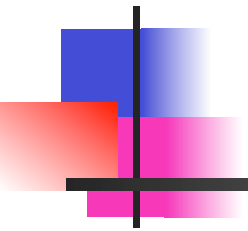


第四章　二叉树

北京大学软件与微电子学院

主要内容

- **4.1** 二叉树的概念
- **4.2** 二叉树的主要性质
- **4.3** 二叉树的抽象数据类型
- **4.4** 周游二叉树
- **4.5** 二叉树的实现
- **4.6** 二叉搜索树
- **4.7** 堆与优先队列
- **4.8 Huffman**编码树

4.1 二叉树的概念

- **4.1.1** 二叉树的定义及相关概念
- **4.1.2** 满二叉树

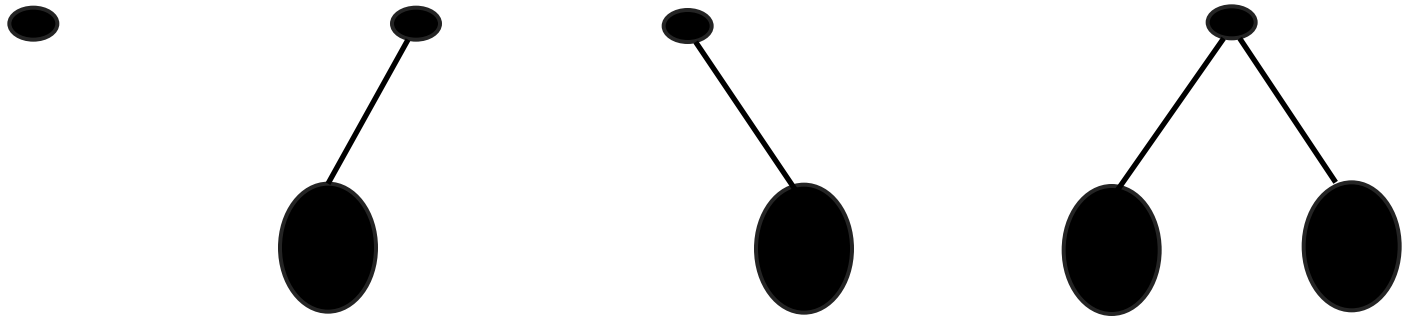
  完全二叉树

  扩充二叉树

二叉树的定义

- 二叉树由结点的有限集合构成:
  - 或者为空集
  - 或者由一个根结点及两棵不相交的分别称作这个根的左子树和右子树的二叉树(它们也是结点的集合)组成
- 这是个递归的定义。二叉树可以是空集合，因此根可以有空的左子树或右子树，或者左右子树皆为空

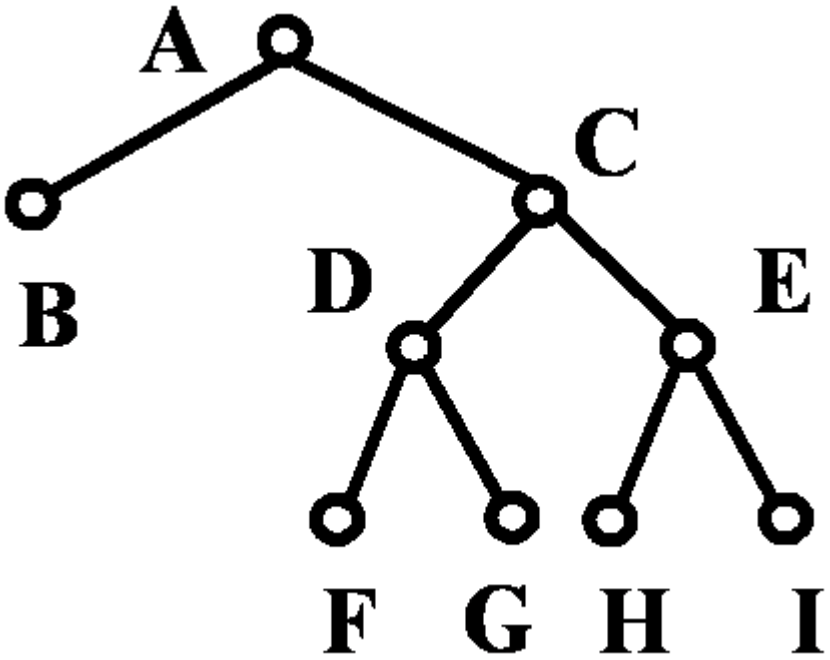
二叉树的五种基本形态



(a)空 (b)独根 (c)空右 (d)空左 (e)左右都不空

满二叉树

- 如果一棵二叉树的任何结点，或者是树叶，或者恰有两棵非空子树，则此二叉树称作满二叉树

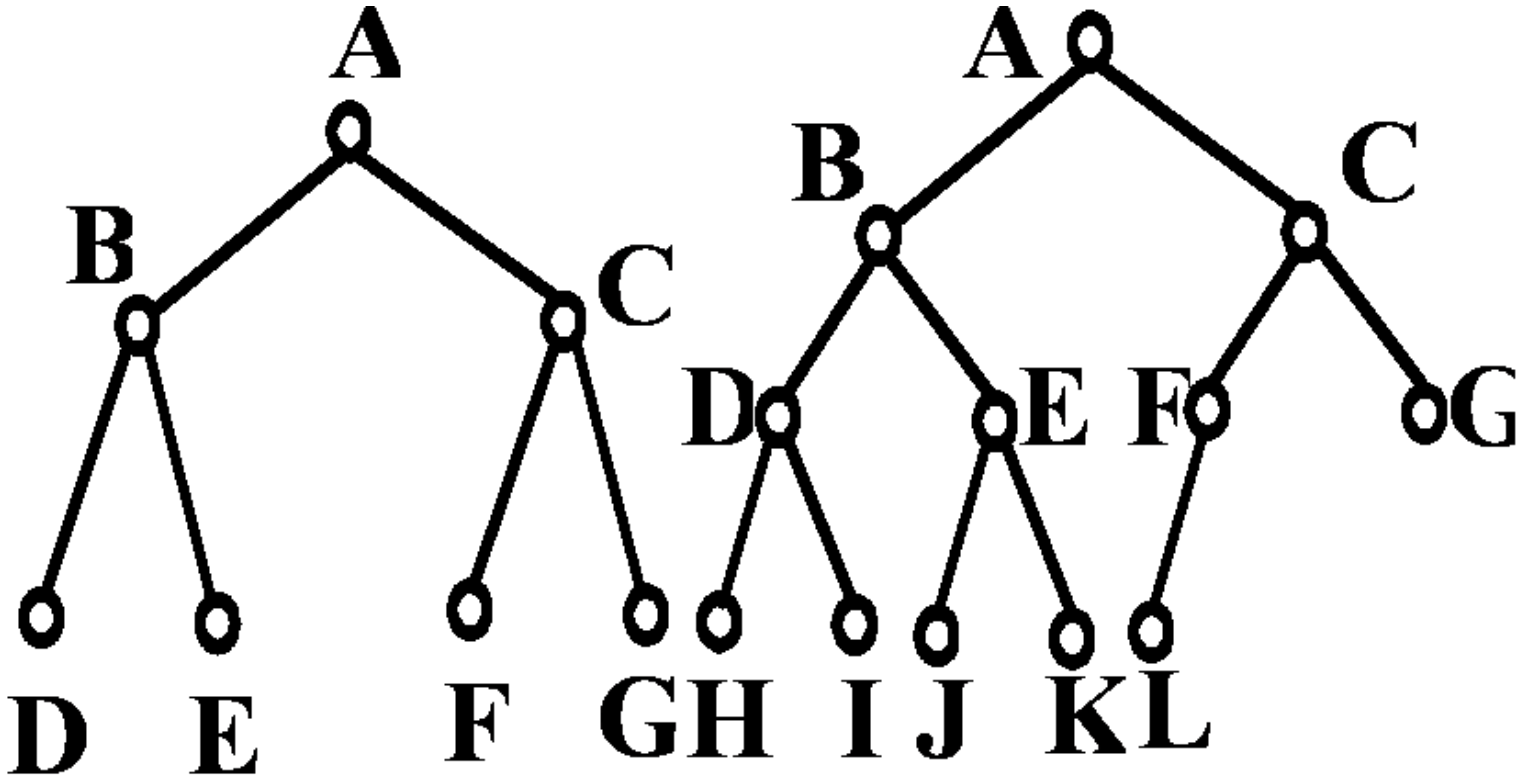


完全二叉树

- 若一颗二叉树
  - 最多只有最下面的两层结点度数可以小于2
  - 最下面一层的结点都集中在该层最左边的若干位置上，

  则称此二叉树为完全二叉树
- 在许多算法和算法分析中都明显地或隐含地用到完全二叉树的概念

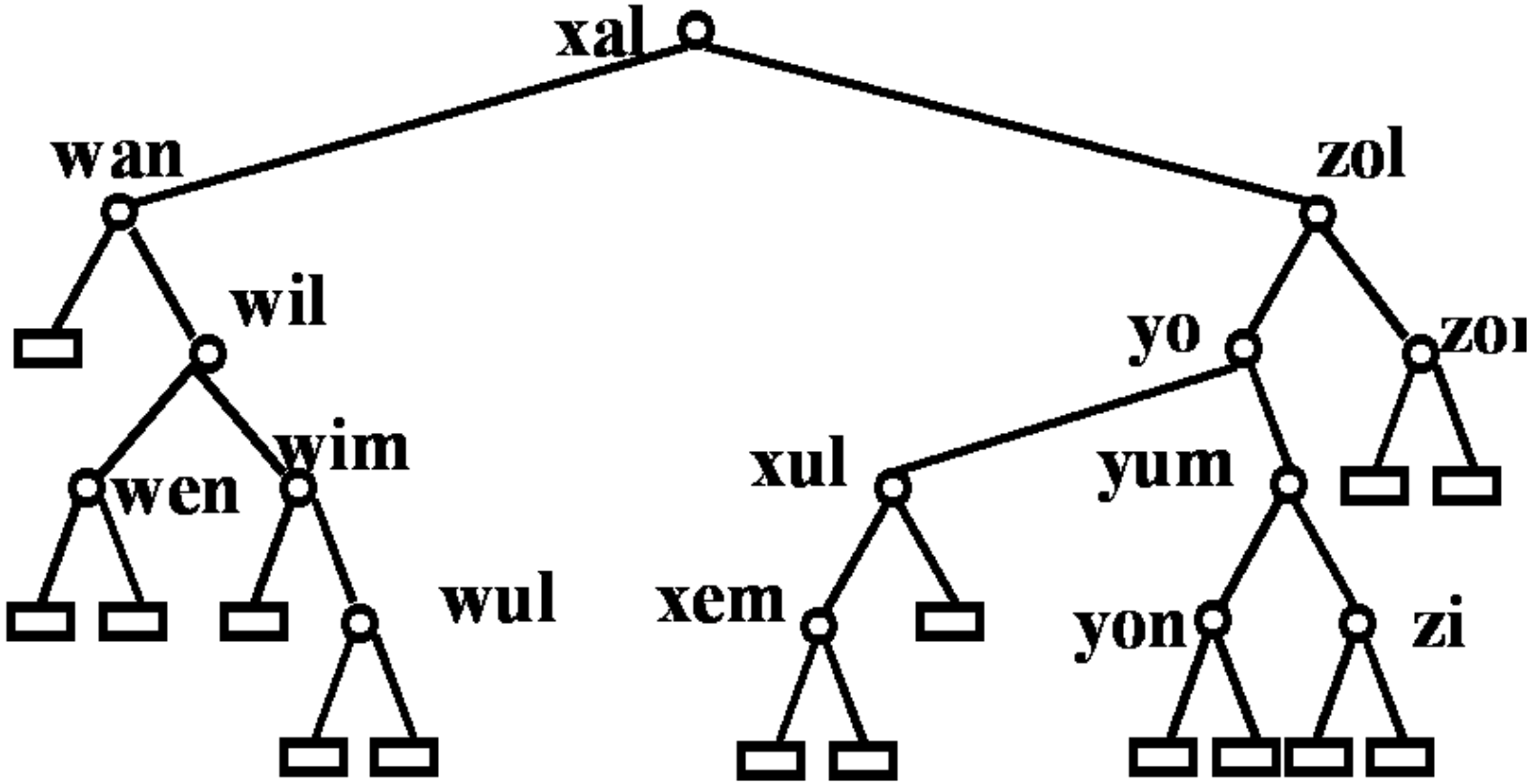
完全二叉树



扩充二叉树

- 当二叉树里出现空的子树时，就增加新的、特殊的结点——空树叶
  - 对于原来二叉树里度数为1的分支结点，在它下面增加一个空树叶
  - 对于原来二叉树的树叶，在它下面增加两个空树叶
- 扩充的二叉树是满二叉树，新增加的空树叶(外部结点)的个数等于原来二叉树的结点(内部结点)个数加1

扩充二叉树



扩充二叉树

- 外部路径长度**E**　从扩充的二叉树的根到每个外部结点的路径长度之和
- 内部路径长度**I**　扩充的二叉树里从根到每个内部结点的路径长度之和
- **E**和**I**两个量之间的关系为　**E=I+2n**

4.2 二叉树的主要性质

1. 满二叉树定理：非空满二叉树树叶数等于其分支结点数加1。

证明：设二叉树结点数为n，叶结点数为m，分支结点数为b。

有n（总结点数= m（叶)+b（分支）　　（公式4.1）

∵ 每个分支，恰有两个子结点（满），故有2*b条边；一颗二叉树，除根结点外，每个结点都恰有一条边联接父结点，故共有n-1条边。即n - 1 = 2b　　（公式4.2）

∴由(公式4.1)，(公式4.2)得 n-1=m+b-1 = 2b，得出

m(叶) = b（分支）+ 1

4.2 二叉树的性质

2. **满二叉树定理推论**：一个非空二叉树的空子树(指针)数目等于其结点数加1。

证明：设二叉树T，将其所有空子树换为树叶，记新 的扩充满二叉树为T’ 。所有原来T的结点现在是T’ 的分支结点。根据满二叉树定理，新添加的树叶数目等于T结点个数加1。而每个新添加的树叶对应T的一个空子树。

因此T中空子树数目等于T中结点数加1。

4.2 二叉树的性质

3. 任何一颗二叉树，度为0的结点比度为2的结点多一个

证明：设有n个结点的二叉树的度为0、1、2的结点数分别为=n_0，n_1，n_2，则

$$n = n_0 + n_1 + n_2 \qquad \text{（公式4.3）}$$

设边数为e。因为除根以外，每个结点都有一条边进入，故 $n = e + 1$。由于这些边是有度为1和2的的结点射出的，因此$e = n_1 + 2 \cdot n_2$，于是

$$n = e + 1 = n_1 + 2 \cdot n_2 + 1 \qquad \text{（公式4.4）}$$

因此由公式（1）（2）得

$$n_0 + n_1 + n_2 = n_1 + 2 \cdot n_2 + 1$$

即 $n_0 = n_2 + 1$

4.2 二叉树的性质

- **4.** 二叉树的第**i**层（根为第**0**层，**i≥1**）最多有2^i个结点
- **5.** 高度为**k(**深度为**k-1**。只有一个根结点的二叉树的高度为**1**，深度为**0)**的二叉树至多有2^k-1个结点
- **6.** 有**n**个结点（**n>0**）的完全二叉树的高度为$\lceil \log_2 (n+1) \rceil$ ($\lceil \log_2 (n+1) \rceil - 1$)

4.2 二叉树的性质

- 二叉树的高度定义为二叉树中层数最大的叶结点的层数加1
- 二叉树的深度定义为二叉树中层数最大的叶结点的层数

4.3 二叉树的抽象数据类型

- 定义了二叉树的逻辑结构之后，我们需要考虑在二叉树逻辑结构之上的各种可能运算，这些运算应该适合二叉树的各种应用 ：
  - 二叉树的某些运算是针对整棵树的
    - 初始化二叉树
    - 合并两棵二叉树
  - 二叉树的大部分运算都是围绕结点进行的
    - 访问某个结点的左子结点、右子结点、父结点
    - 访问结点存储的数据。

4.3 二叉树的抽象数据类型

- 二叉树结点抽象数据类型**BinaryTreeNode**是带有参数 **T** 的模板，**T**是存储在结点中的数据类型
  - 每个元素结点都有**leftchild()**和**rightchild()**左右子结点结构
  - 另外每个结点还包含一个数据域**value()**。
- 为了强调抽象数据类型与存储无关，我们并没有具体规定该抽象数据类型的存储方式

4.3 二叉树的抽象数据类型

```
template <class T>
  class BinaryTreeNode
  {   //申明二叉树为结点类的友元类，便于访问私有
      //数据成员
      friend class BinaryTree;
            private:
            //二叉树结点数据域
            T  element;
       // 实现时，可以补充private的左子结点
       //右子结点定义
```

4.3 二叉树的抽象数据类型

```
public:
   BinaryTreeNode();                          //缺省构造函数
   BinaryTreeNode(const T& ele);//拷贝构造函数
   //给定了结点值和左右子树的构造函数
   BinaryTreeNode(const T& ele,
                  BinaryTreeNode<T>* l,
                        BinaryTreeNode<T>* r);
    T  value() const;//返回当前结点的数据
    //返回当前结点指向左子树
    BinaryTreeNode<T>*  leftchild() const;
    //返回当前结点指向右子树
    BinaryTreeNode<T>*  rightchild() const;
```

4.3 二叉树的抽象数据类型

```
    //设置当前结点的左子树
     void  setLeftchild(BinaryTreeNode<T>*) ;
    //设置当前结点的右子树
     void  setRightchild(BinaryTreeNode<T>*) ;
    //设置当前结点的数据域
     void  setValue(const T& val);
    //判定当前结点是否为叶结点,若是返回true
     bool  isLeaf() const;
    //重载赋值操作符
     BinaryTreeNode<T>& operator=
          (const BinaryTreeNode<T>& Node);
};
```

4.3 二叉树的抽象数据类型

- 二叉树的抽象数据类型的C++定义BinaryTree，没有具体规定该抽象数据类型的存储方式

```
template <class T>
class BinaryTree {
 private:
    //二叉树根结点指针
    BinaryTreeNode<T>*  root;
    //从二叉树的root结点开始
    //查找current结点的父结点
    BinaryTreeNode<T>*
    GetParent(BinaryTreeNode<T>* root,
    BinaryTreeNode<T>* current);
```

4.3 二叉树的抽象数据类型

```
public:
  BinaryTree(root=NULL);          //构造函数
  ~BinaryTree() {DeleteBinaryTree(root);};//析构函数
  bool isEmpty() const;           //判定二叉树是否为空树
  //返回二叉树根结点
  BinaryTreeNode<T>* Root(){return root;};
  //返回current结点的父结点
  BinaryTreeNode<T>* Parent(BinaryTreeNode<T>*
                                    current);
  //返回current结点的左兄弟
   BinaryTreeNode<T>* LeftSibling(
                          BinaryTreeNode<T>* current);
```

4.3 二叉树的抽象数据类型

```
//返回current结点的右兄弟
BinaryTreeNode<T>* RightSibling(
                        BinaryTreeNode<T>* current);
// 以elem作为根结点，leftTree作为树的左子树,
//rightTree作为树的右子树，构造一棵新的二叉树
void CreateTree(const T& elem,
                BinaryTree<T>& leftTree,
                        BinaryTree<T>& rightTree);
//前序周游二叉树或其子树
void PreOrder(BinaryTreeNode<T>* root);
//中序周游二叉树或其子树
void InOrder(BinaryTreeNode<T>* root);
```

4.3 二叉树的抽象数据类型

```
    //后序周游二叉树或其子树
    void PostOrder(BinaryTreeNode<T>* root);
    //按层次周游二叉树或其子树
    void LevelOrder(BinaryTreeNode<T>* root);
    //删除二叉树或其子树
    void DeleteBinaryTree(BinaryTreeNode<T>*
    root);
};
```

4.4 周游二叉树

- 周游　系统地访问数据结构中的结点。每个结点都正好被访问到一次。
- 周游一棵二叉树的过程实际上就是把二叉树的结点放入一个线性序列的过程，或者说把二叉树进行线性化

4.4 周游二叉树

- 二叉树周游
  - **4.4.1** 深度优先周游二叉树
  - **4.4.2** 非递归深度优先周游二叉树
  - **4.4.3** 广度优先周游二叉树

深度优先周游二叉树

我们变换一下根结点的周游顺序，可以得到以下三种方案：

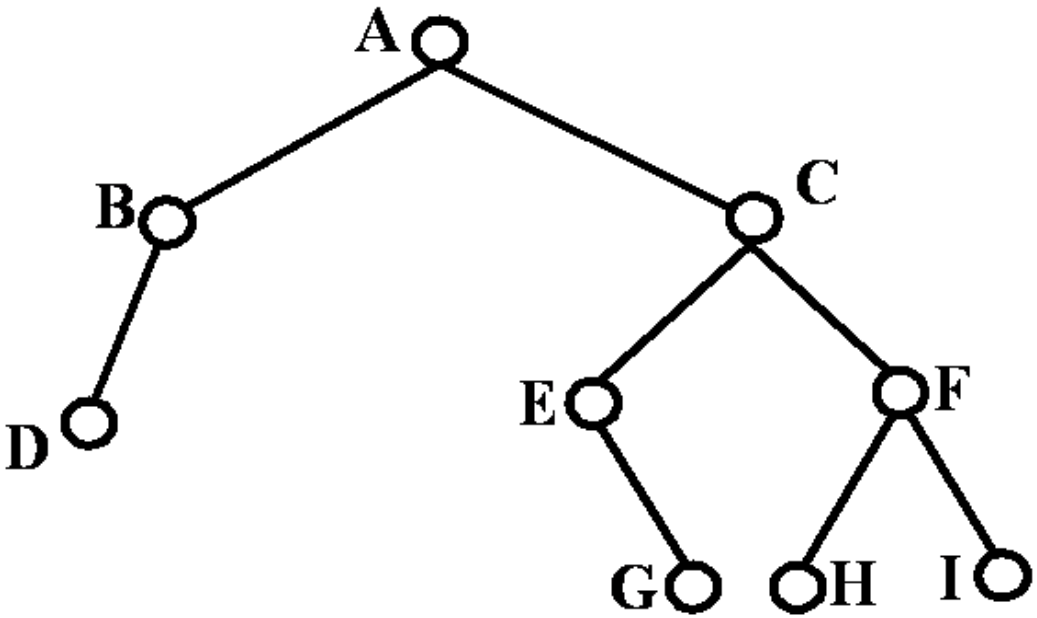
① 前序周游(tLR次序)：访问根结点；前序周游左子树；前序周游右子树。

② 中序周游(LtR次序)：中序周游左子树；访问根结点；中序周游右子树。

③ 后序周游(LRt次序)：后序周游左子树；后序周游右子树；访问根结点。

深度优先周游二叉树

- 深度周游如下二叉树



深度优先周游二叉树

- 深度周游二叉树结果
  - ① 前序周游：**ABDCEGFHI**
  - ② 中序周游：**DBAEGCHFI**
  - ③ 后序周游：**DBGEHIFCA**

深度优先周游二叉树（递归实现）

```
template<class T>
void BinaryTree<T>::DepthOrder
   (BinaryTreeNode<T>* root)  {
        if(root!=NULL){
        Visit(root);            //前序
        DepthOrder(root->leftchild());   //访问左子树
        Visit(root);            //中序
        DepthOrder(root->rightchild());//访问右子树
        Visit(root);            //后序
        }
}
```

非递归深度优先周游二叉树

- 栈是实现递归的最常用的结构
- 深度优先二叉树周游是递归定义的
- 利用一个栈来记下尚待周游的结点或子树，以备以后访问。

非递归前序周游二叉树

- 思想：
  - 遇到一个结点，就访问该结点，并把此结点推入栈中，然后下降去周游它的左子树；
  - 周游完它的左子树后，从栈顶托出这个结点，并按照它的右链接指示的地址再去周游该结点的右子树结构。

```
template<class T>  void
  BinaryTree<T>::PreOrderWithoutRecusion
  (BinaryTreeNode<T>* root)
  //非递归前序遍历二叉树或其子树
```

非递归前序周游二叉树

```
{
   using std::stack;        //使用STL中的stack
   stack<BinaryTreeNode<T>* > aStack;
   BinaryTreeNode<T>* pointer=root; while(!
   aStack.empty()||pointer){
    if(pointer){
    //访问当前结点
    Visit(pointer->value());
    //当前结点地址入栈
    aStack.push(pointer);
```

非递归前序周游二叉树

```
    //当前链接结构指向左孩子
    pointer=pointer->leftchild();
    }
    else {//左子树访问完毕，转向访问右子树
          pointer=aStack.top();
          aStack.pop();          //栈顶元素退栈
          //当前链接结构指向右孩子
          pointer=pointer->rightchild();
           }
    } //end while
}
```

非递归中序周游二叉树

- 思想：
  - 遇到一个结点，就把它推入栈中，并去周游它的左子树
  - 周游完左子树后，从栈顶托出这个结点并访问之，然后按照它的右链接指示的地址再去周游该结点的右子树。
- **template<class T> void BinaryTree<T>::InOrderWithoutRecusion(BinaryTreeNode<T>* root)**

  //非递归中序遍历二叉树或其子树

非递归中序周游二叉树

```
{
  using std::stack;   //使用STL中的stack
  stack<BinaryTreeNode<T>* > aStack;
  BinaryTreeNode<T>* pointer=root;
  while(!aStack.empty()||pointer) {
   if(pointer){
   //当前结点地址入栈
   aStack.push(pointer);
   //当前链接结构指向左孩子
   pointer=pointer->leftchild();
```

非递归中序周游二叉树

```
    }//end if
    else {//左子树访问完毕，转向访问右子树
            pointer=aStack.top();
            Visit(pointer->value());//访问当前结点
            //当前链接结构指向右孩子
            pointer=pointer->rightchild();
            aStack.pop();                    //栈顶元素退栈
            }//end else
  } //end while
}
```

非递归后序周游二叉树

- 思想：
  - 遇到一个结点，把它推入栈中，周游它的左子树
  - 周游结束后，还不能马上访问处于栈顶的该结点，而是要再按照它的右链接结构指示的地址去周游该结点的右子树
  - 周游遍右子树后才能从栈顶托出该结点并访问之

非递归后序周游二叉树

- 解决方案：
  - 需要给栈中的每个元素加上一个特征位，以便当从栈顶托出一个结点时区别是从栈顶元素左边回来的(则要继续周游右子树)，还是从右边回来的(该结点的左、右子树均已周游)
  - 特征为**Left**表示已进入该结点的左子树，将从左边回来；特征为**Right**表示已进入该结点的右子树，将从右边回来

非递归后序周游二叉树

- 栈中的元素类型定义StackElement

```
enum Tags{Left,Right};      //特征标识定义
template <class T>
class StackElement          //栈元素的定义
{
public:
    //指向二叉树结点的链接
    BinaryTreeNode<T>* pointer;
    //特征标识申明
    Tags tag;
};
```

非递归后序周游二叉树

```
template<class T>
void BinaryTree<T>::PostOrderWithoutRecusion
(BinaryTreeNode<T>* root)
//非递归后序遍历二叉树或其子树
{
   using std::stack;//使用STL栈部分
   StackElement<T> element;
   stack<StackElement<T > > aStack;//栈申明
   BinaryTreeNode<T>* pointer;
   if(root==NULL)
        return;//空树即返回
```

非递归后序周游二叉树

```
//else
pointer=root;
while(true){//进入左子树
  while(pointer!=NULL){
      element.pointer=pointer;
      element.tag=Left;
      aStack.push(element);
      //沿左子树方向向下周游
      pointer=pointer->leftchild();
      }
   //托出栈顶元素
   element=aStack.top();
```

非递归后序周游二叉树

```
aStack.pop();
pointer=element.pointer;
//从右子树回来
while(element.tag==Right){
    Visit(pointer->value());//访问当前结点
    if(aStack.empty())
        return;
    //else{
    element=aStack.top();
```

非递归后序周游二叉树

```
            aStack.pop();//弹栈
            pointer=element.pointer;
           // }//end else
        }//end while
        //从左子树回来
        element.tag=Right;
        aStack.push(element);
        //转向访问右子树
        pointer=pointer->rightchild();
    }//end while
}
```

问题讨论

- 前序周游算法是否还可以简洁一些？
- 前序、中序、后序框架的算法通用性？
  - 例如后序框架是否支持前序、中序访问？若支持，怎么改动？
- 非递归周游的意义？
  - 栈中存放了什么？

思考题

- 习题: 给定结点类型为**BinaryTreeNode**的三个指针**p**、**q**、**rt**，假设**rt**↑为根结点，求距离结点**p**↑和结点**q**↑最近的这两个结点的共同祖先结点。
- 上机题：表达式二叉树。把计算机内部的一棵表达式二叉树，输出为相应的中缀表达式（可以带括号，但不允许冗余括号）

非递归前序周游二叉树——简洁

- 思想：
  - 遇到一个结点，就访问该结点，并把此结点的非空右结点推入栈中，然后下降去周游它的左子树；
  - 周游完左子树后，从栈顶托出一个结点，并按照它的右链接指示的地址再去周游该结点的右子树结构。

```
template<class T>  void
  BinaryTree<T>::PreOrderWithoutRecusion
  (BinaryTreeNode<T>* root)
  //非递归前序遍历二叉树或其子树
```

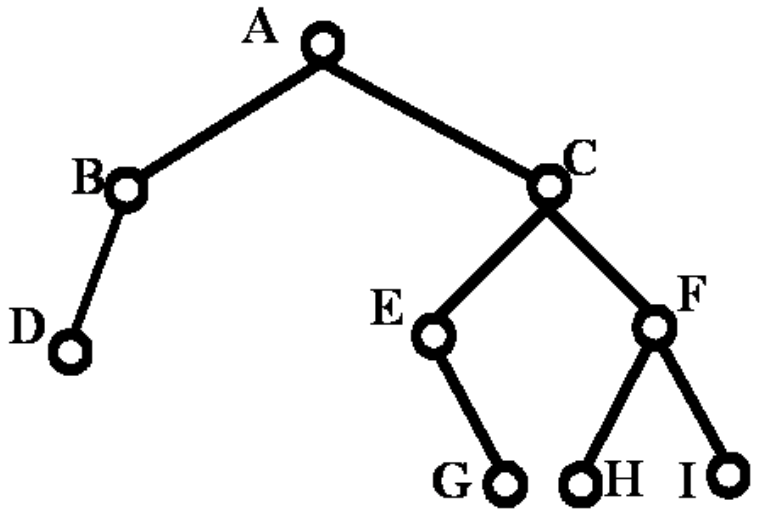
非递归前序周游二叉树

```
{
using std::stack;          //使用STL中的stack
stack<BinaryTreeNode<T>* > aStack;
BinaryTreeNode<T>* pointer=root;
aStack.push(NULL); //  栈底监视哨
while(pointer){  // 或者!aStack.empty()
    Visit(pointer->value());    //访问当前结点
    if (pointer->rightchild() != NULL)   //右孩子入栈
        aStack.push(pointer->rightchild());
    if (pointer->leftchild() != NULL)
        pointer = pointer->leftchild();   //左路下降
    else { //左子树访问完毕，转向访问右子树
        pointer=aStack.top();
        aStack.pop();  //栈顶元素退栈 }
} }
```

广度优先周游二叉树

- 从二叉树的第一层（根结点）开始，自上至下逐层遍历；在同一层中，按照从左到右的顺序对结点逐一访问。

- **例如：** ABCDEFGHI



广度优先周游二叉树

```
template<class T>
Void BinaryTree<T>::LevelOrder
(BinaryTreeNode<T>* root)
{
    using std::queue;          //使用STL的队列
    queue<BinaryTreeNode<T>*> aQueue;
    BinaryTreeNode<T>* pointer=root;
    if(pointer)
        aQueue.push(pointer);
    while(!aQueue.empty())
```

广度优先周游二叉树

```
{
    //取队列首结点
    pointer=aQueue.front();
    Visit(pointer->value());//访问当前结点
    aQueue.pop();
    //左子树进队列
    if(pointer->leftchild())
        aQueue.push(pointer->leftchild());
    //右子树进队列
    if(pointer->rightchild())
        aQueue.push(pointer->rightchild());
}//end while
}
```

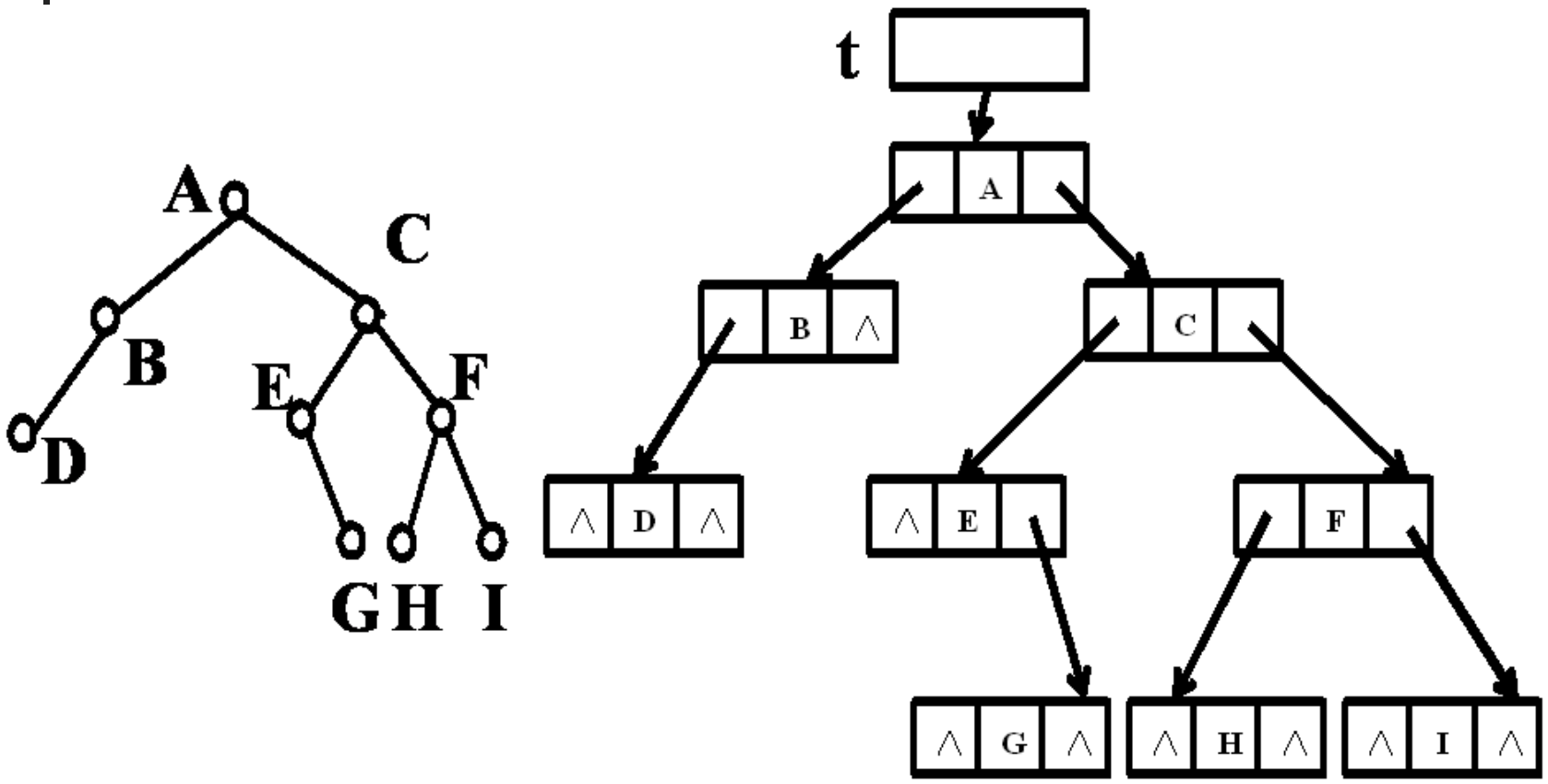
4.5 二叉树的实现

- **4.5.1** 用指针实现二叉树
- **4.5.2** 空间开销分析
- **4.5.3** 用数组实现完全二叉树
- **4.5.4** 穿线二叉树

用指针实现二叉树

- 二叉树是非线性的树形结构，在存储器里表示树形结构的最自然的方法是链接的方法
- 在每个结点中除存储结点本身的数据外再设置两个指针字段**llink**和**rlink**，分别指向结点的左子女和右子女
- 当结点的某个子女为空时，则相应的指针为空指针
- 结点的形式为

用指针实现二叉树



用指针实现二叉树

- 二叉树还可以有其他的链接表示法
  - 例如，在树的每个结点中除用**llink**和**rlink**分别指向子女和兄弟外，再增加一个指向父母的指针**parent**，形成三重链接的二叉树，称为“三叉链表”
  - 有了指向父母的指针就给了我们“向上”访问的能力，这在一些复杂的应用中是需要的

用指针实现二叉树

- 扩展二叉树结点抽象数据类型**BinaryTreeNode,**为每个元素结点添加**left**和**right**左右子结点结构

```
private:
  //二叉树结点指向左子树的指针
  BinaryTreeNode<T>*  left;
  //二叉树结点指向右子树的指针
  BinaryTreeNode<T>*  right;
```

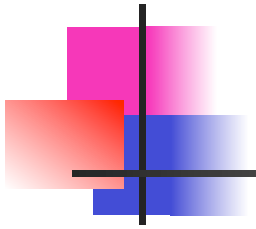
用指针实现二叉树

- 二叉链表表示的二叉树成员函数实现

```
template<class T>
   bool BinaryTree<T>:: isEmpty() const
   { //判定二叉树是否为空树
        return ((root)? false :true);
   }
template<class T>
BinaryTreeNode<T>*BinaryTree<T>::GetParent
   (BinaryTreeNode<T>* root,
   BinaryTreeNode<T>* current)
```

用指针实现二叉树

```
{//从二叉树的root结点开始，查找current结点
 //父结点
 BinaryTreeNode<T>* temp;
 if(root==NULL)
      return NULL;
 //找到父结点
 if((root->leftchild()==current)||
    (root->rightchild()==current))
      return root;
```



用指针实现二叉树

```
//递归寻找父结点
  if((temp=GetParent
          (root->leftchild(),current))!
  =NULL)
     return temp;
  else return GetParent
                (root->rightchild(),current);
```

用指针实现二叉树

```
template<class T>
BinaryTreeNode<T>*
   BinaryTree<T>::Parent(BinaryTreeNode<T>*
   current)
 {//返回current结点的父结点指针
   if((current==NULL)||(current==root))
        return NULL;//空结点或者current为根结点时
   //调用递归函数寻找父结点
   return GetParent(root,current);
 }
```

用指针实现二叉树

```
template<class T>
BinaryTreeNode<T>*BinaryTree<T>::LeftSibling
(BinaryTreeNode<T>* current)
{//返回current结点的左兄弟
   if(current)//current不为空
   {    //返回current结点的父结点
        BinaryTreeNode<T>* temp=Parent(current);
        //如果父结点为空，或者current没有左兄弟
        If((temp==NULL)||current==temp->leftchild())
                return  NULL;
        else return temp->leftchild();}//end if
   return NULL;
}
```

用指针实现二叉树

```
template<class T>
   BinaryTreeNode<T>* BinaryTree<T>::RightSibling
   (BinaryTreeNode<T>* current)
{//返回current结点的右兄弟
   if(current){
        //返回current结点的父结点
        BinaryTreeNode<T>* temp=Parent(current);
        //如果父结点为空，或者current没有右兄弟
        if((temp==NULL)||current==temp->rightchild())
                return  NULL;
        else return temp->rightchild();}//end if
   return NULL;
}
```

用指针实现二叉树

```
template<class T>void
BinaryTree<T>:: CreateTree (const T& elem,
BinaryTree<T>& leftTree, BinaryTree<T>& rightTree)
{//由leftTree,rightTree和elem创建一棵新树，根结点是
 //elem，左子树是leftTree，右子树是rightTree。其中this、
 //leftTree、rightTree必须是不同的三棵树
 root=new BinaryTreeNode<T>(elem,leftTree.root,
                                    rightTree.root);
  //原来两棵子树的根结点指空，避免访问
  leftTree.root=rightTree.root=NULL;
}
```

用指针实现二叉树

```
template<class T>void BinaryTree<T>::
DeleteBinaryTree(BinaryTreeNode<T>* root)
{//以后序周游的方式删除二叉树
   if(root){
        DeleteBinaryTree(root->left);//递归删除左子树
        DeleteBinaryTree(root->right);//递归删除右子树
        delete root;//删除根结点
   }
}
```

空间开销分析

- 结构性开销γ是指为了实现数据结构而花费的辅助空间比例。
- 这些辅助空间不是用来存储数据记录，而是为了保存数据结构的逻辑特性或为了方便运算。

$$\gamma(\text{结构性开销}) = \frac{\text{辅助结构存储量}}{\text{整个结构占用的存储总量}}$$

空间开销分析

- 存储密度α (≤1)表示数据结构存储的效率

$$\alpha(\text{存储密度}) = \frac{\text{数据本身存储量}}{\text{整个结构占用的存储总量}}$$

- 显然α = 1 - γ
- 如果所有的存储空间都分配给了数据，则这个存储结构叫紧凑结构，否则叫非紧凑结构，紧凑结构的存储密度为1，非紧凑结构的存储密度小于1
- 显然二叉链表的存储是非紧凑结构。存储密度越大，则存储空间的利用效率越高

空间开销分析

根据满二叉树定理：一半的指针是空的

- 如果只有叶结点存储数据，分支结点为内部结构结点（如**Huffman**树），则开销取决于二叉树是否满（越满存储效率越高）
- 对于简单的每个结点存两个指针、一个数据域
  - 总空间 $(2p + d)n$
  - 结构性开销：$2pn$
  - 如果 $p = d$，则 $2p/(2p + d) = 2/3$

空间开销分析

- 去掉满二叉树叶结点中的指针

$$\frac{n/2(2p)}{n/2(2p) + dn} = \frac{p}{p + d}$$

则结构性开销为 1/2 （假设$p = d$）

- 如果只在叶结点存数据，则结构性开销为$2pn/(2pn + d(n+1)) \Rightarrow 2/3$ （假设$p = d$）
- 注意：区分叶结点和分支结点又需要额外的算法时间。

空间开销分析

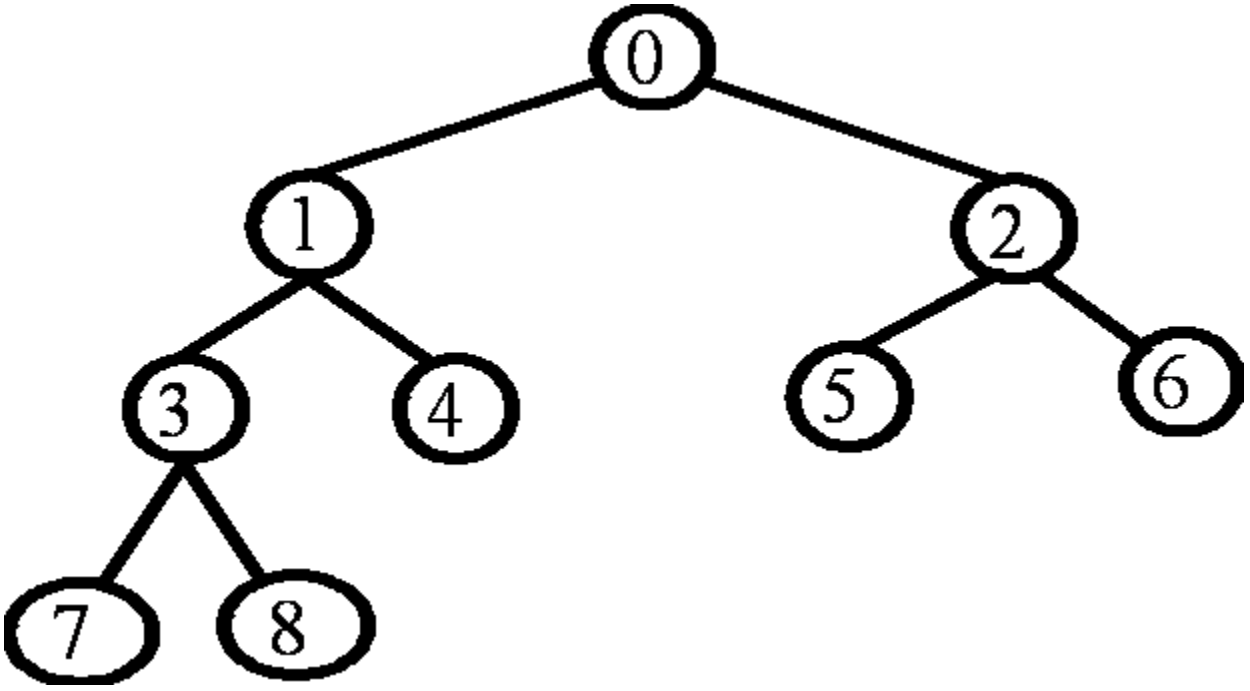
- 在C++中，可以用两种方法来实现不同的分支结点与叶结点：
  - 用union联合类型定义
  - 使用C++的子类来分别实现分支结点与叶结点，并采用虚函数isLeaf来区别分支结点与叶结点
- 早期有人利用结点指针的一个空闲位（一个bit就可以）来存储结点所属的类型
- 也有人利用指向叶结点的指针或者叶结点中的指针域来存储该叶结点的值。
- 目前，计算机内存资源并不紧张的时候，一般不提倡这种单纯追求效率，而丧失可读性的做法

用数组实现完全二叉树

- 当我们要求一个二叉树紧凑存储，并且在处理过程中，该二叉树结构的大小和形状不发生激烈的动态变化时，可以采用顺序的方法存储
- 用顺序方法存储二叉树，就是要把所有结点按照一定的次序顺序存储到一片连续的存储单元中
- 适当安排这个结点的线性序列，可以使结点在序列中的相互位置反映出二叉树结构的部分信息
- 但一般说来这样的信息是不足以刻画整个结构的，还要在结点中附加一些其他的必要信息，以完全地反映整个结构

用数组实现完全二叉树

按层次顺序将一棵有**n**个结点的完全二叉树的所有结点从**0**到**n-1**编号，就得到结点的一个线性序列



完全二叉树的下标公式

- 完全二叉树中除最下面一层外，各层都被结点充满了，每一层结点个数恰是上一层结点个数的两倍。因此，从一个结点的编号就可以推知它的父母，左、右子女，兄弟等结点的编号
  - 当**2i+1≤n**时，结点**i**的左子女是结点**2i+1**，否则结点**i**没有左子女
  - 当**2i+2≤n**时，结点**i**的右子女是结点**2i+2**，否则结点**i**没有右子女

完全二叉树的下标公式

- 当0<i<n时，结点i的父母是结点[(i-1)/2]
- 当i为偶数且0<i<n时，结点i的左兄弟是结点i-1，否则结点i没有左兄弟
- 当i为奇数且i+1<n时，结点i的右兄弟是结点i+1，否则结点i没有右兄弟

完全二叉树的顺序存储总结

- 完全二叉树结点的层次序列足以反映二叉树的结构
  - 所有结点按层次顺序依次存储在一片连续的存储单元中，则根据一个结点的存储地址就可算出它的左右子女，父母的存储地址，就好像明显地存储了相应的指针一样
  - 存储完全二叉树的最简单，最节省空间的存储方式
- 完全二叉树的顺序存储，在存储结构上是线性的，但在逻辑结构上它仍然是二叉树型结构

穿线二叉树

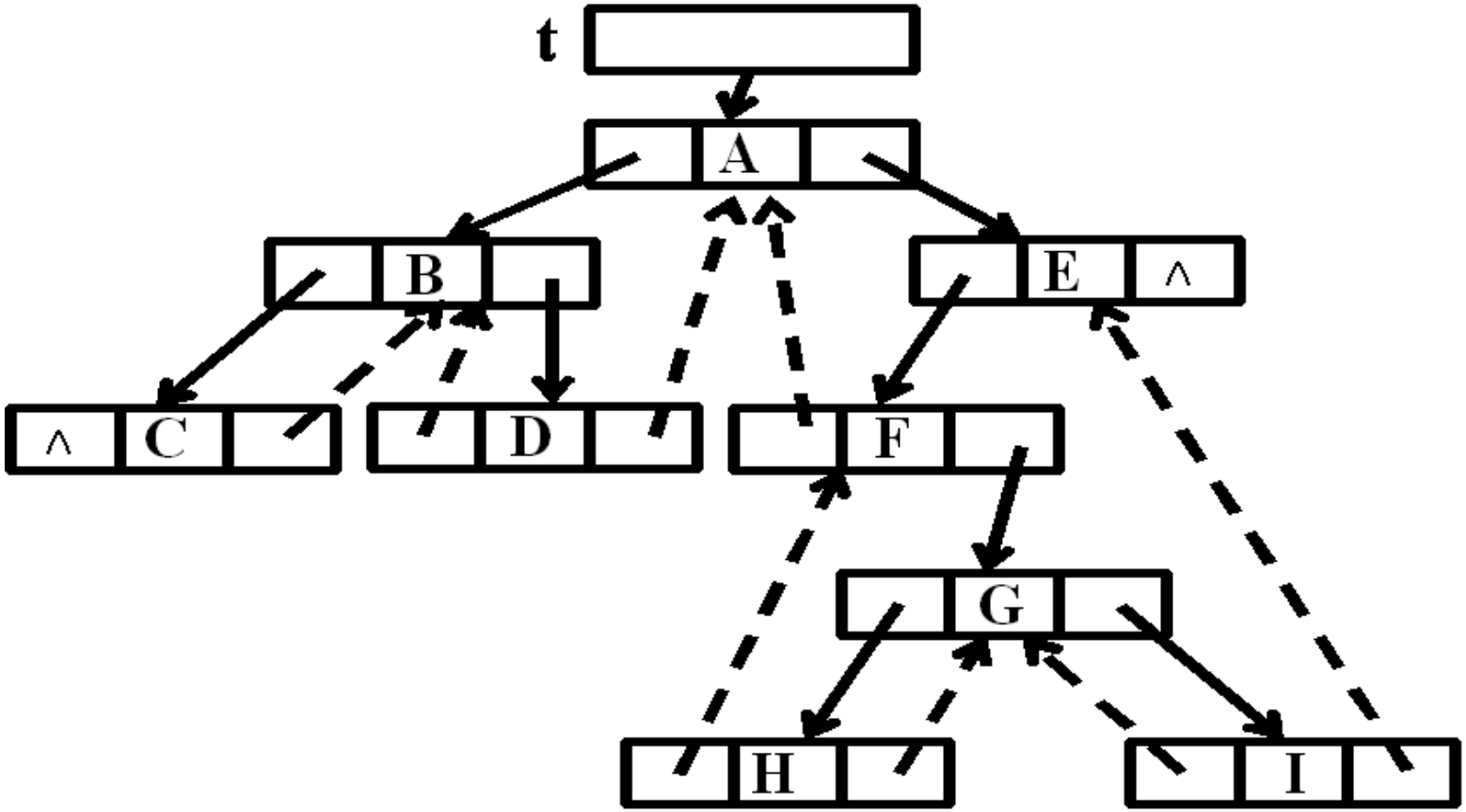
- 穿线树：在二叉链表存储形式的二叉树中，把结点中空指针利用成为周游线索
  - 原来为空的左指针指向结点在某种周游序列下的前驱
  - 原来为空的右指针指向结点在同一种周游序列下的后继

  这样的二叉树称为穿线树
- 可以有中序穿线树，前序穿线树，后序穿线树。每种穿线树可以只穿一半
- 目的：利用空指针的存储空间，建立周游线索

穿线二叉树

- 为了区分线索和指针，需在每个结点中增加两个标志位，分别标识左右指针域是实际指针还是线索
  - lTag = 0,　left为左子女指针
  - lTag = 1,　left为前驱线索
  - rTag= 0,　right为右子女指针
  - rTag = 1,　right为后继指针

中序穿线二叉树 ：示例



穿线二叉树结点类

```
template <class T>
class ThreadBinaryTreeNode
{
private:
    int lTag,rTag;//左右标志位
    //线索或左右子树
    ThreadBinaryTreeNode<T> *left,*right;
    T    element;
public:
    ThreadBinaryTreeNode();                  //缺省构造函数
    ThreadBinaryTreeNode(const T)      //拷贝构造函数
    :element(T),left(NULL),right(NULL),lTag(0),rTag(0)
    {};
```

穿线二叉树结点类

```
T& value() const{return element};
ThreadBinaryTreeNode<T>* leftchild()   const
                    {return left};
ThreadBinaryTreeNode<T>* rightchild() const
                    {return right};
 void setValue(const T& type){element=type;};
//析构函数
 virtual ~ThreadBinaryTreeNode();
};
```

中序穿线二叉树类

```
template <class T> class ThreadBinaryTree{
private:
   ThreadBinaryTreeNode<T>* root;//根结点指针
public:
   ThreadBinaryTree(){root=NULL;};//构造函数
   virtual ~ThreadBinaryTree(){DeleteTree(root);};
   //返回根结点指针
   ThreadBinaryTreeNode<T>* getroot(){return root;};
   //中序线索化二叉树
   void InThread(ThreadBinaryTreeNode<T>* root);
   //中序周游
   void InOrder(ThreadBinaryTreeNode<T>* root);
};
```

中序线索化二叉树：递归实现

```
template <class T>      void
ThreadBinaryTree<T>::InThread
        (ThreadBinaryTreeNode<T>*root,
         ThreadBinaryTreeNode<T>* &pre)
{
  if(root!=NULL) {
        //中序线索化左子树
        InThread(root->leftchild(),pre);
        if(root->leftchild()==NULL){
           //建立前驱线索
           root->left=pre;
```

中序线索化二叉树：递归实现

```
        if (pre) root->lTag=1;
    }
    if((pre)&&(pre->rightchild()==NULL))
    {//建立后继线索
        pre->right=root;
        pre->rTag=1;
    }//end if
    pre=root;
    InThread(root->rightchild(),pre); //中序线索化右子树
    }//end if
}
```

周游穿线树

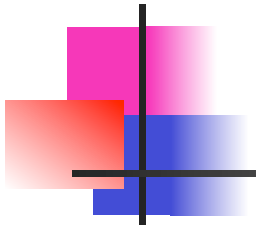
- 中序周游中序穿线树：先从穿线树的根出发，一直沿左指针，找到“最左”（它一定是中序的第一个结点）；然后反复地找结点的中序后继
- 一个结点的右指针如果是线索，则右指针就是下一个要周游的结点，如果右指针不是线索，则它的中序后继是其右子树的“最左”结点

中序周游穿线树

```
template<class T>void
   ThreadBinaryTree<T>::InOrder(
        ThreadBinaryTreeNode<T>* root)
{
   ThreadBinaryTreeNode<T>* pointer;
   //是否为空二叉树
   if(root==NULL)        return;
        else pointer=root;
   //找“最左下”结点
   while(pointer->leftchild()!=NULL)
        pointer=pointer->leftchild();
   //访问当前结点并找出当前结点的中序后继
```

中序周游穿线树

```
while(1)
{
   Visit(pointer->value());                  //访问当前结点
   if(pointer->rightchild()==NULL) return;
   if(pointer->rTag==1)
     pointer=pointer->rightchild();//按照线索寻找后继
   else{//按照指针寻找后继
          pointer=pointer->rightchild();
          while(pointer->lTag==0)
              pointer=pointer->leftchild();  //沿左链下降
       }//end else
   }//end while
}
```



问题讨论

- 是否能够改进第一次找“最左下”结点的代码？
- 是否能够改进整个代码？

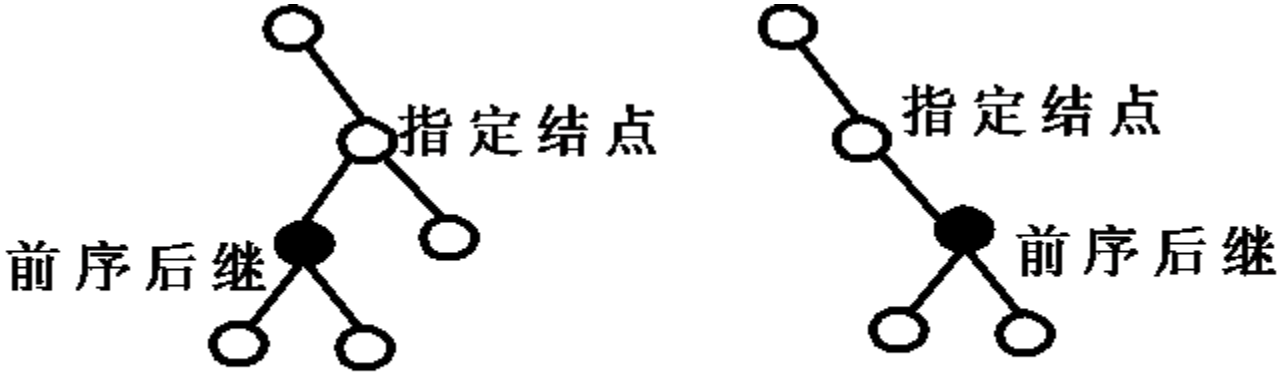
穿线二叉树

- 中序穿线树里找指定结点在前序下的后继结点
- 算法中需要用到的一个重要事实：若一个树叶是某子树的中序下的最后一个结点，则它必是该子树的前序下最后一个结点

穿线二叉树

- 情况一：当指定结点不是树叶时
  - 若指定结点有左子女，则左子女是它的前序后继
  - 若指定结点没有左子女，则右子女是它的前序后继

这种情况下，问题的解决非常简单，不需要线索的帮助



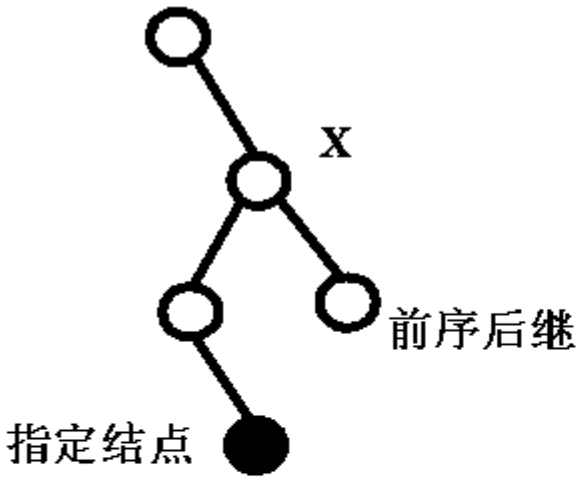
（a）指定结点有左子女　　（b）指定结点无左子女

穿线二叉树

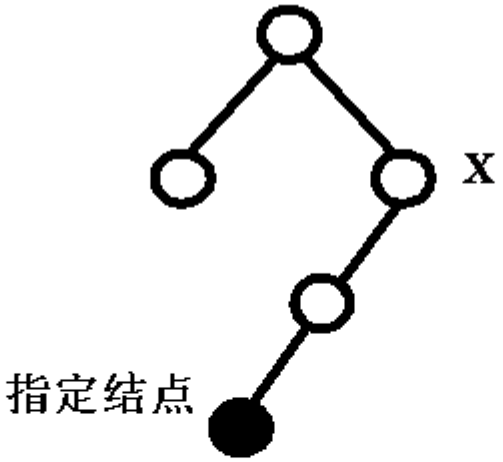
- 情况二：当指定结点是树叶时
  - 若指定结点是“某结点x”的左子树中按前序周游列出的最后一个结点，且该结点x又有右子女，则指定结点的前序后继就是该结点x的右子女
  - 若指定结点不是任何结点的左子树中按前序周游列出的最后一个结点(c)；或者虽然是某结点x的左子树中按前序周游列出的最后一个结点，但该结点x没有右子女(b)，则指定结点没有前序后继(参看下图)，这种情况下解决问题的关键在于找出上述的“某结点x”

穿线二叉树

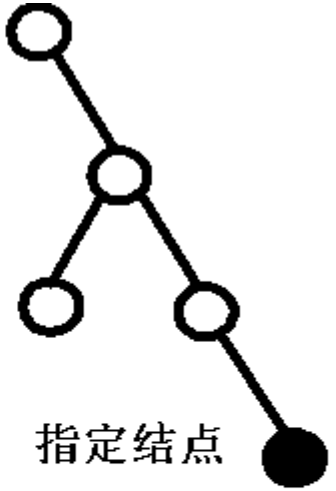
- 情况二：



（a）结点 x 有右子女



（b）结点 x 无右子女



（c）不能找到结点 x，使指定结点是结点 x 的左子树中按前序的最后结点

穿线二叉树

- 我们知道指定结点的右线索是指向一个祖先结点x的，指定结点是结点x的左子树的中序最后一个结点
- 由前面事实可知，指定结点也是结点x的左子树的前序最后一个结点。因此，指定结点的右线索所指的结点就是我们这里要找的某结点x
- 于是我们可以利用中序穿线树中右线索的帮助来找指定结点的前序后继了。

在中序穿线二叉树里找指定结点在前序下的后继

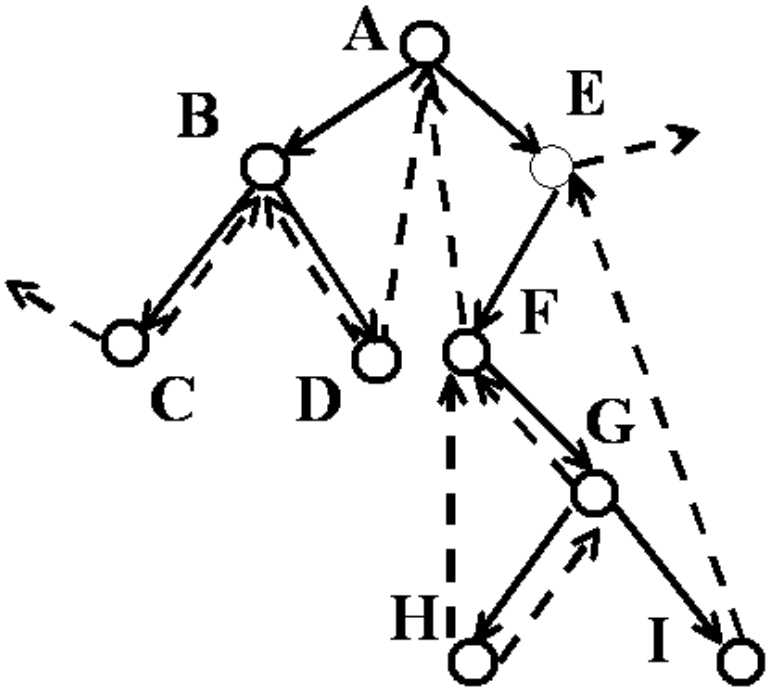
```
template<class T>ThreadBinaryTreeNode<T>*
ThreadBinaryTree<T>::FindNextinInorderTree
   (ThreadBinaryTreeNode<T>* pointer)
{//在中序穿线树中找指定结点在前序下的后继
ThreadBinaryTreeNode<T>*temppointer=
   NULL;
//指定结点有左子女
if(pointer->lTag==0)
   return pointer->leftchild();
```

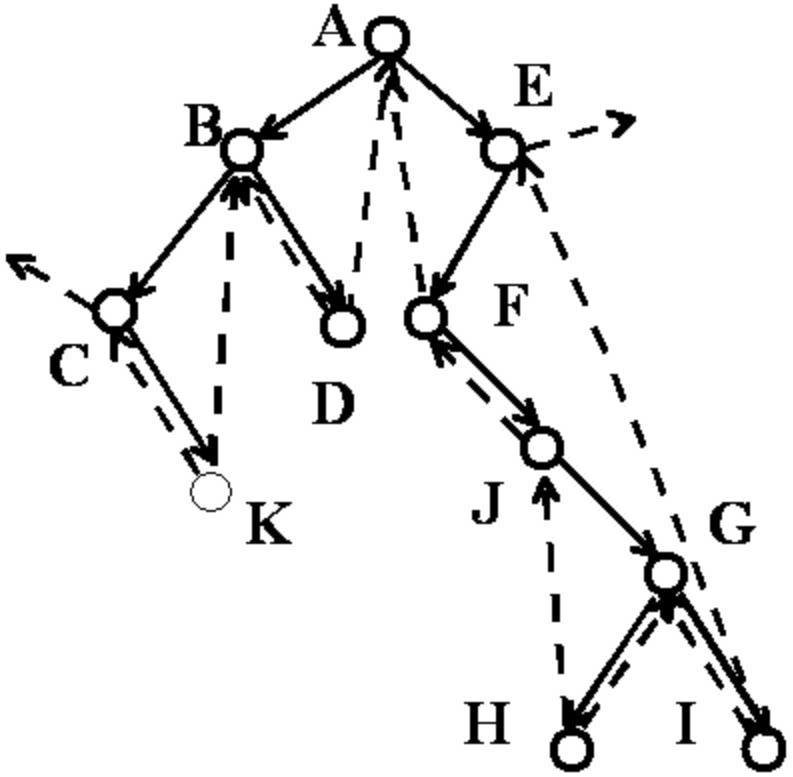
在中序穿线二叉树里找指定结点在前序下的后继

```
else
    temppointer=pointer
//temppointer=pointer->leftchild();
while(temppointer->rTag==1)
    temppointer=temppointer->rightchild();
temppointer=temppointer->rightchild();
return temppointer;
}
```

穿线树的结点插入

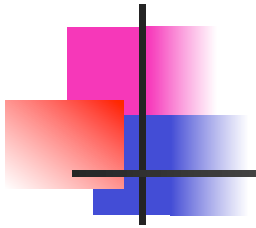
- 结点F的右边插入J，结点C的右边插入K





穿线树的结点插入

- 往中序穿线树里插入结点的算法，规定插入这样进行：
  - **newpointer**指向要插入的新结点，**pointer**指向穿线二叉树里的一个结点
  - 将新结点插进来作为**pointer**指向的结点的右子树的根。**pointer**指向的结点的原来的右子树现在作为新结点的右子树(新结点的左子树为空)。即在中序序列里，新结点刚好插到**p**所指向的结点的后面.



- **Pointer的新后继结点是Newpointer**
- **Newpointer的后继是pointer->rightchild()**
- 如果**Pointer**的右子树不空，则右子树的最左结点线索指向**Newpointer**；若空，则**pointer**的右线索给**Newpointer**继承

穿线树的结点插入

```
template<class T>  void
ThreadBinaryTree<T> ::InsertNode
   (ThreadBinaryTreeNode<T>*pointer,
   ThreadBinaryTreeNode<T>*  newpointer)
{//往中序穿线树里插入一个新结点
   ThreadBinaryTreeNode<T>*
              temppointer=NULL;
//找指定结点的中序后继
if(pointer->rightchild()==NULL)
   temppointer=NULL;
```

穿线树的结点插入

```
else if(pointer->rTag==1)//右儿子为线索
   temppointer=pointer->rightchild();
else
   {//右儿子为指针
         temppointer=pointer->rightchild();
         while(temppointer->lTag==0)
         temppointer=temppointer->leftchild();
        }
//temppointer指针指向pointer结点的中序后继
//建立指定结点的中序后继的左线索
```

穿线树的结点插入

```
if((temppointer!=NULL)&&(temppointer->lTag==1))
        temppointer->left=newpointer;
//建立新结点的右指针或右线索
newpointer->rTag=pointer->rTag;
newpointer->right=pointer->rightchild();
//插入新结点
pointer->rTag=0;
pointer->right=newpointer;
//建立新结点左线索
newpointer->lTag=1;
newpointer->left=pointer;
}
```

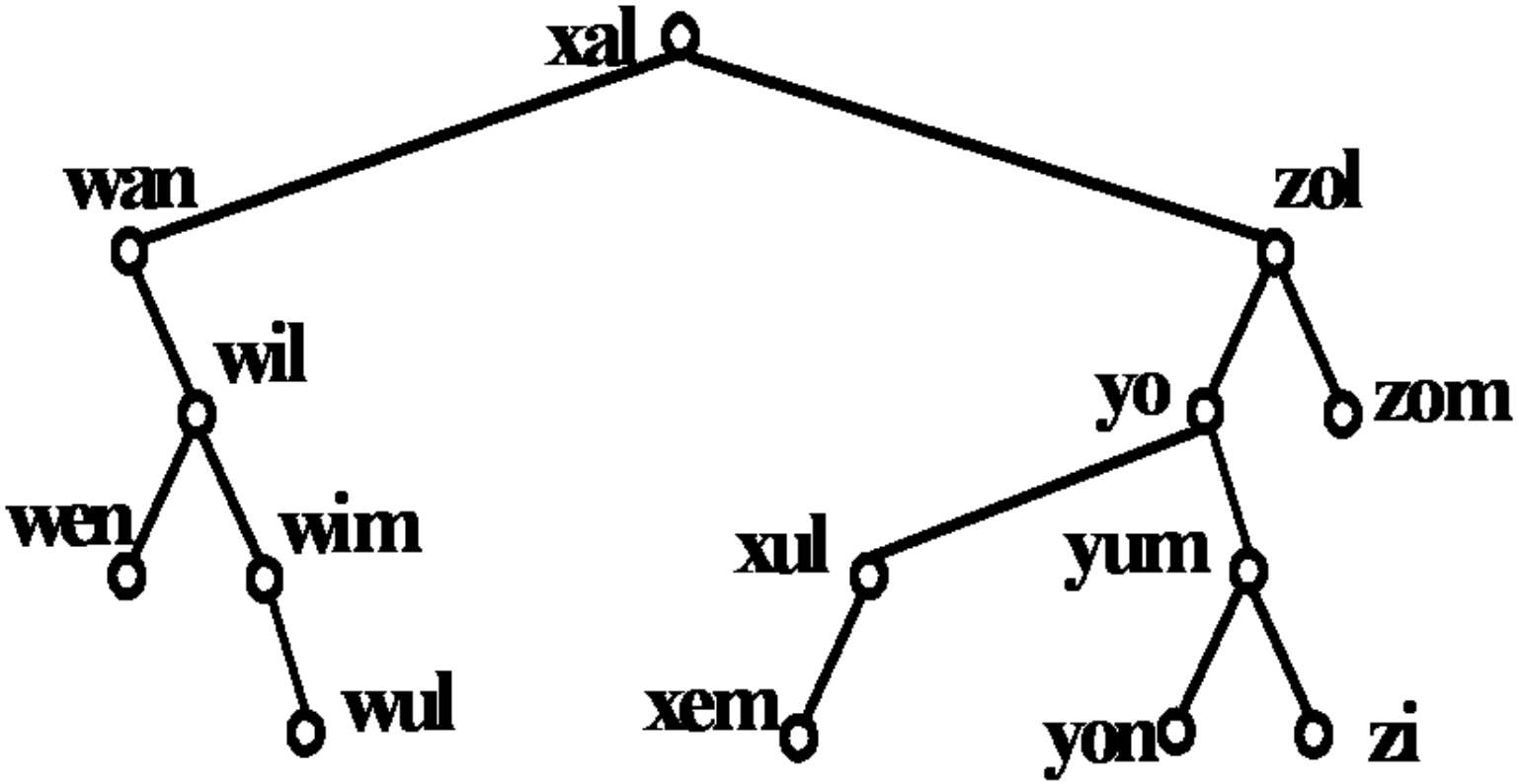
穿线树总结

- 任何包括n个结点的二叉树的二叉链表中，2n个指针中都只有n-1个用来指示结点的左右子女，而另外n+1个为空。这显然是浪费存储空间的
- 利用空指针空间的一个办法就是用指向结点在中序下的前驱结点和后继结点的指针来代替这些空的指针
- 穿线树的最大优点是：由于有了线索的存在而使得周游二叉树和找结点在指定次序下的前驱、后继的算法变得直截了当

4.6 二叉搜索树

- 二叉搜索树（BST）
  - 或者是一颗空树；
  - 或者是具有下列性质的二叉树：对于任何一个结点，设其值为K，则该结点的左子树(若不空)的任意一个结点的值都小于K；该结点的右子树(若不空)的任意一个结点的值都大于或等于K；而且它的左右子树也分别为二叉搜索树
- 二叉搜索树的性质：按照中序周游将各结点打印出来，将得到按照由小到大的排列

BST图示



二叉搜索树

- 二叉搜索树的效率就在于只需检索二个子树之一
  - 从根结点开始，在二叉搜索树中检索值K。如果根结点储存的值为K，则检索结束。
  - 如果K小于根结点的值，则只需检索左子树
  - 如果K大于根结点的值，就只检索右子树
- 这个过程一直持续到K被找到或者我们遇上了一个树叶
- 如果遇上树叶仍没有发现K，那么K就不在该二叉搜索树中

二叉搜索树的插入

- 往二叉搜索树里插入新结点，要保证插入后仍符合二叉搜索树的定义
- 插入是这样进行的：将待插入结点的关键码值与树根的关键码值比较，若待插入的关键码值小于树根的关键码值，则进入左子树，否则进入右子树
- 在子树里又与子树根比较，如此进行下去，直到把新结点插入到二叉树里作为一个新的树叶

二叉搜索树的插入

```
template<class T>
void BinarySearchTree<T>::InsertNode(
        BinaryTreeNode<T>* root ,
        BinaryTreeNode<T>* newpointer)
{//向二叉搜索树插入新结点
   BinaryTreeNode<T>* pointer=NULL;
   if(root==NULL){
   //用指针newpointer初始化二叉搜索树树根，赋值实现
        Initialize(newpointer);
        return;
   }
   else pointer=root;
```

二叉搜索树的插入

```
while(1){
        if(newpointer->value()==pointer->value())
                return ;
        //相等则不用插入
        else if(newpointer->value()<pointer->value())
        {
           if(pointer->leftchild()==NULL){
                pointer->left=newpointer;//作为左子树
                return;
           }
          else pointer=pointer->leftchild();
}
```

二叉搜索树的插入

```
    else{
        //作为右子树
        if(pointer->rightchild()==NULL) {
                pointer->right=newpointer;
                return;
        }
        else   pointer=pointer->rightchild();
        }//end else
    }//end while
}
```

二叉搜索树

- 对于给定的关键码集合，为建立二叉搜索树，可以从一个空的二叉搜索树开始，将关键码一个个插进去
- 将关键码集合组织成二叉搜索树，实际上起了对集合里的关键码进行排序的作用，按中序周游二叉搜索树，就能得到排好的关键码序列。

二叉搜索树的删除

- 从二叉搜索树里删除一个结点时，不能把以这个结点为根的子树都删除掉，只能删除掉这一个结点，并且还要保持二叉搜索树原来的性质。
- 设p，p1，r是指针变量，p↑表示s要删除的结点，p1↑表示p↑的父母结点，则删除可以按如下规定进行：
  - 若结点p↑没有左子树，则用右子树的根代替被删除的结点p↑
  - 若结点p↑有左子树，则在左子树里找按中序周游的最后一个结点r↑，将r↑的右指针置成指向p↑的右子树的根，然后用结点p↑的左子树的根去代替被删除的结点p↑

二叉搜索树的删除

```
template<class T>       void
BinarySearchTree<T>::DeleteNode(
                BinaryTreeNode<T>* pointer)
{//二叉搜索树的删除
   BinaryTreeNode<T>* temppointer=NULL;
   BinaryTreeNode<T>*parent=GetParent(root,pointer);
   //被删结点无左子树吗?
   if(pointer->leftchild() == NULL)
   {//被删除结点是根结点吗?
        if(parent == NULL)
              root=pointer->rightchild();
```

二叉搜索树的删除

```
else if(parent->leftchild()==pointer)
        parent->left=pointer->rightchild();
    else
        parent->right=pointer->rightchild();
    delete pointer;
    pointer=NULL;
    return;
}//end if
else temppointer=pointer->leftchild();
//在左子树中找对称序的最后一个结点
while(temppointer->rightchild()!=NULL)
    temppointer=temppointer->rightchild();
```

二叉搜索树的删除

```
//被删除结点的右子树作为temppointer的右子树
temppointer->right=pointer->rightchild();
//被删除结点的左子树根代替被删除结点
if(parent==NULL)
        root=pointer->leftchild();
else if(parent->leftchild()==pointer)
        parent->left=pointer->leftchild();
else
        parent->right=pointer->leftchild();
delete pointer;
pointer=NULL;
return;
}
```

改进方案

- 设p，p1，r是指针变量，p↑表示s要删除的结点，p1↑表示p↑的父母结点，则删除可以按如下规定进行：
  - 若结点p↑没有左子树，则用右子树的根代替被删除的结点p↑
  - 若结点p↑有左子树，则在左子树里找按中序周游的最后一个结点r↑，将r↑的右指针置成指向p↑的右子树的根，然后用结点r↑去代替被删除的结点p↑

改进的二叉树结点删除算法

```
template <class T>      void
   BinarySearchTree<T>::DeleteNodeEx
  (BinaryTreeNode<T>* pointer)
{//若待删除结点不存在，返回
   if( pointer == NULL )
        return;
   //保存替换结点
   BinaryTreeNode<T> * temppointer;
   //保存替换结点的父结点
   BinaryTreeNode<T> * tempparent = NULL;
```

改进的二叉树结点删除算法

```
//保存删除结点的父结点
   BinaryTreeNode<T> * parent =
   GetParent(root ,pointer );
//如果待删除结点的左子树为空，就将它的右子树代替它
   if( pointer->leftchild() == NULL ){
   //将右子树连接到待删除结点的父的合适位置
        if( parent == NULL )
                root = pointer->rightchild();
        else if( parent->leftchild() == pointer )
                parent->left = pointer->rightchild();
```

改进的二叉树结点删除算法

```
        else   parent->right = pointer->rightchild();
        delete pointer;
        pointer=NULL;
        return;
}//end if
//当待删除结点左子树不为空，就在左子树中寻找最大结点替
//换待删除结点
temppointer = pointer->leftchild();
while(temppointer->rightchild() != NULL )
```

改进的二叉树结点删除算法

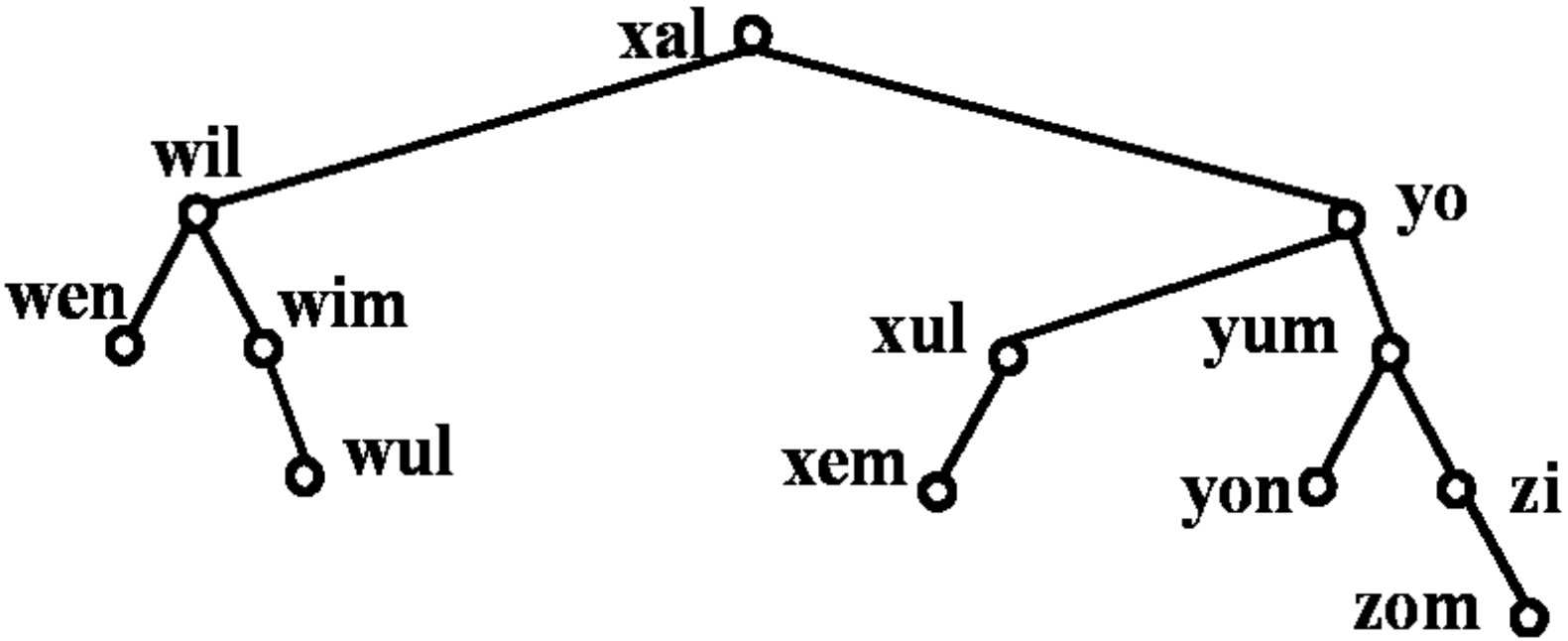
```
{
    tempparent = temppointer;
    temppointer = temppointer->rightchild();
}
//删除替换结点
if(tempparent==NULL)
    pointer->left=temppointer->leftchild();
else   tempparent->right=temppointer->leftchild();
//用替换结点去替代真正的删除结点
```

改进的二叉树结点删除算法

```
if(parent==NULL)
      root=temppointer;
else if( parent->leftchild() == pointer )
          parent->left=temppointer;
 else parent->right=temppointer;
temppointer->left=pointer->leftchild();
temppointer->right=pointer->rightchild();
delete pointer;
pointer=NULL;
return;
}
```

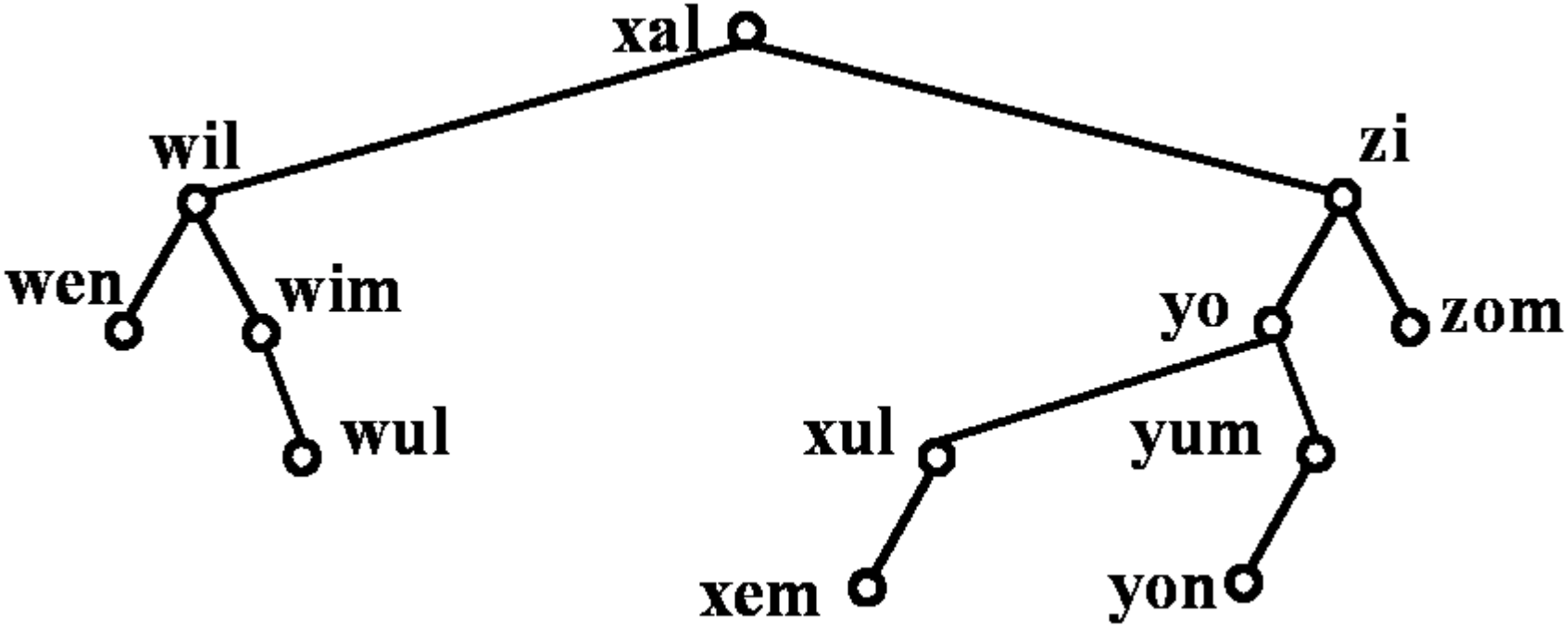
二叉树结点删除算法(未改进)

- 从二叉排序树中删除**wan**和**zol**后得到的二叉排序树



二叉树结点删除算法(改进)

- 从二叉排序树中删除**wan**和**zol**后得到的二叉排序树



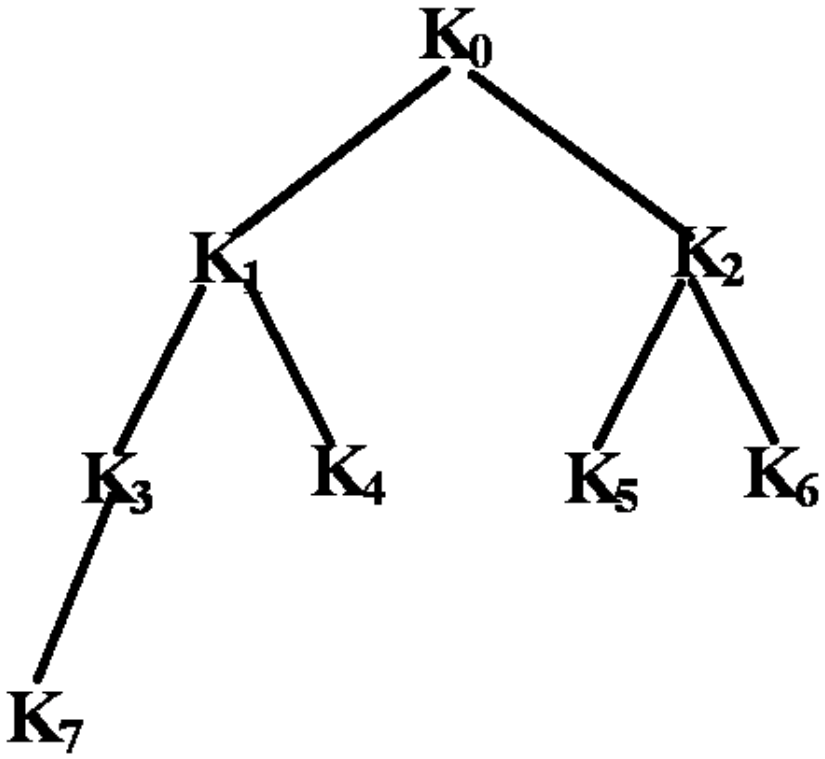
二叉搜索树总结

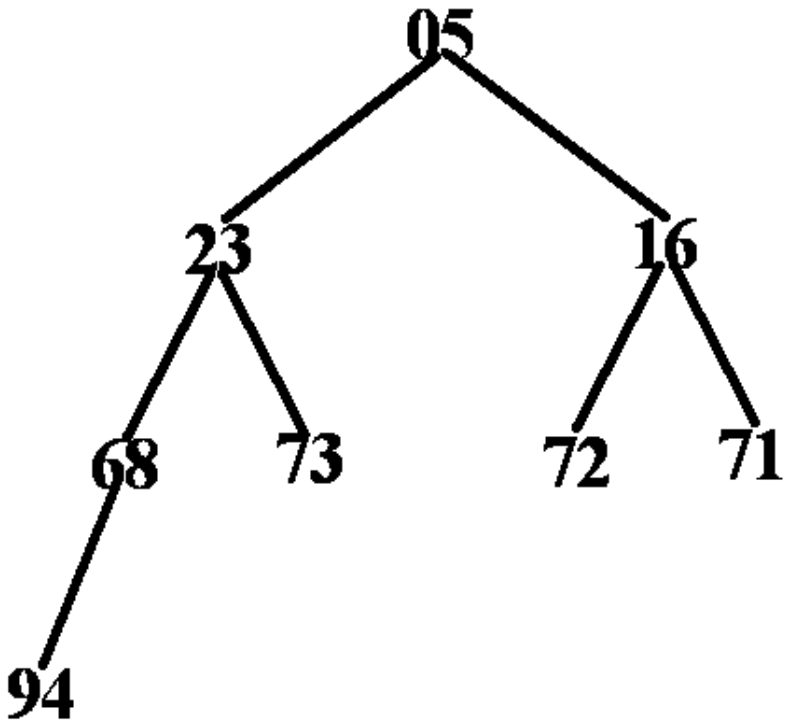
- 树形结构的一个重要应用是用来组织索引，二叉搜索树是适用于内存储器的一种重要的树形索引
- 二叉搜索树的插入和删除运算非常简单。往二叉搜索树里插入新结点或删除已有结点，要保证操作结束后仍符合二叉搜索树的定义

4.7 堆与优先队列

- 最小值堆：最小值堆是一个关键码序列 $\{K_0, K_1, \ldots K_{n-1}\}$，它具有如下特性：
  - $K_i \leq K_{2i+1}$ $(i=0, 1, \ldots, n/2-1)$
  - $K_i \leq K_{2i+2}$
- 类似可以定义最大值堆

堆的示例





堆的性质

- 堆实际上是一个完全二叉树的层次序列，可以用数组表示
- 堆中储存的数是局部有序的
  - 结点储存的值与其子女储存的值之间存在某种联系。有两种不同的堆，决定于其关于联系的定义
  - 堆中任何一个结点与其兄弟之间都没有必然的联系
- 堆不唯一。从逻辑角度看，堆实际上是一种树型结构

堆的类定义

```
template <class T>
class MinHeap      //最小堆ADT定义
{
private:
   T* heapArray;   //存放堆数据的数组
   int CurrentSize;//当前堆中元素数目
   int MaxSize;    //堆所能容纳的最大元素数目
   void BuildHeap();//建堆
public:
   //构造函数,n表示初始化堆的最大元素数目
   MinHeap(const int n);
```

堆的类定义

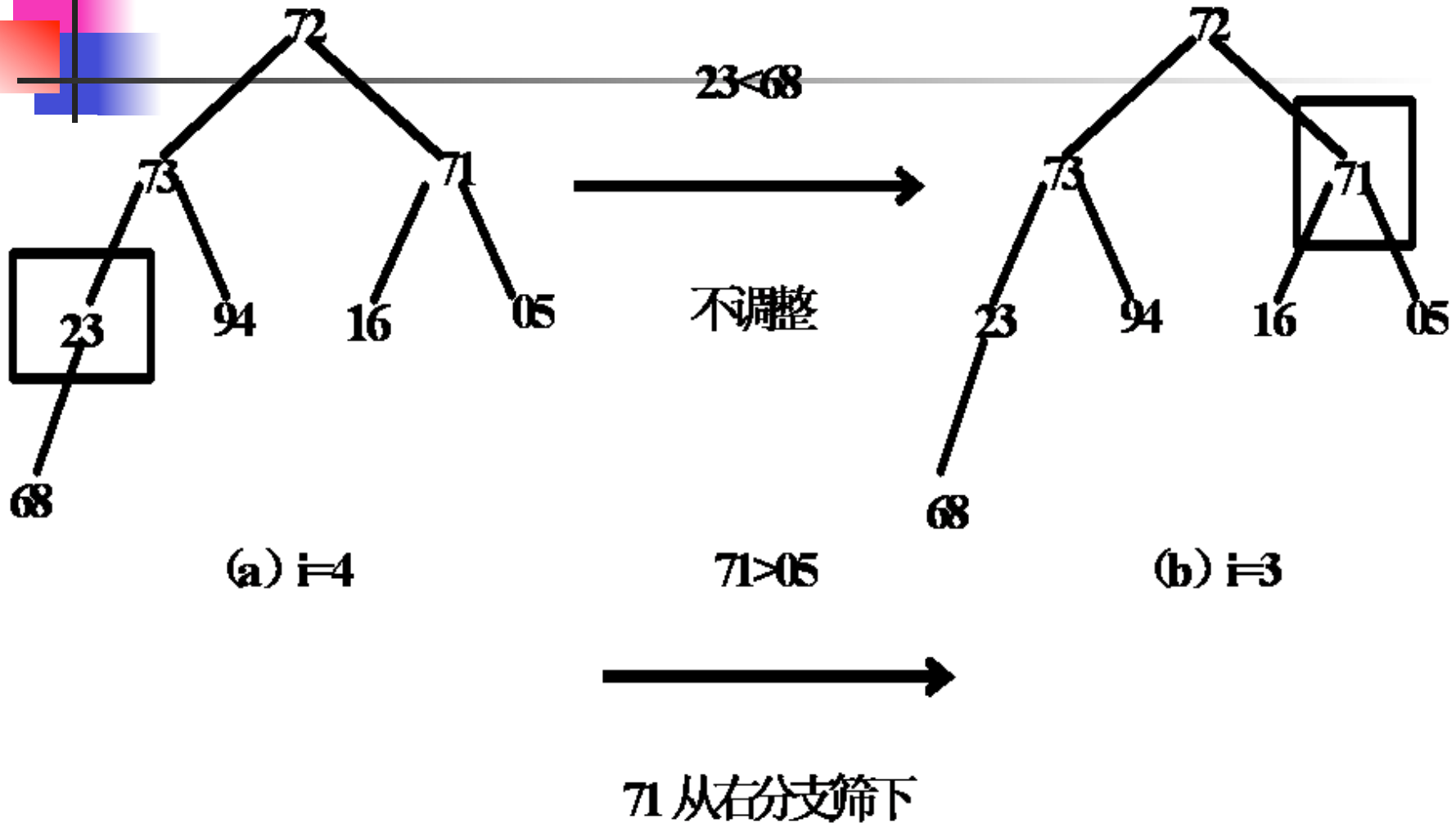
```
//析构函数
virtual ~MinHeap(){delete []heapArray;};
//如果是叶结点，返回TRUE
bool isLeaf(int pos) const;
//返回左孩子位置
int leftchild(int pos) const;
//返回右孩子位置
int rightchild(int pos) const;
// 返回父结点位置
int parent(int pos) const;
```

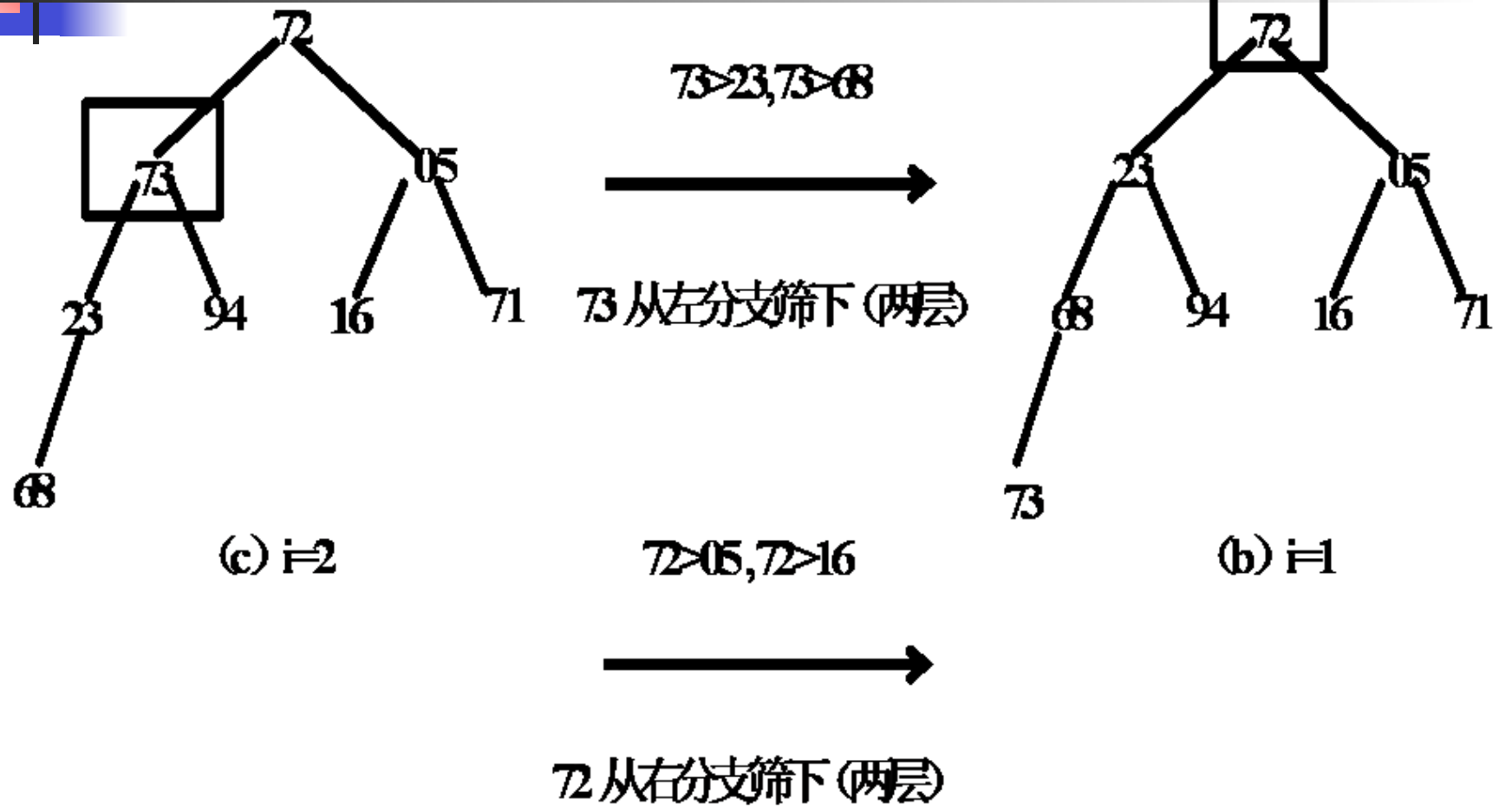
堆的类定义

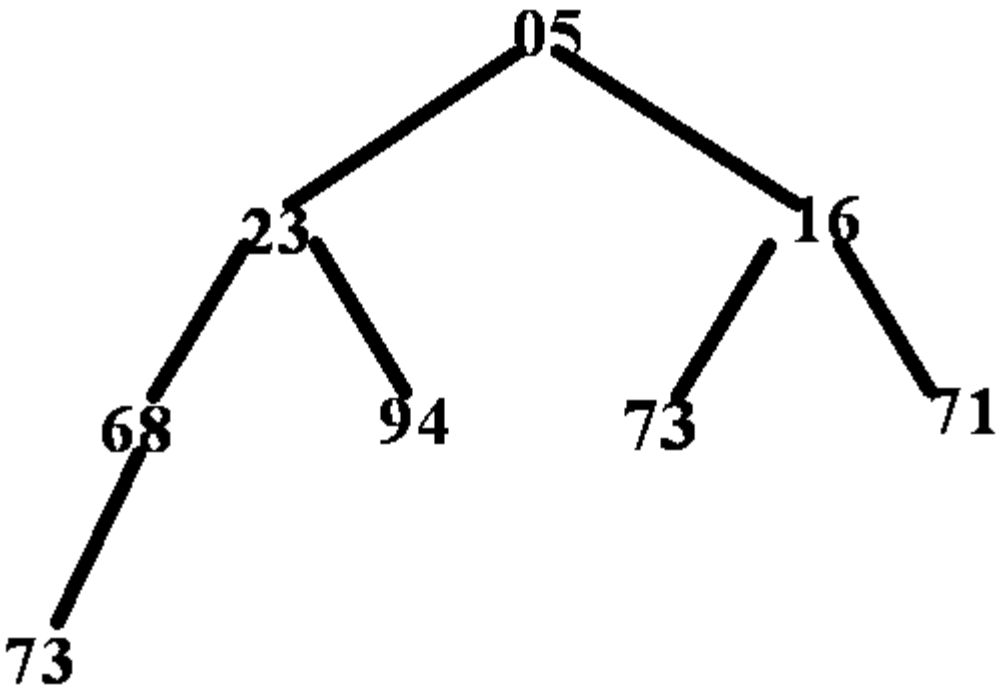
```
// 删除给定下标的元素
bool Remove(int pos, T& node);
//向堆中插入新元素newNode
bool Insert(const T& newNode);
//从堆顶删除最小值
T& RemoveMin();
//从position向上开始调整，使序列成为堆
void SiftUp(int position);
//筛选法函数，参数left表示开始处理的数组下标
void SiftDown(int left);
}
```

建堆过程

- 不必将值一个个地插入堆中，通过交换形成堆
- 假设根的左、右子树都已是堆，并且根的元素名为R。这种情况下，有两种可能：
  - (1) R的值小于或等于其两个子女，此时堆已完成；
  - (2) R的值大于其某一个或全部两个子女的值，此时R应与两个子女中值较小的一个交换，结果得到一个堆，除非R仍然大于其新子女的一个或全部的两个。这种情况下，我们只需简单地继续这种将R“拉下来”的过程，直至到达某一个层使它小于它的子女，或者它成了叶结点







（e）建成的堆

堆成员函数的实现

```
template<T>
MinHeap<T>::MinHeap(const int n)
{
   if(n<=0)
        return;
   CurrentSize=0;
   MaxSize=n;//初始化堆容量为n
   heapArray=new T[MaxSize];//创建堆空间
   //此处进行堆元素的赋值工作
   BuildHeap();
}
```

堆成员函数的实现

```
template<class T>
bool MinHeap<T>::isLeaf(int pos) const
{
   return (pos>=CurrentSize/
   2)&&(pos<CurrentSize);
}
template<class T>
int MinHeap<T>::leftchild(int pos) const
{
   return 2*pos+1;//返回左孩子位置
}
```

堆成员函数的实现

```
template<class T>
int MinHeap<T>::rightchild(int pos) const
{
  return 2*pos+2;//返回右孩子位置
}
template<class T>
int MinHeap<T>::parent(int pos) const
{
  return (pos-1)/2;//返回父结点位置
}
```

筛选法

```
template <class T>
void MinHeap<T>::SiftDown(int position)
{
   int i=position;//标识父结点
   int j=2*i+1;//标识关键值较小的子结点
   T    temp=heapArray[i];//保存父结点
   //过筛
   while(j<CurrentSize){
      if((j<CurrentSize-1)&&
        (heapArray[j]>heapArray[j+1]))
```

筛选法

```
            j++;//j指向数值较小的子结点
            if(temp>heapArray[j]){
                  heapArray[i]=heapArray[j];
                  i=j;
                  j=2*j+1;//向下继续
            }//end if
            else break;
      }//end if
      heapArray[i]=temp;
}
```

建堆

- 从堆的第一个分支结点heapArray[CurrentSize/2-1]开始，自底向上逐步把以各分支结点为根的子树调整成堆

```
template<class T>
void MinHeap<T>::BuildHeap()
{
    //反复调用筛选函数，问题：CurrentSize<2?
    for (int i=CurrentSize/2-1; i>=0; i--)
        SiftDown(i);
}
```

插入新元素

```
template <class T>
bool MinHeap<T>::Insert(const T& newNode)
//向堆中插入新元素newNode
{
  if(CurrentSize==MaxSize)//堆空间已经满
      return FALSE;
  heapArray[CurrentSize]=newNode;
  SiftUp(CurrentSize);//向上调整
  CurrentSize++;
}
```

向上筛选调整堆

```
template<class T>
void MinHeap<T>::SiftUp(int position)
{//从position向上开始调整，使序列成为堆
   int temppos=position;
   T temp=heapArray[temppos];

   while((temppos>0)&&(heapArray[parent(temppos)]
   >temp))  //请比较父子结点直接swap的方法
   {
    heapArray[temppos]=heapArray[parent(temppos)];
   temppos=parent(temppos);
   }
   heapArray[temppos]=temp;
}
```

移出最小值(优先队列出队)

- 要求保持完全二叉树形状，并且剩下的n-1个结点值仍然符合堆的性质
- 可以将堆中最后一个位置上的元素(数组中实际的最后一个元素)移到根的位置上，利用siftdown对堆重新调整

```
template<T>
T& MinHeap<T>::RemoveMin()
                        {//从堆顶删除最小值
    if(CurrentSize==0)
    {
            //空堆
            cout<<"Can't Delete";
            exit(1);
    }
```

移出最小值(优先队列出队)

```
    else
    {
        //交换堆顶和最后一个元素
        swap(0,--CurrentSize);
        if(CurrentSize>1)  // <=1就不要调整了
              //从堆顶开始筛选
              SiftDown(0);
        return heapSize[CurrentSize];
    }//end else
}
```

删除元素

```
template<class T>
bool MinHeap<T>::Remove(int pos, T& node)
{// 删除给定下标的元素
   if((pos<0)||(pos>=CurrentSize))
        return false;
   //指定元素置于最后
   T temp=heapArray[pos];
   heapArray[pos]=heapArray[--CurrentSize];
   SiftUp(pos);//上升筛
   SiftDown(pos);//向下筛，不是SiftDown(0);
   node=temp;
   return true;
}
```

建堆效率

- n个结点的堆，高度d = floor($\log_2 n$ + 1)。根为第0层，则第i层结点个数为2^i，
- 考虑一个元素在堆中向下移动的距离。
  - 大约一半的结点深度为d-1，不移动（叶）。
  - 四分之一的结点深度为d-2，而它们至多能向下移动一层。
  - 树中每向上一层，结点的数目为前一层的一半，而子树高度加一。因而元素移动的最大距离的总数为

$$\sum_{i=1}^{\log n}(i-1)\frac{n}{2^i}=O(n)$$

建堆效率

- 所以，这种算法时间代价为O(n)
- 由于堆有log n层深，插入结点、删除普通元素和删除最小元素的平均时间代价和最差时间代价都是O(log n)

优先队列

- 堆是优先队列的一种自然的实现方法。优先队列存储对象，并根据需要释放具有最小（大）值的对象
- 有些优先队列的应用要求能够改变已存储于队列中的对象的优先权，典型实现方法需要一个辅助数据结构

Huffman编码树

- 要求给出一个具有n个外部结点的扩充二叉树
  - 该二叉树每个外部结点K_i有一个w_i与之对应，作为该外部结点的权
  - 这个扩充二叉树的叶结点带权外部路径长度总和

$$\sum_{i=0}^{n-1} w_i \cdot l_i$$ 最小

（注意不管内部结点，也不用有序）

  - 权越大的叶结点离根越近；如果某个叶的权较小，可能就会离根较远

建立Huffman编码树

- 首先，按照“权”(例如频率)将字母排为一列。
- 接着，拿走前两个字母(“权”最小的两个字母)，再将它们标记为Huffman树的树叶，将这两个树叶标为一个分支结点的两个子女，而该结点的权即为两树叶的权之和。将所得“权”放回序列中适当位置，使“权”的顺序保持。
- 重复上述步骤直至序列中只剩一个元素，则Huffman树建立完毕。

前缀编码

- 一个编码集合中，任何一个字符的编码都不是另外一个字符编码的前缀，这种编码叫作前缀编码
- 这种前缀特性保证了代码串被反编码时，不会有多种可能。例如，
  - 对于上面8个字符，编码为Z(111100)， K(111101)，F(11111)，C(1110)，U(100)，D(101)，L(110)，E(0)。
  - 这是一种前缀编码，对于代码“000110”，可以翻译出唯一的字符串“EEEL”

Huffman编码树的应用

- 设 $D=\{d_0, \ldots, d_{n-1}\}$，

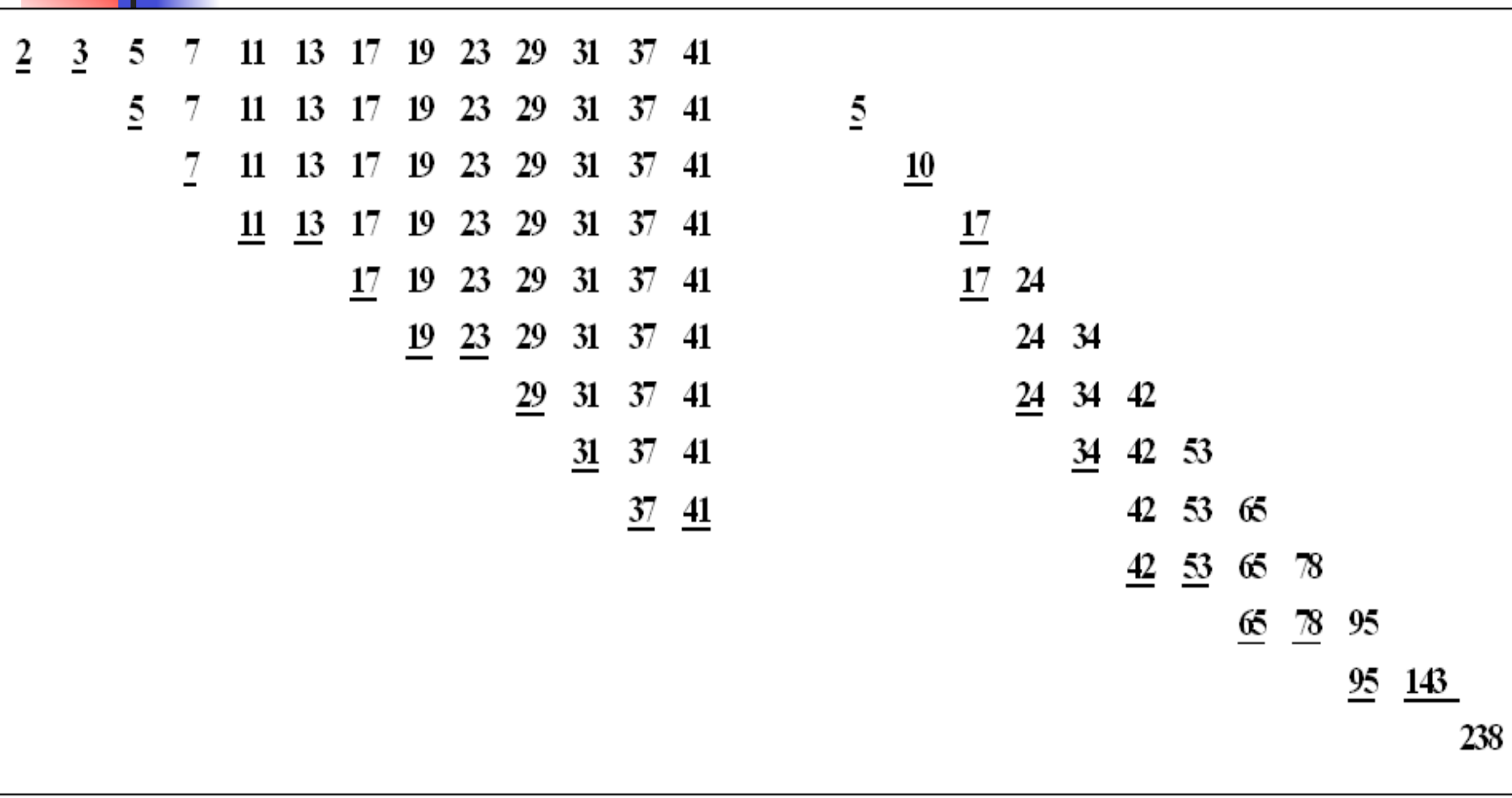
  $W=\{W_0, \ldots, W_{n-1}\}$

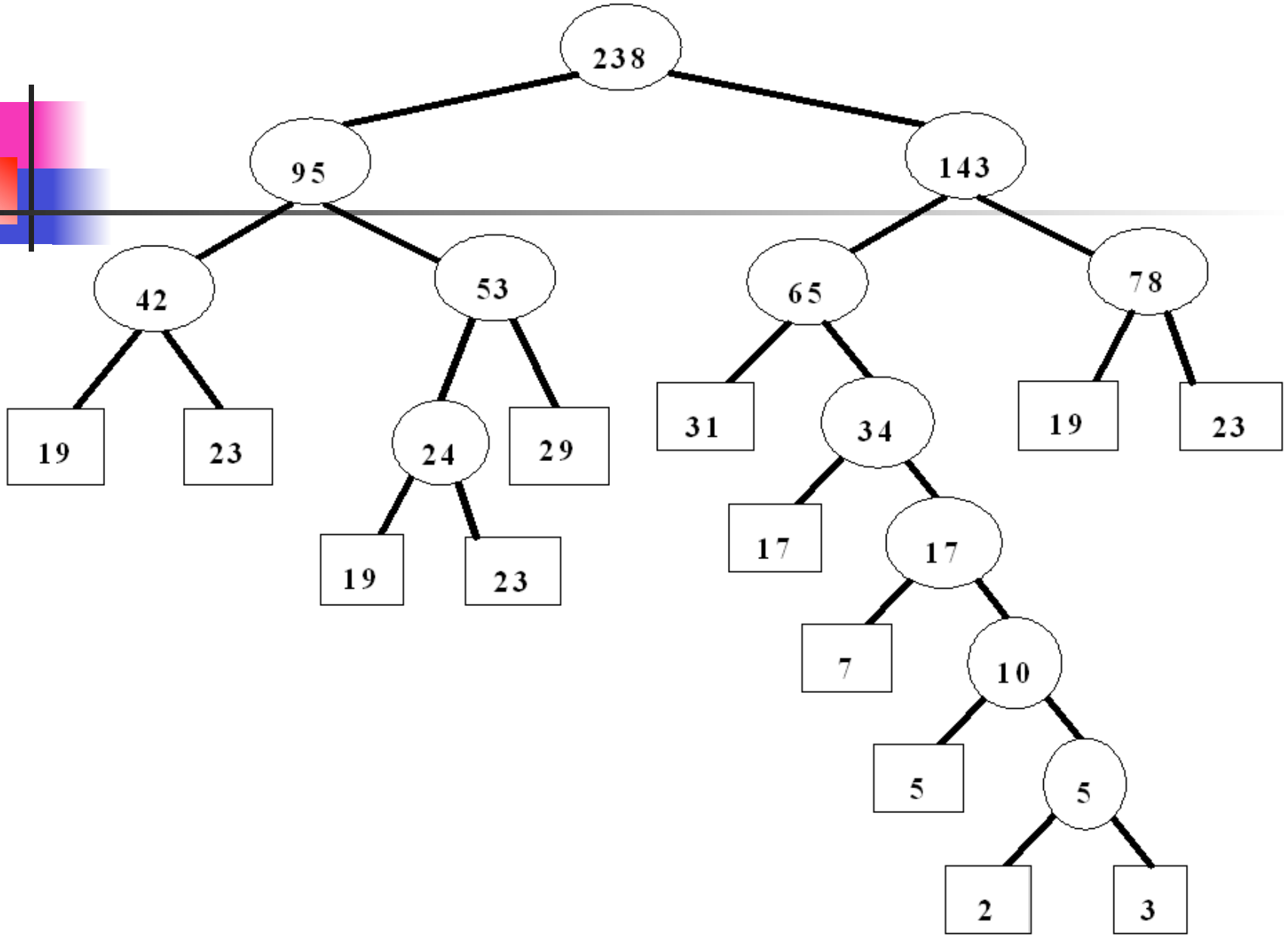
D为需要编码的字符集合，W为D中各字符出现的频率，要对D里的字符进行二进制编码，使得：

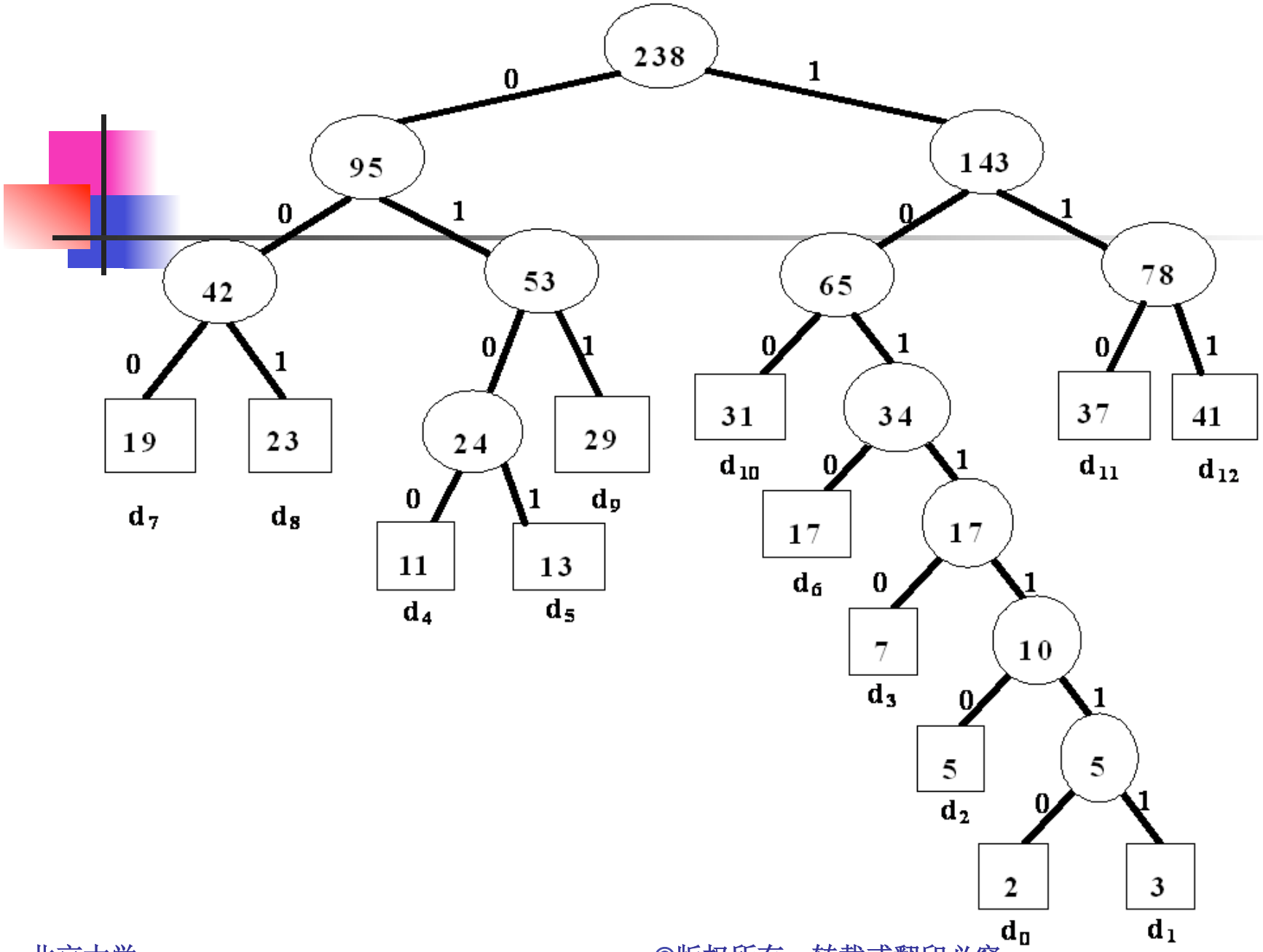
- 通信编码总长最短
- 若$d_i \neq d_j$，则d_i的编码不可能是d_j的编码的开始部分(前缀)

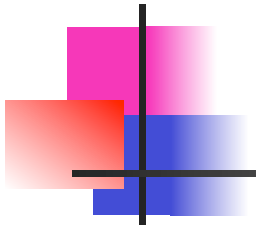
Huffman编码树的应用

- 利用Huffman算法可以这样编码：用d_0，d_1，…，d_{n-1}作外部结点，W_0，W_1，…，W_{n-1}作外部结点的权，构造具有最小带权外部路径长度的扩充二叉树。
- 把从每个结点引向其左子女的边标上号码**0**，把从每个结点引向其右子女的边标上号码**1**。从根到每个叶子的路径上的号码连接起来就是这个叶子代表的字符的编码









- 各字符的二进制编码为：

  d_0：1011110　d_1：1011111

  d_2：101110　d_3：10110

  d_4：0100　d_5：0101

  d_6：1010　d_7：000

  d_8：001　d_9：011

  d_{10}：100　d_{11}：110

  d_{12}：111

- 出现频率越大的字符其编码越短

Huffman编码树的应用

- 用**Huffman**算法构造出的扩充二叉树给出了各字符的编码，同时也用来译码。
- 从二叉树的根开始，用需要译码的二进制位串中的若干个相邻位与二叉树边上标的**0**，**1**相匹配，确定一条到达树叶的路径。一旦到达树叶，则译出了一个字符，再回到树根，从二进制位串中的下一位开始继续译码

Huffman树类

```
template <class T>
class HuffmanTree
{
private:
    HuffmanTreeNode<T>* root;//Huffman树的树根
    //把ht1和ht2为根的Huffman子树合并成一棵以parent为
    //根的二叉树
    void MergeTree(HuffmanTreeNode<T> &ht1,
                   HuffmanTreeNode<T> &ht2,
                   HuffmanTreeNode<T>* parent);
```

Huffman树类

```
   //删除Huffman树或其子树
   void DeleteTree(HuffmanTreeNode<T>* root);
public:
   //构造Huffman树，weight是存储权值的数组，n是数组长度
   HuffmanTree(T weight[],int n);
   //析构函数
   virtual ~HuffmanTree(){DeleteTree(root);};
}
```

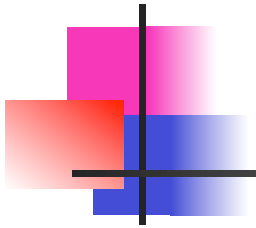
Huffman树的构造

```
template<class T>
HuffmanTree<T>::HuffmanTree(T weight[], int n)
{
   //定义最小值堆
   MinHeap<HuffmanTreeNode<T>> heap(n);
    HuffmanTreeNode<T>
   *parent,&firstchild,&secondchild;
   HuffmanTreeNode<T>* NodeList=new
   HuffmanTreeNode<T>[n];
   for(int i=0;i<n;i++){
   NodeList[i].element=weight[i];

   NodeList[i].parent=NodeList[i].left=NodeList[i].right=
   NULL;
   heap.Insert(NodeList[i]);//向堆中添加元素
   }//end for
```

Huffman树的构造

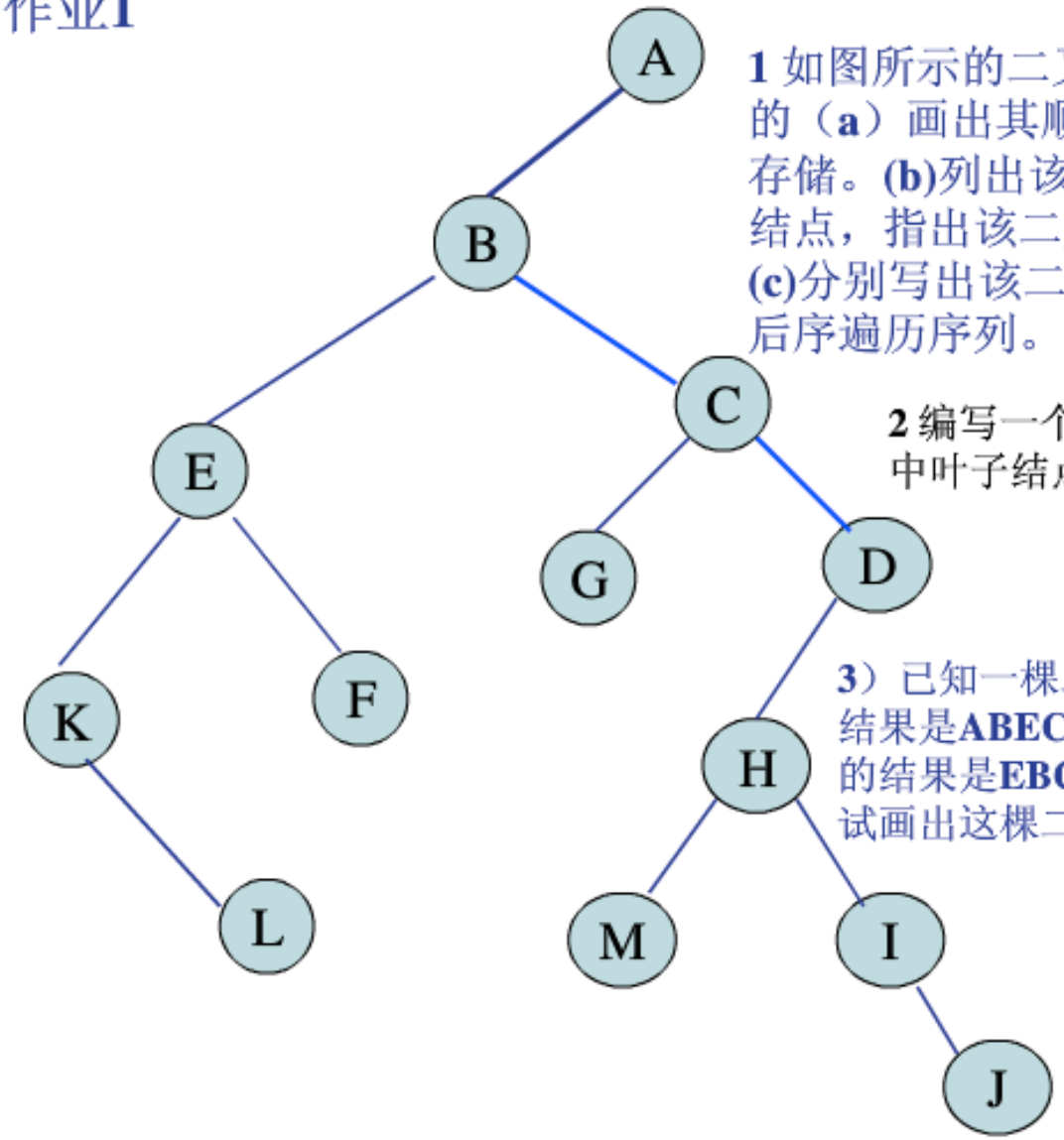
```
for(i=0;i<n-1;i++)
    {//通过n-1次合并建立Huffman树
     parent=new HuffmanTreeNode<T>;
     firstchild=heap.RemoveMin();//选择权值最小的结点
     secondchild=heap.RemoveMin();//选择权值次小的结点
     //合并权值最小的两棵树
     MergeTree(firstchild,secondchild,parent);
     heap.Insert(*parent);//把parent插入到堆中去
     root=parent;//建立根结点
    }//end for
delete []NodeList;
}
```



本章总结

- 二叉树的主要概念与相关性质
- 二叉树的抽象数据类型、存储表示与实现效率
- 二叉树的遍历策略
- 穿线树
- 二叉搜索树及其应用
- 堆的概念、性质与构造
- **Huffman**树的主要思想与具体应用

作业1

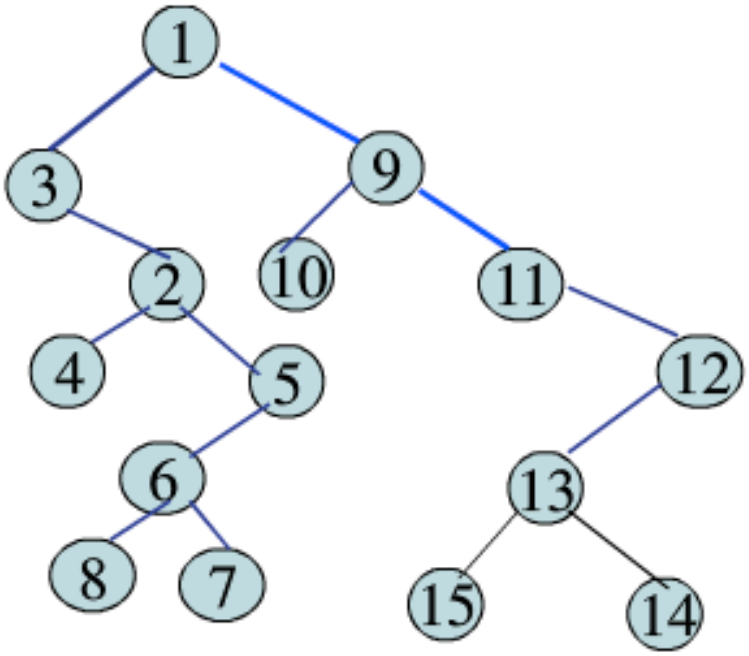


1 如图所示的二叉树
的（a）画出其顺序存储和二叉链表存储。**(b)**列出该二叉树的叶子结点，指出该二叉树的深度
(c)分别写出该二叉树的先序，中序，后序遍历序列。

2 编写一个算法统计二叉树中叶子结点的个数

3）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。

1）将如图所示的二叉树进行中序线索化。



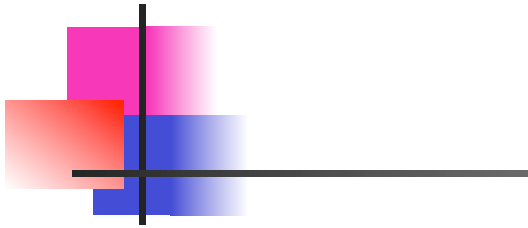
作业：2

2 试找出分别满足下面条件的所有二叉树：

（1）前序序列和中序序列相同；　（2）中序序列和后序序列相同；

（3）前序序列和后序序列相同；　（4）前序、中序、后序序列均相同

3 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



The End