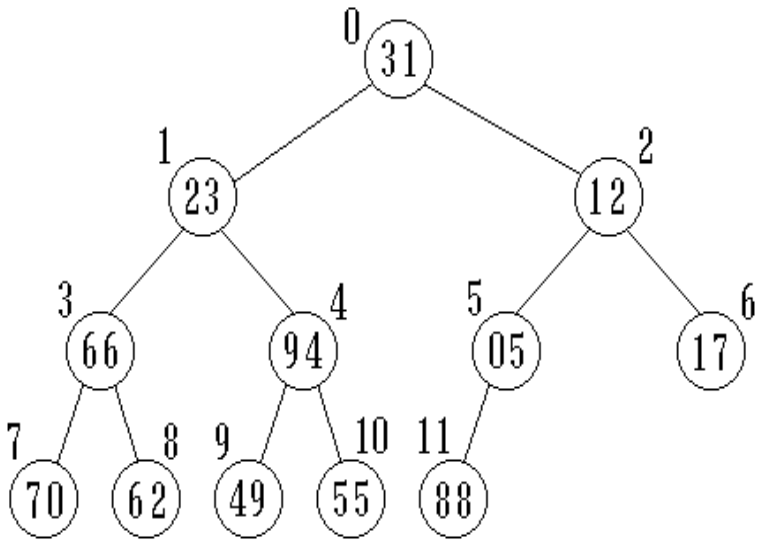
- 若i == 0, 则 i 无双亲
  若i > 0, 则 i 的双亲为$\lfloor (i-1)/2 \rfloor$
- 若2*i+1 < n, 则 i 的左子女为2*i+1
  若2*i+2 < n, 则 i 的右子女为2*i+2
- 若 i 为偶数, 且i != 0, 则其左兄弟为i-1
  若 i 为奇数, 且i != n-1, 则其右兄弟为i+1
- i 所在层次为$\lfloor \log_2 (i+1) \rfloor$

二叉树的抽象数据类型

```
template <class Type> class BinaryTree {
public:
    BinaryTree ( );
    BinaryTree ( BinTreeNode<Type> * lch,
        BinTreeNode<Type> * rch, Type item );
    int IsEmpty ( );
    BinTreeNode<Type> *Parent ( );
    BinTreeNode<Type> *LeftChild ( );
    BinTreeNode<Type> *RightChild ( );
    int Insert ( const Type &item );
    int Find ( const Type &item ) const;
    Type GetData ( ) const;
    const BinTreeNode<Type> *GetRoot ( ) const;
}
```

二叉树的表示

数组表示

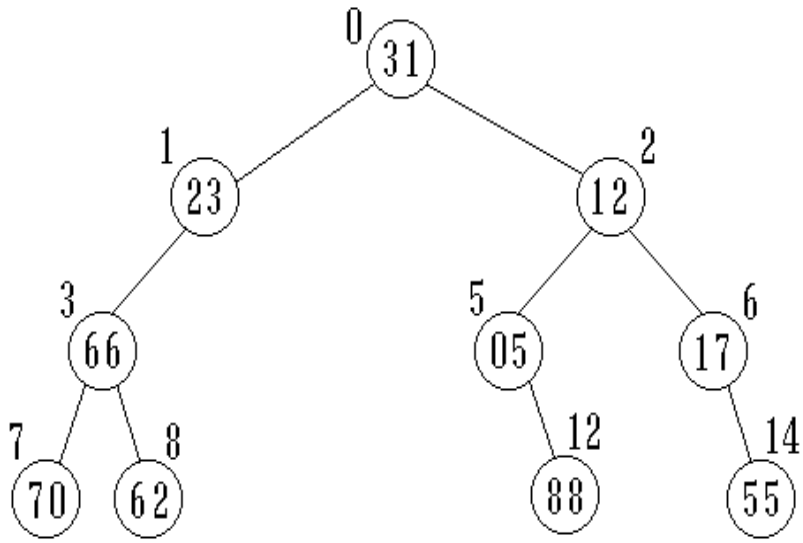


(a)

0	1	2	3	4	5	6	7	8	9	10	11
31	23	12	66	94	05	17	70	62	49	55	88

(b)

完全二叉树的数组表示



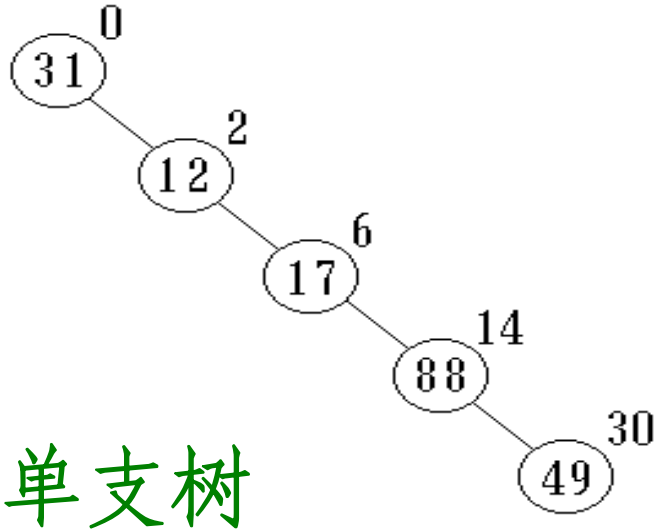
(a)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
31	23	12	66		05	17	70	62				88		55

(b)

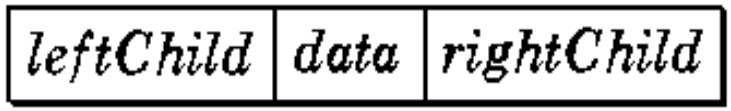
一般二叉树的数组表示

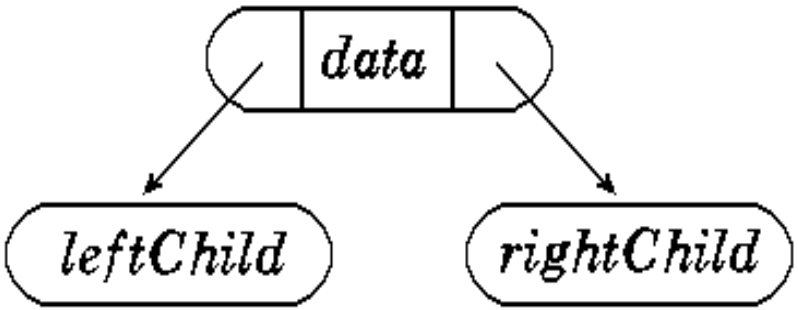
由于一般二叉树必须仿照完全二叉树那样存储，可能会浪费很多存储空间，单支树就是一个极端情况。



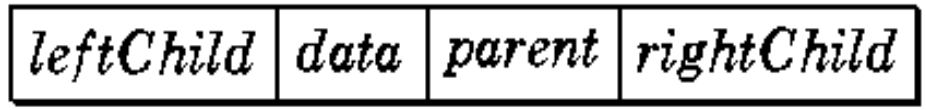
单支树

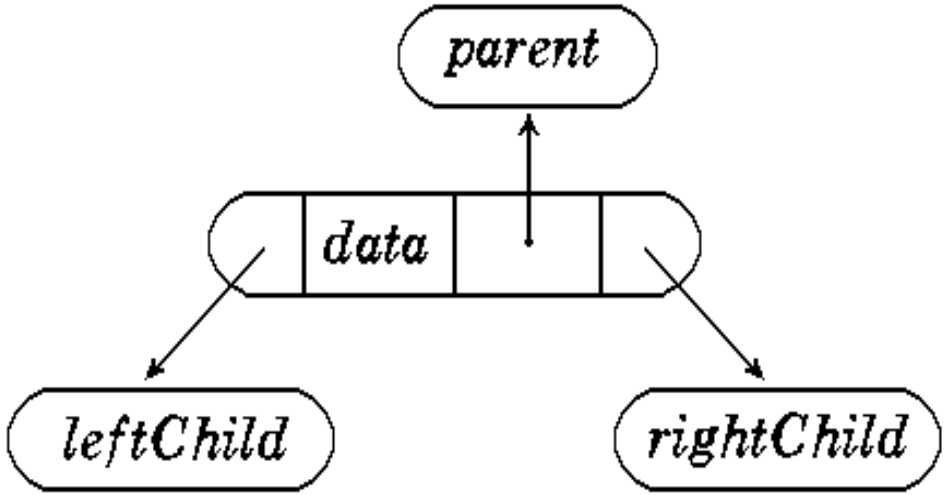
链表表示



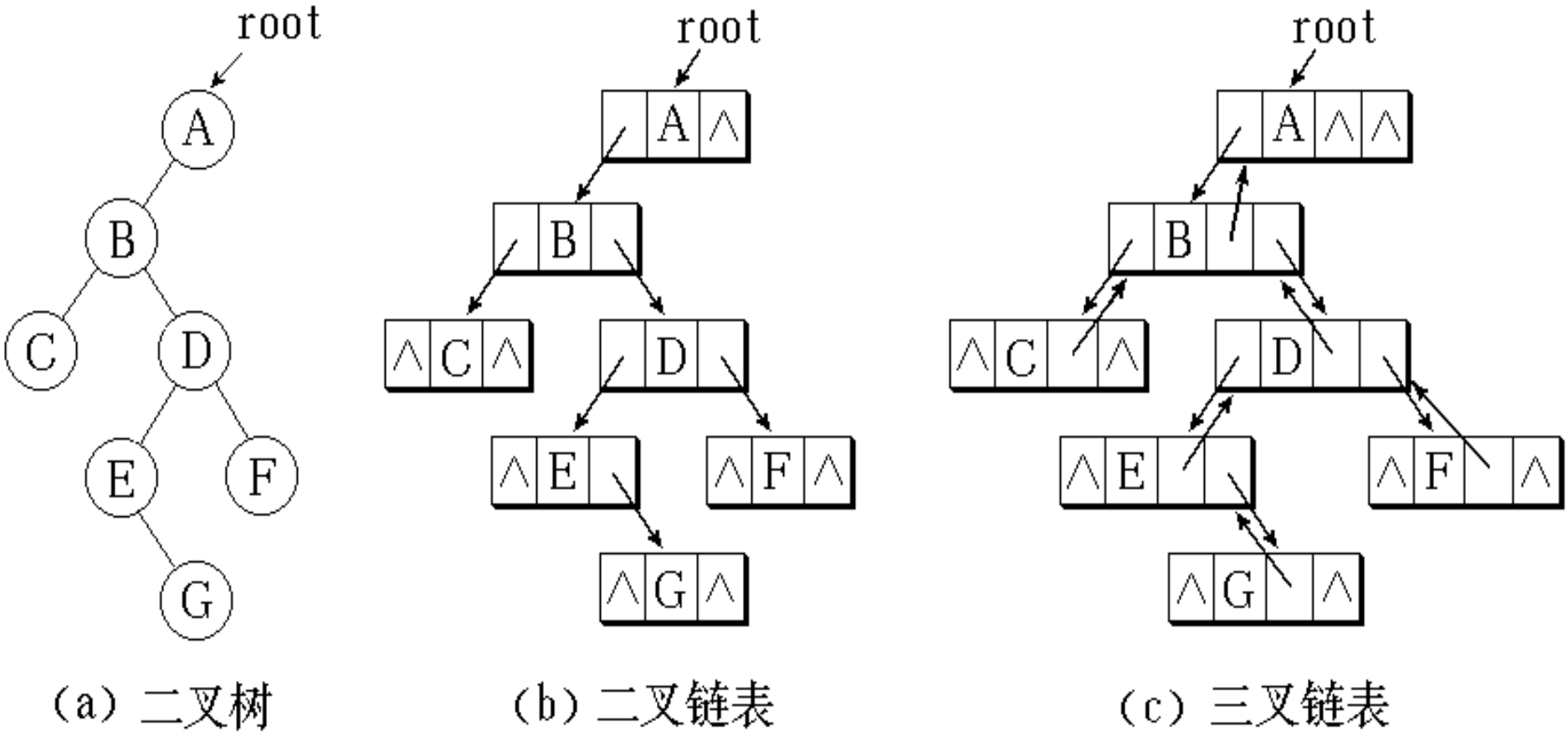


(a) 二叉链表

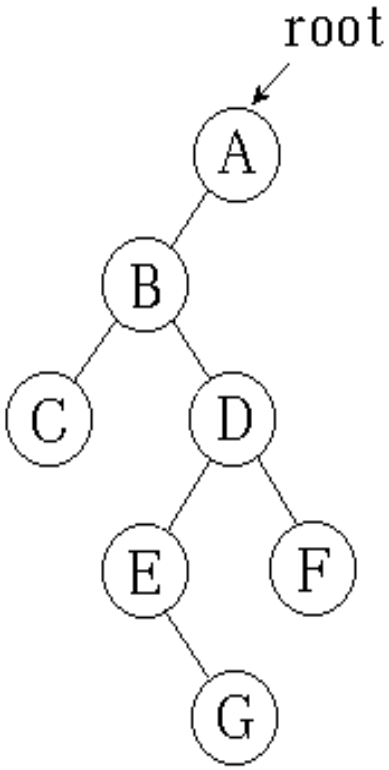




(b) 三叉链表



二叉树链表表示的示例



(a) 二叉树

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

二叉链表的静态结构

二叉树的类定义

```
template <class Type> class BinaryTree;

template <class Type> Class BinTreeNode {
friend class BinaryTree<Type>;
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; }
```

```
private:
    BinTreeNode<Type> *leftChild,  *rightChild;
    Type data;
};

template <class Type> class BinaryTree {
public:
    BinaryTree ( ) : root (NULL) { }
    BinaryTree ( Type value ) : RefValue (value),
            root (NULL) { }
    virtual ~BinaryTree ( ) { destroy ( root ); }
    virtual int IsEmpty ( )
        { return root == NULL ? 1 : 0; }
```

```
virtual BinTreeNode <Type> *Parent (
    BinTreeNode <Type> *current )
    { return root == NULL || root == current?
        NULL : Parent ( root, current ); }
virtual BinTreeNode <Type> *LeftChild (
        BinTreeNode <Type> *current )
    { return root != NULL ?
        current→leftChild : NULL; }
virtual BinTreeNode <Type> *RightChild (
        BinTreeNode <Type> *current )
    { return root != NULL ?
        current→rightChild : NULL; }
```

```
    virtual int Insert ( const Type & item);
    virtual int Find ( const Type &item ) const;
    const BinTreeNode <Type> *GetRoot ( ) const
      { return root; }
    friend istream &operator >> ( istream
          &in, BinaryTree<Type> &Tree )
    friend ostream &operator << ( ostream
          &out, BinaryTree <Type> &Tree )
private:
    BinTreeNode <Type> *root;
    Type RefValue;
```

```
    BinTreeNode <Type> *Parent (
          BinTreeNode <Type> *start,
          BinTreeNode <Type> *current );
    int Insert ( BinTreeNode<Type> * &current,
          const Type &item );
    void Traverse ( BinTreeNode<Type> *current,
          ostream &out ) const
    int Find ( BinTreeNode<Type> *current,
          const Type &item ) const
    void destroy(BinTreeNode<Type> *current);
}
```

二叉树部分成员函数的实现

```
template<class Type> void
BinaryTree<Type>:: destroy
( BinTreeNode<Type> *current ) {
    if ( current != NULL ) {
        destroy ( current→leftChild );
        destroy ( current→rightChild );
        delete current;
    }
}
```

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

```
template <class Type> void
BinaryTree<Type> :: Traverse (
BinTreeNode <Type> *current, ostream &out )
const {
    if ( current != NULL ) {
        out << current→data << ' ';
        Traverse ( current→leftChild, out );
        Traverse ( current→rightChild, out );
    }
}
```

```
template <class Type> istream & operator >>
 ( istream & in, BinaryTree <Type> &Tree ) {
     Type item;
     cout << “构造二叉树 : \n ";
     cout << “输入数据 ( 用 " << Tree.RefValue
          << “结束 ): ";
     in >> item;
     while ( item != Tree.RefValue ) {
        Tree.Insert ( item );
        cout << “输入数据 ( 用 " << Tree.RefValue
             << “结束 ): ";
        in >> item;
    }
    return in;
}
```

```
template <class Type> 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;
}
```

二叉树遍历
(Binary Tree Traversal)

所谓树的遍历，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**

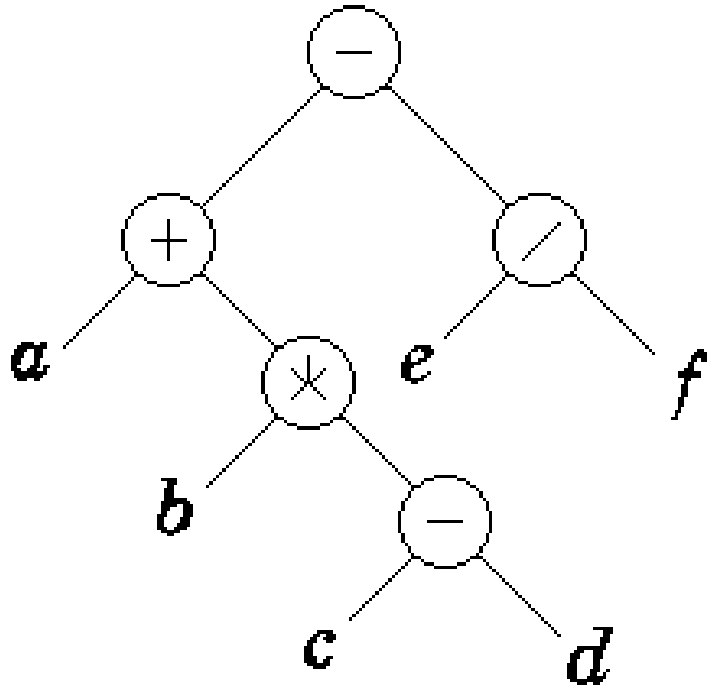
中序遍历 (Inorder Traversal)

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

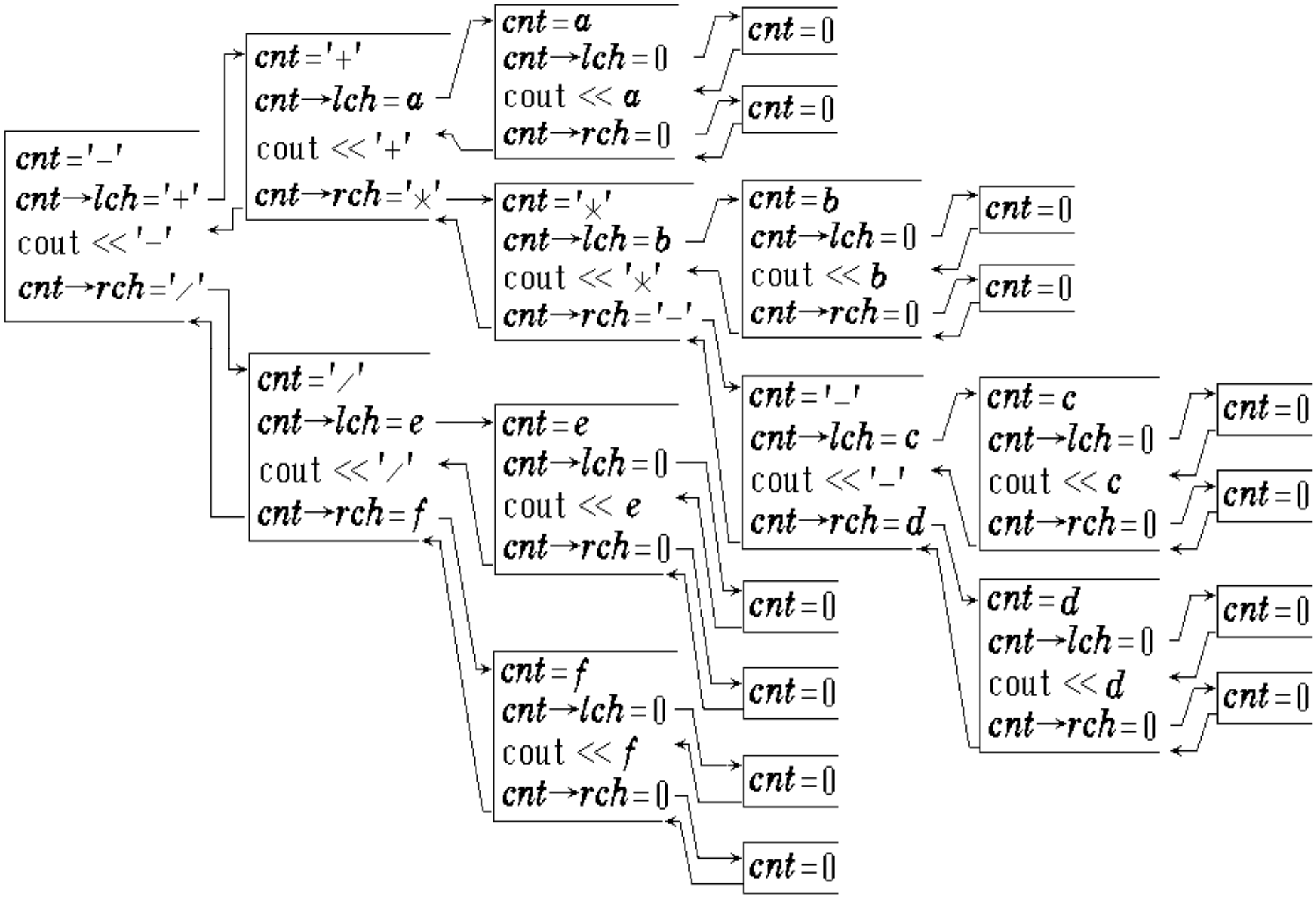
a + b * c - d - e / f



表达式语法树

二叉树递归的中序遍历算法

```
template <class Type>
void BinaryTree <Type>::InOrder ( ) {
    InOrder ( root );
}

template <class Type> void BinaryTree <Type>::
    InOrder ( BinTreeNode <Type> *current ) {
        if ( current != NULL ) {
            InOrder ( current→leftChild );
            cout << current→data;
            InOrder ( current→rightChild );
        }
}
```

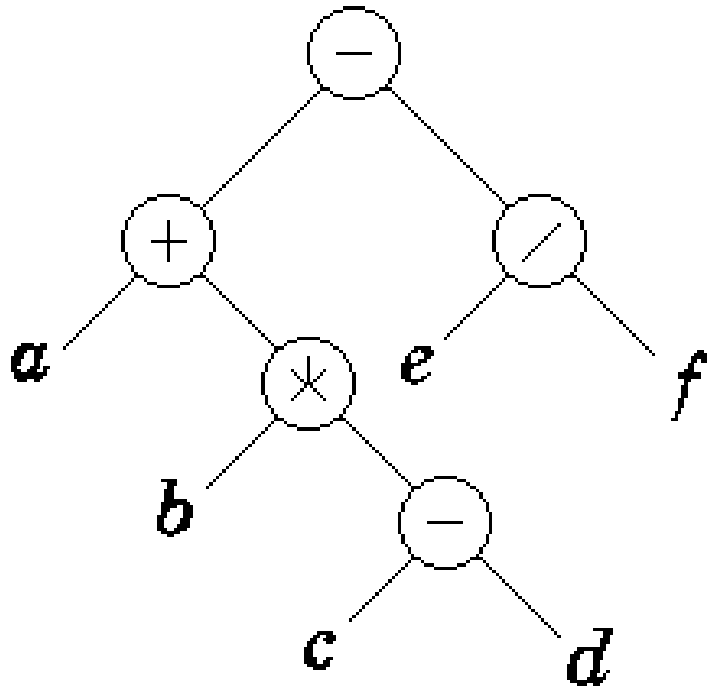


中序遍历二叉树的递归过程图解

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*

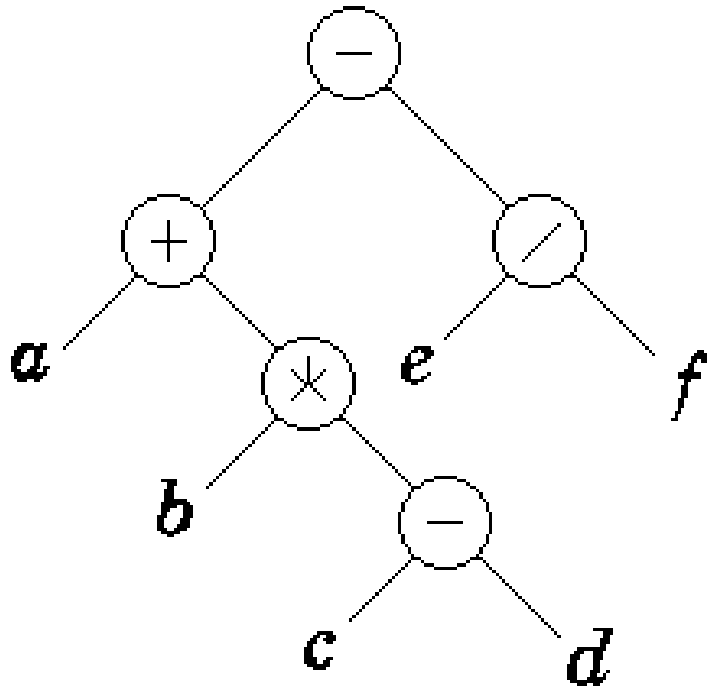
二叉树递归的前序遍历算法

```
template <class Type>
void BinaryTree <Type>::PreOrder ( ) {
    PreOrder ( root );
}

template <class Type> void BinaryTree<Type>::
    PreOrder ( BinTreeNode <Type> *current ) {
    if ( current != NULL ) {
        cout << current→data;
        PreOrder ( current→leftChild );
        PreOrder ( current→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 ( ) {
    PostOrder ( root );
}

template <class Type> void BinaryTree<Type>::
PostOrder ( BinTreeNode <Type> *current ) {
    if ( current != NULL ) {
        PostOrder ( current→leftChild );
        PostOrder ( current→rightChild );
        cout << current→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及二叉树的高度

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

```
template <class Type>
int BinaryTree<Type> :: Depth (
const BinTreeNode <Type> *t ) const {
    if ( t == NULL ) return −1;
    else return 1 + Max ( Depth ( t→leftChild ),
                          Depth ( t→rightChild ) );
}
```

传统方法：利用栈的非递归遍历算法

二叉树结点定义

```
typedef struct BitreeNode {
    TreeDataType  data;
    struct BitreeNode * leftChild,  * rightChild;
} BitreeNode,  * Bitree;
```

传统方法：利用栈的前序遍历非递归算法

```
void PreOrderTraverse ( Bitree T ) {
    StackType S;    BitreeNode * p;
    S.makeEmpty( );  S.Push(null);   p = T;  //初始化
    while ( p ) {
        printf ( p→data );
        if ( p→rightChild ) S.Push ( p→rightChild );
              //预留右子树指针在栈中
        if  ( p→leftChild ) p = p→leftChild;//进左子树
        else { p = S.getTop( );  S.Pop( ); }   //进右子树
    }
}
```

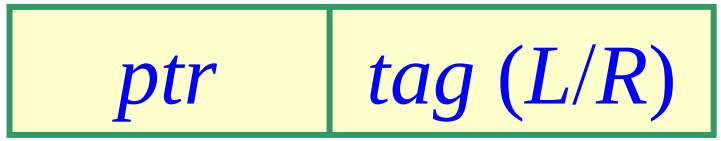
传统方法：利用栈的中序遍历非递归算法

```
void InOrderTraverse ( Bitree T ) {
    StackType S;  BitreeNode * p;
    S.makeEmpty( );   p = T;                    //初始化
    do{
        while ( p ) { S.Push(p);   p = p→leftChild; }
        if ( !S.IsEmpty( ) ) {                  //栈非空
            p = S.getTop( );  S.Pop( );         //退栈
            printf ( p→data);                   //访问根结点
            p = p→rightChild;                   //向右链走
        }
    } while ( p || !S.IsEmpty( ) );
}
```

传统方法：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

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




```
void PostOrderTraverse ( Bitree T ) {
//后序遍历根为 T 的二叉树，存储结构为二叉链
//表，S 是存储所经过二叉树结点的工作栈。其
//中，tag 是结点标记。 当 tag = L 时，表示刚才
//是在左子树中，从左子树退出时，还要去遍历
```

```
//右子树。当 tag = R 时,表示刚才是在右子树//中,在从右子树中退出时,去访问位于栈顶的
//结点。
    StackType S;   BitreeNode * p;   StackNode w;
    S.makeEmpty( );    p = T;                    //初始化
    do {
        while ( p ) {
            w.ptr = p;   w.tag = L;   S.Push(w);
            p = p→leftChild;
        }
        //遍历左子树并把经过结点加左标记进栈
        int continue = 1;          //继续循环标记
```

```
while ( continue && !S.IsEmpty( ) ) {
    w = S.getTop( );   S.Pop( );
    p = w.ptr;
    switch (w.tag) {   //判断栈顶的tag标记
      case L :  w.tag = R;   S.Push(w);
                continue = 0;   //修改栈顶tag
                p = p→rightChild;
                break;
      case R :  printf (p→data);  break;
    }
  }
} while ( p || !S.IsEmpty( ) );
}
```

二叉树遍历的游标类 (Tree Iterator)

二叉树的游标抽象基类

```
template <class Type> class TreeIterator {
public:
    TreeIterator ( const BinaryTree <Type> & BT )
            : T (BT), current (NULL) { }
    virtual ~TreeIterator ( ) { }
    virtual void First ( ) = 0;
    virtual void operator ++ ( ) = 0;
    int operator +( ) const
            {return current != NULL; }
    const Type & operator ( ) ( ) const;
```

```
protected:
    const BinaryTree <Type> & T;
    const BinTreeNode <Type> * current;
private:
    TreeIterator ( const TreeIterator & ) { }
    TreeIterator & operator =
                    ( const TreeIterator & ) const ;
};
```

```
template <class Type>
const Type &TreeIterator<Type> ::
operator ( ) const {
    if ( current == NULL )
      { cout << "非法访问" << endl;  exit (1); }
    return current→GetData ( );
}
```

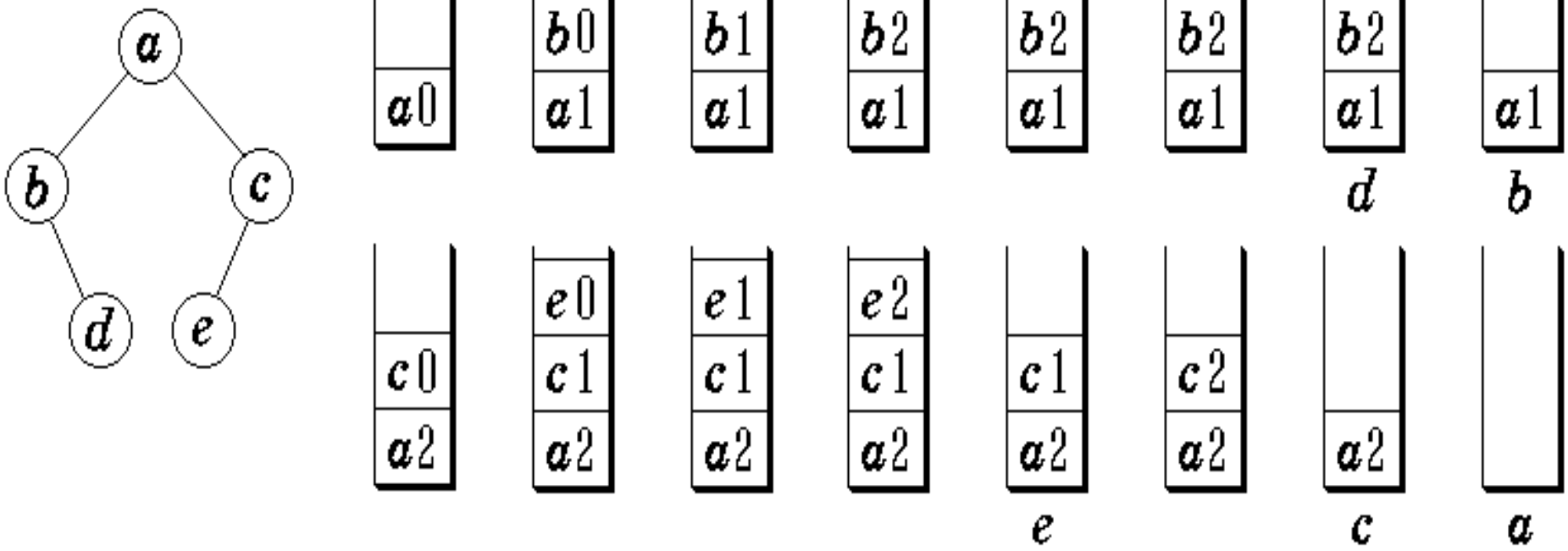
后序游标类

- 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。



后序遍历游标类的定义

```
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 ( const 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
}
```

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) {
        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 ++ ( );
}
```

```
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

中序遍历与后序遍历走过的路线一样，只是在从左子树退出后立即访问根结点，再遍历右子树。中序遍历也要使用一个递归工作栈 ***st***。

中序游标类的定义

```
template <class Type> class InOrder :
    public PostOrder <Type> {
public:
    InOrder ( BinaryTree <Type> & BT ) :
    PostOrder <Type> ( BT ) { }
```

```
    void First ( );
    void operator ++ ( );
};
```

中序游标类operator ++( )操作的实现

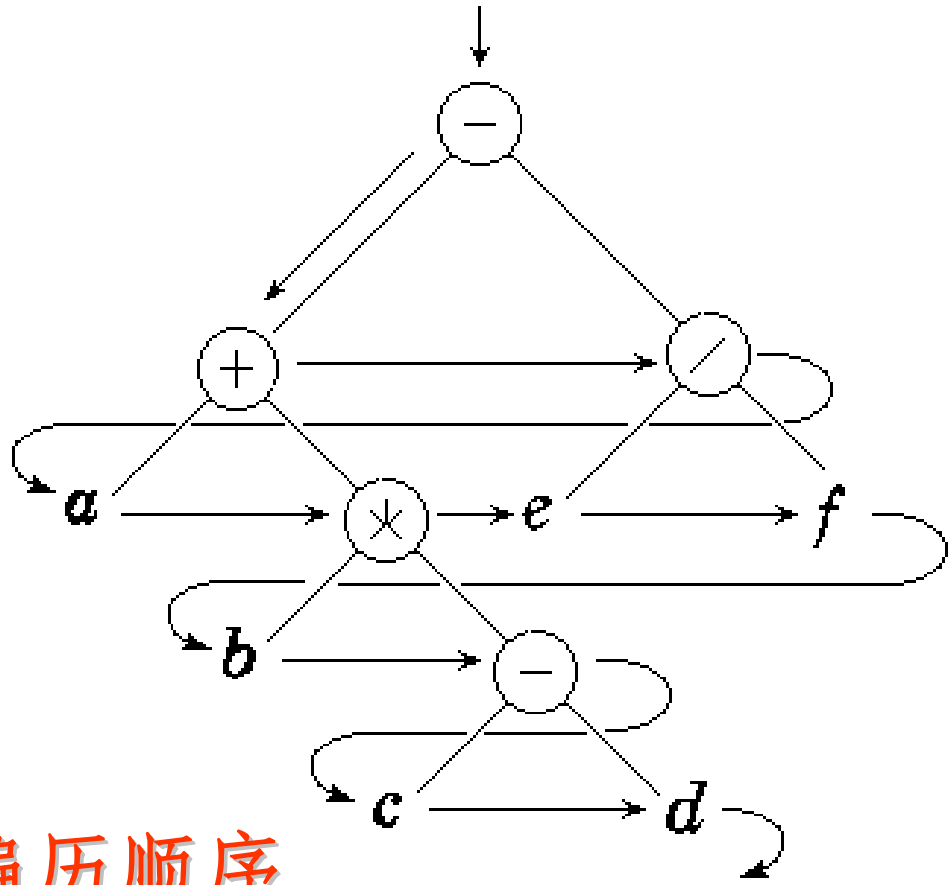
```
template <class Type> void InOrder <Type>::
operator ++ ( ) {
    if ( st.IsEmpty ( ) ) {
      if ( current == NULL )
        { cout << "已经遍历完成" << endl;  exit (1); }
      current = NULL;  return;
    }
    StkNode <Type> Cnode;
```

```
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，成为当前结点
    current = Cnode.Node;
    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> (    //左子女进栈
                Cnode.Node→GetLeft ( ) ) );
    }
}
```

层次序游标类

从根开始逐层访问，用 **FIFO** 队列实现。



遍历顺序

层次序游标类的定义

```
template <class Type> class LevelOrder :
    public TreeIterator <Type> {
public:
    LevelOrder ( const BinaryTree <Type> & BT );
    ~LevelOrder ( ) { }
    void First ( );
    void operator ++ ( );
protected:
    Queue < const BinTreeNode <Type> * > qu;
};
```

层次序游标类的operator ++ ( )操作

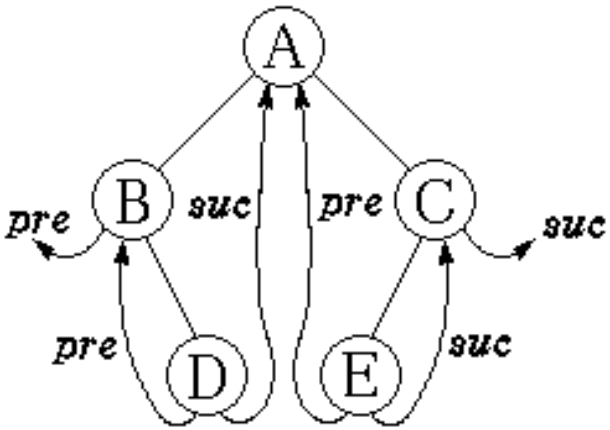
```
template <class Type> LevelOrder <Type>::
    LevelOrder ( const BinaryTree <Type> & BT ) :
        TreeIterator <Type>  ( BT )
            { qu.EnQueue ( T.GetRoot ( ) ); }

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 ( ) );
    operator ++ ( );
}
```

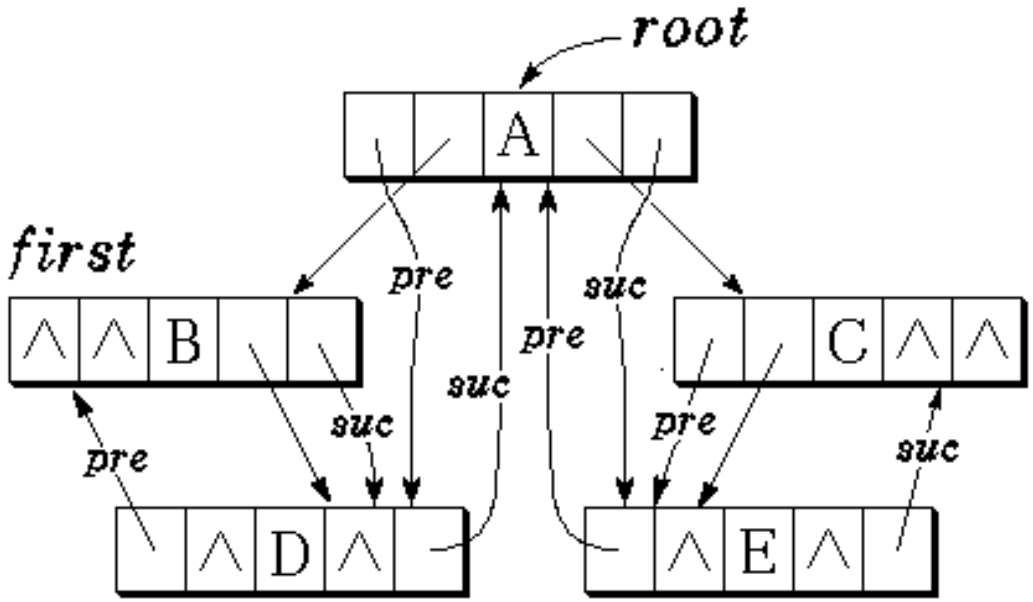
```
template <class Type>
void LevelOrder<Type> :: operator ++ ( ) {
    if ( qu.IsEmpty ( ) ) {
       if ( current == NULL )
         { cout << "已经遍历完成" << endl;  exit (1);}
       current = NULL;  return;
    }
    current = qu.DeQueue ( );                          //退队
    if ( current→GetLeft ( ) != 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
}
```

线索化二叉树 (Threaded Binary Tree)

线索 (Thread)

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

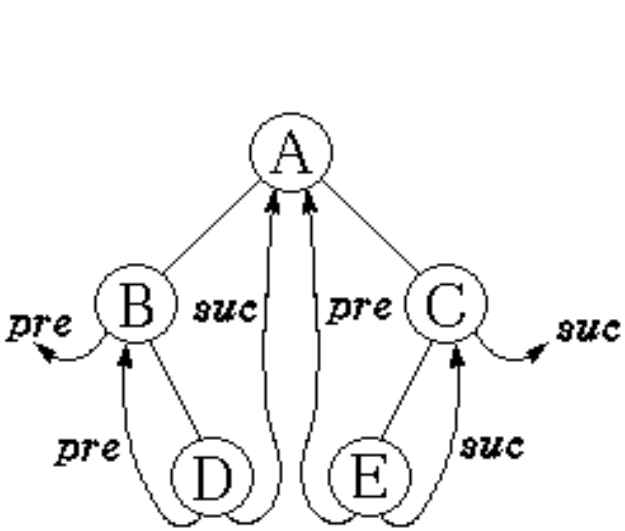


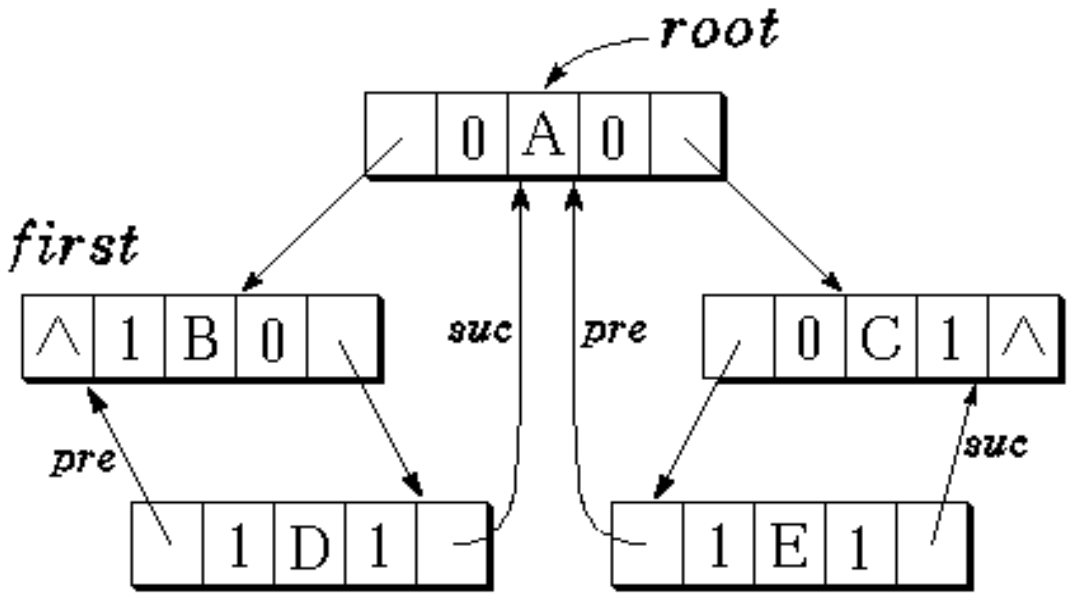


增加 ***Pred*** 指针和 ***Succ*** 指针的二叉树

线索化二叉树及其二叉链表表示

leftChild	*leftThread*	*data*	*rightThread*	*rightChild*





***LeftThread* = 0,** ***LeftChild*为左子女指针**

***LeftThread* = 1,** ***LeftChild*为前驱线索**

***RightThread* = 0,** ***RightChild*为右子女指针**

***RightThread* = 1,** ***RighrightChild*为后继指针**

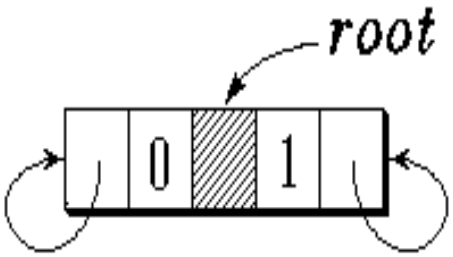
中序线索化二叉树的类定义

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

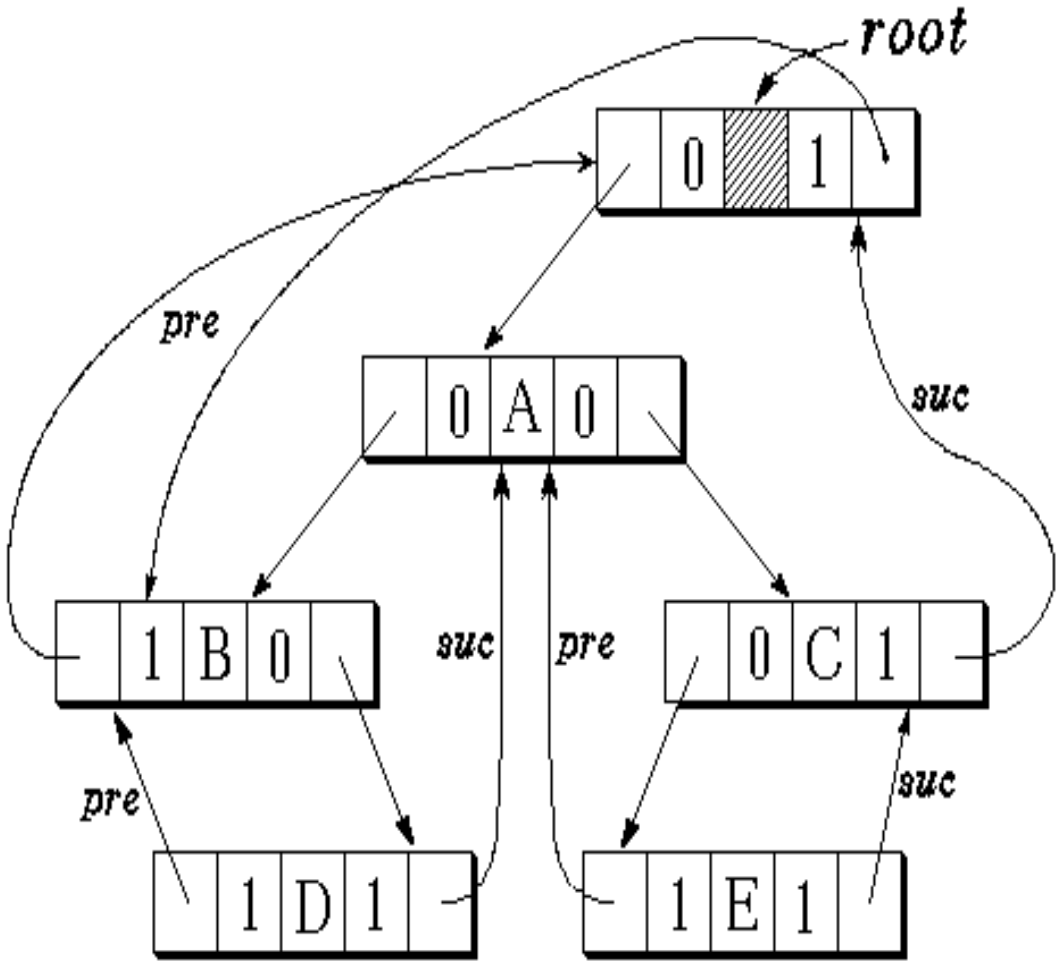
```
template <class Type> class ThreadTree {
        friend class ThreadInorderIterator;
                public:
    ..........
private:
    ThreadNode<Type> *root;
};

template <class Type>
class ThreadInorderIterator {
public:
    ThreadInorderIterator ( ThreadTree<Type>
        &Tree ) : T (Tree) { current = T.root; }
```

```
    ThreadNode<Type> * First ( );
    ThreadNode<Type> * Last ( );
    ThreadNode<Type> * Next ( );
    ThreadNode<Type> * Prior ( );
private:
    ThreadTree<Type> &T;
    ThreadNode<Type> *current;
}
```

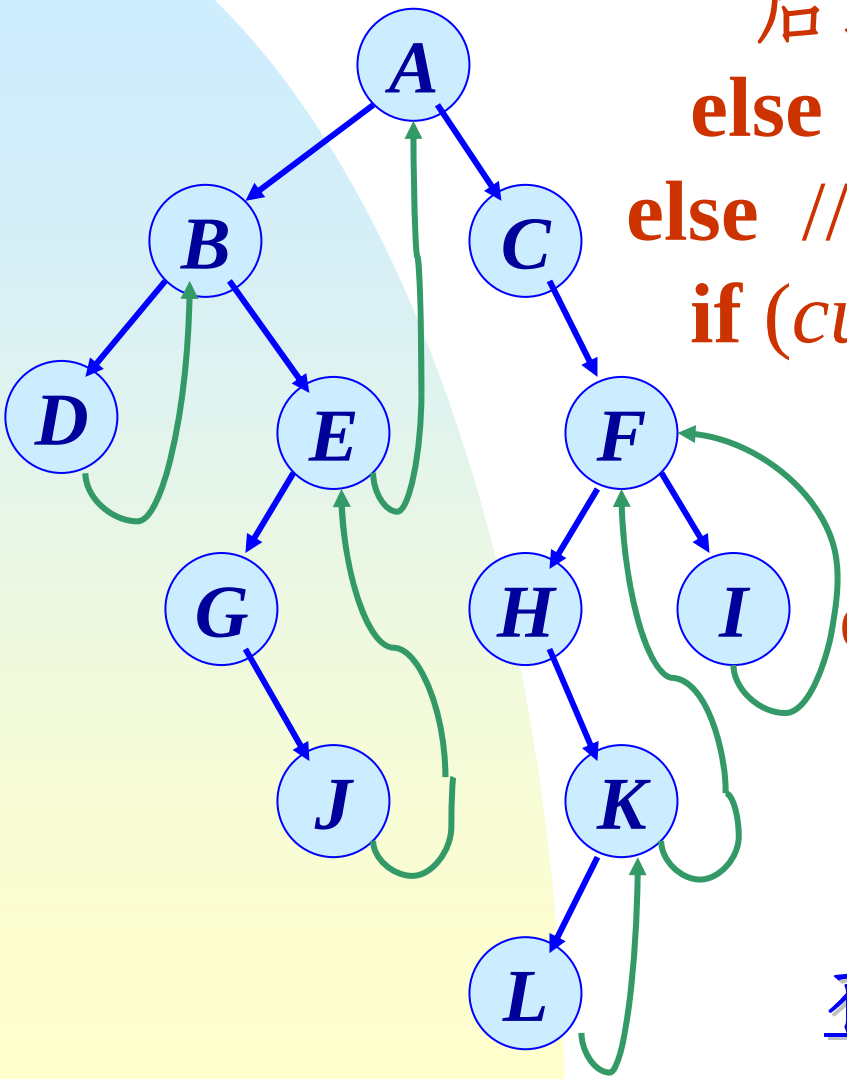
带表头结点的中序穿线链表



(a) 空二叉链表



(b) 非空二叉链表



if (*current*→*rightThread* ==1)

if (*current*→*rightChild* !=*T.root*)

后继为 *current*→*rightChild*

else 无后继

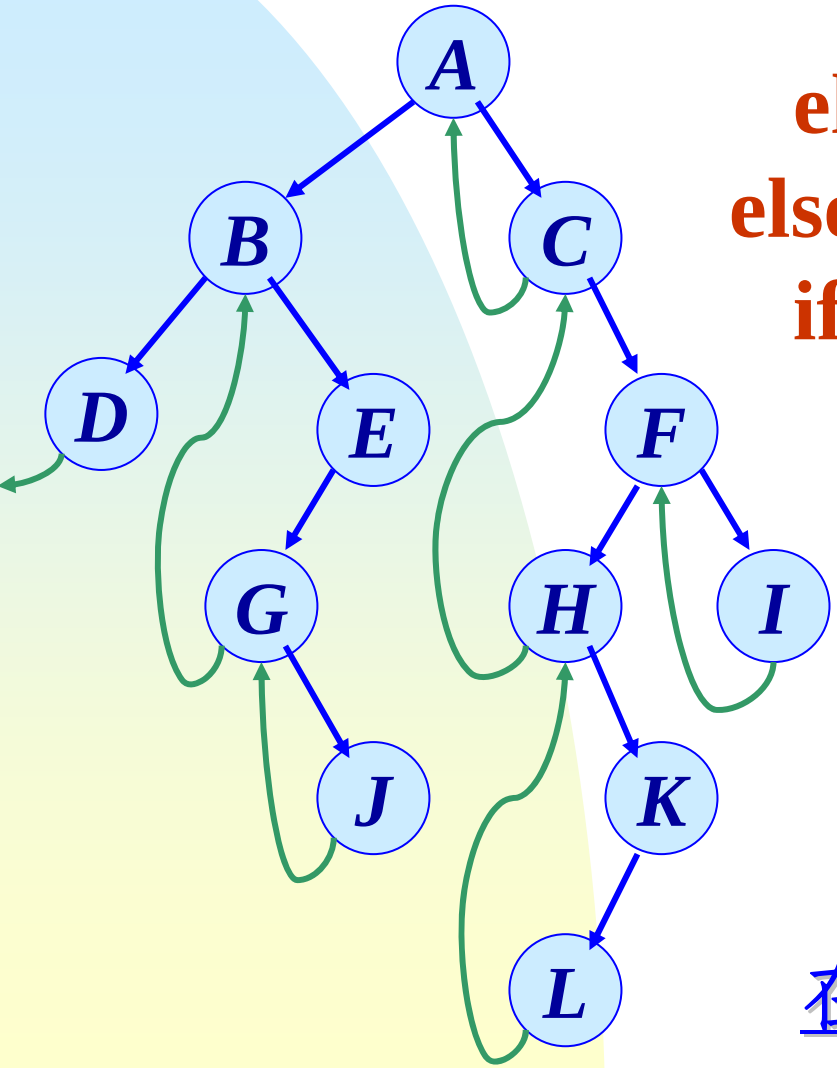
else //*current*→*rightThread* !=1

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后继




```
if (current→leftThread ==1)
  if (current→leftChild !=T.root)
    前驱为current→leftChild
  else 无前驱
else //current→leftThread ==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
```

寻找当前结点
在中序下的前驱

在中序线索化二叉树中的部分成员函数的实现

```
template <class Type> ThreadNode<Type> *
ThreadInorderIterator<Type> :: First ( ) {
    while ( current→leftThread == 0 )
        current = current→leftChild;
    return current;
}

template <class Type> ThreadNode<Type> *
ThreadInorderIterator<Type> :: Next ( ) {
    ThreadNode<Type> *p = current→rightChild;
    if ( current→rightThread == 0 )
```

```
        while ( p→leftThread == 0 ) p = p→leftChild;
        current = p;
        if ( current == T.root ) return NULL;
        else return current;
}

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

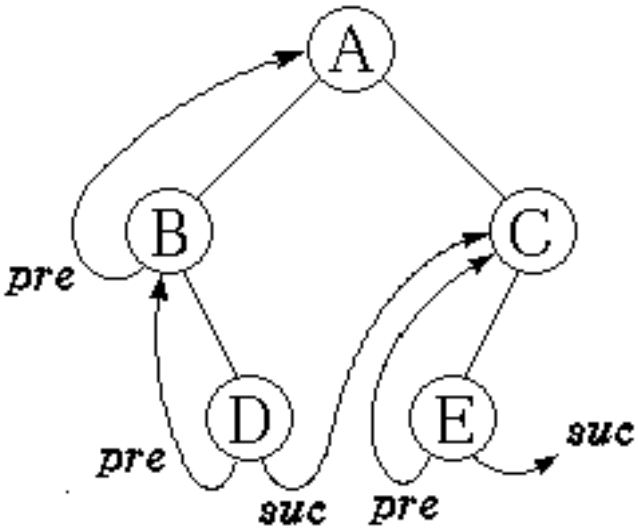
通过中序遍历建立中序线索化二叉树

```
template <class Type>
void ThreadTree<Type> ::
InThread ( ThreadNode<Type> *current,
                        ThreadNode<Type> *pre ) {
   if ( current != NULL ) {
      InThread ( current→leftChild, pre );
      if ( current→leftChild == NULL ) {
         current→leftChild = pre;
         current→leftThread = 1;
      }
```

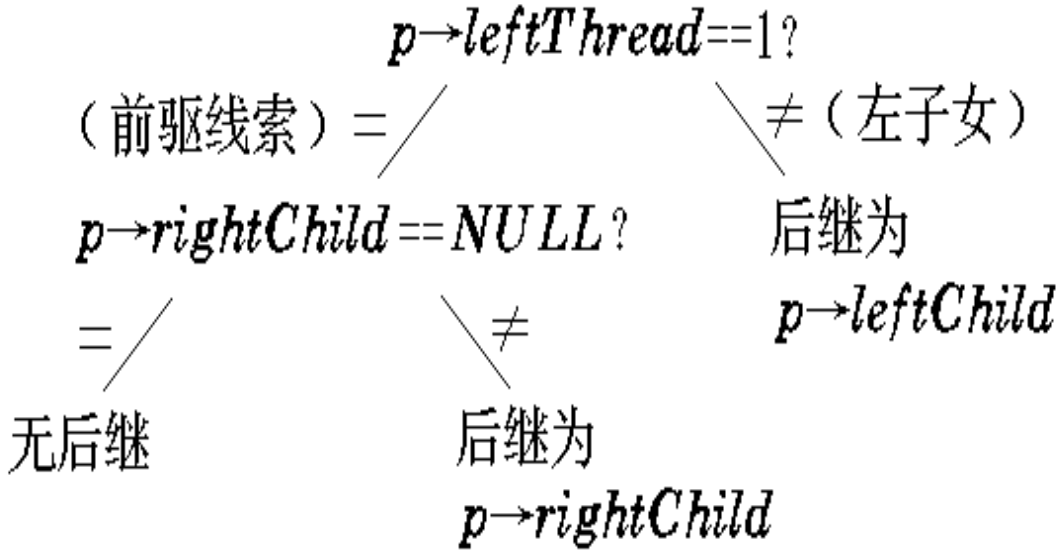
```
        if ( pre→rightChild == NULL ) {
            pre→rightChild = current;
            pre→rightThread = 1;
        }
        pre = current;
        InThread ( current→rightChild, pre );
    }
}

template <class Type> void
ThreadTree<Type>::CreateInThread ( ) {
    ThreadNode<Type> *pre;
    root = new ThreadNode<Type>;
```

```
    root→leftThread = 1;  root→rightThread = 0;
    if ( this == NULL ) {
        root→leftChild = root;
        root→rightChild = root;
    }
    else {
        current = root→leftChild = this;
        root→leftThread = 0;
        pre = root;
        InThread ( current, pre );
        pre→rightChild = root;   pre→rightThread = 1;
    }
}
```

前序线索化二叉树



（a）前序穿线二叉树



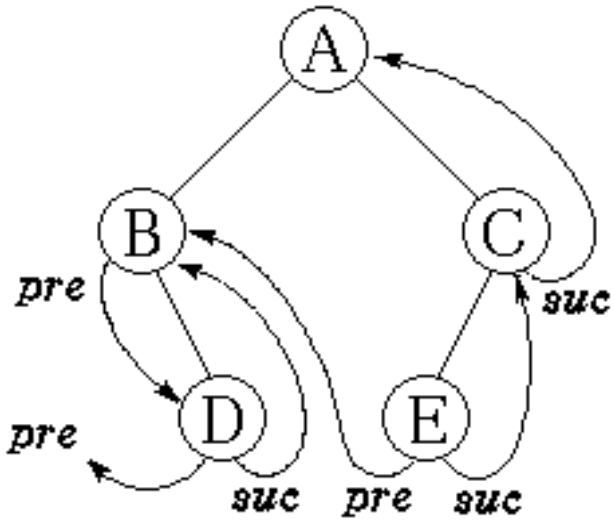
（a）求结点 *p* 的后继

前序序列

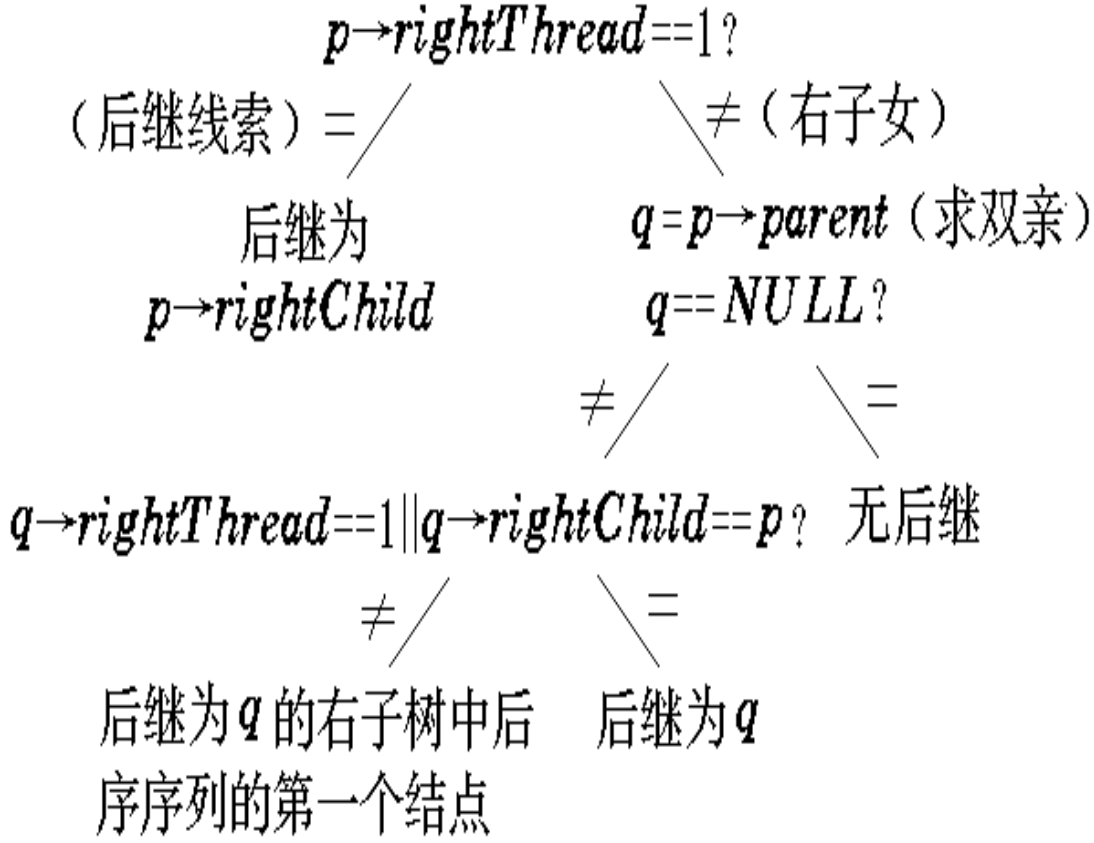
A B D C E

在前序线索化二叉树中
寻找当前结点的后继

后序线索化二叉树



（b）后序穿线二叉树



（a）求结点 p 的后继

后序序列

D B E C A

在后序线索化二叉树中
寻找当前结点的后继

堆 (Heap)

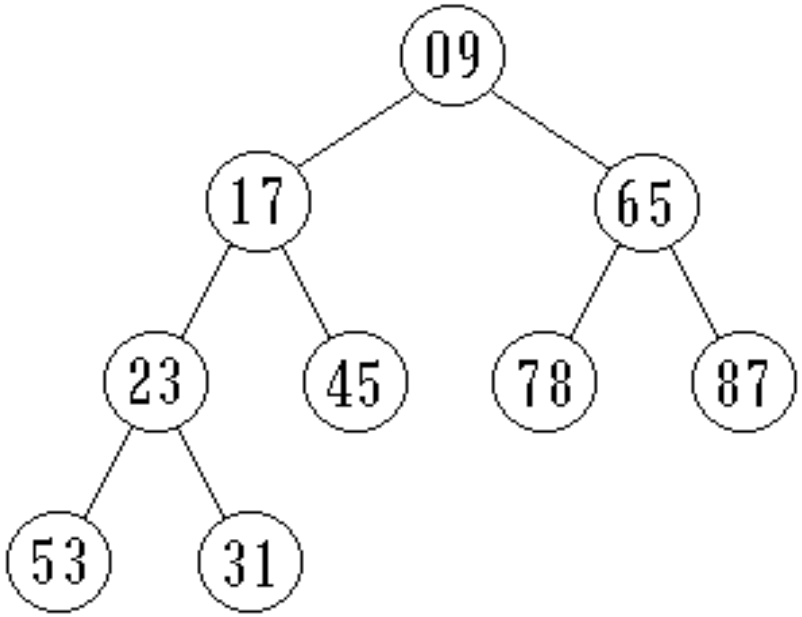
优先级队列

每次出队列的是优先权最高的元素

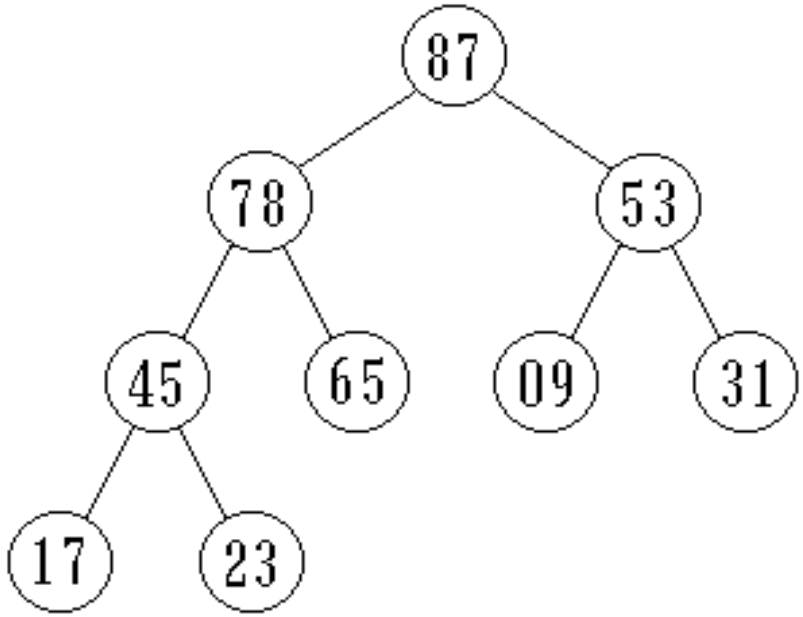
```
template <class Type> class MinPQ {
public:
    Virtual void Insert ( const Type & ) = 0;
    Virtual Type *RemoveMin ( Type & ) = 0;
}
```

最小优先级队列类的定义

堆的定义



(a) 最小堆



(b) 最大堆

完全二叉树的数组表示

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

完全二叉树的数组表示

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

最小堆的类定义

```
template <class Type> class MinHeap :
    public  MinPQ <Type> {
public:
    MinHeap ( int maxSize );
    MinHeap ( Type arr[ ], int n );
    ~MinHeap ( ) { delete [ ] heap; }
    const MinHeap<Type> & operator =
        ( const MinHeap &R );
    int Insert ( const Type &x );
    int RemoveMin ( Type &x );
```

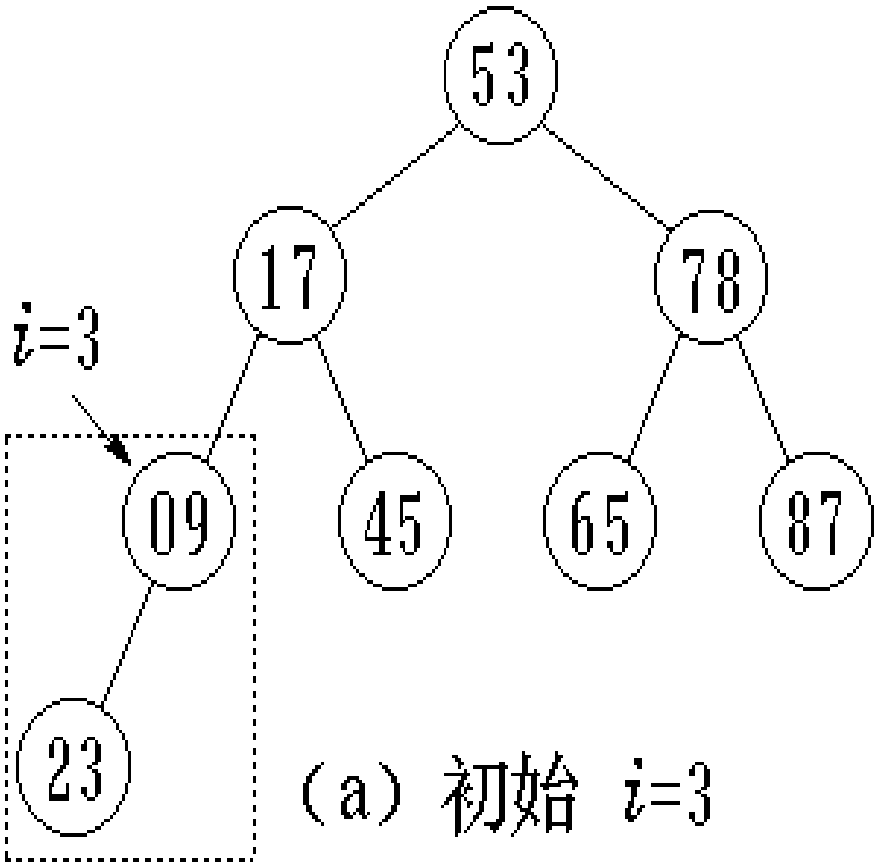
```
    int IsEmpty ( ) const
            { return CurrentSize == 0; }
    int IsFull ( ) const
            { return CurrentSize == MaxHeapSize; }
    void MakeEmpty ( ) { CurrentSize = 0; }
private:
    enum { DefaultSize = 10 };
    Type *heap;
    int CurrentSize;
    int MaxHeapSize;
    void FilterDown ( int i, int m );
    void FilterUp ( int i );
}
```

堆的建立

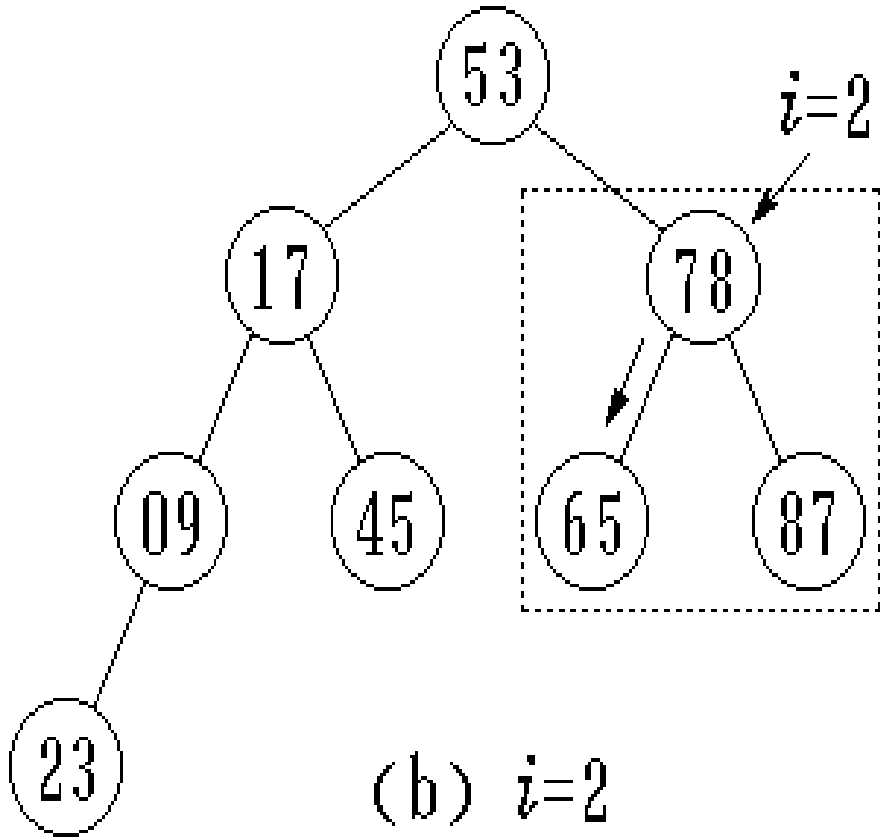
```
template <class Type> MinHeap <Type> ::
MinHeap ( int maxSize ) {
//根据给定大小maxSize,建立堆对象
    MaxHeapSize = DefaultSize < maxSize ?
            maxSize : DefaultSize;          //确定堆大小
    heap = new Type [MaxHeapSize]; //创建堆空间
    CurrentSize = 0;                        //初始化
}

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];
    heap = arr;                  //数组传送
    CurrentSize = n;             //当前堆大小
    int currentPos = (CurrentSize-2)/2;   //最后非叶
    while ( currentPos >= 0 ) {
    //从下到上逐步扩大,形成堆
        FilterDown ( currentPos, CurrentSize-1 );
        //从currentPos开始,到CurrentSize为止, 调整
        currentPos--;
    }
}
```

将一组用数组存放的任意数据调整成堆

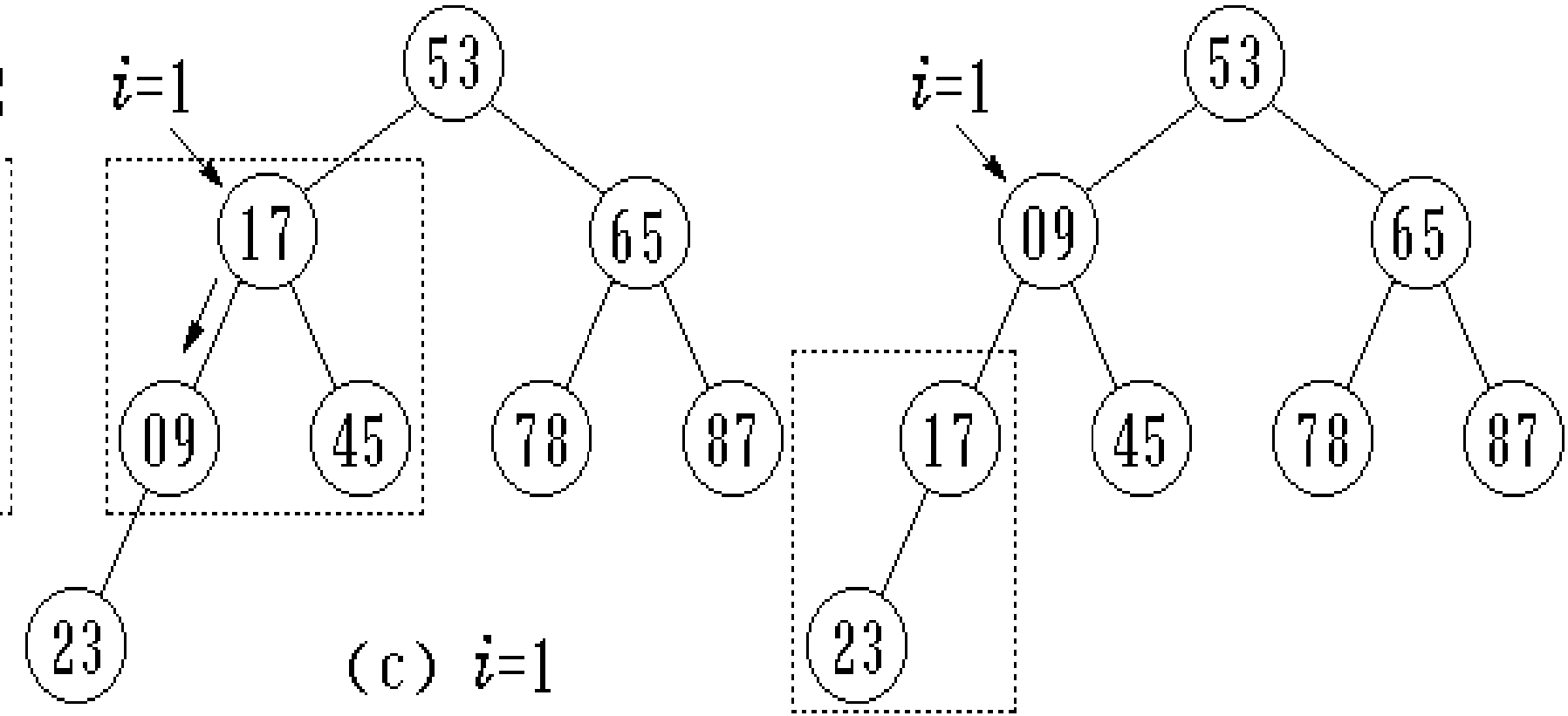


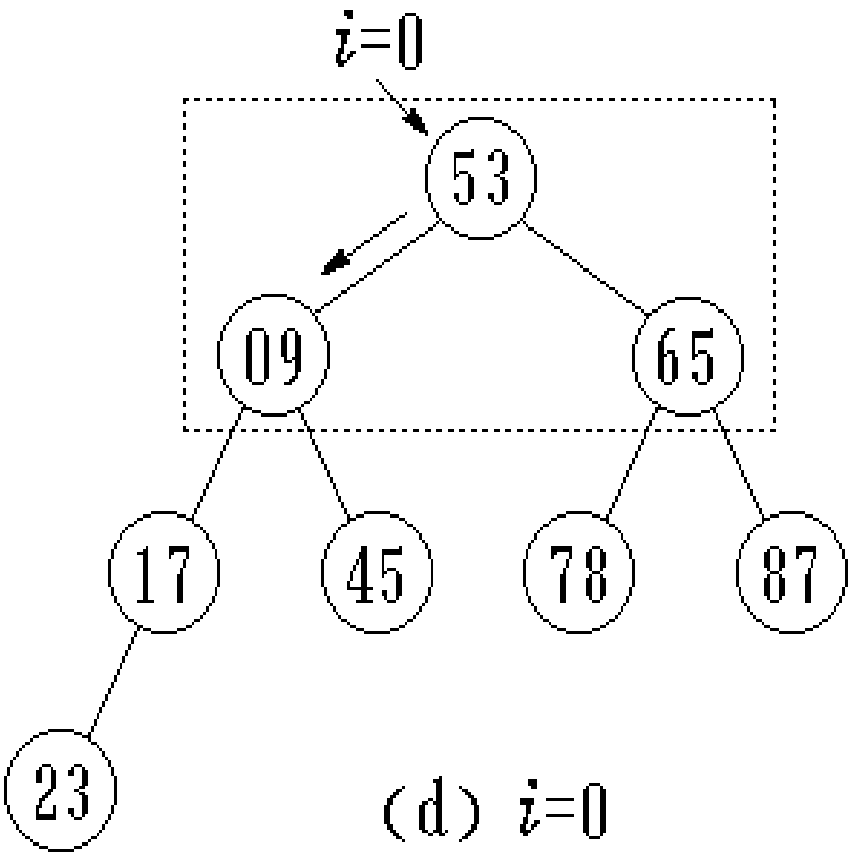
(a) 初始 $i=3$



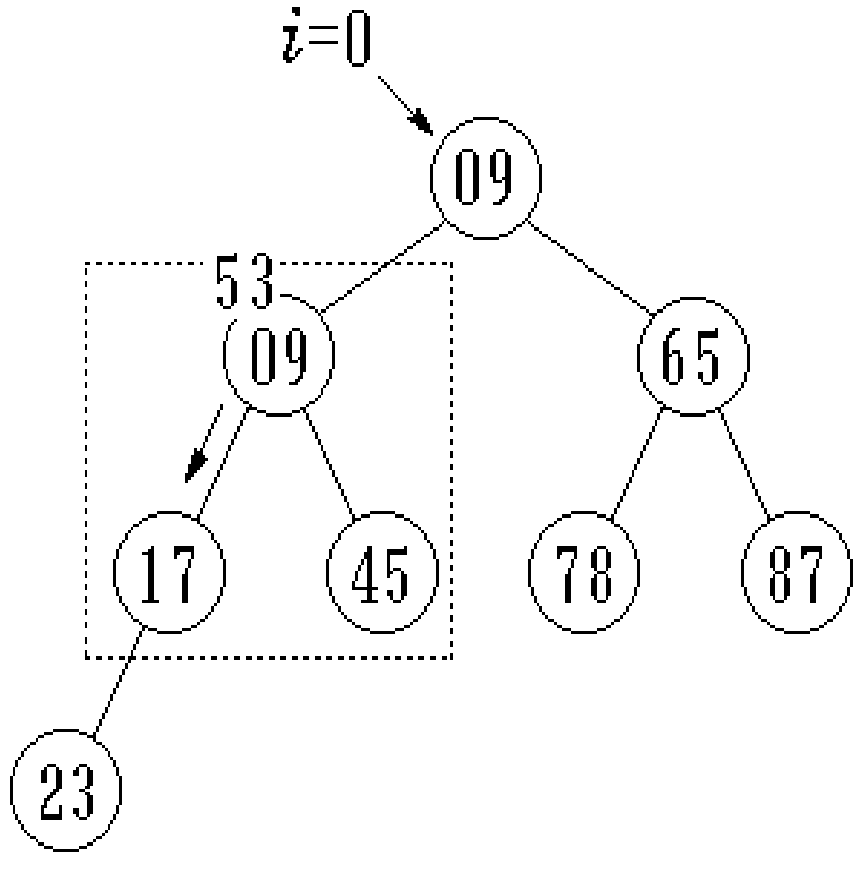
(b) $i=2$

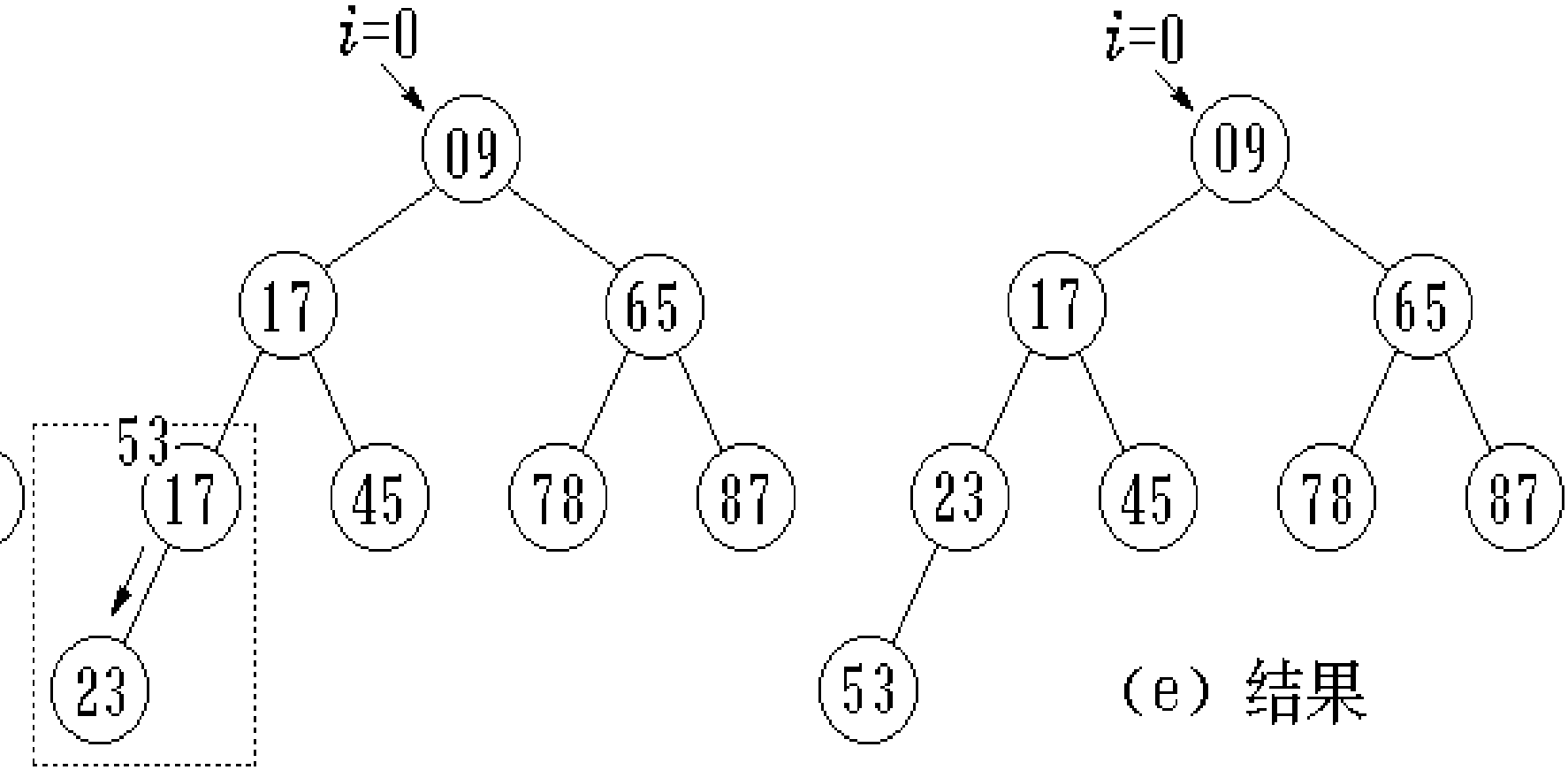
自下向上逐步调整为最小堆


i=1
53
17
65
09
45
78
87
23
(c) i=1
i=1
53
09
65
17
45
78
87
23



(d) $i=0$





(e) 结果

最小堆的向下调整算法

```
template <class Type> void MinHeap<Type> ::
FilterDown ( int start, int EndOfHeap ) {
    int i = start,   j = 2*i+1;        // j 是 i 的左子女
    Type temp = heap[i];
    while ( j <= EndOfHeap ) {
        if ( j < EndOfHeap && heap[j].key >
           heap[j+1].key )  j++;    //两子女中选小者
        if ( temp.key <= heap[j].key ) break;
        else { heap[i] = heap[j];  i = j;   j = 2*j+1; }
    }
    heap[i] = temp;
}
```

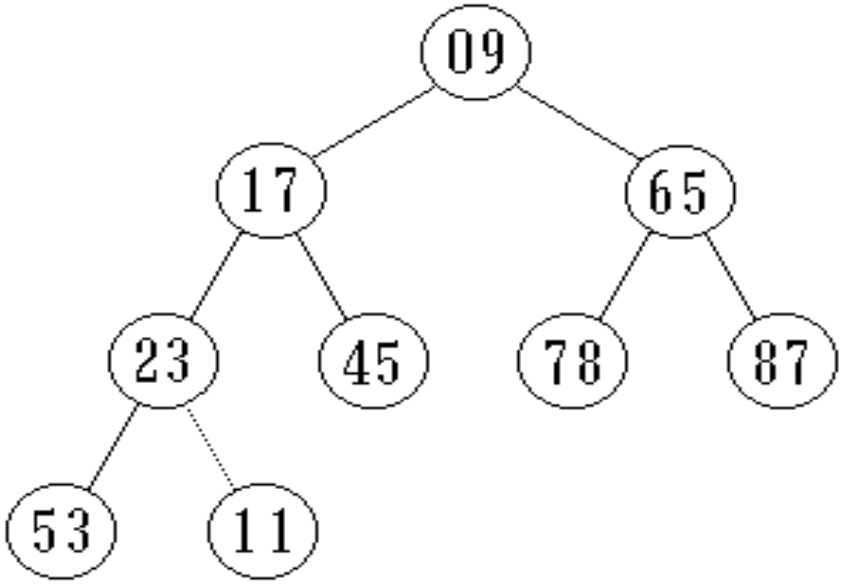
堆的插入

```
template <class Type> int MinHeap<Type> ::
Insert ( const Type &x ) {
//在堆中插入新元素 x
    if ( CurrentSize == MaxHeapSize )     //堆满
      { cout << "堆已满" << endl;  return 0; }
    heap[CurrentSize] = x;              //插在表尾
    FilterUp (CurrentSize);             //向上调整为堆
    CurrentSize++;                      //堆元素增一
    return 1;
}
```

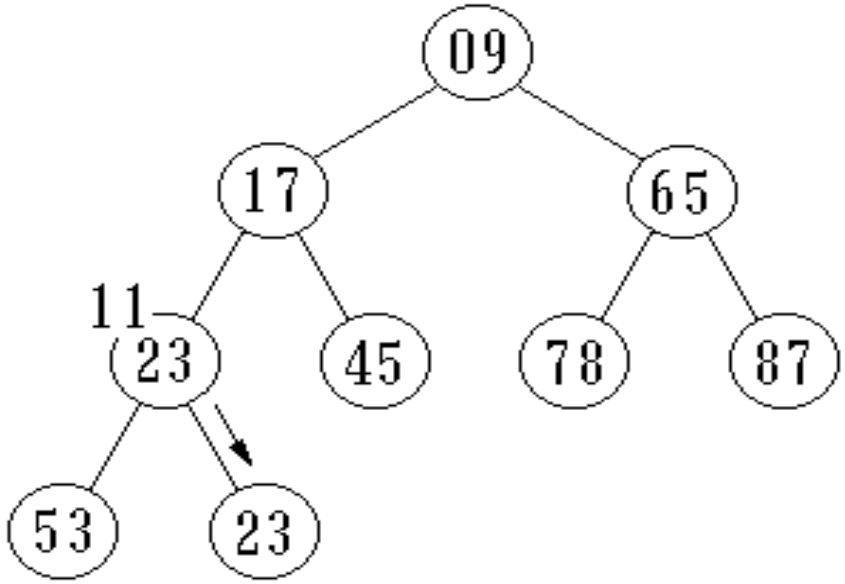
最小堆的向上调整算法

```
template <class Type> void MinHeap<Type> ::
FilterUp ( 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].key <= temp.key ) break;
        else {  heap[j] = heap[i];  j = i;  i = (i −1)/2; }
    }
    heap[j] = temp;
}
```

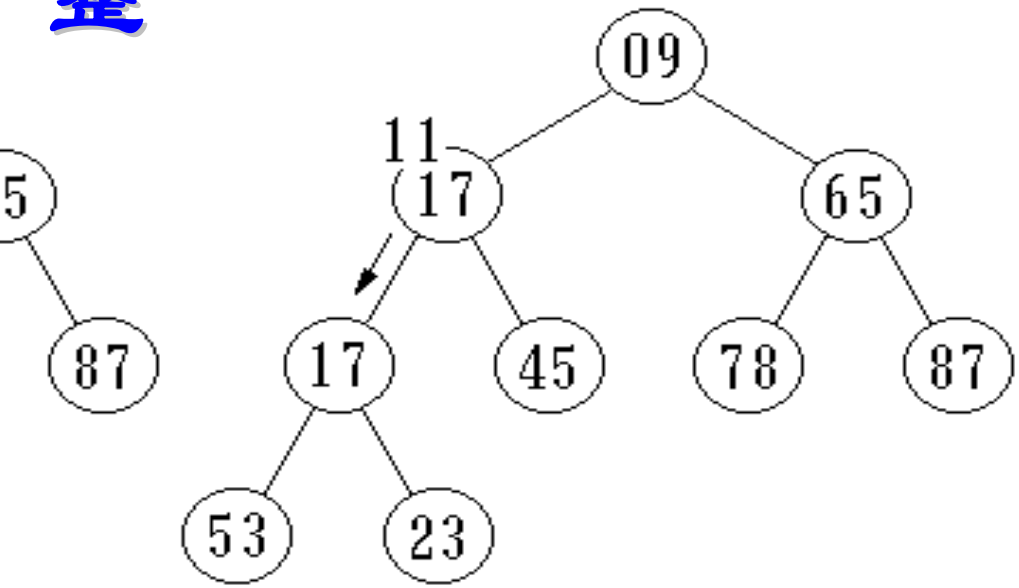
最小堆的向上调整



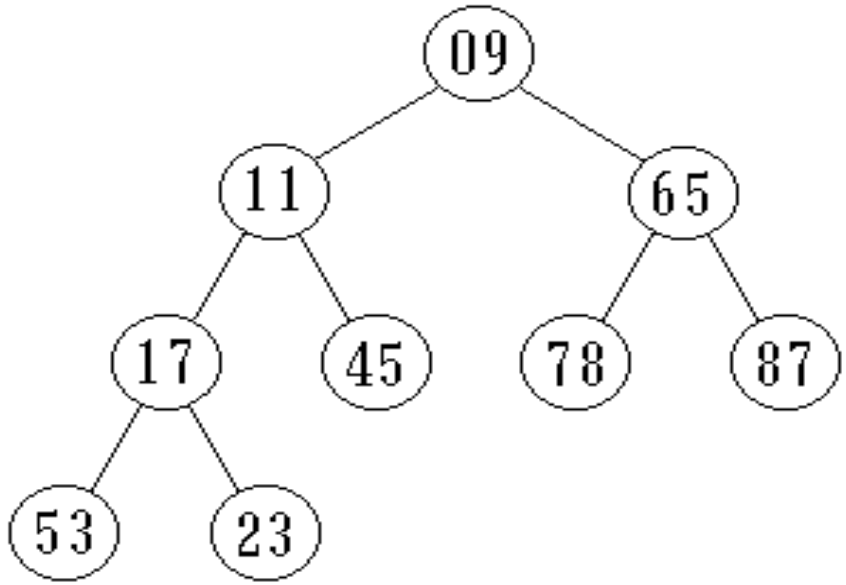
（a）初始 尾部加11



（b） 双亲关键码23下降



下降 （c）双亲关键码17下降



（d）11回填 调整完成

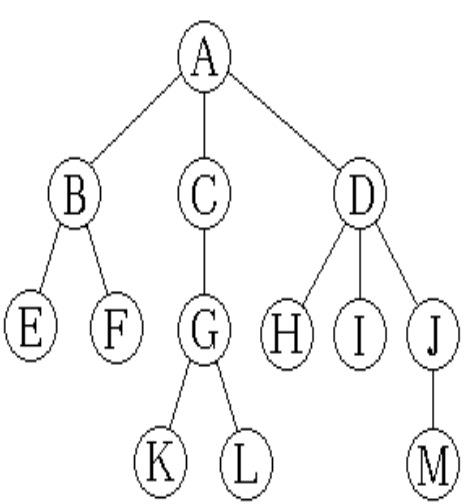
最小堆的删除算法

```
template <class Type> int MinHeap <Type> ::
RemoveMin ( Type &x ) {
    if ( !CurrentSize )
      { cout << " 堆已空 " << endl;  return 0; }
    x = heap[0];                    //最小元素出队列
    heap[0] = heap[CurrentSize-1];
    CurrentSize--;                  //用最小元素填补
    FilterDown ( 0, CurrentSize-1 );
    //从0号位置开始自顶向下调整为堆
    return 1;
}
```

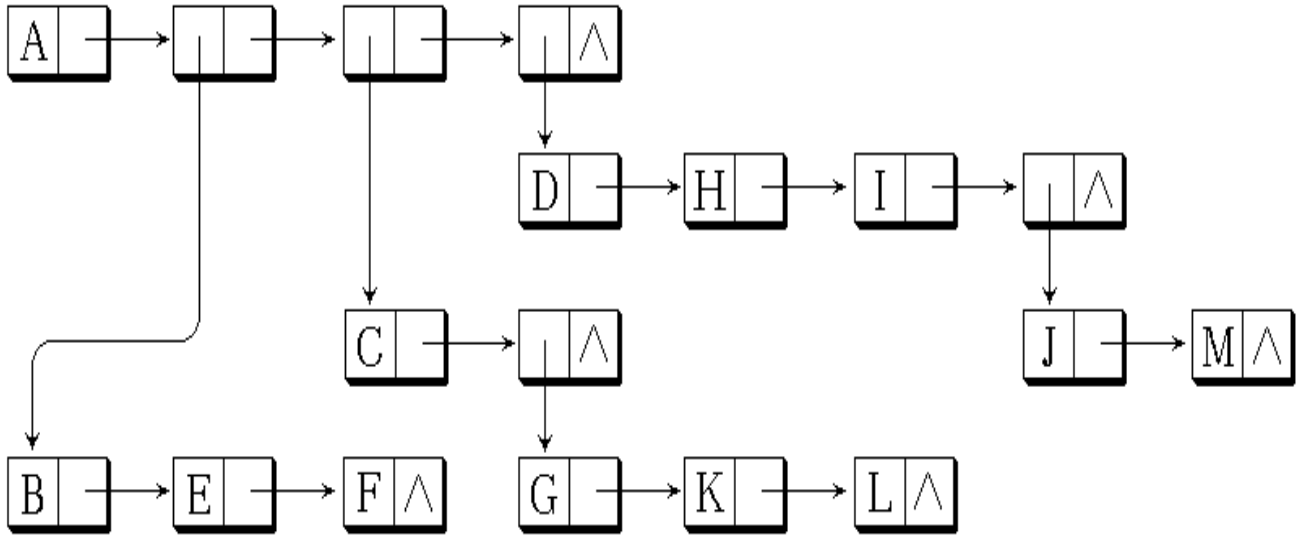
树与森林

树的存储表示

- 广义表表示



(a) 树

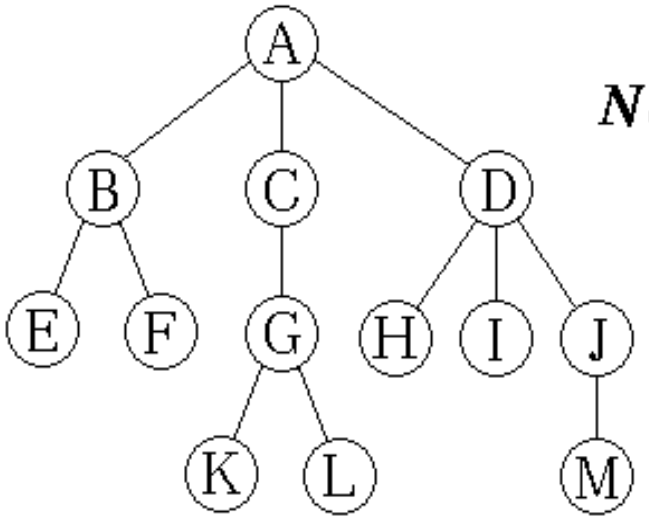


(b) 广义表表示

树的广义表表示

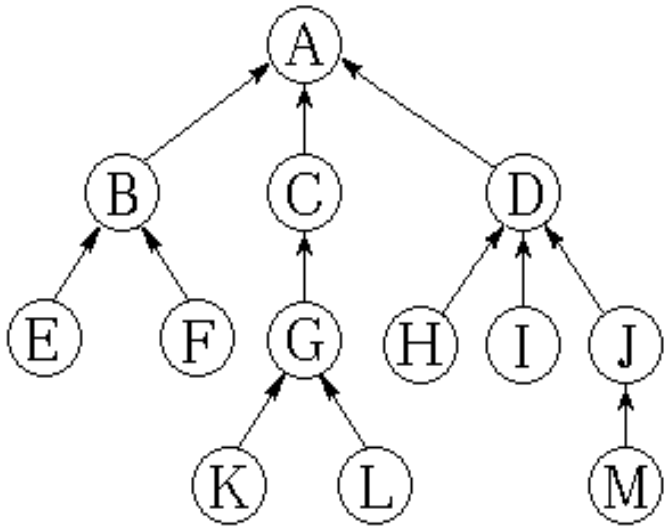
(结点的*utype*域没有画出)

❑ 双亲表示



NodeList	1	2	3	4	5	6	7	8	9	10	11	12	13
data	A	B	E	F	C	G	K	L	D	H	I	J	M
parent	0	1	2	2	1	5	6	6	1	9	9	9	12

（b）双亲表示数组



（c）双亲表示图解

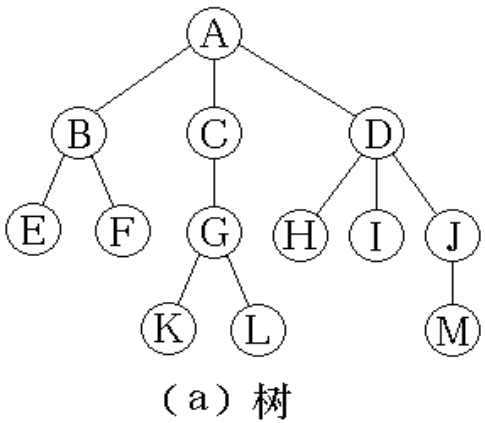
❒ 左子女-右兄弟表示法

第一种解决方案

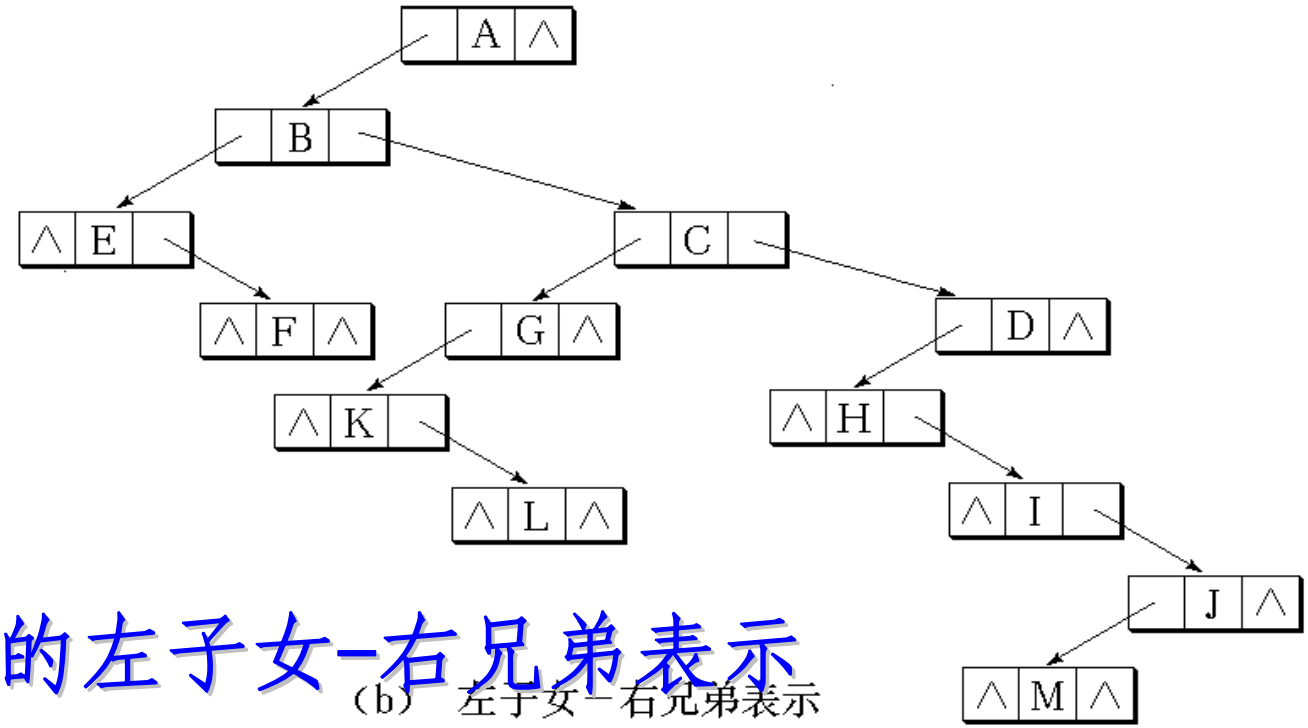
data	$child_1$	$child_2$	$child_3$	.....	$child_d$

第二种解决方案

data	*firstChild*	*nextSibling*



（a）树



（b） 左子女－右兄弟表示

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;
    TreeNode ( Type value=0,
            TreeNode<Type> *fc=NULL,
            TreeNode<Type> *ns=NULL ) : data (value),
            firstChild (fc), nextSibling (ns) { }
};
```

```
template <class Type> class Tree {
public:
    Tree ( ) { root = current = NULL; }
    .......
private:
    TreeNode<Type> *root, *current;
    void PreOrder ( ostream & out,
            TreeNode<Type> *p );
    int Find ( TreeNode<Type> *p, Type target );
    void RemovesubTree ( TreeNode<Type> *p );
    int FindParent ( TreeNode<Type> *t,
                                TreeNode<Type> *p );
}
```

用左子女-右兄弟表示实现的树的部分操作

```
template <class Type> void Tree <Type> ::
BuildRoot ( Type rootVal ) {
//建立树的根结点,并使之成为树的当前结点
    root = current = new TreeNode <Type>
(rootVal); }

template <class Type> int Tree <Type> :: Root ( ) {
//让树的根结点成为树的当前结点
    if ( root == NULL )
        { current = NULL;  return 0; }
    else { current = root;  return 1; }
}
```

```
template <class Type> int Tree<Type>::Parent ( ) {
//在树中寻找当前结点的双亲,使之成为当前结点
    TreeNode<Type> *p = current,  *t;
    if ( current == NULL || current == root )
      { current = NULL;  return 0; }
    t = root;
    int k = FindParent ( t, p );
    return k;
}

template <class Type> int Tree <Type> ::
FirstChild ( ) {
//在树中寻找当前结点的长子,使之成为当前结点
```

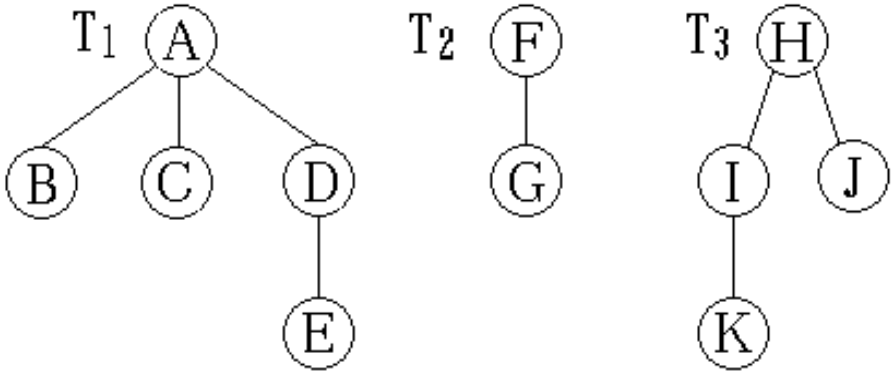
```
    if (current != NULL &&
            current→firstChild != NULL )
        { current = current→firstChild;  return 1; }
    current = NULL;  return 0;
}

Template <class Type> int Tree <Type>::
NextSibling ( ) {
//在树中寻找当前结点的兄弟,使之成为当前结点if
( current != NULL &&
            current→nextSibling != NULL )
        { current = current→nextSibling;  return 1; }
    current = NULL;  return 0;
}
```

```
template <class Type> int Tree<Type> ::
FindParent ( TreeNode<Type> *t,
                                TreeNode<Type> *p ) {
//在根为 t 的树中找 p 的双亲,使之成为当前结点
   TreeNode<Type> *q = t→firstChild;
   while ( q != NULL && q != p ) {
   //循根的长子的兄弟链, 递归在子树中搜索
      if ((int i = FindParent (*q,*p)) != 0 ) return i;
      q = q→nextSibling;
  }
  if ( q != NULL && q ==p )
     { current = t;  return 1; }
  else return 0;       //未找到双亲, 当前结点不变
```

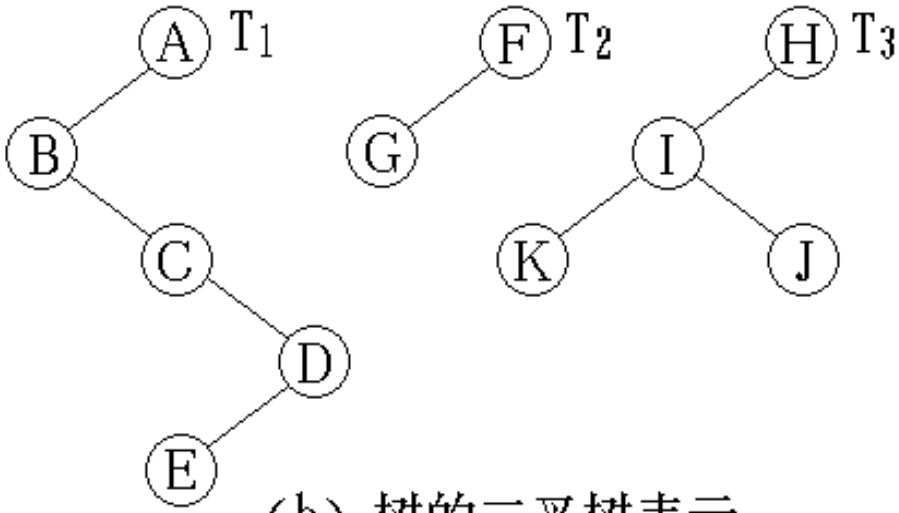
```
}

template <class Type> int Tree<Type> ::
Find ( Type target ) {
    if ( IsEmpty ( ) ) return 0;
    return Find ( root, target );
}

template <class Type> int Tree <Type>::
Find ( TreeNode <Type> *p, Type target ) {
//在根为 p 的树中寻找值为 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:  =1, 搜索成功; =0, 搜索失败
```

```
    int result = 0;
    if ( p→data == target )
        { result = 1;  current = p; }
    else {
        TreeNode<Type> *q = p→firstChild;
        while ( q != NULL &&
                        ! ( result = Find ( q, target ) ) )
            q = q→nextSibling;
    }
    return result;
}
```

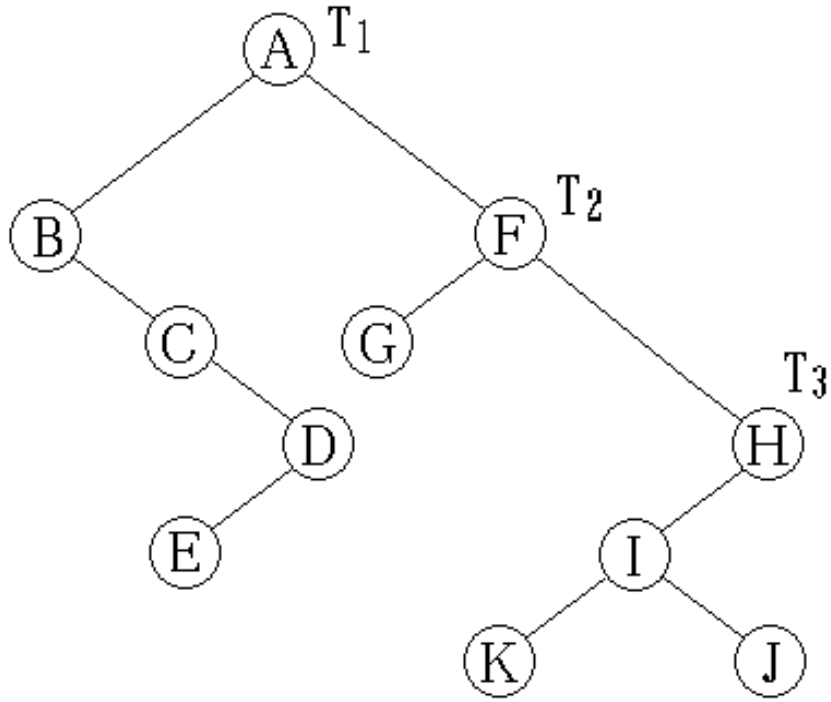
森林与二叉树的转换



（a）3棵树的森林



（b）树的二叉树表示



（c）森林的二叉树表示

森林与二叉树的对应关系

(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 $B(T_{11}, T_{12}, \cdots, T_{1m})$，其中，$T_{11}, T_{12}, \cdots, T_{1m}$ 是 $root(T_1)$ 的子树;

其右子树为 $B(T_2, T_3, \cdots, T_n)$，其中，$T_2, T_3, \cdots, T_n$ 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。

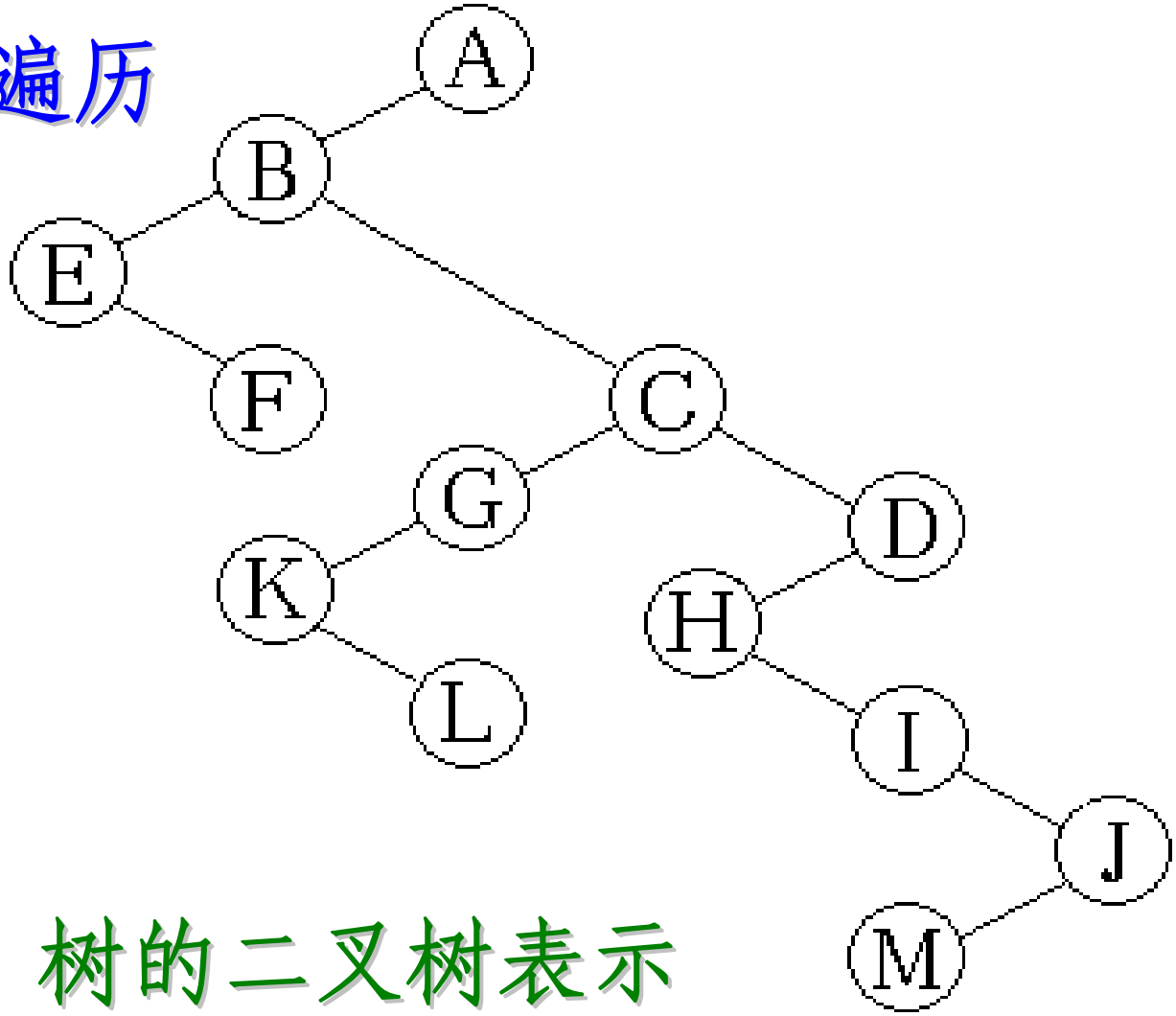
② 如果 ***B*** 非空，则

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。

树的遍历

深度优先遍历



树的二叉树表示

(1) 树的先根次序遍历的递归算法

```
template <class Type>
void Tree<Type> :: PreOrder ( ) {
    if ( !IsEmpty ( ) ) {
        visit ( );                          //访问根结点
        TreeNode<Type> *p = current; //暂存当前结点
        int i = FirstChild ( );     //当前结点转到长子
        while (i) { PreOrder ( );  i = NextSibling ( ); }
        //递归先根遍历各棵子树
        current = p;                        //递归完恢复当前结点
    }
}
```

(2) 树的后根次序遍历的递归算法

```
template <class Type>
void Tree<Type> :: PostOrder ( ) {
    if ( !IsEmpty ( ) ) {
        TreeNode<Type> *p = current; //暂存当前结点
        int i = FirstChild ( );      //当前结点转到长子
        while (i) { PostOrder ( );  i = NextSibling ( ); }
         //递归后根遍历各棵子树
        current = p;                 //递归完恢复当前结点
        visit ( );                   //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;
    do {
        while ( !IsEmpty ( ) ) {      //从根沿长子链向下
            visit ( );  st.Push ( current );    //访问,进栈
            FirstChild ( );
        }
        while ( IsEmpty ( ) && !st.IsEmpty ( ) ) {
            current = st.Pop ( );  NextSibling ( );
        }     //无子女或无兄弟,退栈,转向下一兄弟
    } while ( !IsEmpty ( ) );      //栈空,退出循环
```

```
    current = p;
}
```

(4) 按后根次序遍历树的迭代算法

```
template <class Type>
void Tree<Type> :: PostOrder1 ( ) {
    Stack<TreeNode<Type>*> st;
    TreeNode<Type> *p = current;
    do {
        while ( !IsEmpty ( ) ) {      //从根沿长子链向下
            st.Push ( current );  FirstChild ( );   //进栈
    }
```

```
        while ( IsEmpty ( ) && !st.IsEmpty ( ) ) {
            current = st.Pop ( );  visit ( );
            NextSibling ( );
         //无子女或无兄弟,退栈,访问,转向兄弟
        }
    } while ( !IsEmpty ( ) );
    current = p;
}
```

广度优先(层次次序)遍历

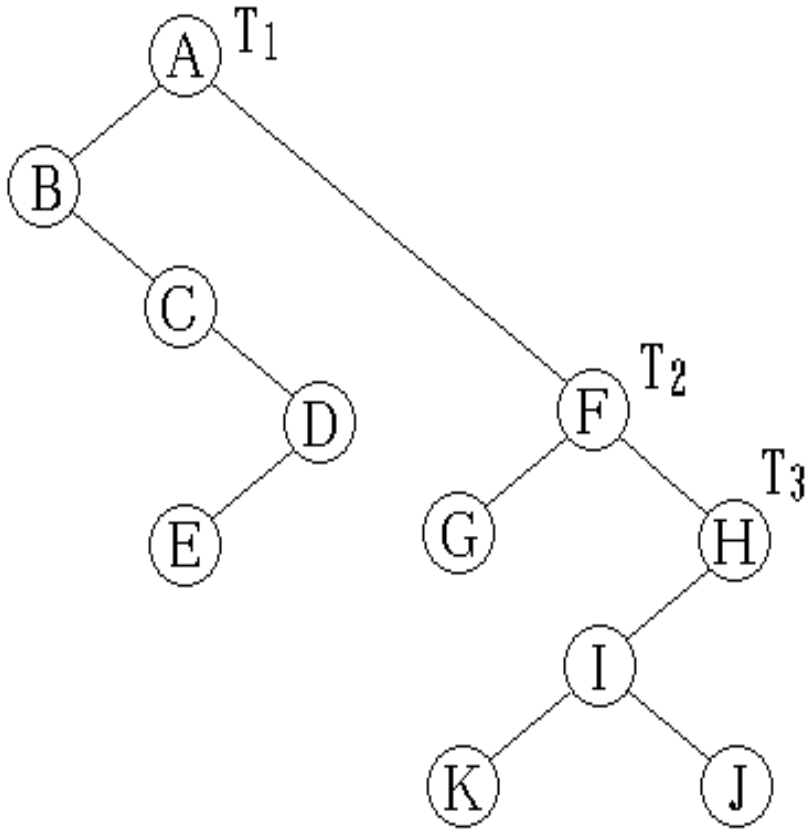
```
template <class Type>
void Tree<Type> :: LevelOrder ( ) {
```

```
    Queue<TreeNode<Type>*> Qu ( DefaultSize );
    TreeNode<Type> *p
    if ( ! IsEmpty ( ) ) {
       p = current;   Qu.EnQueue ( current );
       while ( !Qu.IsEmpty ( ) ) {     //队列不空
          current = Qu.DeQueue ( );  visit ( );
          //退出队列,访问
          FirstChild ( );                  //转向长子
          while ( !IsEmpty ( ) )    //结点指针不空
            { Qu.EnQueue ( current );  NextSibling ( ); }
       }
       current = p;
    }
}
```

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。

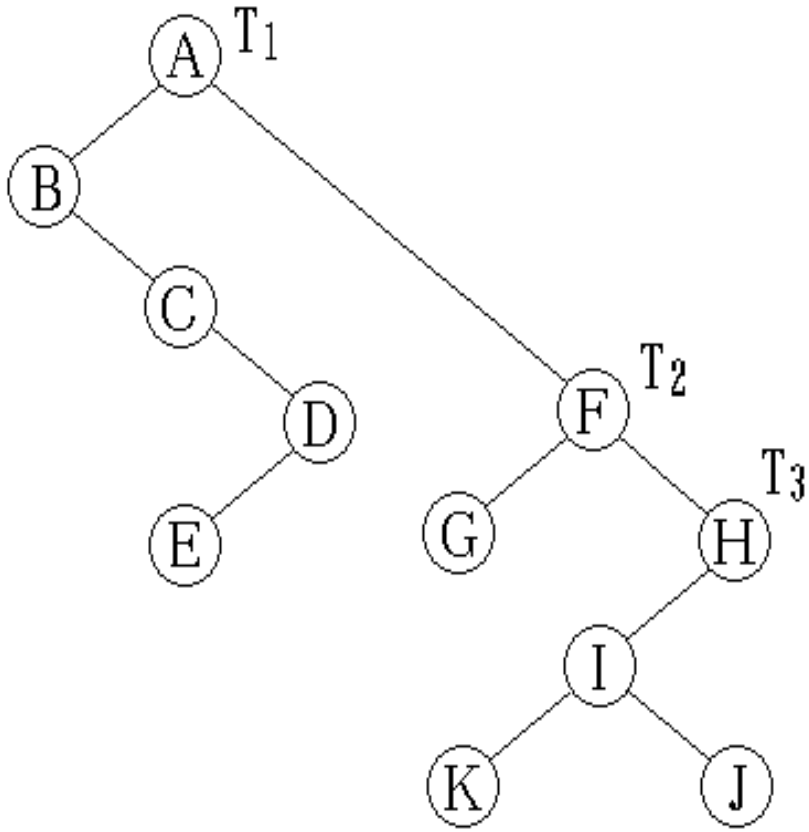


森林的二叉树表示

森林的遍历

(2) 中根次序遍历的规则：

- 若森林F为空，返回；否则
  - 中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；
  - 中根次序序遍历其它树组成的森林。

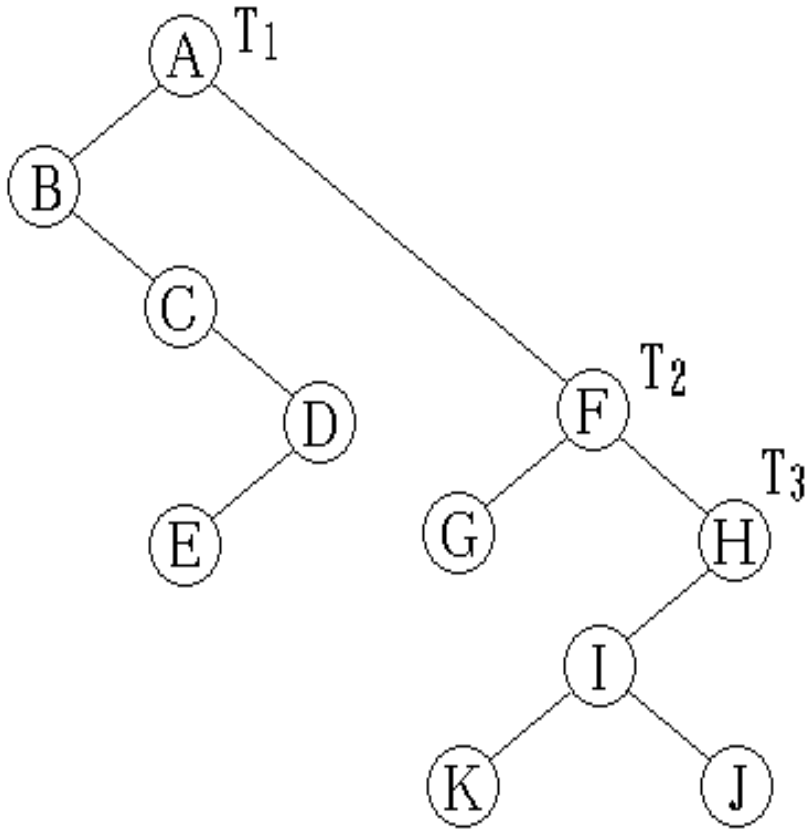


森林的二叉树表示

森林的遍历

(3) 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；
  - 后根次序遍历其它树组成的森林；
  - 访问F的第一棵树的根结点。

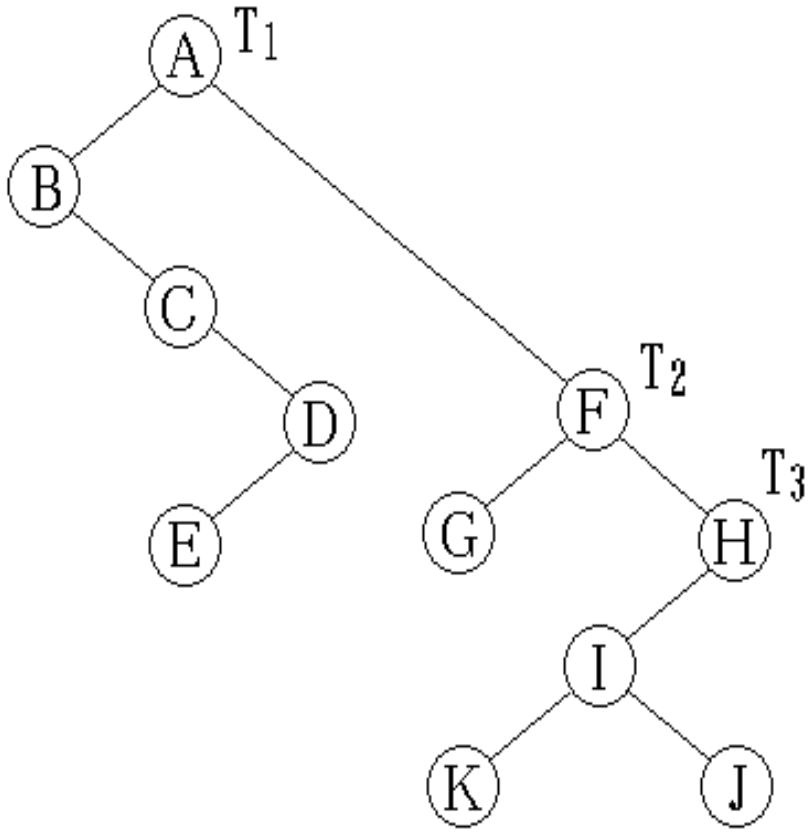


森林的二叉树表示

森林的遍历

(4) 广度优先遍历(层次序遍历)：

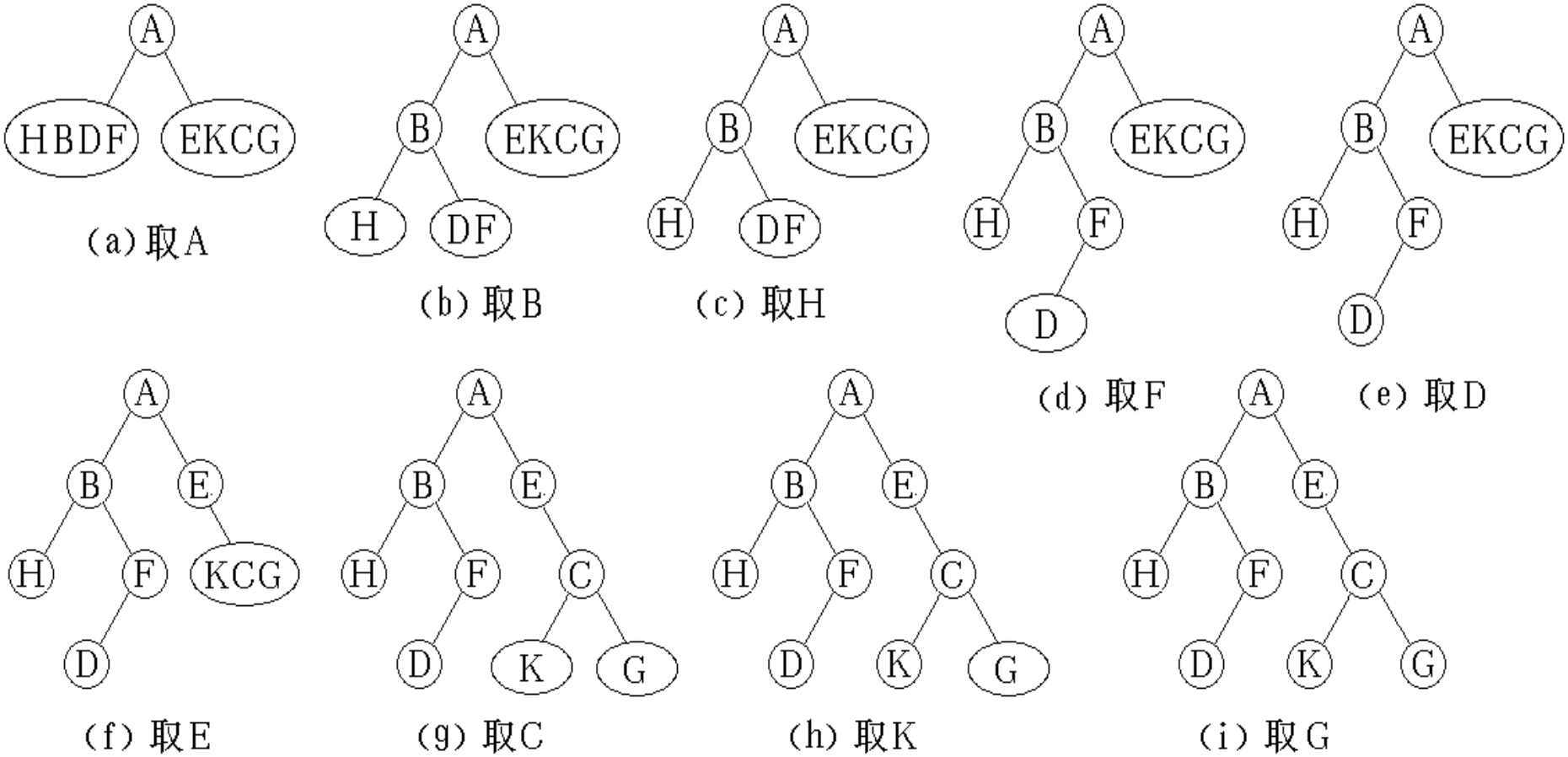
- 若森林F为空，返回；否则
  - 依次遍历各棵树的根结点；
  - 依次遍历各棵树根结点的所有子女；
  - 依次遍历这些子女结点的子女结点。……



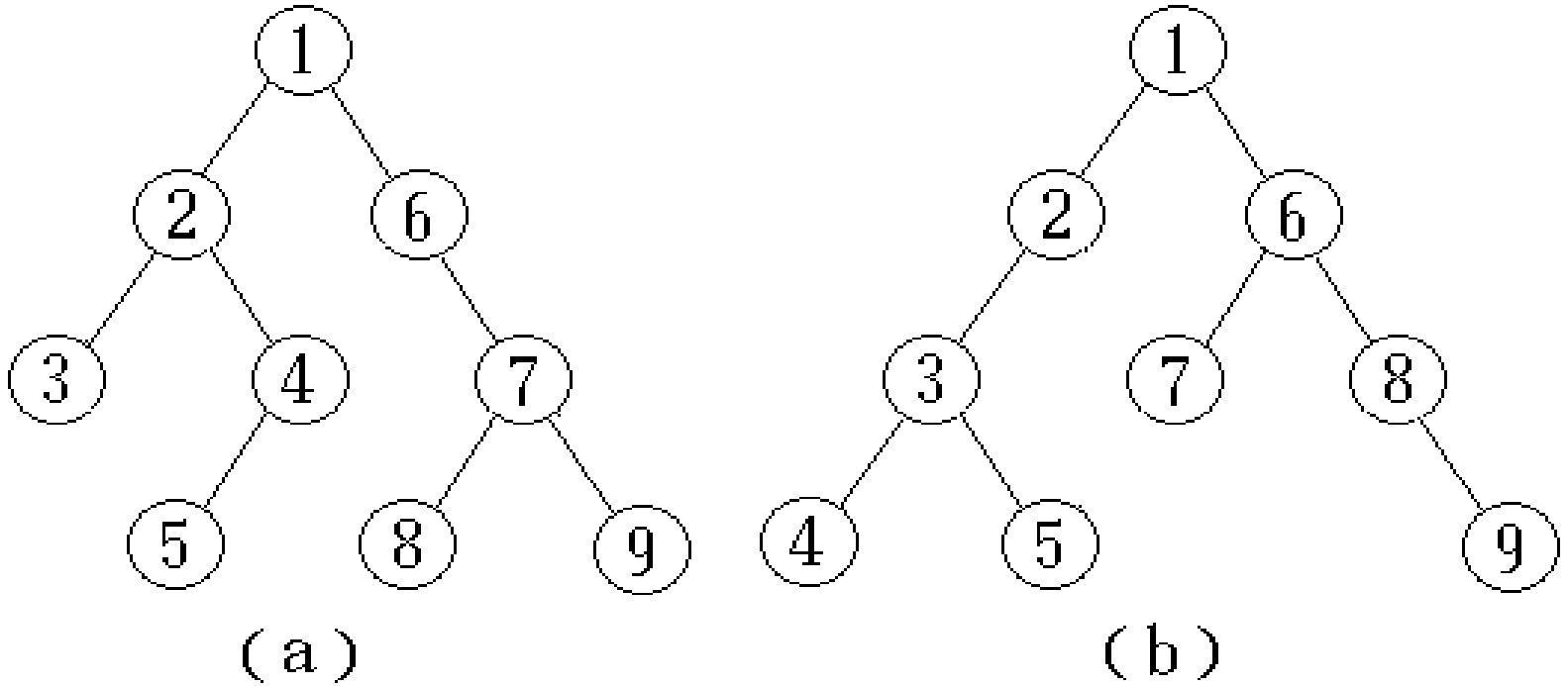
森林的二叉树表示

二叉树的计数

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：

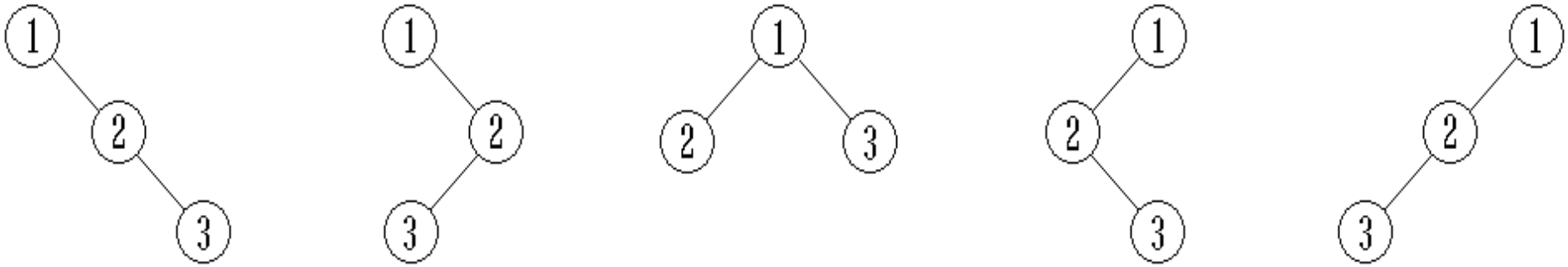


如果前序序列固定不变，给出不同的中序序列，可得到不同的二叉树。

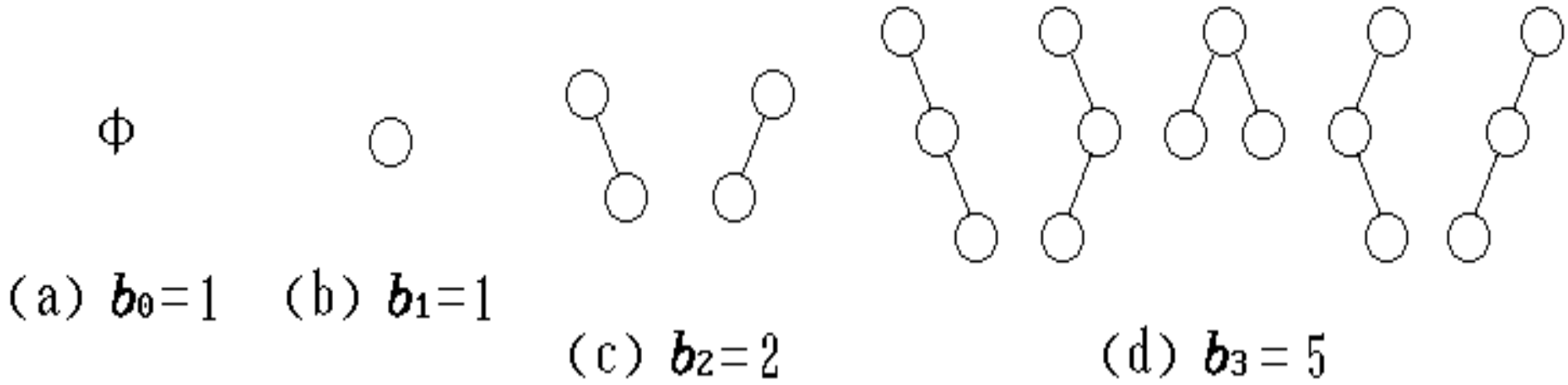


问题是有 ***n*** 个数据值，可能构造多少种不同的二叉树？我们可以固定前序排列，选择所有可能的中序排列。

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个, 1个, 2个, 3个结点的不同二叉树如下

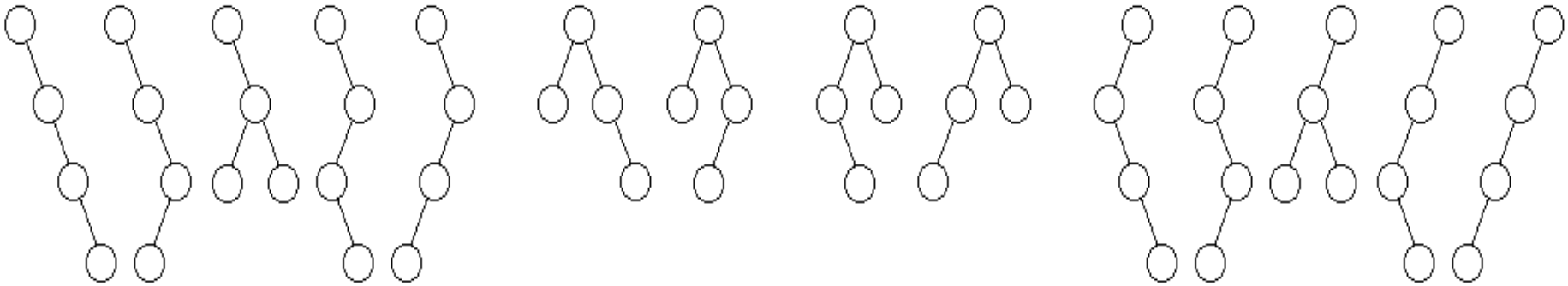


计算具有 *n* 个结点的不同二叉树的棵数

*Catalan*函数

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

例如，具有**4**个结点的不同二叉树

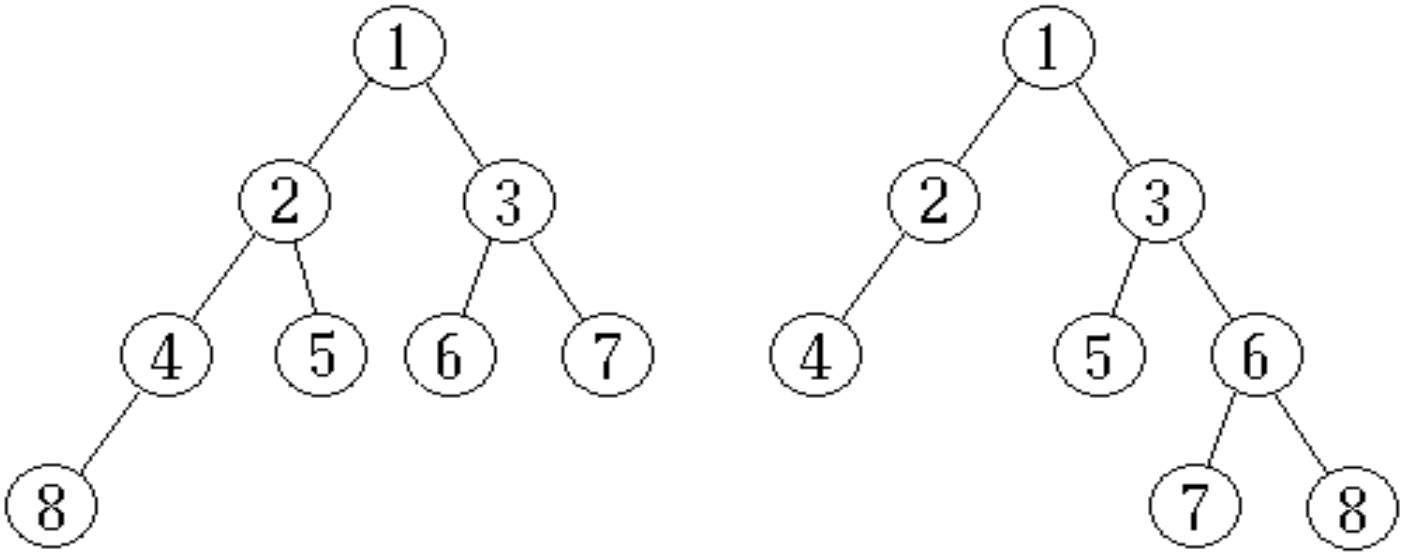


(a) $i=0$　(b) $i=1$　(c) $i=2$　(d) $i=3$

霍夫曼树 (Huffman Tree)

路径长度 (Path Length)

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



具有不(a)同路径长度的(b)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

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

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

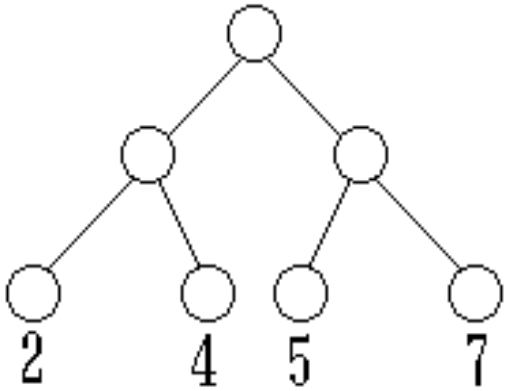
其路径长度最小者为 $PL = \sum_{i=0}^{n-1} \lfloor \log_2(i+1) \rfloor$

带权路径长度 (Weighted Path Length, WPL)

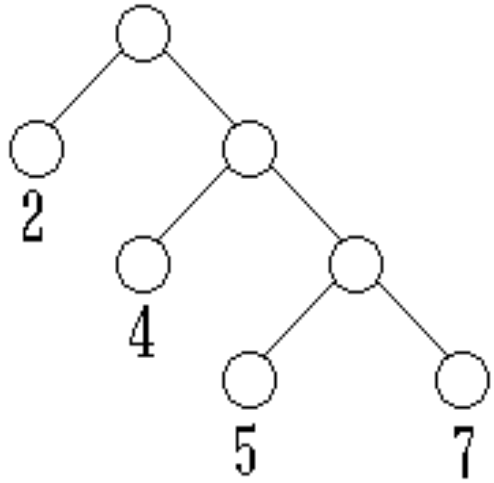
树的带权路径长度是树的各叶结点所带的权值与该结点到根的路径长度的乘积的和。

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

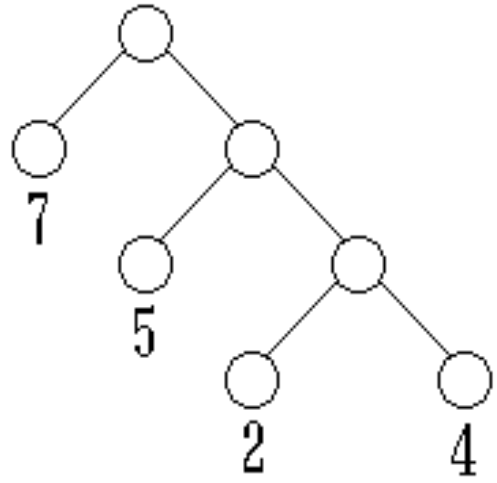
具有不同带权路径长度的扩充二叉树



(a) $WPL = 36$



(b) $WPL = 46$



(c) $WPL = 35$

霍夫曼树

带权路径长度达到最小的扩充二叉树即为霍夫曼树。

在霍夫曼树中，权值大的结点离根最近。

霍夫曼算法

(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$F = \{T_0, T_1, T_2, \cdots, T_{n-1}\}$，其中每一棵扩充二叉树 T_i 只有一个带有权值 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:

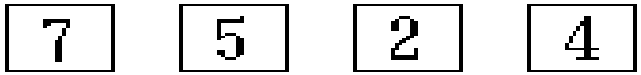
① 在 F 中选取两棵根结点的权值最小的扩充二叉树, 做为左、右子树构造一棵新的二叉树。置新的二叉树的根结点的权值为其左、右子树上根结点的权值之和。

② 在 F 中删去这两棵二叉树。

③ 把新的二叉树加入 F。

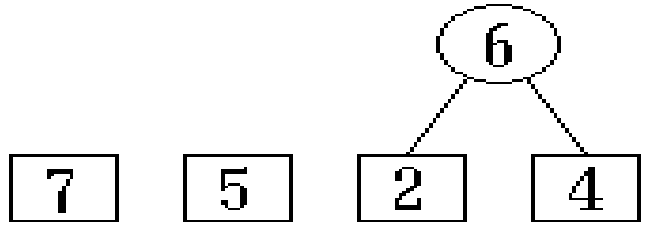
霍夫曼树的构造过程

F：{7}{5}{2}{4}



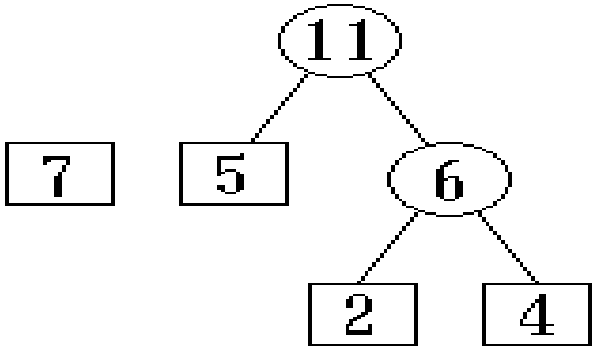
（a）初始

F：{7}{5}{6}



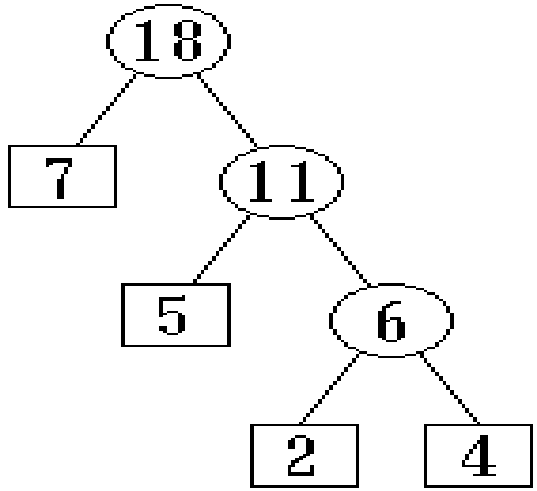
（b）合并{2}{4}

F：{7}{11}



（c）合并{5}{6}

F：{18}



（d）合并{7}{11}

扩充二叉树的类定义

```
template <class Type> class ExtBinTree;

template <class Type> class Element {
friend class ExtBinTree;
private:
    Type data;
    Element<Type> *leftChild, *rightChild;
};

template <class Type> class ExtBinTree {
public:
    ExtBinTree ( ExtBinTree<Type> &bt1,
                 ExtBinTree<Type> &bt2 ) {
```

```
        root→leftChild = bt1.root;
        root→rightChild = bt2.root;
        root→data.key = bt1.root→data.key +
                                bt2.root→data.key;
    }
protected:
    const int DefaultSize = 20;
    Element <Type> *root;          //扩充二叉树的根
}
```

建立霍夫曼树的算法

```
template <class Type>
void HuffmanTree (Type *fr,  int n, ExtBinTree
<Type> & newtree ) {
   ExtBinTree <Type> & first, & second;
   ExtBinTree <Type> Node[DafualtSize];
   MinHeap < ExtBinTree <Type> > hp;  //最小堆
   if ( n > DefaultSize ) {
      cout << "大小 n " << n << "超出了数组边界 "
           << endl;   return;
   }
   for ( int i = 0; i < n; i++ ) {
      Node[i].root→data.key = fr[i];
```

```
        Node[i].root→leftChild =
                Node[i].root→rightChild = NULL;
    }    //传送初始权值
    hp.MinHeap ( Node, n );
    for ( int i = 0; i < n−1; i++ ) {
     //建立霍夫曼树的过程，做n−1趟
        hp.DeleteMin ( first );      //选根权值最小的树
        hp.DeleteMin ( second );  //选根权值次小的树
        newtree = new ExtBinTree <Type>
                    ( first, second );    //建新的根结点
        hp.Insert ( newtree );       //形成新树插入
    }
}
```

霍夫曼编码

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。

因各字符出现的概率为{ 2/18, 7/18, 4/18, 5/18 }。

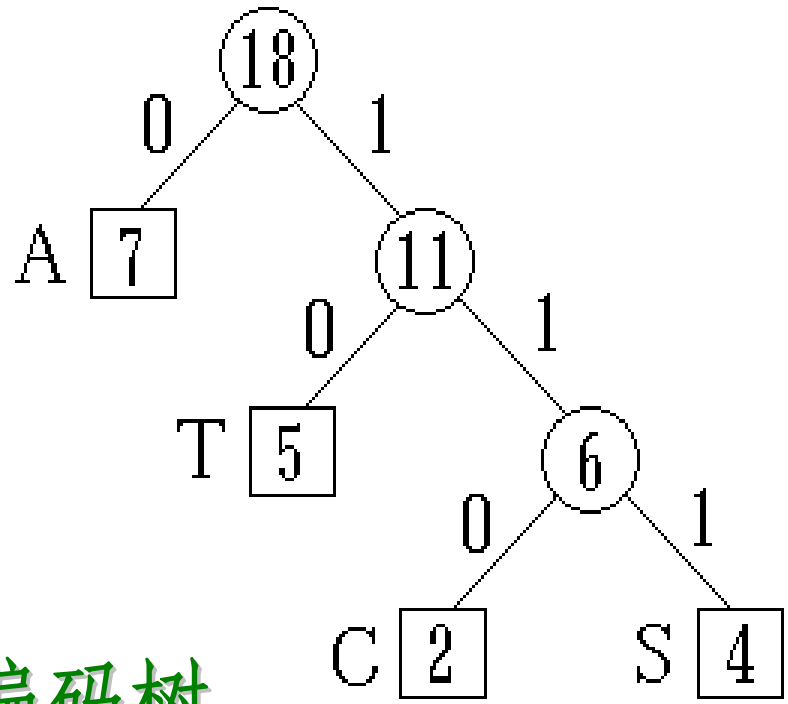
化整为{2, 7, 4, 5}，以它们为各叶结点上的权值，建立霍夫曼树。左分支赋0，右分支赋1，得霍夫曼编码(变长编码)。

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霍夫曼树的带权路径长度**WPL**。

霍夫曼编码是一种无前缀编码。解码时不会混淆。



霍夫曼编码树

小结　需要复习的知识点

- 树　树的定义、树的基本运算
  - 树的分层定义是递归的
  - 树中结点个数与高度的关系
- 二叉树　二叉树定义、基本运算
  - 二叉树性质
  - 二叉树结点个数与高度的关系
  - 不同二叉树棵数
  - 完全二叉树的顺序存储

- 完全二叉树的双亲、子女和兄弟的位置
- 二叉树的前序·中序·后序·层次遍历
- 前序·中序·后序的线索化二叉树中前驱与后继的查找方法
- 应用二叉树遍历的递归算法

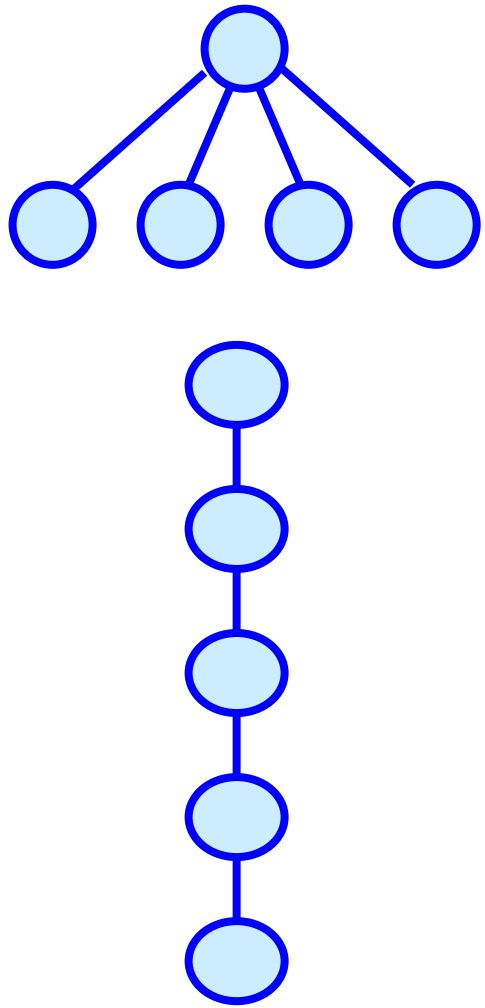
- 霍夫曼树
  - 霍夫曼树的构造方法
  - 霍夫曼编码
  - 带权路径长度的计算
- 树的存储
  - 树的广义表表示与双亲表示
  - 树与二叉树的对应关系
  - 树的先根·后根·层次遍历

- 堆 堆的定义、堆的插入与删除
  - 形成堆时用到的向下调整算法
  - 形成堆时比较次数的上界估计
  - 堆插入时用到的向上调整算法
  - 堆的插入与删除算法

【例1】在结点个数为n ($n > 1$)的各棵树中，高度最小的树的高度是多少？它有多少叶结点？多少分支结点？高度最大的树的高度是多少？它有多少叶结点？多少分支结点？

【解答】结点个数为 n 时，高度最小的树的高度为 1，有 2层；有 $n-1$ 个叶结点，1 个分支结点；

高度最大的树的高度为 $n-1$，有 n 层；它有 1 个叶结点，$n-1$个分支结点



【例2】已知一棵二叉树的前序遍历的结果序列是

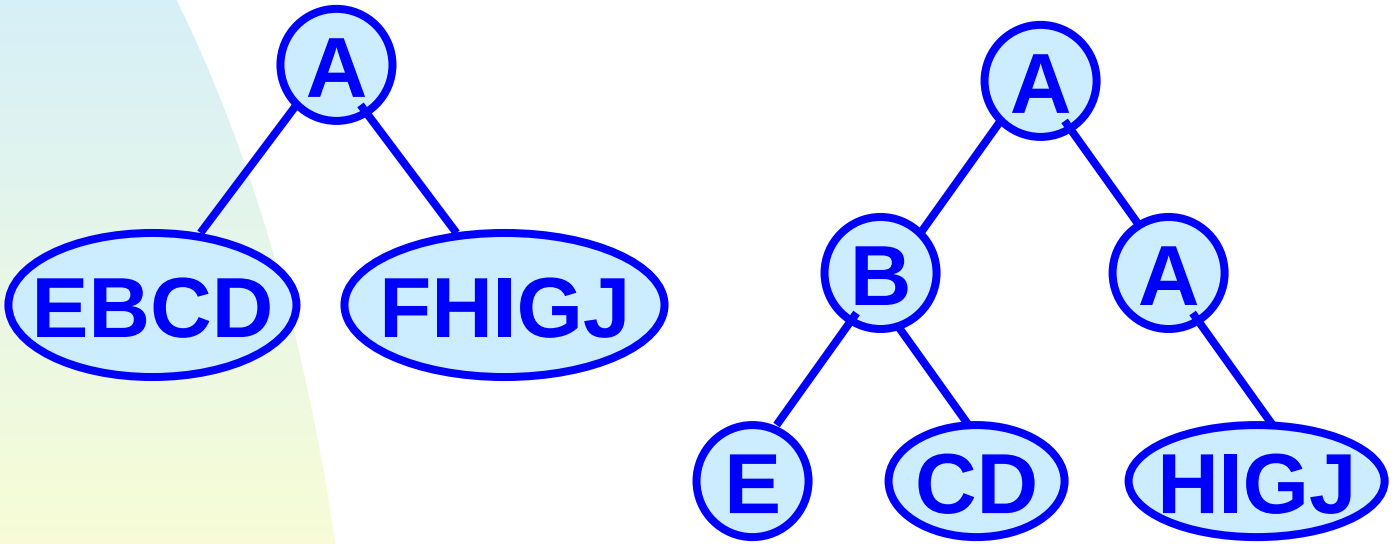
ABECDFGHIJ

中序遍历的结果序列是

EBCDAFHIGJ

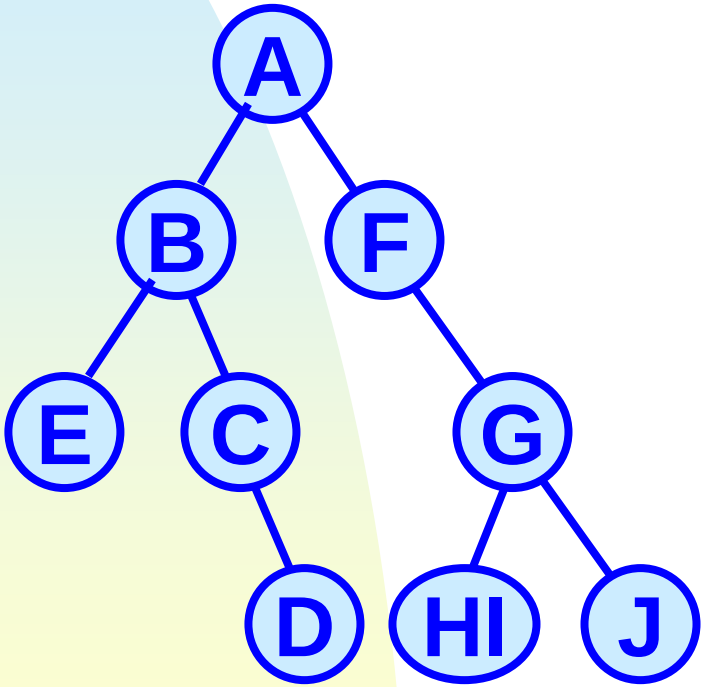
试画出这棵二叉树。

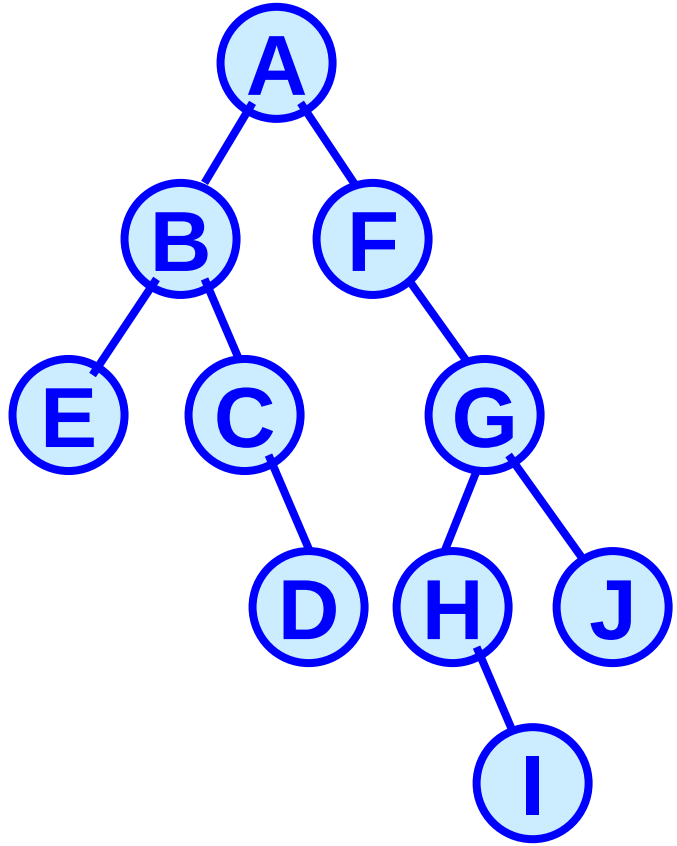
【解答】前序序列 ABECDFGHIJ，
中序序列 EBCDAFHIGJ 时：



前序序列 ABECDFGHIJ

中序序列 EBCDAFHIGJ





【例3】若用二叉链表作为二叉树的存储表示，试针对以下问题编写递归算法:

(1) 统计二叉树中叶结点个数。

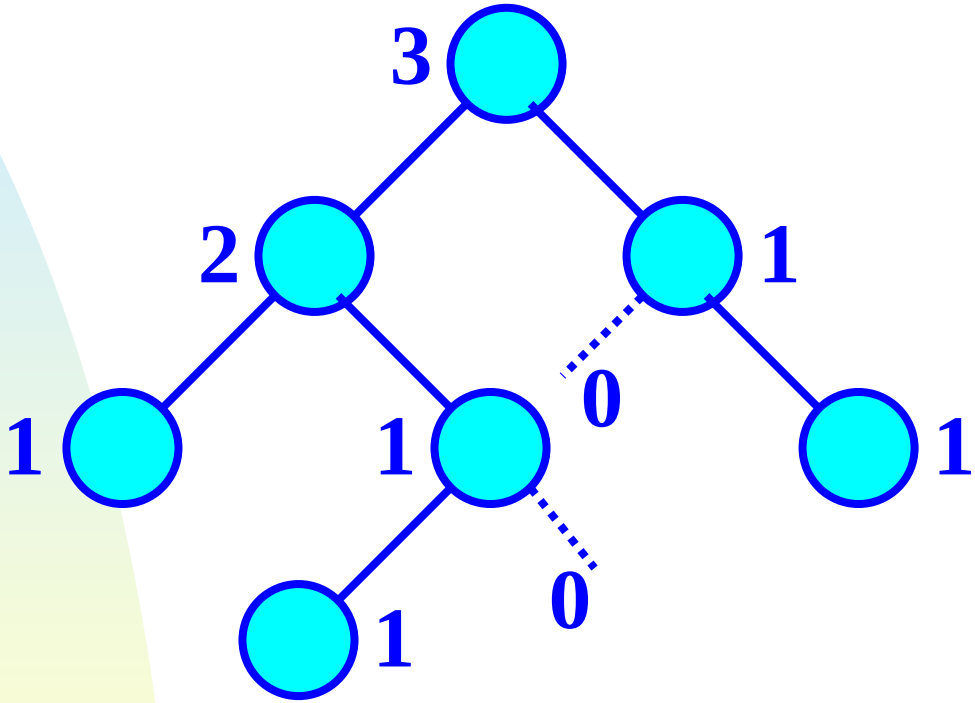
(2) 以二叉树为参数, 交换每个结点的左子女和右子女。

【解答】

(1) 定义二叉树结构

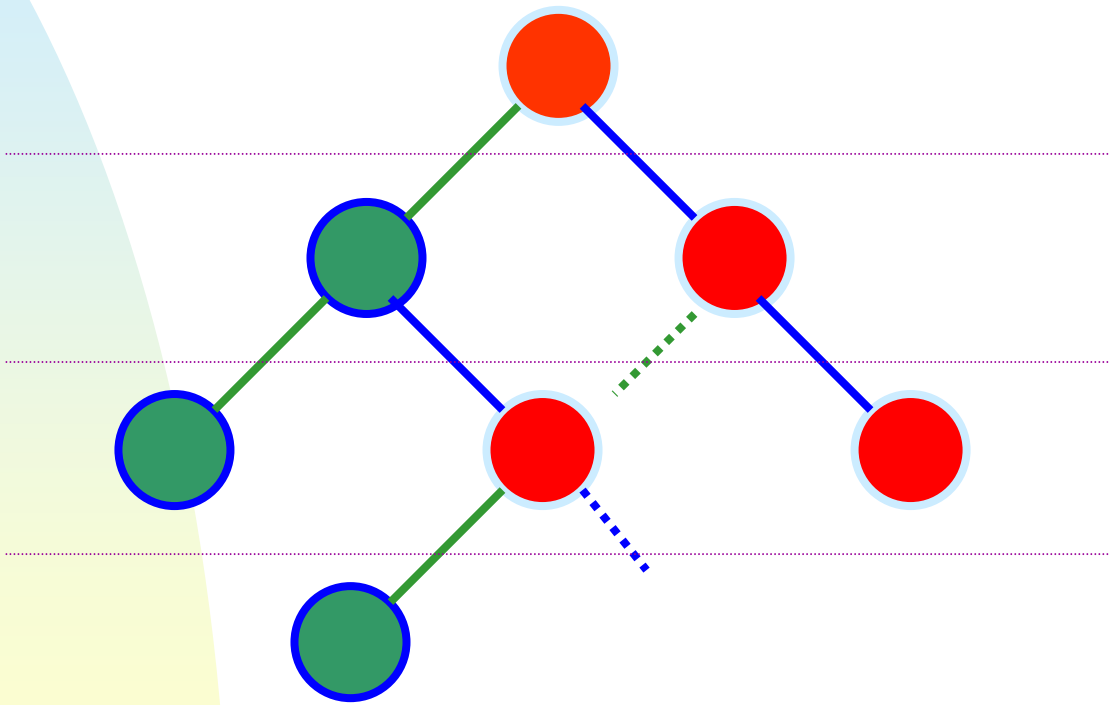
```
template <class Type>
    class BinaryTree;   //二叉树
template <class Type>
    class BinTreeNode   //二叉树结点
```

(2) 统计二叉树中叶结点个数



```
template <class Type>
int leaf ( BinTreeNode<Type>* t ) {
    int leaves;
    if ( !t ) leaves = 0;
    else if ( !t→leftChild &&
              !t→rightChild ) leaves = 1;
        else leaves = leaf ( t→leftChild )
                    + leaf ( t→rightChild );
    return *leaves;
}
```

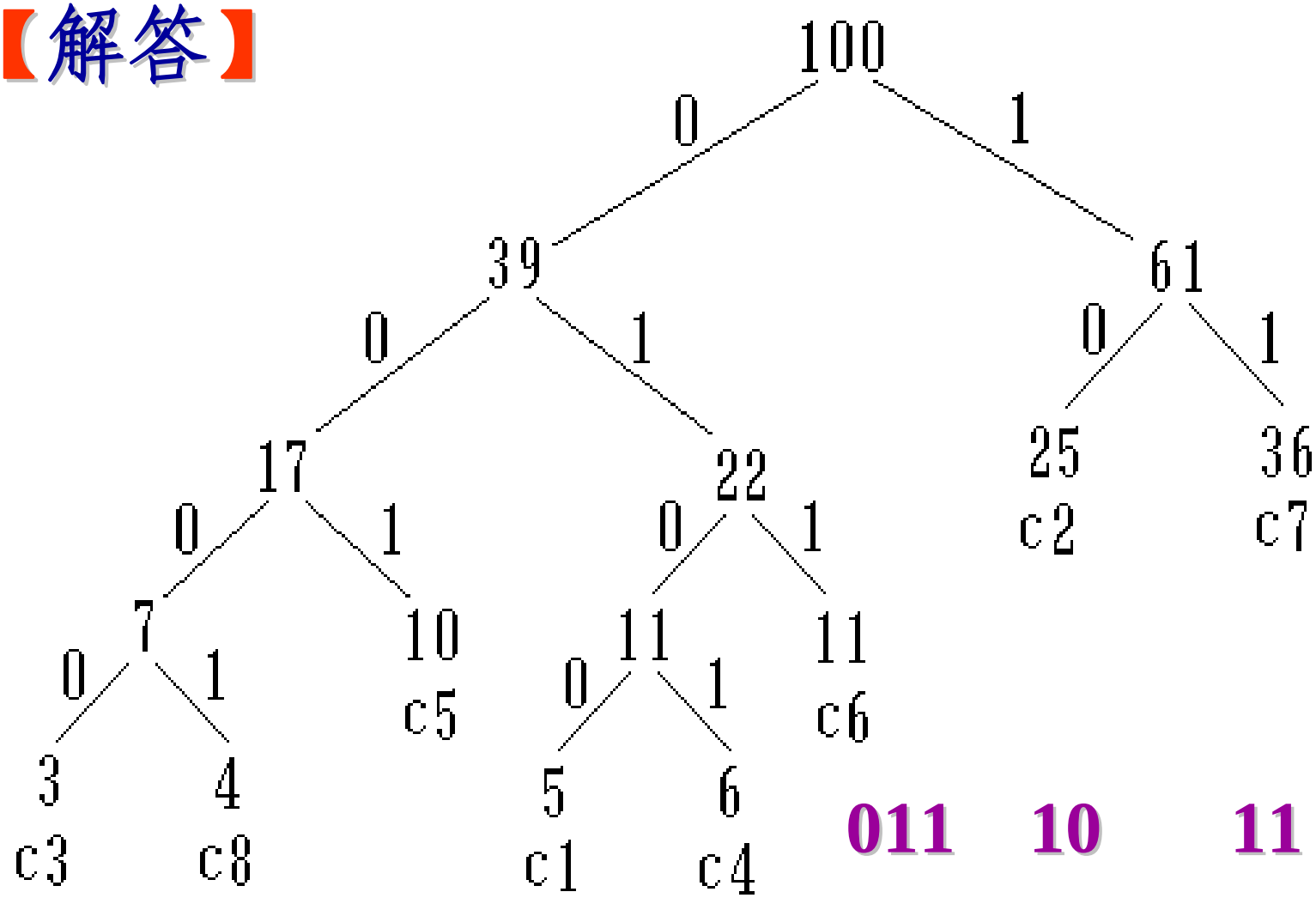
(3) 交换每个结点的左子女和右子女



```
template <class Type>
void exchange ( BinTreeNode<Type>* t ) {
    BinTreeNode<Type>* p;
    if ( t→leftChild || t→rightChild ) {
        //非叶结点，交换左、右子女
        p = t→leftChild;
        t→leftChild = t→rightChild ;
        t→rightChild = p;
        exchange ( t→leftChild );
        exchange ( t→rightChild );
    }
}
```

【例4】假定用于通信的电文仅由 8 个字母 c1, c2, c3, c4, c5, c6, c7, c8 组成, 各字母在电文中出现的频率分别为 5, 25, 3, 6, 10, 11, 36, 4。试为这 8 个字母设计不等长 Huffman 编码, 并给出该电文的总码数。

【解答】



011 10 11

0000 0001 001 0100 0101

则*Huffman*编码为

c1	c2	c3	c4	c5	c6	c7	c8
0100	10	0000	0101	001	011	11	0001

电文总码数为

$$4 * 5 + 2 * 25 + 4 * 3 + 4 * 6 +$$
$$3 * 10 + 3 * 11 + 2 * 36 + 4 * 4$$
$$= 257$$

【例5】下面是对二叉树进行中序遍历的递归算法。

```
template <class Type>
void inorder ( BinTreeNode<Type> * t ) {
    if ( t ) {
        inorder ( t→leftChild );
        cout << t→data;
        inorder ( t→rightChild );
    }
}
```

将算法中的第二个递归语句消去，这相当于尾递归，可用循环实现

```
template <class Type>
void inorder ( BinTreeNode<Type> * t ) {
   while ( t ) {
        inorder ( t→leftChild );
        cout << t→data;
        t = t→rightChild;
    }
}
```

接着,将算法中的第一个递归语句消去,这必须借助于栈,记忆回退的路径.

```
template <class Type>
void InOrder ( BinTreeNode<Type>* t ) {
    StackType S;
    BinTreeNode<Type>* p = t;
    S.makeEmpty( );              //初始化
```

```
    do{
        while ( p )
            { S.Push(p);   p = p→leftChild; }
        if ( !S.IsEmpty( ) ) {
            p = S.getTop( );  S.Pop( );
            cout << p→data << endl;
            p = p→rightChild;
        }
    } while ( p || !S.IsEmpty( ) );
}
```