第 6 章 树与森林

6-1 写出用广义表表示法表示的树的类声明，并给出如下成员函数的实现：

(1) **operator** >> ( ) 接收用广义表表示的树作为输入，建立广义表的存储表示；

(2) 复制构造函数 用另一棵表示为广义表的树初始化一棵树；

(3) **operator** == ( ) 测试用广义表表示的两棵树是否相等；

(4) **operator** << ( ) 用广义表的形式输出一棵树；

(5) 析构函数 清除一棵用广义表表示的树。

【解答】

```
#include <iostream.h>
#define maxSubTreeNum 20;                          //最大子树(子表)个数
class GenTree;                                     //GenTree 类的前视声明

class GenTreeNode {                                //广义树结点类的声明
friend class GenTree;
private:
    int utype;                                     //结点标志：=0, 数据; =1, 子女
    GenTreeNode * nextSibling;                     //utype=0, 指向第一个子女;
                                                   //utype=1 或 2, 指向同一层下一兄弟
    union {                                        //联合
        char RootData;                             //utype=0, 根结点数据
        char Childdata;                            //utype=1, 子女结点数据
        GenTreeNode *firstChild;                   //utype=2, 指向第一个子女的指针
    }
public:
    GenTreeNode ( int tp, char item ) : utype (tp), nextSibling (NULL)
      { if ( tp == 0 ) RootData = item;   else ChildData = item; }
      //构造函数：构造广义表表示的树的数据结点
    GenTreeNode ( GenTreeNode *son = NULL ) : utype (2), nextSibling (NULL), firstChild ( son ) { }
      //构造函数：构造广义表表示的树的子树结点
    int nodetype ( ) { return utype; }             //返回结点的数据类型
    char GetData ( ) { return data; }              //返回数据结点的值
    GenTreeNode * GetFchild ( ) { return firstChild; }   //返回子树结点的地址
    GenTreeNode * GetNsibling ( ) { return nextSibling; }  //返回下一个兄弟结点的地址
    void setInfo ( char item ) { data = item; }    //将结点中的值修改为 item
    void setFchild ( GenTreeNode * ptr ) { firstChild = ptr; }   //将结点中的子树指针修改为 ptr
    void setNsinbilg ( GenTreeNode * ptr ) { nextSibling = ptr; }
};

class GenTree {                                    //广义树类定义
private:
    GenTreeNode *first;                            //根指针
```

```
    char retValue;                                          //建树时的停止输入标志
    GenTreeNode *Copy ( GenTreeNode * ptr );                //复制一个 ptr 指示的子树
    void Traverse ( GenListNode * ptr );                    //对 ptr 为根的子树遍历并输出
    void Remove ( GenTreeNode *ptr );                       //将以 ptr 为根的广义树结构释放
    friend int Equal ( GenTreeNode *s, GenTreeNode *t );    //比较以 s 和 t 为根的树是否相等
public:
    GenTree ( );                                            //构造函数
    GenTree ( GenTree& t );                                 //复制构造函数
    ~GenTree ( );                                           //析构函数
    friend int operator == ( GenTree& t1, GenTree& t2 );    //判两棵树 t1 与 t2 是否相等
    friend istream& operator >> ( istream& in, GenTree& t );//输入
    friend ostream& operator << ( ostream& out, GenTree& t );//输出
}
```

(1) **operator >> ( )** 接收用广义表表示的树作为输入，建立广义表的存储表示

```
istream& operator >> ( istream& in, GenTree& t ) {
//友元函数，从输入流对象 in 接受用广义表表示的树，建立广义表的存储表示 t。
    t.ConstructTree ( in, retValue );
    return in;
}

void GenTree :: ConstructTree ( istream& in, char& value ) {
//从输入流对象 in 接受用广义表表示的非空树，建立广义表的存储表示 t。
    Stack <GenTreeNode* > st (maxSubTreeNum);               //用于建表时记忆回退地址
    GenTreeNode * p, q, r;    Type ch;
    cin >> value;                                           //广义树停止输入标志数据
    cin >> ch;    first = q = new GenTreeNode ( 0, ch );    //建立整个树的根结点
    cin >> ch;    if ( ch == '(' ) st.Push ( q );           //接着应是'('，进栈
    cin >> ch;
    while ( ch != value ) {                                 //逐个结点加入
        switch ( ch ) {
          case '(' :   p = new GenTreeNode <Type> ( q );    //建立子树, p->firstChild = q
                       r = st.GetTop( );    st.Pop( );      //从栈中退出前一结点
                       r->nextSibling = p;                  //插在前一结点 r 之后
                       st.Push( p );    st.Push( q );       //子树结点及子树根结点进栈
                       break;
          case ')' :   q->nextSibling = NULL;   st.pop( );  //子树建成，封闭链，退到上层
                       if ( st.IsEmpty ( ) == 0 ) q = st.GetTop( );  //栈不空，取上层链子树结点
                       else return 0;                       //栈空，无上层链，算法结束
                       break;
          case ',' :   break;
          default :    p = q;
                       if ( isupper (ch) ) q = new GenTreeNode ( 0, ch );  //大写字母，建根结点
```

```
                else q = new GenTreeNode ( 1, ch );                    //非大写字母，建数据结点
                p->nextSibling = q;                                    //链接
            }
            cin >> ch;
        }
    }
```

(2) 复制构造函数 用另一棵表示为广义表的树初始化一棵树；

```
GenTree :: GenTree ( const GenTree& t ) {                              //共有函数
    first = Copy ( t.first );
}

GenTreeNode* GenTree :: Copy ( GenTreeNode *ptr ) {
//私有函数，复制一个 ptr 指示的用广义表表示的子树
    GenTreeNode *q = NULL;
    if ( ptr != NULL ) {
        q = new GenTreeNode ( ptr->utype, NULL );
        switch ( ptr->utype ) {                                        //根据结点类型 utype 传送值域
          case 0 : q->RootData = ptr->RootData;   break;               //传送根结点数据
          case 1 : q->ChildData = ptr->ChildData;   break;             //传送子女结点数据
          case 2 : q->firstChild = Copy ( ptr->firstChild );   break;  //递归传送子树信息
        }
        q->nextSibling = Copy ( ptr->nextSibling );                    //复制同一层下一结点为头的表
    }
    return q;
}
```

(3) **operator** == ( ) 测试用广义表表示的两棵树是否相等

```
int operator == ( GenTree& t1, GenTree& t2 ) {
    //友元函数 ： 判两棵树 t1 与 t2 是否相等，若两表完全相等，函数返回 1，否则返回 0。
    return Equal ( t1.first, t2.first );
}

int Equal ( GenTreeNode *t1, GenTreeNode *t2 ) {
//是 GenTreeNode 的友元函数
    int x;
    if ( t1 == NULL && t2 == NULL ) return 1;                          //表 t1 与表 t2 都是空树，相等
    if ( t1 != NULL && t2 != NULL && t1->utype == t2->utype ) {        //两子树都非空且结点类型相同
        switch ( t1->utype ) {                                         //比较对应数据
          case 0 :  x = ( t1->RootData == t2->RootData ) ? 1 : 0;      //根数据结点
                    break;
          case 1 :  x = ( t1->ChildData == t2->ChildData ) ? 1 : 0;    //子女数据结点
                    break;
```

```
            case 2 : x = Equal ( t1->firstChild,    t2->firstChild );              //递归比较其子树
        }
        if ( x ) return Equal ( t1->nextSibling,    t2->nextSibling );
                        //相等，递归比较同一层的下一个结点；不等，不再递归比较
    }
    return 0;
}
```

(4) **operator** << ( ) 用广义表的形式输出一棵树

```
ostream& operator << ( ostream& out, GenTree& t ) {
//友元函数，将树 t 输出到输出流对象 out。
    t.traverse ( out, t.first );
    return out;
}

void GenTree :: traverse ( ostream& out, GenTreeNode * ptr ) {
//私有函数，广义树的遍历算法
    if ( ptr != NULL ) {
        if ( ptr->utype == 0 ) out << ptr->RootData << '(';                  //根数据结点
        else if ( ptr->utype == 1 ) {                                        //子女数据结点
                out << ptr->ChildData;
                if ( ptr->nextSibling != NULL ) out << ',';
            }
        else {                                                               //子树结点
                traverse ( ptr->firstChild );                                //向子树方向搜索
                if ( ptr->nextSibling != NULL ) out << ',';
        }
        traverse ( ptr->nextSibling );                                       //向同一层下一兄弟搜索
    }
    else out << ')';
}
```

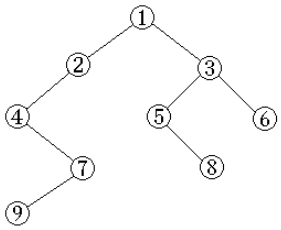
(5) 析构函数 清除一棵用广义表表示的树

```
GenTree :: ~ GenTree ( ) {
//用广义表表示的树的析构函数，假定 first ≠ NULL
    Remove ( first );
}

void GenTree :: Remove ( GenTreeNode *ptr ) {
    GenTreeNode * p;
    while ( ptr != NULL ) {
        p = ptr->nextSibling;
        if ( p->utype == 2 ) Remove ( p->firstChild );                       //在子树中删除
```

```
            ptr->nextSibling = p->nextSibling;    delete ( p );                    //释放结点 p
        }
    }
```

6-2 列出右图所示二叉树的叶结点、分支结点和每个结点的层次。



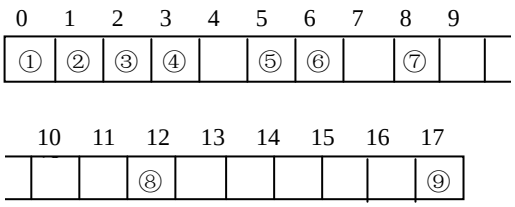
【解答】

二叉树的叶结点有⑥、⑧、⑨。分支结点有①、②、③、④、⑤、⑦。

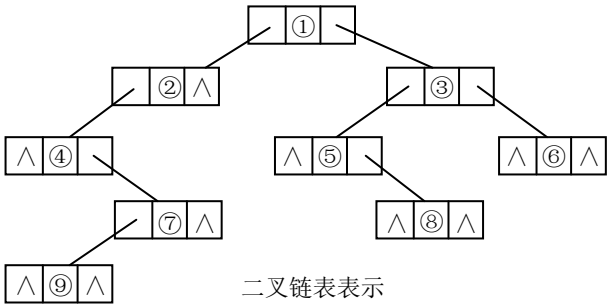
结点①的层次为 0；结点②、③的层次为 1；结点④、⑤、⑥的层次为 2；结点⑦、⑧的层次为 3；结点⑨的层次为 4。

6-3 使用 (1) 顺序表示和 (2) 二叉链表表示法，分别画出右图所示二叉树的存储表示。

【解答】



顺序表示



二叉链表表示

6-4 用嵌套类写出用链表表示的二叉树的类声明。

【解答】

```
#include <iostream.h>
template <class Type> class BinaryTree {
private:
    template <class Type> class BinTreeNode {
    public:
        BinTreeNode<Type> *leftChild, *rightChild;
        Type data;
    }
    Type RefValue;
    BinTreeNode<Type> * root;
    BinTreeNode<Type> *Parent ( BinTreeNode<Type> *start, BinTreeNode<Type> *current );
    int Insert ( BinTreeNode<Type> *current, const Type& item );
    int Find ( BinTreeNode<Type> *current, const Type& item ) const;
    void destroy ( BinTreeNode<Type> *current );
    void Traverse ( BinTreeNode<Type> *current, ostream& out ) const;
public:
    BinaryTree ( ) : root ( NULL ) { }
    BinaryTree ( Type value ) : RefValue ( value ), root ( NULL ){ }
    ~BinaryTree ( ) { destroy (root); }
    BinTreeNode ( ) : leftChild ( NULL ), rightChild ( NULL ) { }
    BinTreeNode ( Type item ) : data ( item ), leftChild ( NULL ), rightChild ( NULL ) { }
```

```
    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; }
    int IsEmpty ( ) { return root == NULL ? 1 : 0; }
    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> * RighttChild ( BinTreeNode<Type> *current )
      { return current != NULL ? current->rightChild : NULL; }
    int Insert ( const Type& item );
    BinTreeNode<Type> * Find ( const Type& item );
    BinTreeNode<Type> * GetRoot ( ) const { return root; }
    friend istream& operator >> ( istream& in, BinaryTree<Type>& Tree );          //输入二叉树
    friend ostream& operator << ( ostream& out, BinaryTree<Type>& Tree );         //输出二叉树
}
```

6-5 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？它有多少个叶结点？多少个分支结点？高度最大的树的高度是多少？它有多少个叶结点？多少个分支结点？

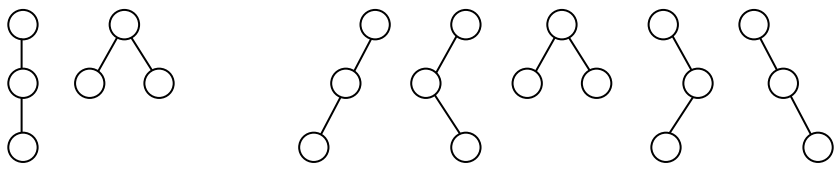
【解答】

结点个数为 n 时，高度最小的树的高度为 1，有 2 层；它有 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。

6-6 试分别画出具有 3 个结点的树和 3 个结点的二叉树的所有不同形态。

【解答】

具有 3 个结点的树　　具有 3 个结点的二叉树



6-7 如果一棵树有 n_1 个度为 1 的结点，有 n_2 个度为 2 的结点，… ，n_m 个度为 m 的结点，试问有多少个度为 0 的结点？试推导之。

【解答】

总结点数 $n = n_0 + n_1 + n_2 + \cdots + n_m$

总分支数 $e = n-1 = n_0 + n_1 + n_2 + \cdots + n_m - 1$

$= m*n_m + (m-1)*n_{m-1} + \cdots + 2*n_2 + n_1$

则有 $n_0 = \left(\sum_{i=2}^{m} (i-1)n_i \right) + 1$

6-8 试分别找出满足以下条件的所有二叉树:

(1) 二叉树的前序序列与中序序列相同;

(2) 二叉树的中序序列与后序序列相同;

(3) 二叉树的前序序列与后序序列相同。

【解答】

(1) 二叉树的前序序列与中序序列相同：空树或缺左子树的单支树；

(2) 二叉树的中序序列与后序序列相同：空树或缺右子树的单支树；

(3) 二叉树的前序序列与后序序列相同：空树或只有根结点的二叉树。

6-9 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的个数。

(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) 统计二叉树中叶结点个数

```
int BinaryTree<Type> :: leaf ( BinTreeNode<Type> * ptr ) {
    if ( ptr == NULL ) return 0;
    else if ( ptr->leftChild == NULL && ptr->rightChild == NULL ) return 1;
    else return leaf ( ptr->leftChild ) + leaf ( ptr->rightChild );
}
```

(2) 交换每个结点的左子女和右子女

```
void BinaryTree<Type> :: exchange ( BinTreeNode<Type> * ptr ) {
    BinTreeNode<Type> * temp;
    if ( ptr->leftChild != NULL || ptr->rightChild != NULL ) {
        temp = ptr->leftChild;
        ptr->leftChild = ptr->rightChild;
        ptr->rightChild = temp;
        exchange ( ptr->leftChild );
        exchange ( ptr->rightChild );
    }
}
```

6-10 一棵高度为 h 的满 k 叉树有如下性质: 第 h 层上的结点都是叶结点, 其余各层上每个结点都有 k 棵非空子树, 如果按层次自顶向下, 同一层自左向右, 顺序从 1 开始对全部结点进行编号, 试问:

(1) 各层的结点个数是多少?

(2) 编号为 i 的结点的父结点(若存在)的编号是多少?

(3) 编号为 i 的结点的第 m 个孩子结点(若存在)的编号是多少?

(4) 编号为 i 的结点有右兄弟的条件是什么? 其右兄弟结点的编号是多少?

(5) 若结点个数为 n, 则高度 h 是 n 的什么函数关系?

【解答】

(1) k^i (i = 0, 1, ……, h)

(2) $\left\lfloor \frac{i+k-2}{k} \right\rfloor$

(3) (i−1)*k + m + 1

(4) (i−1) % k ≠ 0 或 $i \le \left\lfloor \frac{i+k-2}{k} \right\rfloor * k$ 时有右兄弟，右兄弟为 i + 1。

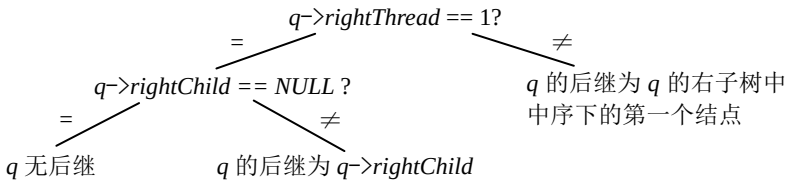
(5) $h = \log_k (n*(k-1)+1)-1$ (n = 0 时 h = −1)

6−11 试用判定树的方法给出在中序线索化二叉树上：

【解答】

(1) 搜索指定结点的在中序下的后继。

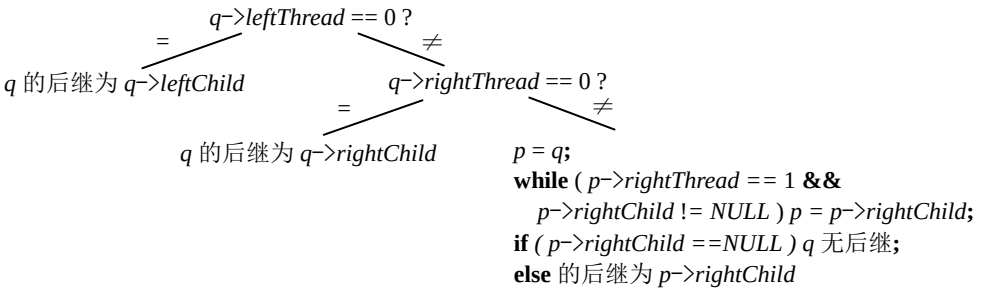
设指针 q 指示中序线索化二叉树中的指定结点，指针 p 指示其后继结点。



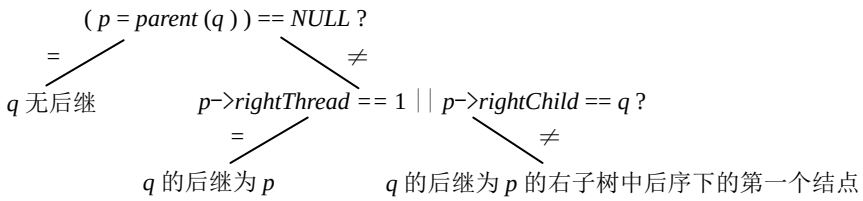
找 q 的右子树中在中序下的第一个结点的程序为：

```
p = q->rightChild;
while ( p->leftThread == 0 ) p = p->leftChild;   // p 即为 q 的后继
```

(2) 搜索指定结点的在前序下的后继。



(3) 搜索指定结点的在后序下的后继。



可用一段遍历程序来实现寻找 p 的右子树中后序下的第一个结点：即该子树第一个找到的叶结点。找到后立即返回它的地址。

6−12 已知一棵完全二叉树存放于一个一维数组 T[n]中，T[n]中存放的是各结点的值。试设计一个算法，从 T[0]开始顺序读出各结点的值，建立该二叉树的二叉链表表示。

【解答】

```
template <class Type>                                   //建立二叉树
istream& operator >> ( istream& in, BinaryTree<Type>& t ) {
    int n;
    cout << "Please enter the number of node : ";   in >> n;
```

```
    Type *A = new Type[n+1];
    for ( int i = 0; i < n; i++ ) in >> A[i];
    t. ConstructTree( T, n, 0, ptr );                    //以数组建立一棵二叉树
    delete [ ] A;
    return in;
}

template <class Type>
void BinaryTree<Type> :: ConstructTree ( Type T[ ], int n, int i, BinTreeNode<Type> *& ptr ) {
//私有函数 ：将用 T[n]顺序存储的完全二叉树，以 i 为根的子树转换成为用二叉链表表示的
//以 ptr 为根的完全二叉树。利用引用型参数 ptr 将形参的值带回实参。
    if ( i >= n ) ptr = NULL;
    else {
        ptr = new BinTreeNode<Type> ( T[i] );                //建立根结点
        ConstructTree ( T, n, 2*i+1, ptr->leftChild );
        ConstructTree ( T, n, 2*i+2, ptr->rightChild );
    }
}
```

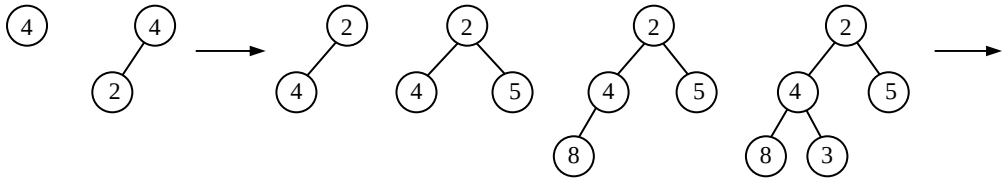
6-13 试编写一个算法，把一个新结点 l 作为结点 s 的左子女插入到一棵中序线索化二叉树中，s 原来的左子女变成 l 的左子女。

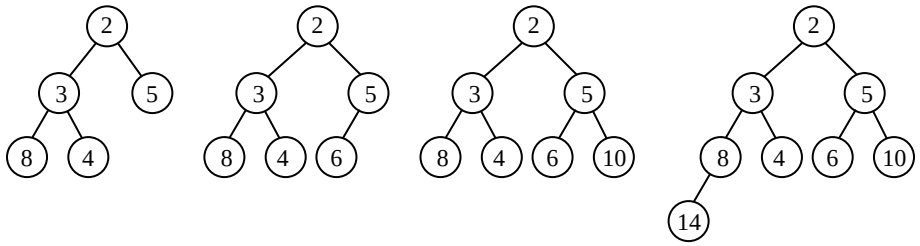
【解答】

```
template<class Type>
void ThreadTree<Type> :: leftInsert ( ThreadNode<Type> *s, ThreadNode<Type> * l ) {
  if ( s != NULL && l != NULL ) {
    l->leftChild = s->leftChild;   l->leftThread = s->leftThread;    //预先链接
    l->rightChild = s;   l->rightThread = 1;                 //新插入结点*l 的后继为*s
    s->leftChild = l;   s->leftThread = 0;                   //*l 成为*s 的左子女
    if ( l->leftThread == 0 ) {                              //*l 的左子女存在
      ThreadNode<Type> * p = l->leftChild;
      while ( p->rightThread == 0 )                          //找*l 的左子树中序下最后一个结点
        p = p->rightChild;
      p->rightChild = l;                                     //该结点的后继为*l
    }
  }
}
```

6-14 写出向堆中加入数据 4, 2, 5, 8, 3, 6, 10, 14 时，每加入一个数据后堆的变化。

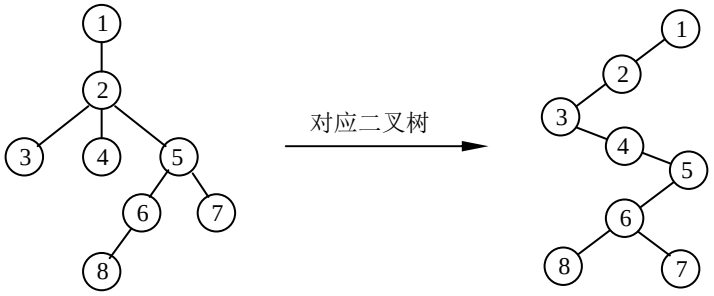
【解答】以最小堆为例：





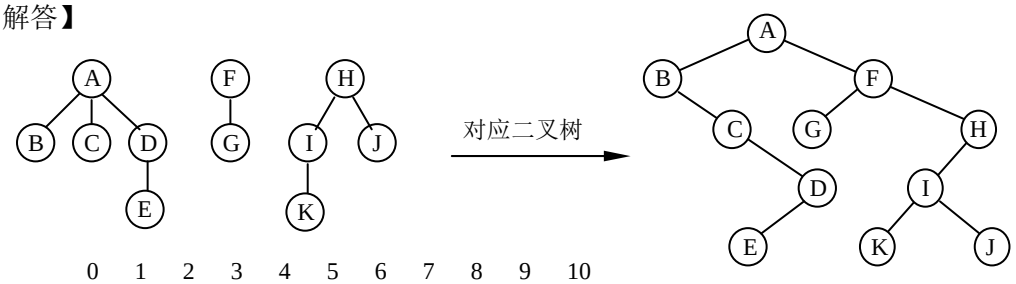
6-16 请画出右图所示的树所对应的二叉树。

【解答】



6-17 在森林的二叉树表示中，用 llink 存储指向结点第一个子女的指针，用 rlink 存储指向结点下一个兄弟的指针，用 data 存储结点的值。如果我们采用静态二叉链表作为森林的存储表示，同时按森林的先根次序依次安放森林的所有结点，则可以在它们的结点中用只有一个二进位的标志 ltag 代替 llink，用 rtag 代替 rlink。并设定若 ltag = 0，则该结点没有子女，若 ltag ≠ 0，则该结点有子女；若 rtag = 0，则该结点没有下一个兄弟，若 rtag ≠ 0，则该结点有下一个兄弟。试给出这种表示的结构定义，并设计一个算法，将用这种表示存储的森林转换成用 llink－rlink 表示的森林。

【解答】



	0	1	2	3	4	5	6	7	8	9	10
llink	1	-1	-1	4	-1	6	-1	8	9	-1	-1
data	A	B	C	D	E	F	G	H	I	K	J
rlink	5	2	3	-1	-1	7	-1	-1	10	-1	-1

森林的左子女-右兄弟表示的静态二叉链表

	0	1	2	3	4	5	6	7	8	9	10
ltag	1	0	0	1	0	1	0	1	1	0	0
data	A	B	C	D	E	F	G	H	I	K	J
rtag	1	1	1	0	0	1	0	0	1	0	0

森林的双标记表示

(1) 结构定义

```
template <class Type> class LchRsibNode {        //左子女-右兄弟链表结点类的定义
protected:
  Type data;                                     //结点数据
  int llink, rlink;                              //结点的左子女、右兄弟指针
```

```
public:
   LchRsibNode ( ) : llink(NULL), rlink(NULL) { }
   LchRsibNode ( Type x ) : llink(NULL), rlink(NULL), data(x) { }
}

template <class Type> class DoublyTagNode {       //双标记表结点类的定义
protected:
   Type data;                                     //结点数据
   int ltag, rtag;                                //结点的左子女、右兄弟标记
public:
   DoublyTagNode ( ) : ltag(0), rtag(0) { }
   DoublyTagNode ( Type x ) : ltag(0), rtag(0), data(x) { }
}

template <class Type> class staticlinkList        //静态链表类定义
      : public LchRsibNode<Type>, public DoublyTagNode <Type>{
private:
   LchRsibNode<Type> *V;                          //存储左子女-右兄弟链表的向量
   DoublyTagNode <Type> *U;                       //存储双标记表的向量
   int MaxSize, CurrentSize;                      //向量中最大元素个数和当前元素个数
public:
   dstaticlinkList ( int Maxsz ) : MaxSize ( Maxsz ), CurrentSize (0) {
      V = new LchRsibNode <Type> [Maxsz];
      U = new DoublyTagNode <Type> [Maxsz];
   }
```

(2) 森林的双标记先根次序表示向左子女−右兄弟链表先根次序表示的转换

```
void staticlinkList<Type> :: DtagF−LchRsibF ( ) {
    Stack<int> st;    int k;
    for ( int i = 0;    i < CurrentSize;    i++ ) {
        switch ( U[i].ltag ) {
          case 0 : switch ( U[i].rtag ) {
                     case 0 :   V[i].llink = V[i].rlink = −1;
                                if ( st.IsEmpty ( ) == 0 )
                                   { k = st.GetTop ( );    st.Pop ( );    V[k].rlink = i + 1; }
                                break;
                     case 1 :   V[i].llink = −1;    V[i].rlink = i + 1;    break;
                   }
                   break;
          case 1 : switch ( U[i].rtag ) {
                     case 0 :   V[i].llink = i + 1;    V[i].rlink = −1;    break;
                     case 1 :   V[i].llink = i + 1;    st.Push ( i );
                   }
```

```
        }
    }
}
```

6-18 二叉树的双序遍历(Double-order traversal)是指：对于二叉树的每一个结点来说，先访问这个结点，再按双序遍历它的左子树，然后再一次访问这个结点，接下来按双序遍历它的右子树。试写出执行这种双序遍历的算法。

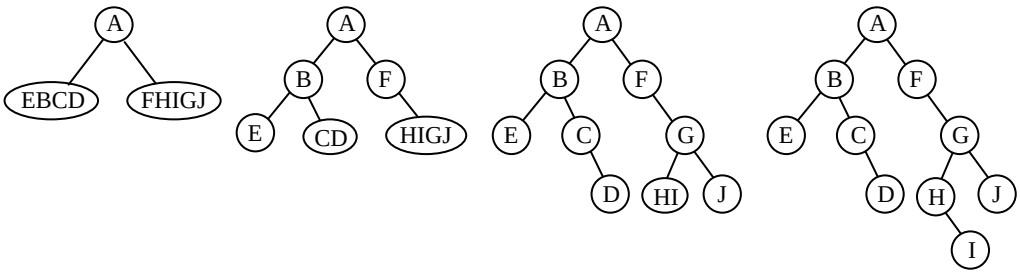
【解答】

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

6-19 已知一棵二叉树的前序遍历的结果是 ABECDFGHIJ, 中序遍历的结果是 EBCDAFHIGJ, 试画出这棵二叉树。

【解答】

当前序序列为 ABECDFGHIJ，中序序列为 EBCDAFHIGJ 时，逐步形成二叉树的过程如下图所示：

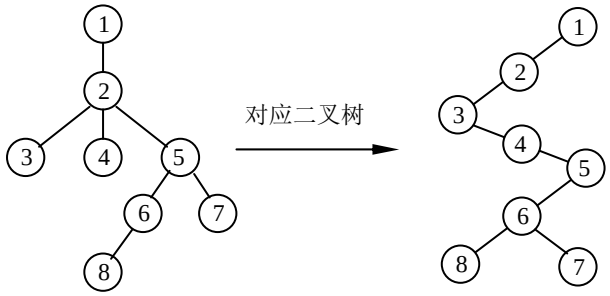


6-20 已知一棵树的先根次序遍历的结果与其对应二叉树表示(长子-兄弟表示)的前序遍历结果相同, 树的后根次序遍历结果与其对应二叉树表示的中序遍历结果相同。试问利用树的先根次序遍历结果和后根次序遍历结果能否唯一确定一棵树? 举例说明。

【解答】

因为给出二叉树的前序遍历序列和中序遍历序列能够唯一地确定这棵二叉树，因此，根据题目给出的条件，利用树的先根次序遍历结果和后根次序遍历结果能够唯一地确定一棵树。

例如，对于题 6-16 所示的树

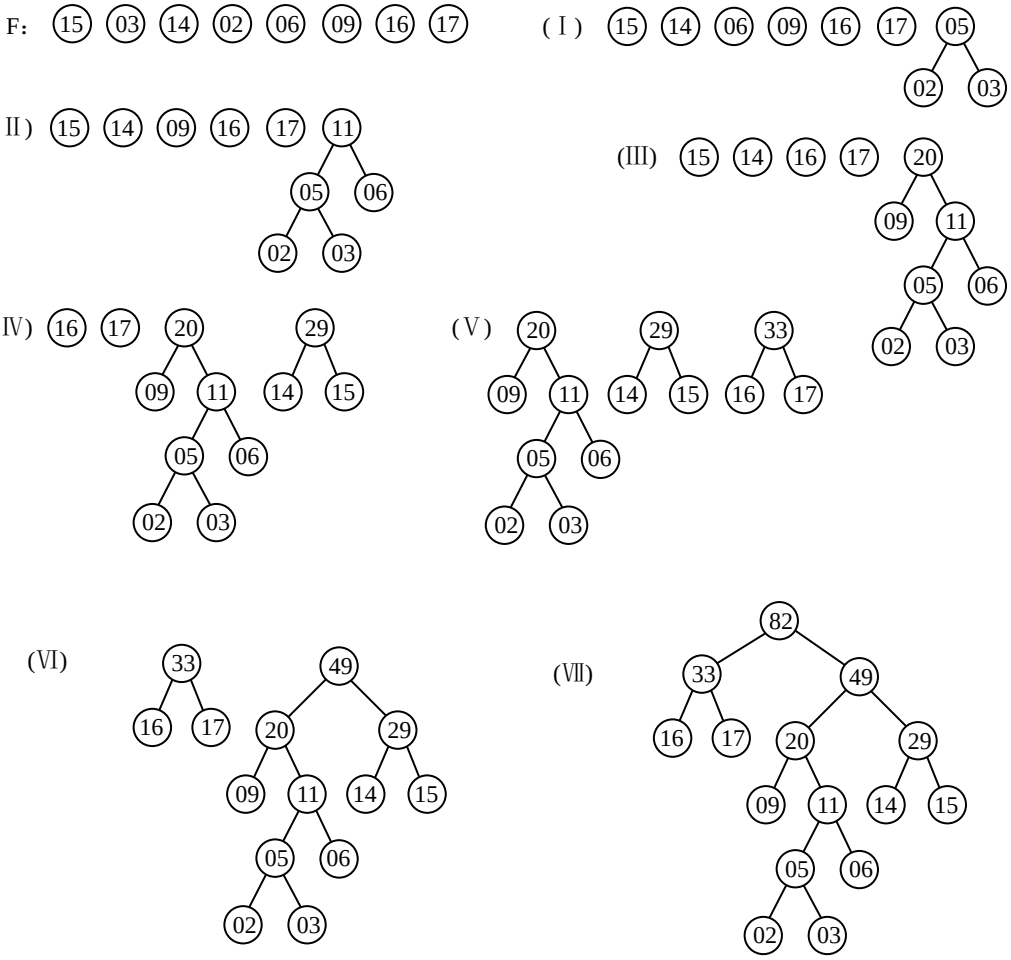


对应二叉树的前序序列为 1, 2, 3, 4, 5, 6, 8, 7；中序序列为 3, 4, 8, 6, 7, 5, 2, 1。

原树的先根遍历序列为1, 2, 3, 4, 5, 6, 8, 7；后根遍历序列为 3, 4, 8, 6, 7, 5, 2, 1。

6-21 给定权值集合{15, 03, 14, 02, 06, 09, 16, 17}，构造相应的霍夫曼树，并计算它的带权外部路径长度。

【解答】



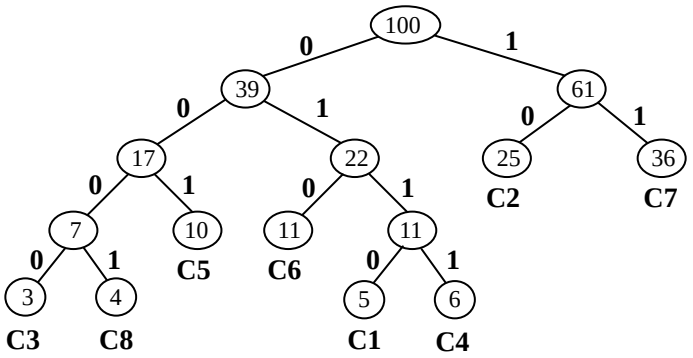
此树的带权路径长度 WPL = 229。

6-22 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】已知字母集 { c1, c2, c3, c4, c5, c6, c7, c8 }，频率 {5, 25, 3, 6, 10, 11, 36, 4 }，则 Huffman 编码为

c1	c2	c3	c4	c5	c6	c7	c8
0110	10	0000	0111	001	010	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

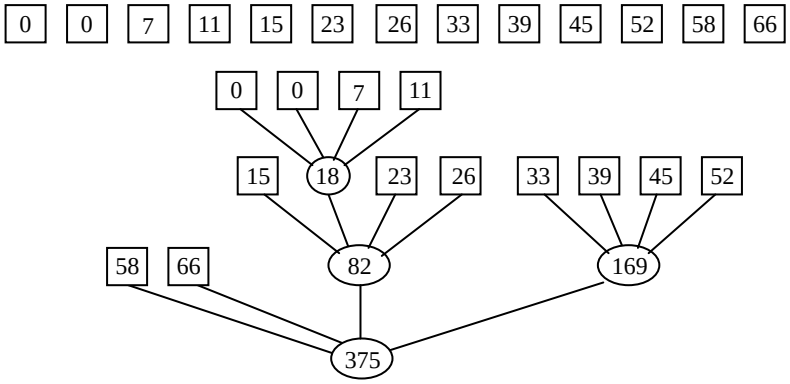


6-23 给定一组权值: 23, 15, 66, 07, 11, 45, 33, 52, 39, 26, 58, 试构造一棵具有最小带权外部路径长度的扩充 4 叉树，要求该 4 叉树中所有内部结点的度都是 4, 所有外部结点的度都是 0。这棵扩充 4 叉树的带

权外部路径长度是多少?

【解答】权值个数 n = 11，扩充 4 叉树的内结点的度都为 4，而外结点的度都为 0。设内结点个数为 n_4，外结点个数为 n_0，则可证明有关系 $n_0 = 3 * n_4 + 1$。由于在本题中 $n_0 = 11 \neq 3 * n_4 + 1$，需要补 2 个权值为 0 的外结点。此时内结点个数 $n_4 = 4$。

仿照霍夫曼树的构造方法来构造扩充 4 叉树，每次合并 4 个结点。



此树的带权路径长度 WPL = 375 + 82 + 169 + 18 = 644。