数据结构与算法
第四章 二叉树

张铭

http://db.pku.edu.cn/mzhang/DS/

北京大学信息科学与技术学院

“数据结构与算法”教学小组

©版权所有，转载或翻印必究

主要内容

 ©版权所有，转载或翻印必究

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

©版权所有，转载或翻印必究

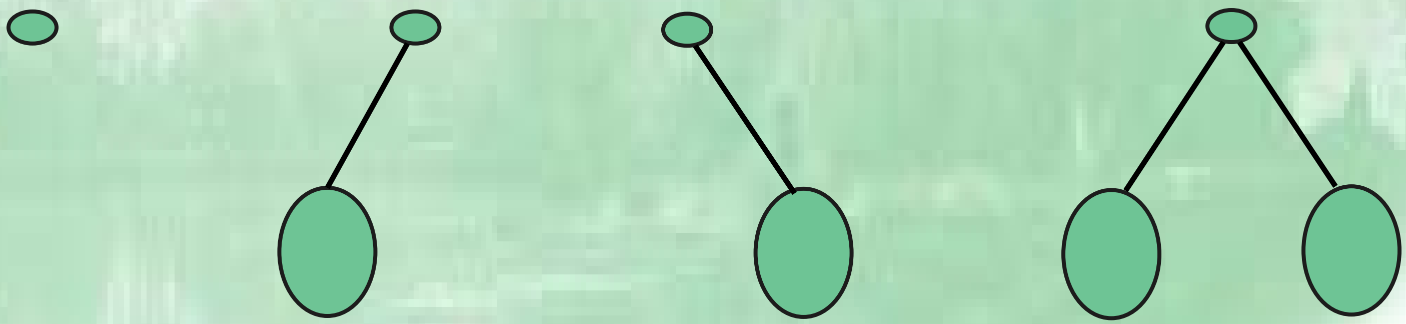
二叉树的定义

- 树的特例
- 递归的定义：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

©版权所有，转载或翻印必究

二叉树的五种基本形态

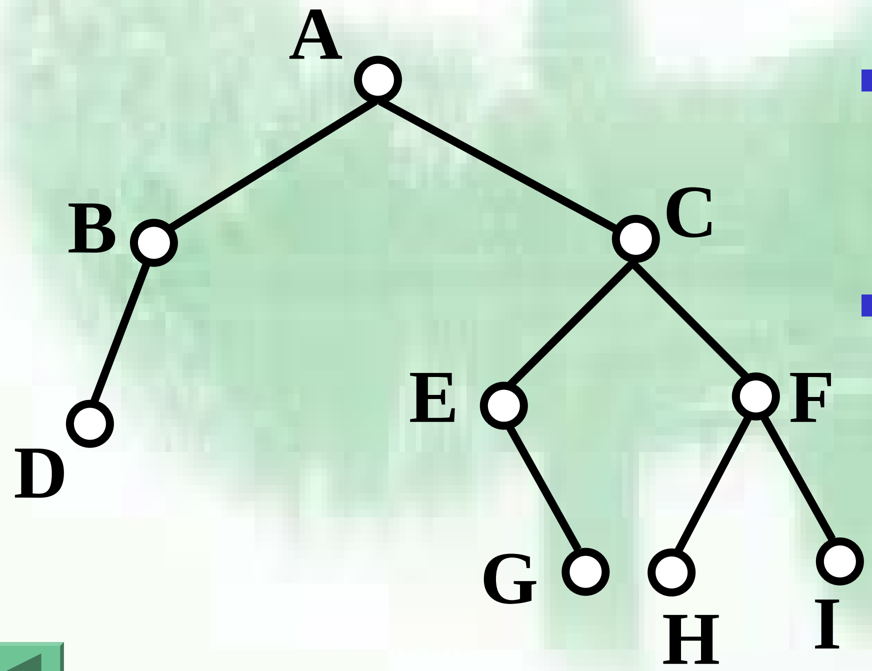
- 二叉树可以是空集合，因此根可以有空的左子树或右子树，或者左右子树皆为空



(a)空 (b)独根 (c)空右 (d)空左 (e)左右都不空

©版权所有，转载或翻印必究

二叉树的相关术语

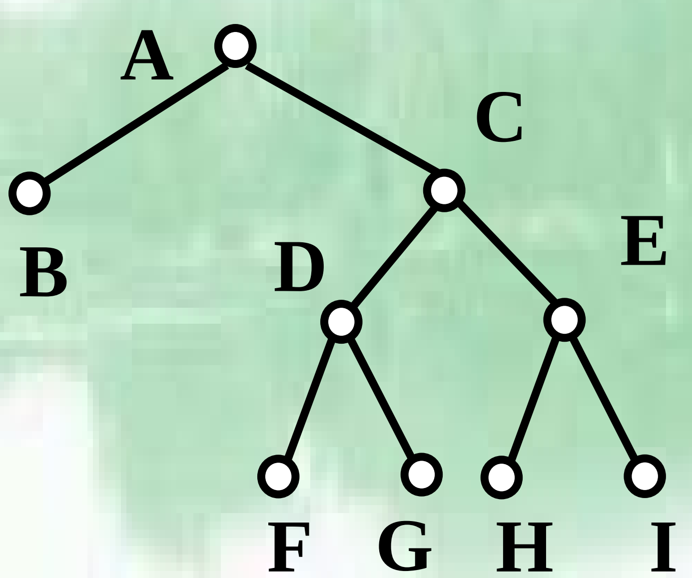


- 结点
  - 根结点、子结点、父结点、左子结点、右子结点
  - 分支结点、叶结点
- 边
  - 路径、路径长度
  - 祖先、后代
- 深度、高度、层数、宽度
  - 二叉树的高度定义为二叉树中层数最大的叶结点的层数加1
  - 二叉树的深度定义为二叉树中层数最大的叶结点的层数

©版权所有，转载或翻印必究

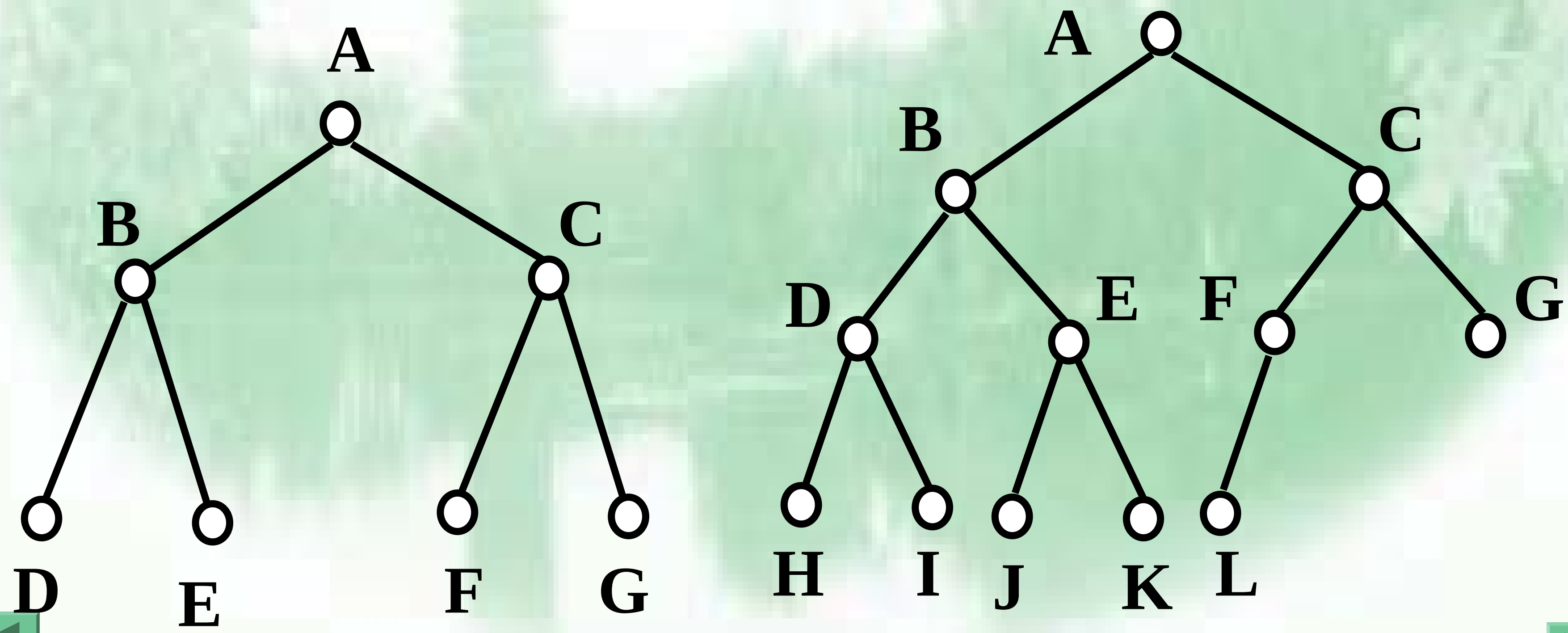
满二叉树

- 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



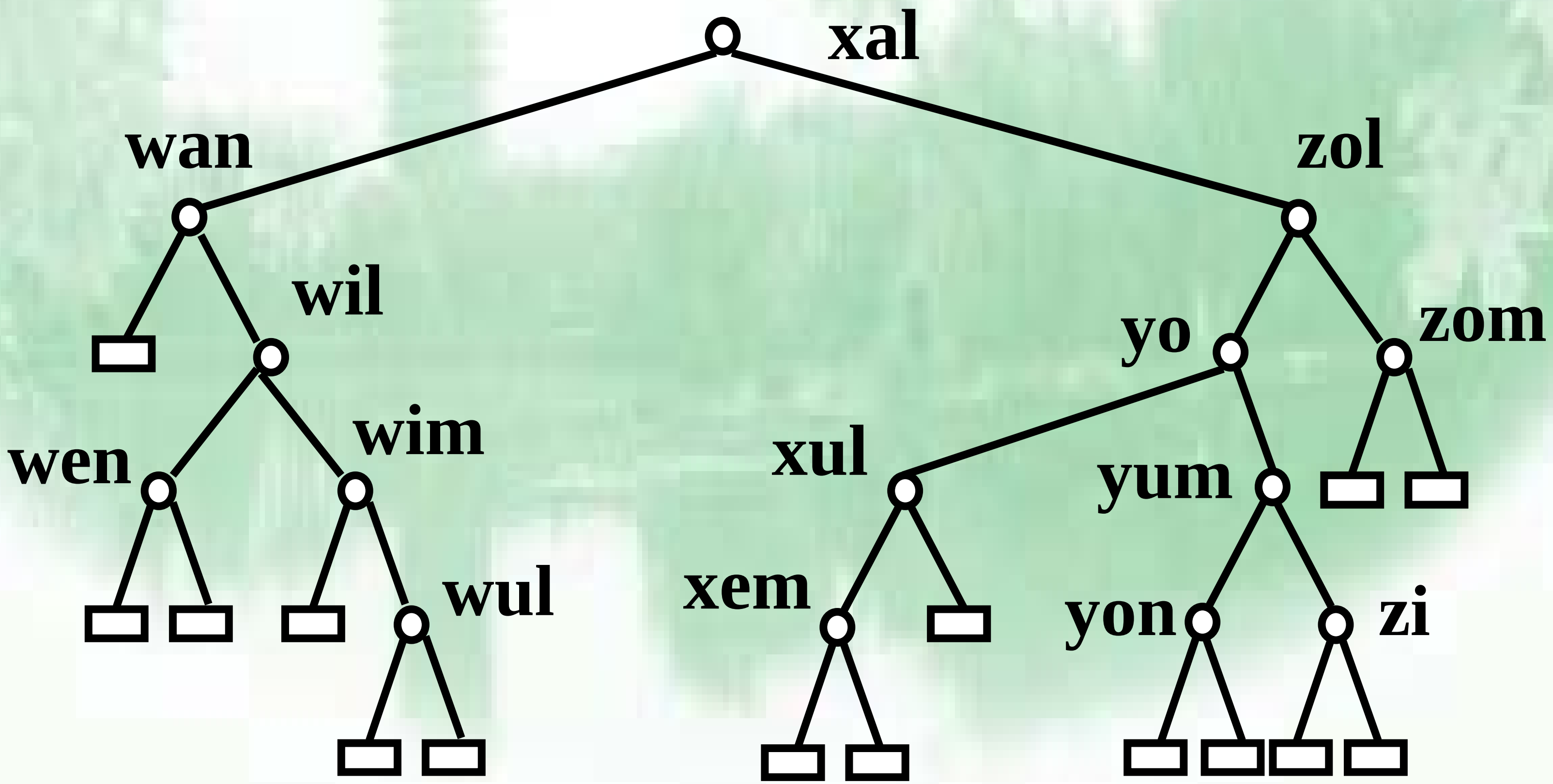
完全二叉树

- 最多只有最下面的两层结点度数可以小于2
- 最下一层的结点都集中在最左边



扩充二叉树

- 所有空子树，都增加空树叶



扩充二叉树

- 扩充二叉树是满二叉树
  - 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
- 路径长度
  - 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：设结点总数为n，叶结点数m，分支结点数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

©版权所有，转载或翻印必究

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。

©版权所有，转载或翻印必究

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 $n_0 = n_2 + 1$

©版权所有，转载或翻印必究

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

©版权所有，转载或翻印必究

4.3 二叉树的抽象数据类型

- 逻辑结构 + 运算：
  - 针对整棵树
    - 初始化二叉树
    - 合并两棵二叉树
  - 围绕结点
    - 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

```
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:
            T  element;    //二叉树结点数据域
         // 实现时，可以补充private的左右
         //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* l,
                   BinaryTreeNode* 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;
```


©版权所有，转载或翻印必究

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*) ;
//设置当前结点的右子树
void  setRightchild(BinaryTreeNode*) ;
//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)
};
```


©版权所有，转载或翻印必究

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;
```


©版权所有，转载或翻印必究

4.3 二叉树的抽象数据类型

```
public:
  BinaryTree();                          //构造函数
  ~BinaryTree();                         //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();
  //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);
```


©版权所有，转载或翻印必究

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)；
```


©版权所有，转载或翻印必究

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);
};
```


©版权所有，转载或翻印必究

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
- 二叉树的结点的线性化

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

©版权所有，转载或翻印必究

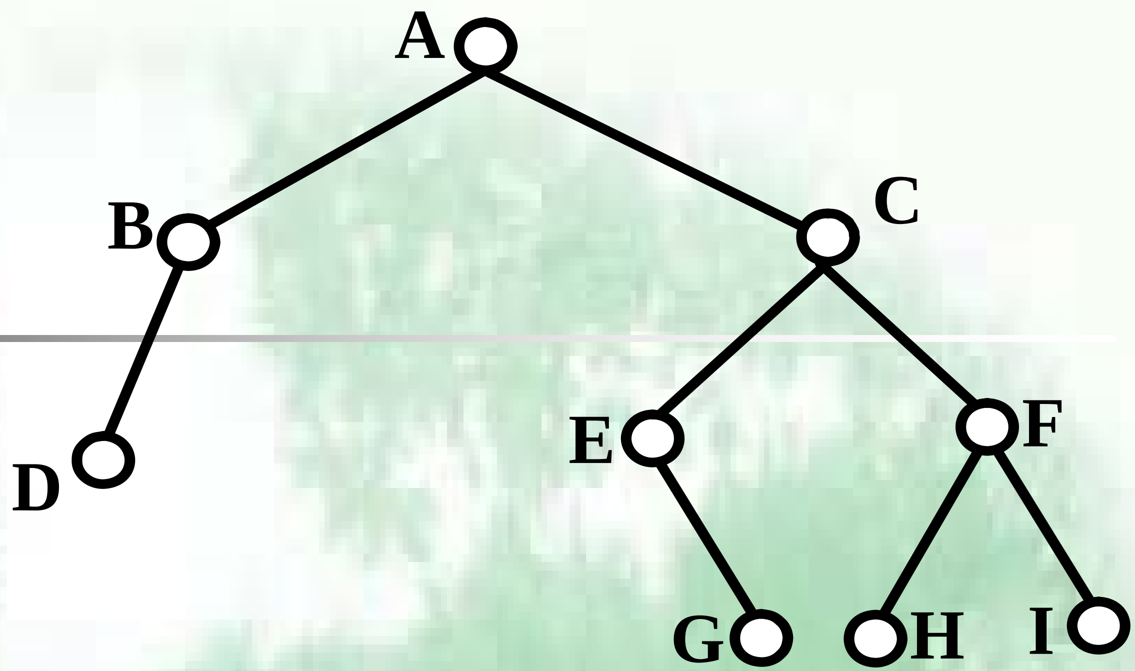
深度优先周游二叉树

变换根结点的周游顺序，得到三种方案：

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。



- ① 前序周游：**ABDCEGFHI**
- ② 中序周游：**DBAEGCHFI**
- ③ 后序周游：**DBGEHIFCA**

©版权所有，转载或翻印必究

深度优先周游二叉树（递归）

```
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
    }
}
```


©版权所有，转载或翻印必究

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

非递归深度优先周游二叉树

- 深度优先二叉树周游是递归定义的
  - 栈是实现递归的最常用的结构
    - 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——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
```


©版权所有，转载或翻印必究

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 }
}}
```

©版权所有，转载或翻印必究

非递归中序周游二叉树

- 遇到一个结点
  - 把它推入栈中
  - 周游其左子树
- 周游完左子树
  - 从栈顶托出该结点并访问之
  - 按照其右链地址周游该结点的右子树

©版权所有，转载或翻印必究

非递归中序周游二叉树

- template<class T> void BinaryTree<T>::InOrderWithoutRecusion(BinaryTreeNode<T>* root) {

```
using std::stack;          //使用STL中的stack
stack<BinaryTreeNode<T>* > aStack;
BinaryTreeNode<T>* pointer=root;
while(!aStack.empty()||pointer) {
    if(pointer){
        // Visit(pointer->value()); 前序访问点
        aStack.push(pointer); //当前结点地址入栈
       //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();
        aStack.pop();              //栈顶元素退栈
        Visit(pointer->value());//访问当前结点
        //当前链接结构指向右孩子
        pointer=pointer->rightchild();
}//end else
} //end while
}
```

非递归后序周游二叉树

- 遇到一个结点
  - 把它推入栈中
  - 周游它的左子树
- 左子树周游结束后
  - 按照其右链地址周游该结点的右子树
- 周游遍右子树后
  - 从栈顶托出该结点并访问之

©版权所有，转载或翻印必究

非递归后序周游二叉树

- 左子树返回 **vs** 右子树返回 **?**
- 给栈中元素加上一个特征位
  - **Left**表示已进入该结点的左子树，将从左边回来
  - **Right**表示已进入该结点的右子树，将从右边回来

©版权所有，转载或翻印必究

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;
};
```


©版权所有，转载或翻印必究

非递归后序周游二叉树

```
template<class T>
void BinaryTree<T>::PostOrderWithoutRecusion
(BinaryTreeNode<T>* root)
//非递归中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
```


©版权所有，转载或翻印必究

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();
```


©版权所有，转载或翻印必究

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
}
```

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

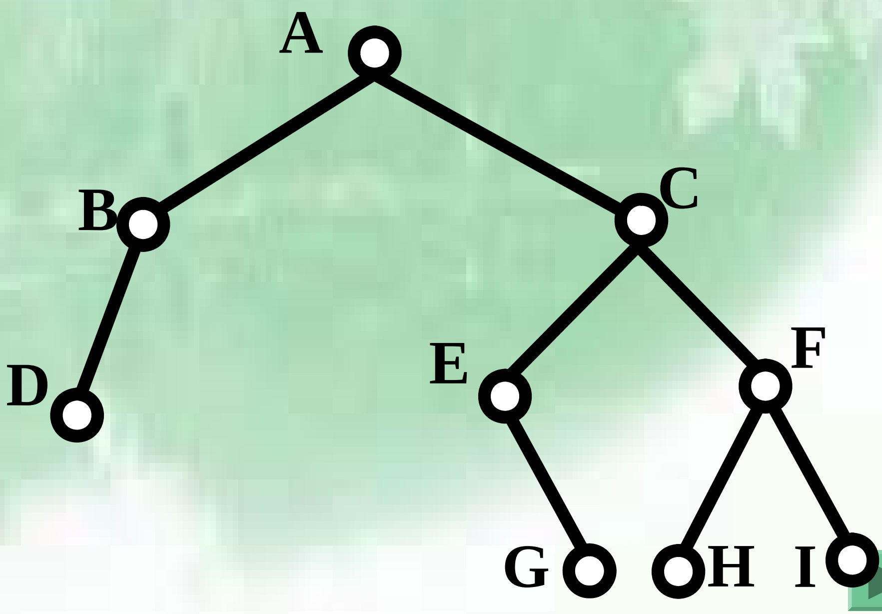
©版权所有，转载或翻印必究

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：**

  A B C D E F G H I



广度优先周游二叉树

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


©版权所有，转载或翻印必究

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
}
```

算法代价分析

时间

- 在各种周游中，每个结点都被访问且只被访问一次，时间代价为**O(n)**
- 如果计算非递归保存入出栈（或队列）时间
  - 广度优先，正好每个结点入/出队一次，**O(n)**
  - 前序、中序，某些结点入/出栈一次，不超过**O(n)**
  - 后序周游，每个结点分别从左、右边各入/出一次，**O(n)**

空间

- 最好**O(log n)**，最坏**O(n)**
- 深度优先
  - 栈的深度与树的高度有关
- 广度优先
  - 与树的最大宽度有关

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

©版权所有，转载或翻印必究

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

©版权所有，转载或翻印必究

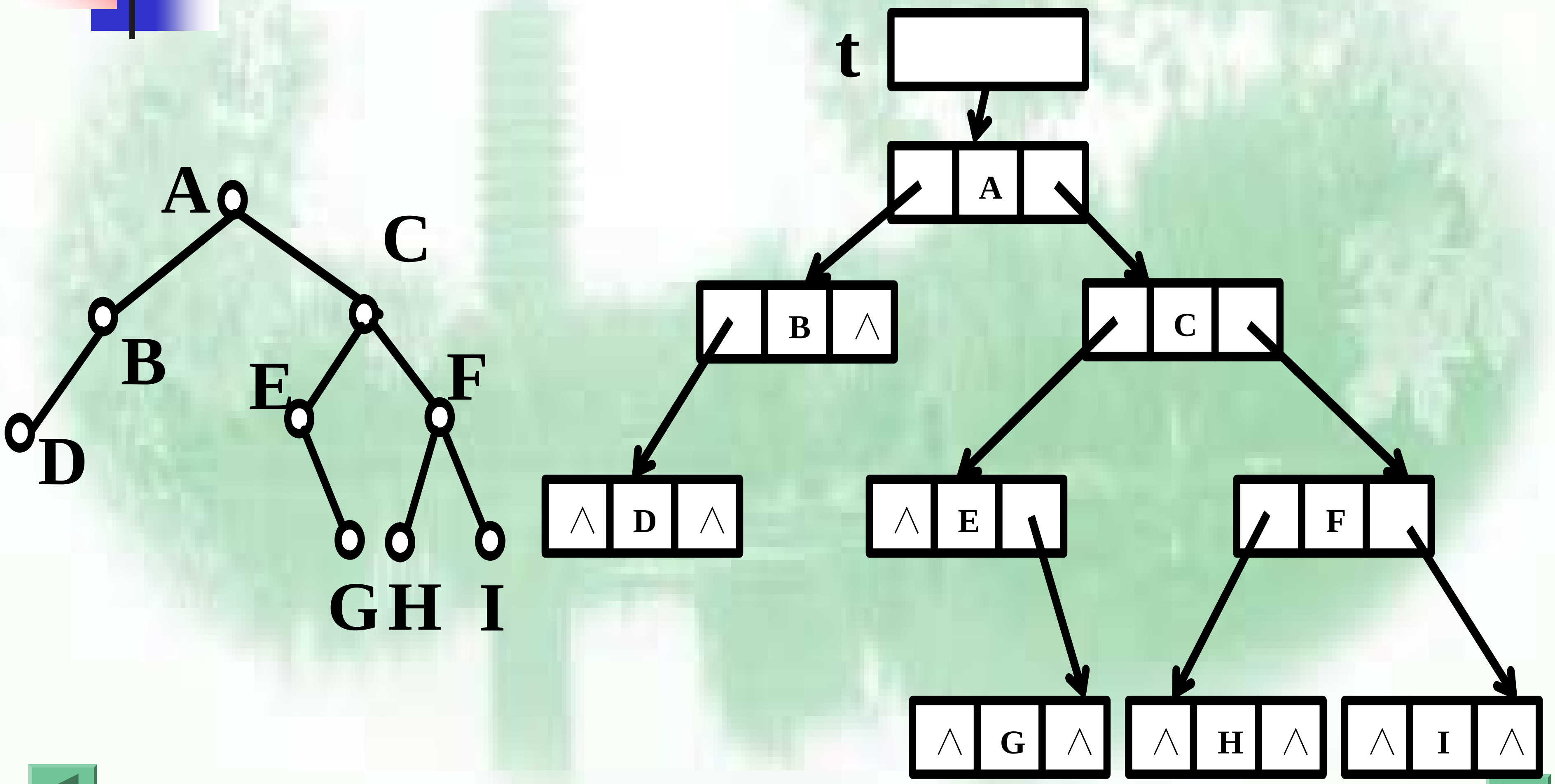
用指针实现二叉树

- 非线性结构最自然的方法是**链接**的方法
- **指针left和right**，分别指向结点的左子女和右子女
  - 空指针

left	info	right

 ©版权所有，转载或翻印必究

用指针实现二叉树



用指针实现二叉树

- 其他的链接表示法
  - 例如“三叉链表”
    - 增加一个指向父母的指针**parent**
    - 具有“向上”访问的能力

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

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或其子树
   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
   }
};
```


©版权所有，转载或翻印必究

4.5.2 空间开销分析

- 结构性开销γ
  - 辅助空间
    - 保存数据结构的逻辑特性
    - 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
- 紧凑结构 vs 非紧凑结构
- 二叉链表的存储是非紧凑结构

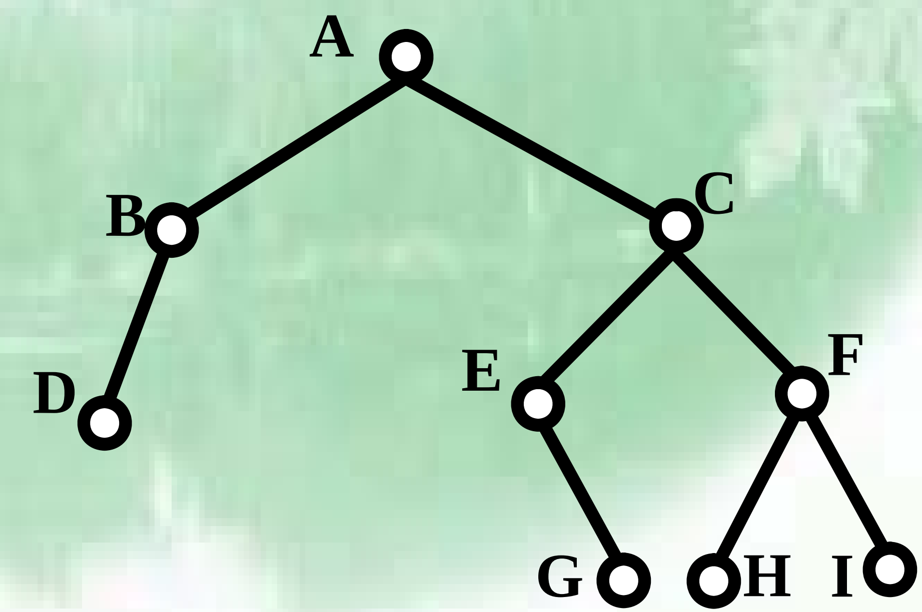
©版权所有，转载或翻印必究

空间开销分析

根据满二叉树定理：一半的指针是空的

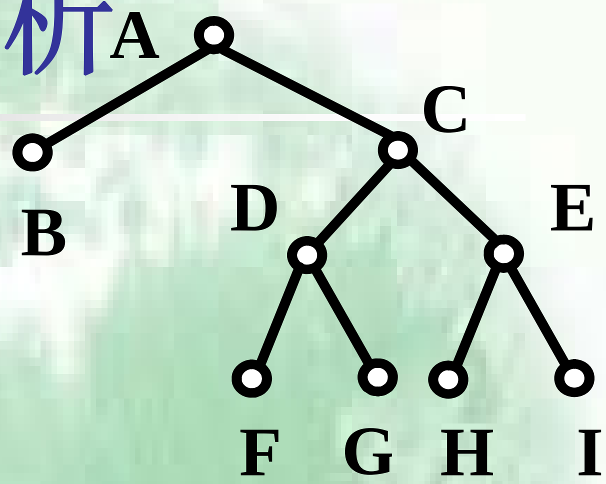
- 每个结点存两个指针、一个数据域
  - 总空间 $(2p + d)n$
  - 结构性开销：$2pn$
  - 如果 $p = d$, 则

    $2p/(2p + d) = 2/3$



©版权所有，转载或翻印必究

空间开销分析



- 去掉满二叉树叶结点中的指针

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

则结构性开销为 1/2 （假设$p = d$）

- 如果只在叶结点存数据，d n/2

在分支结点存储指针，2p n/2 = p n

则结构性开销为

  - $pn / (pn + dn / 2) \Rightarrow 2/3$ （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

 ©版权所有，转载或翻印必究

空间开销分析

- C+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
- 早期节省内存资源
  - 利用结点指针的一个空闲位（一个**bit**）来存储结点所属的类型
  - 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

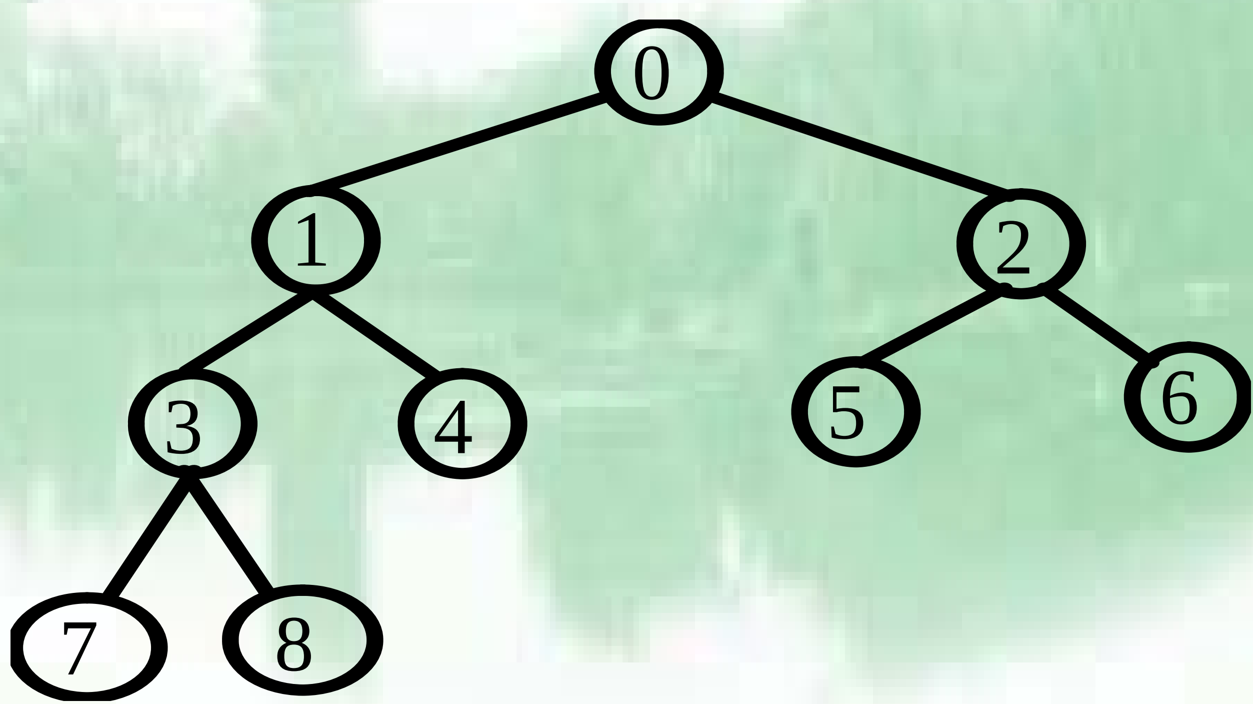
4.5.3 用数组实现完全二叉树

- 顺序方法存储二叉树
  - 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
  - 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
- 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

©版权所有，转载或翻印必究

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

- 从结点的编号就可以推知其父母、子女、兄弟的编号
  - 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

©版权所有，转载或翻印必究

完全二叉树的下标公式

- 当0<i<n时，结点i的父母是结点$\lfloor (i-1)/2 \rfloor$
- 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

4.5.4 穿线二叉树

- 穿线树：在二叉链表中
  - 空左指针指向结点在某种周游序列下的前驱
  - 空的右指针指向结点在同一种周游序列下的后继
- 中序穿线树，前序穿线树，后序穿线树
  - 半索 vs 全索
- 目的：利用空指针的存储空间，建立周游线索

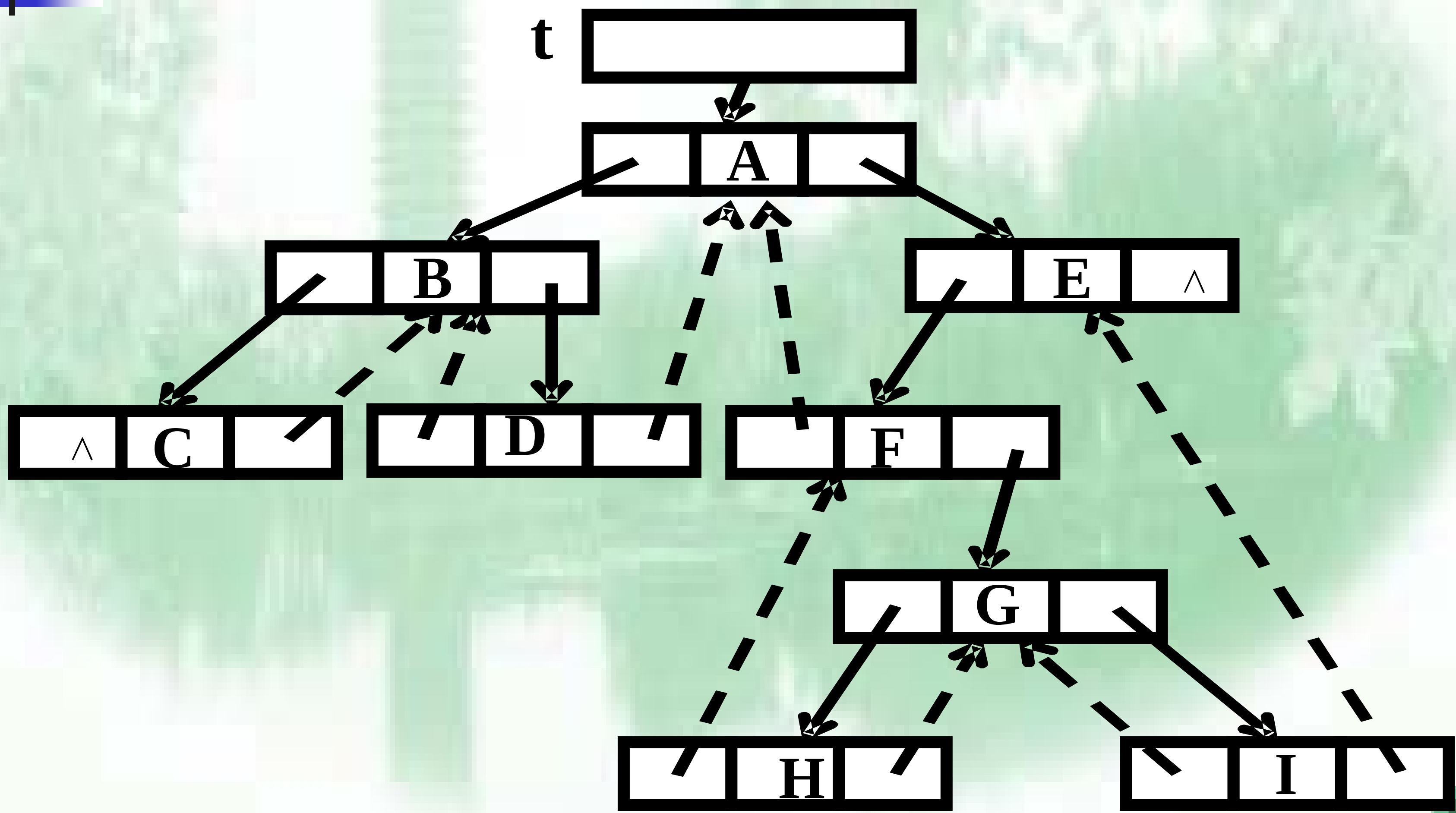
©版权所有，转载或翻印必究

穿线二叉树

- 标志位lTag和rTag：区分线索和指针
  - 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

left	lTag	info	rTag	right

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)
    {};
```


©版权所有，转载或翻印必究

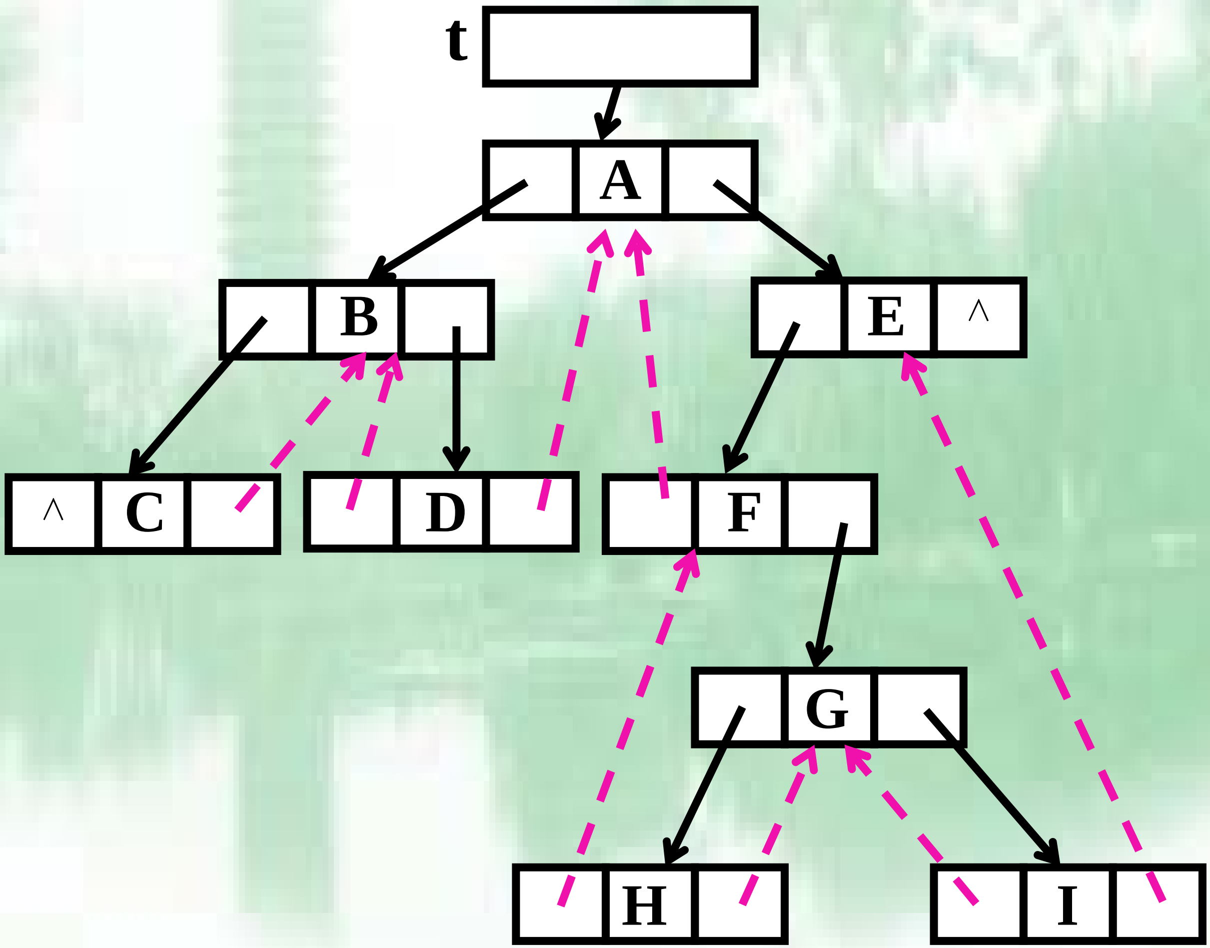
穿线二叉树结点类

```
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);
};
```

中序穿线二叉树：示例



 ©版权所有，转载或翻印必究

中序线索化二叉树：递归实现

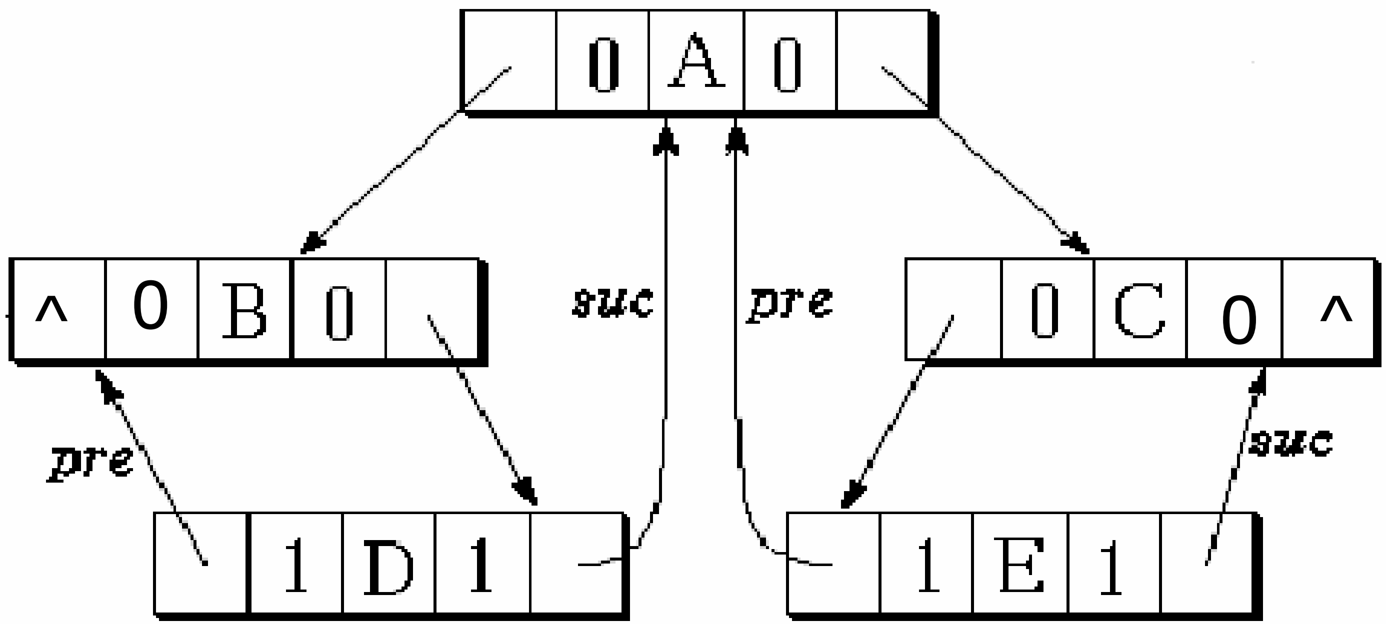
```
template <class T>      void
ThreadBinaryTree<T>::InThread
        (ThreadBinaryTreeNode<T>*root,
         ThreadBinaryTreeNode<T>* &pre)  {
  if (root!=NULL){
        //中序线索化左子树
        InThread(root->leftchild(),pre);
        if(root->leftchild()==NULL){
            //建立前驱线索
            root->left=pre;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


©版权所有，转载或翻印必究

中序线索化二叉树：递归实现

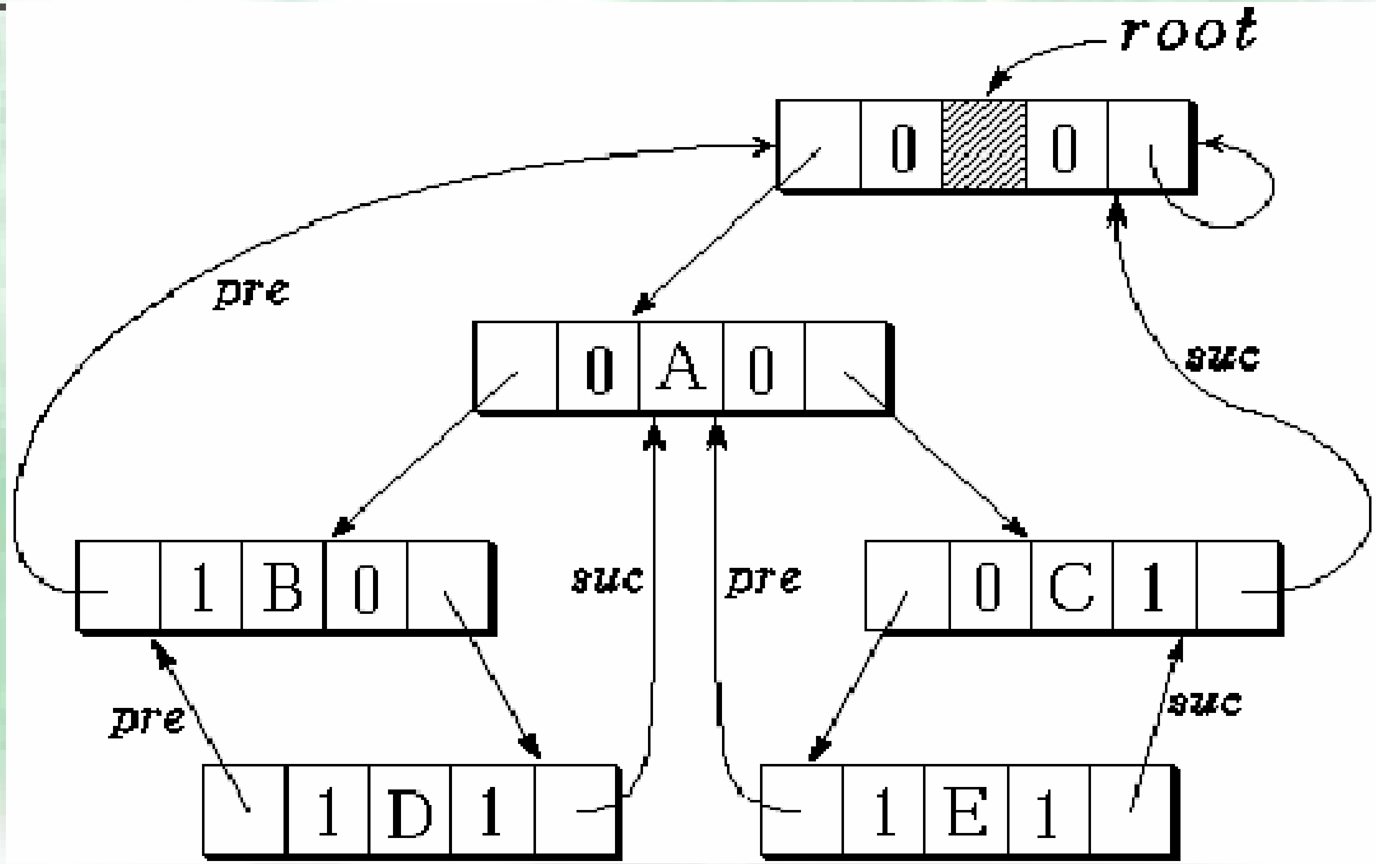
```
        if (pre)  root->lTag=1;
  }
  if((pre)&&(pre->rightchild()==NULL))
  {//建立后继线索
        pre->right=root;
        pre->rTag=1;
  }//end if
  pre=root;
  InThread(root->rightchild(),pre); //中序线索化右子树
}//end if
}
```


©版权所有，转载或翻印必究



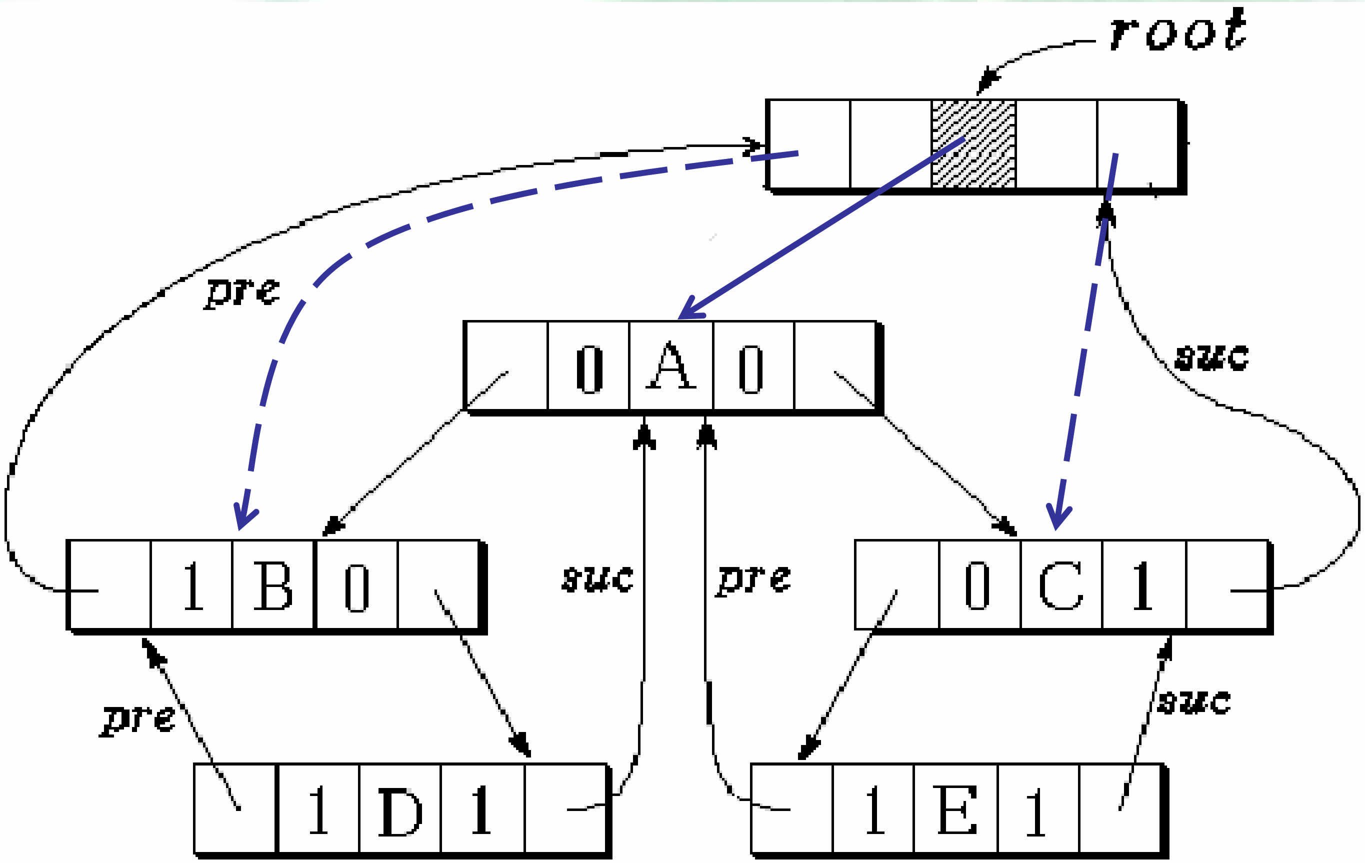
 ©版权所有，转载或翻印必究

带头结点的中序穿线二叉树



©版权所有，转载或翻印必究

另一种头结点



 ©版权所有，转载或翻印必究

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；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

©版权所有，转载或翻印必究

1	2	3	4	5	6	7	8	9
15	17	18	22	35	51	60	88	93

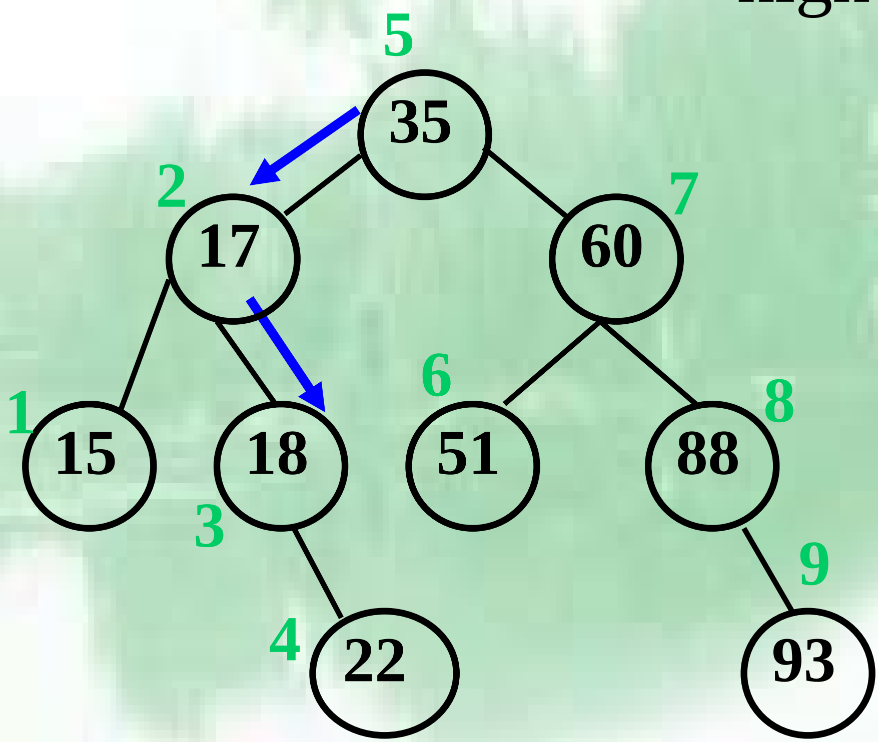
low

mid

high

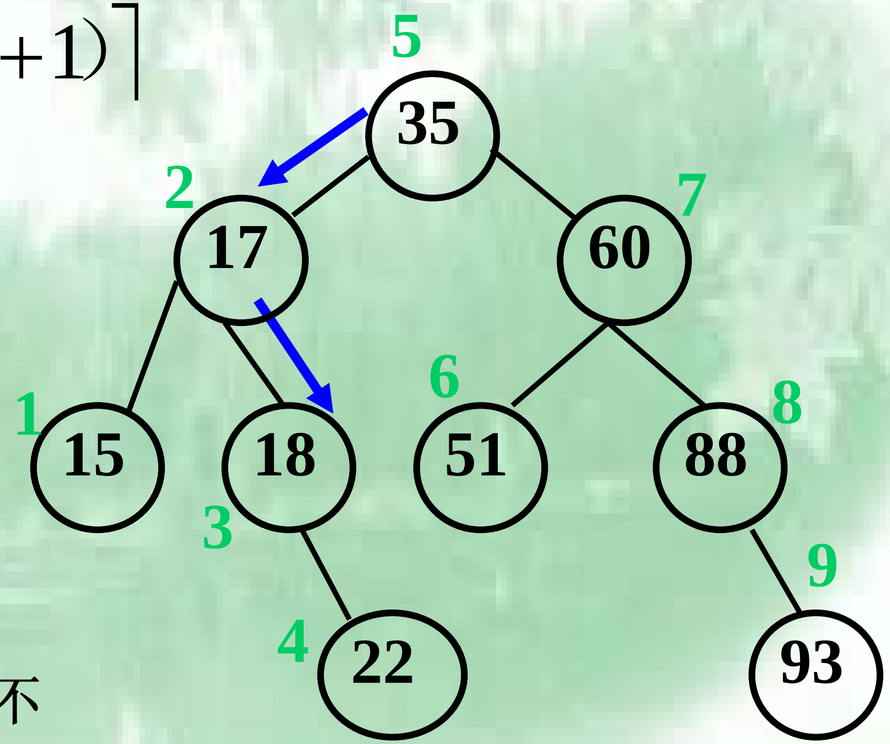
二分法检索

判定树

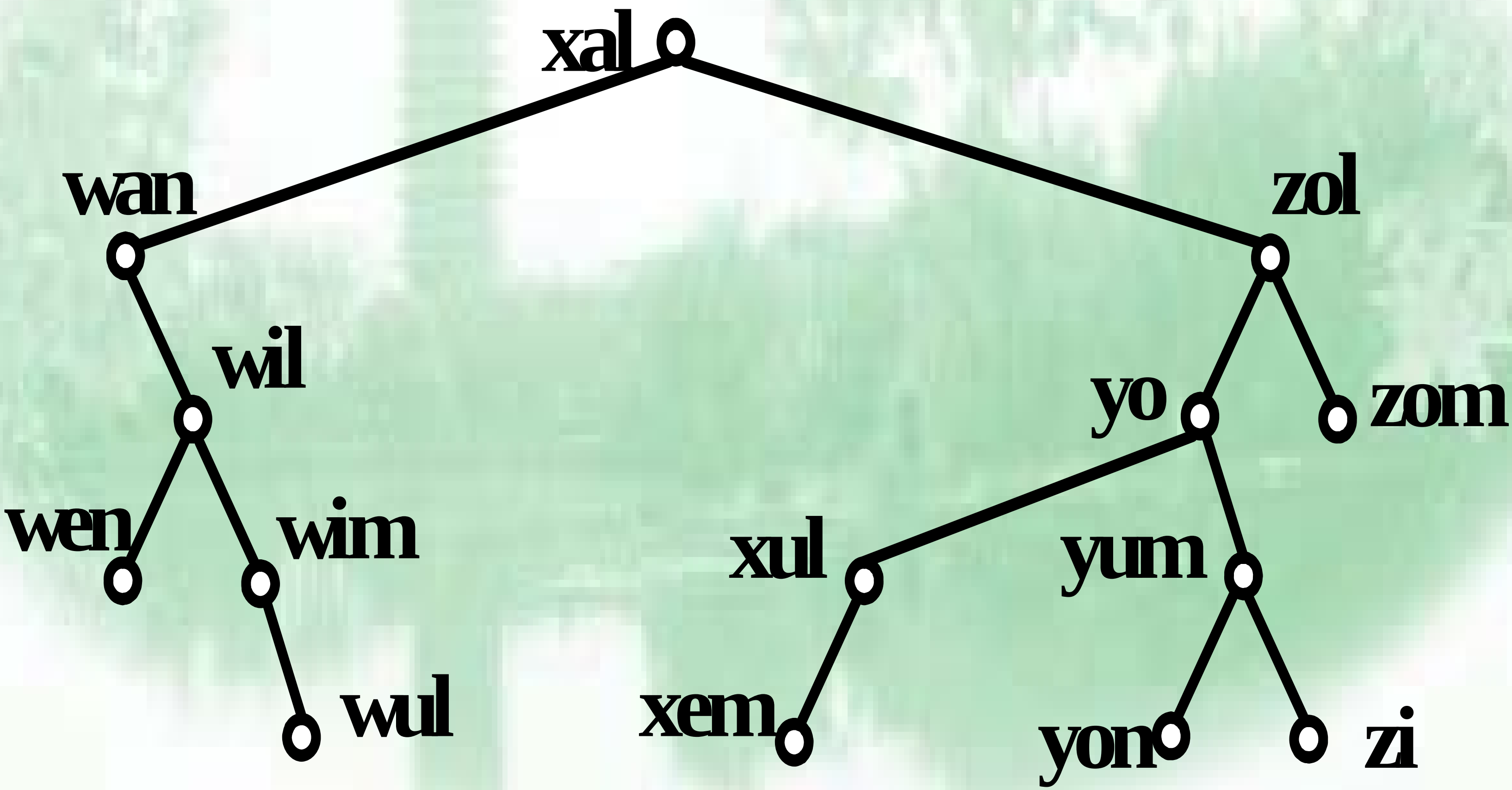


二分法检索性能分析

- 最大检索长度为 $\lceil \log_2(n+1) \rceil$
- 失败的检索长度是 $\lceil \log_2(n+1) \rceil$ 或 $\lfloor \log_2(n+1) \rfloor$
- 在算法复杂性分析中
  - **log n**是以**2**为底的对数
  - 以其他数值为底，算法量级不变



BST图示



©版权所有，转载或翻印必究

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被找到或者遇上了树叶
- 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

©版权所有，转载或翻印必究

二叉搜索树的插入

- 插入新结点要符合二叉搜索树的定义
  - 将新结点的关键码值与树根的关键码值比较，若新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

©版权所有，转载或翻印必究

二叉搜索树的插入

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


©版权所有，转载或翻印必究

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();
}
```


©版权所有，转载或翻印必究

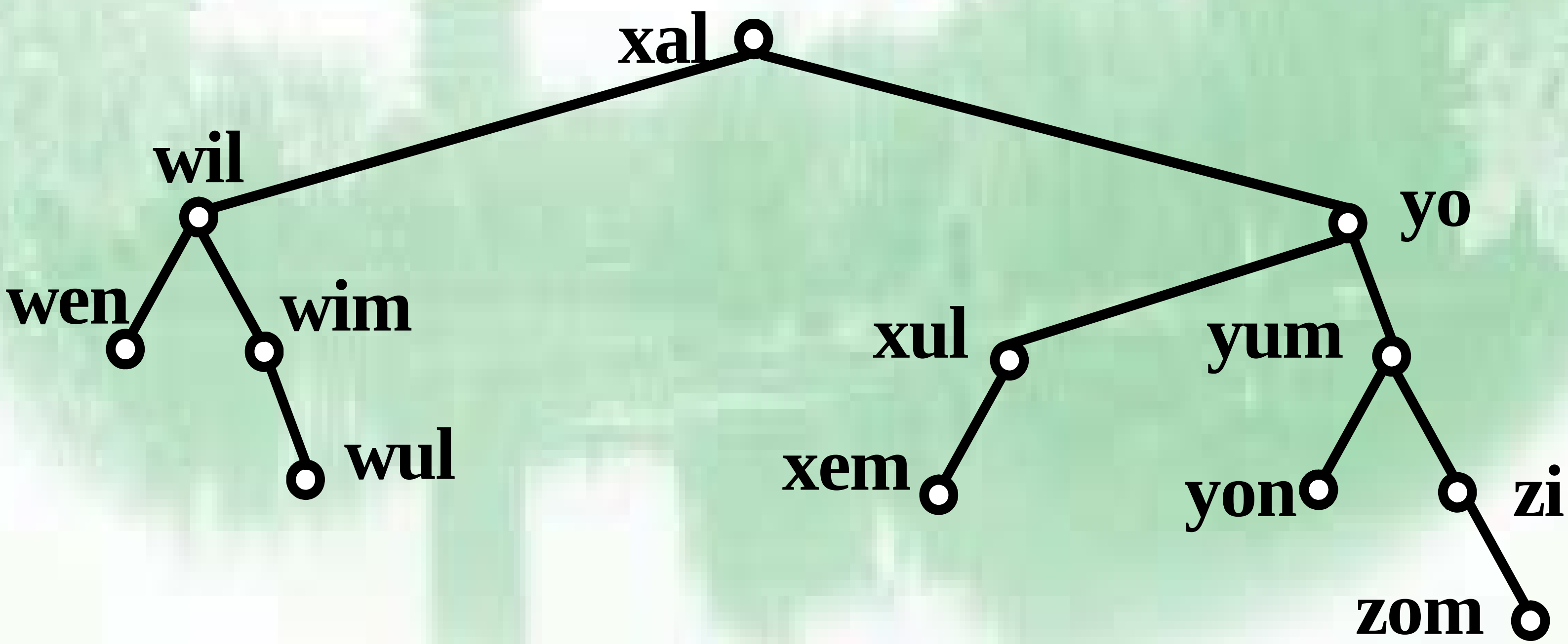
二叉搜索树的插入

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


©版权所有，转载或翻印必究

二叉树结点删除算法(未改进)

- 从教材**P112**图**4.18**二叉排序树中删除**wan**和**zol**后得到的二叉排序树



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**↑

©版权所有，转载或翻印必究

改进的BST结点删除算法(简化)

```
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;
```


©版权所有，转载或翻印必究

```
//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 )
      temppointer = pointer->rightchild();
  else {
    //左子树不为空，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 ) {
       tempparent = temppointer;
       temppointer = temppointer->rightchild();
     } // end of while
```


©版权所有，转载或翻印必究

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


©版权所有，转载或翻印必究

删除rt右子树中最小结点

```
template <class T>
BinaryTreeNode*
  BST::deletemin(BinaryTreeNode <T> *& rt) {
    assert(rt != NULL);
    if (rt->left != NULL)
        return deletemin(rt->left);
    else { //找到右子树中最小，删除
        BinaryTreeNode <T> *temp = rt;
        rt = rt->right;
        return temp;
    }
  }
```

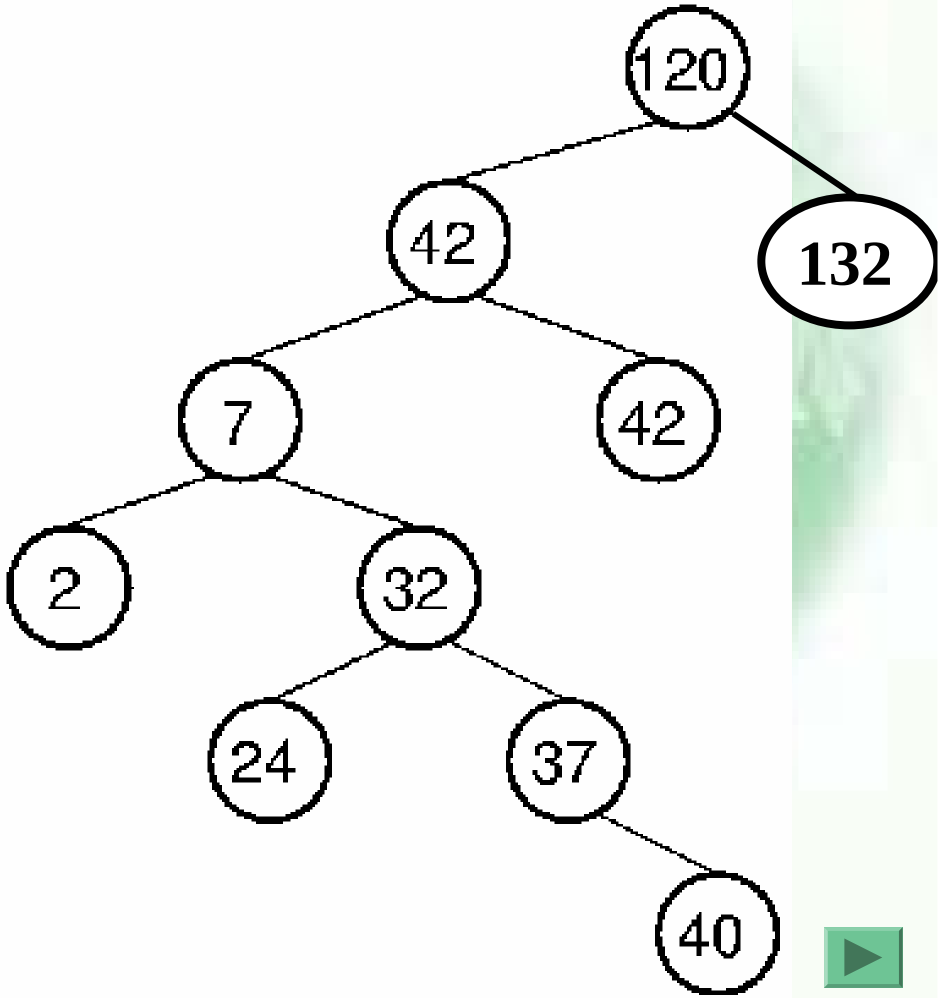

©版权所有，转载或翻印必究

```
template <class T>
void BST::removehelp(BinaryTreeNode <T> *& rt, const T
  val) {
    if (rt==NULL) cout<<val<<" is not in the tree.\n";
    else if (val < rt->value())
        removehelp(rt->left, val);
    else if (val > rt->value())
        removehelp(rt->right, val);
    else {                     // 真正的删除
        BinaryTreeNode <T> * temp = rt;
        if (rt->left == NULL)  rt = rt->right;
        else if (rt->right == NULL)  rt = rt->left;
        else {
            temp = deletemin(rt->right);
            rt->setValue(temp->value());
        }
        delete temp;
} }
```


©版权所有，转载或翻印必究

允许有重复关键码时

- 重复关键码应有规律地出现，例如右子树
  - 插入、检索、删除
- 删除，应删右子树中最小



二叉搜索树总结

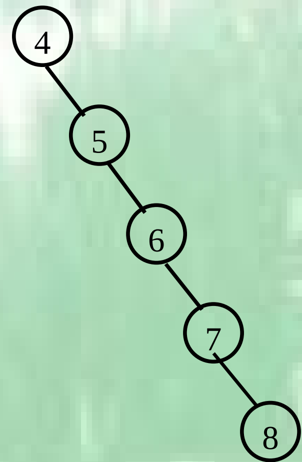
- 组织内存索引
  - •二叉搜索树是适用于内存储器的一种重要的树形索引
  - •外存常用B/B+树
- 保持性质 vs 保持性能

  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

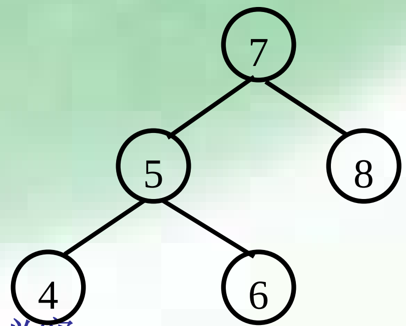
平衡的二叉搜索树(AVL)

- BST受输入顺序影响
  - 最好O(log *n*)
  - 最坏O(*n*)
- AVL树使得BST始终保持O(log *n*)级的平衡状态（12.2.2节）
  - 任何结点的左子树和右子树高度最多相差1

输入顺序为 4、5、6、7、8



输入顺序为 7、5、4、6、8



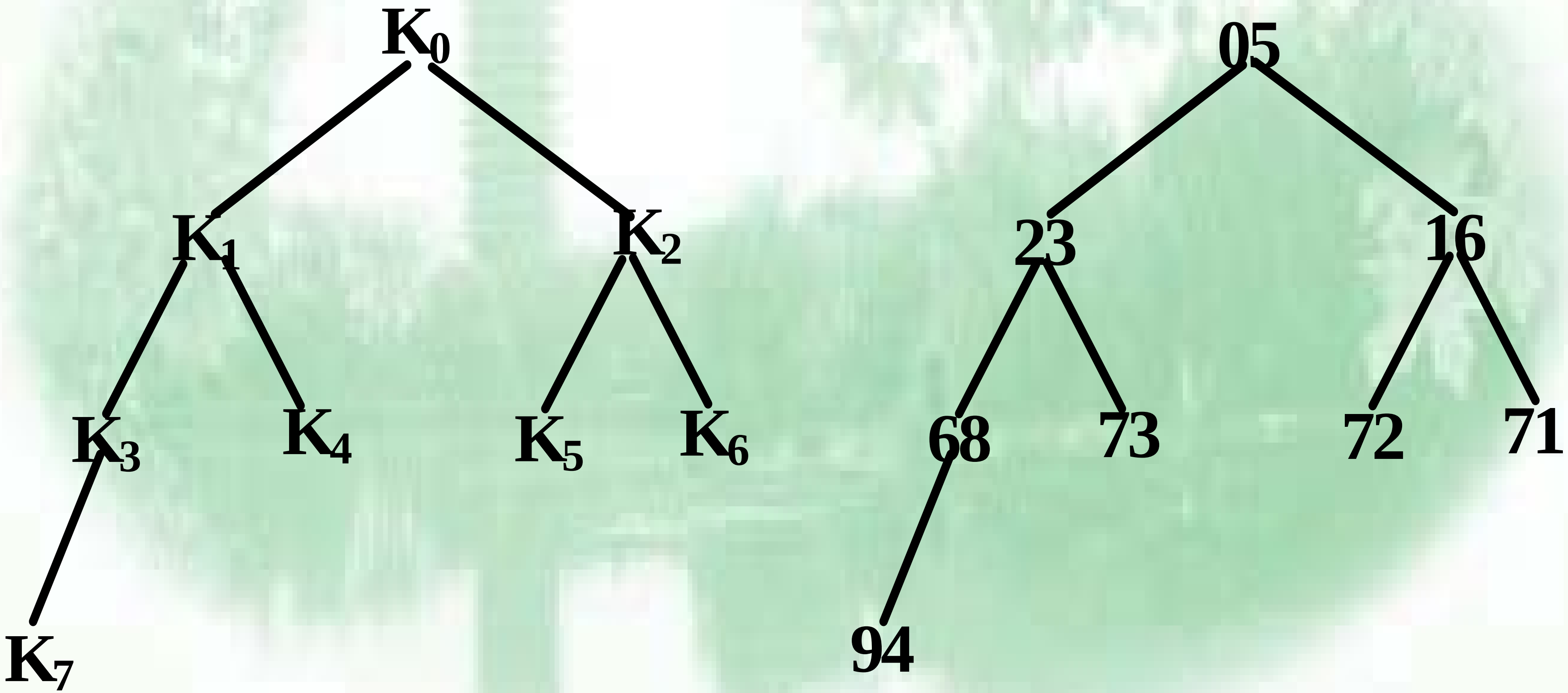
©版权所有，转载或翻印必究

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：
  - $K_i \leqslant K_{2i+1}$　(i=0，1，…，n/2-1)
  - $K_i \leqslant K_{2i+2}$
- 类似可以定义最大值堆

©版权所有，转载或翻印必究

堆的示例



©版权所有，转载或翻印必究

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
  - 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

©版权所有，转载或翻印必究

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);
```


©版权所有，转载或翻印必究

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;
```


©版权所有，转载或翻印必究

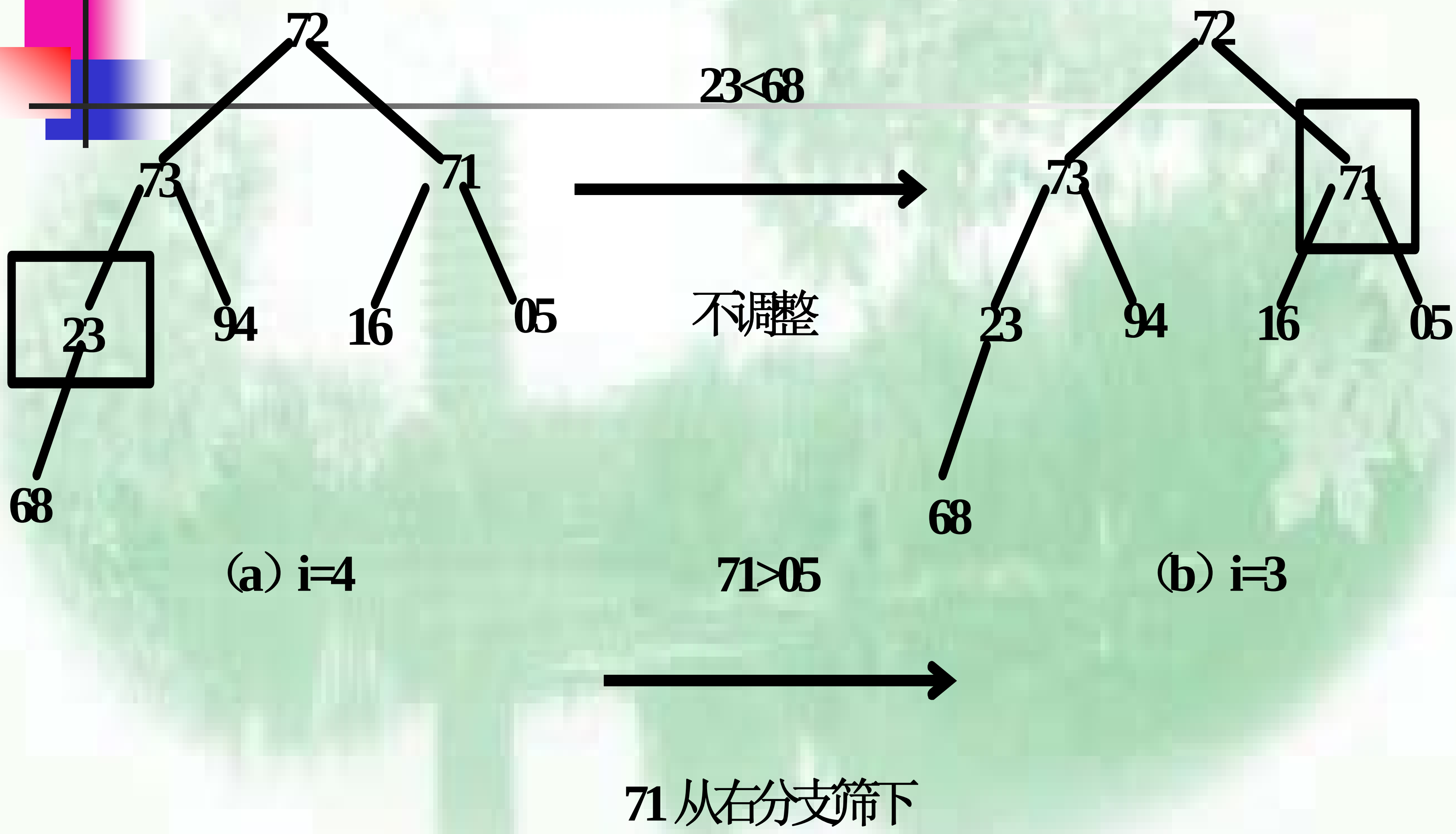
堆的类定义

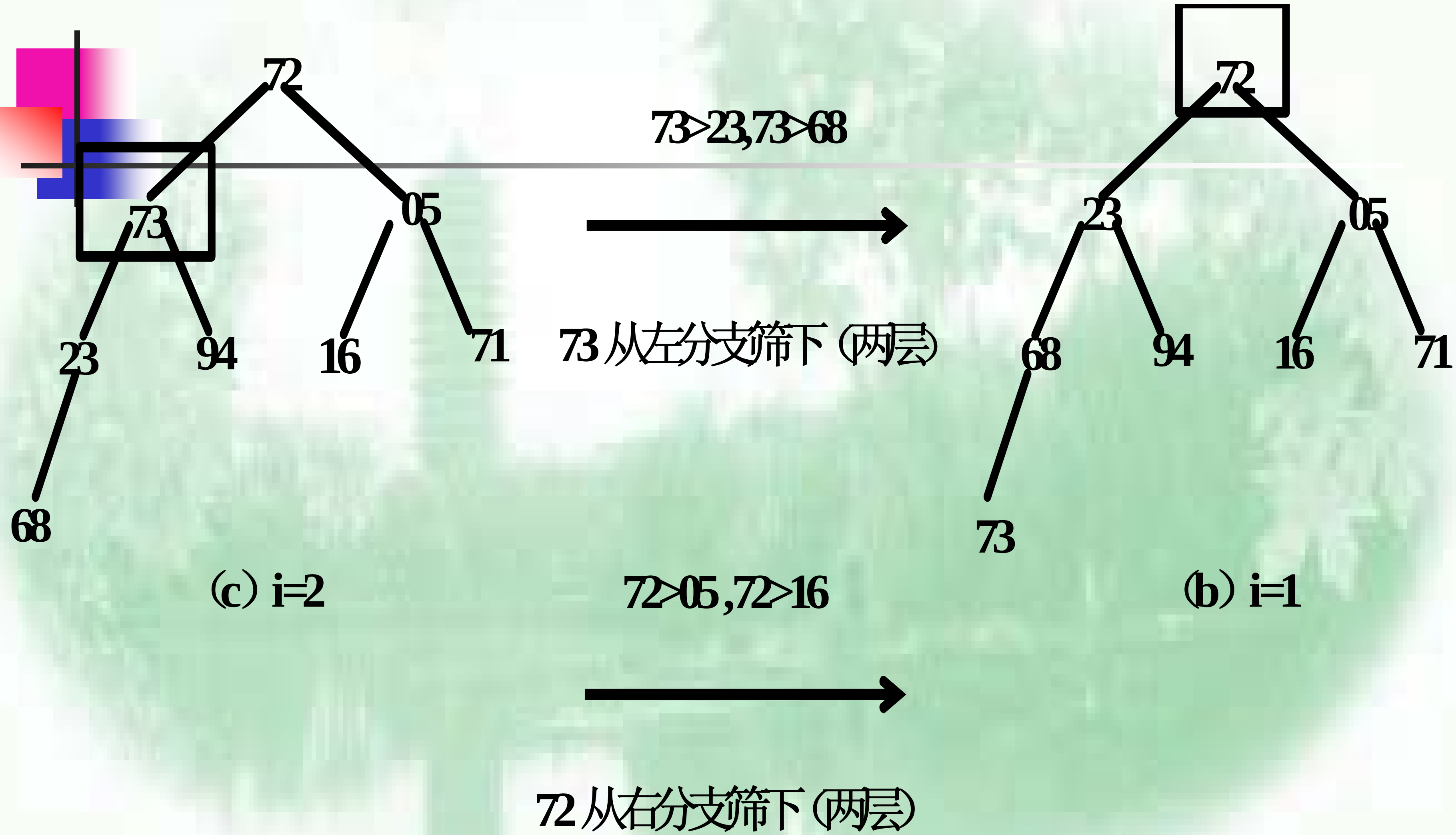
```
// 删除给定下标的元素
bool Remove(int pos, T& node);
//向堆中插入新元素newNode
bool Insert(const T& newNode);
//从堆顶删除最小值
T& RemoveMin();
//从position向上开始调整，使序列成为堆
void SiftUp(int position);
//筛选法函数，参数left表示开始处理的数组下标
void SiftDown(int left);
}
```

建最小值堆过程

- 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

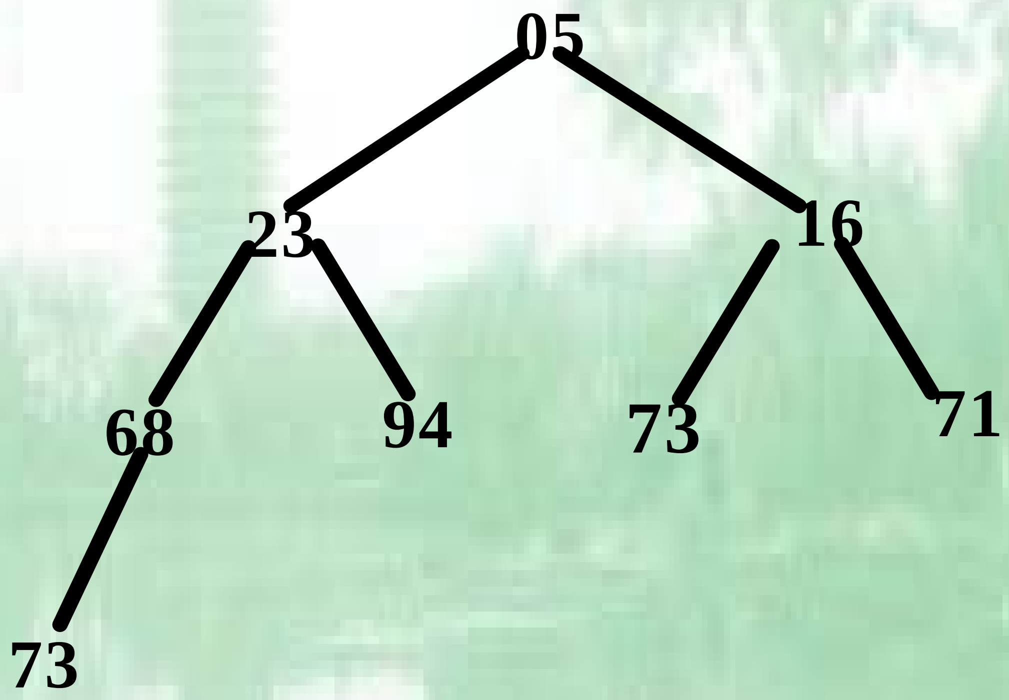
©版权所有，转载或翻印必究


72
73
71
23
94
16
05
68
23<68
不调整
(a) i=4
71>05
(b) i=3
71从右分支筛下


72
73
05
23
94
16
71
68
73>23,73>68
73从左分支筛下（两层）
72
23
05
68
94
16
71
73
72>05,72>16
72从右分支筛下(两层)

(c) i=2

(b) i=1



（e）建成的堆

©版权所有，转载或翻印必究

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
}
```


©版权所有，转载或翻印必究

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
}
```

 ©版权所有，转载或翻印必究

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]))
```


©版权所有，转载或翻印必究

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;
}
```


©版权所有，转载或翻印必究

建堆

- 从第一个分支结点heapArray[CurrentSize/2-1]开始，自底向上逐步把以子树调整成堆

```
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);
}
```


©版权所有，转载或翻印必究

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++;
}
```


©版权所有，转载或翻印必究

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;
}
```


©版权所有，转载或翻印必究

移出最小值(优先队列出队)

- 保持完全二叉树形状，剩下的n-1个结点值仍符合堆的性质
- 将堆中最后一个位置上的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);
        }
```


©版权所有，转载或翻印必究

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
}
```


©版权所有，转载或翻印必究

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$d = \lfloor \log_2 n + 1 \rfloor$。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

- 建堆算法时间代价为O(n)
- 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)

©版权所有，转载或翻印必究

优先队列

- 堆可以用于实现优先队列
- 优先队列
  - 根据需要释放具有最小（大）值的对象
  - 最大树、 左高树**HBLT**、**WBLT**、**MaxWBLT**
- 改变已存储于优先队列中对象的优先权
  - 辅助数据结构帮助找到对象

©版权所有，转载或翻印必究

4.8 Huffman树和编码问题

- 等长编码
- 数据压缩和不等长编码
- 前缀编码
- 扩充二叉树与**Huffman**编码

©版权所有，转载或翻印必究

等长编码

- 计算机二进制编码
  - ASCII码
  - 中文编码
- 等长编码
  - 假设所有代码都等长，则表示n个不同的代码需要$\log_2 n$位
  - 字符的使用频率相等
- 空间效率

©版权所有，转载或翻印必究

数据压缩和不等长编码

- 频率不等的字符

Z	K	F	C	U	D	L	E
2	7	24	32	37	42	42	120

- 可以利用字符的出现频率来编码
  - 经常出现的字符的编码较短，反之不常出现的字符编码较长
- 数据压缩既能节省磁盘空间，又能提高运算速度。（外存时空权衡的规则）

©版权所有，转载或翻印必究

数据压缩和不等长编码（续）

- 不等长编码是文件压缩技术的核心
- Huffman编码是最简单的文件压缩技术，它给出了这种编码方法的思想

©版权所有，转载或翻印必究

前缀编码

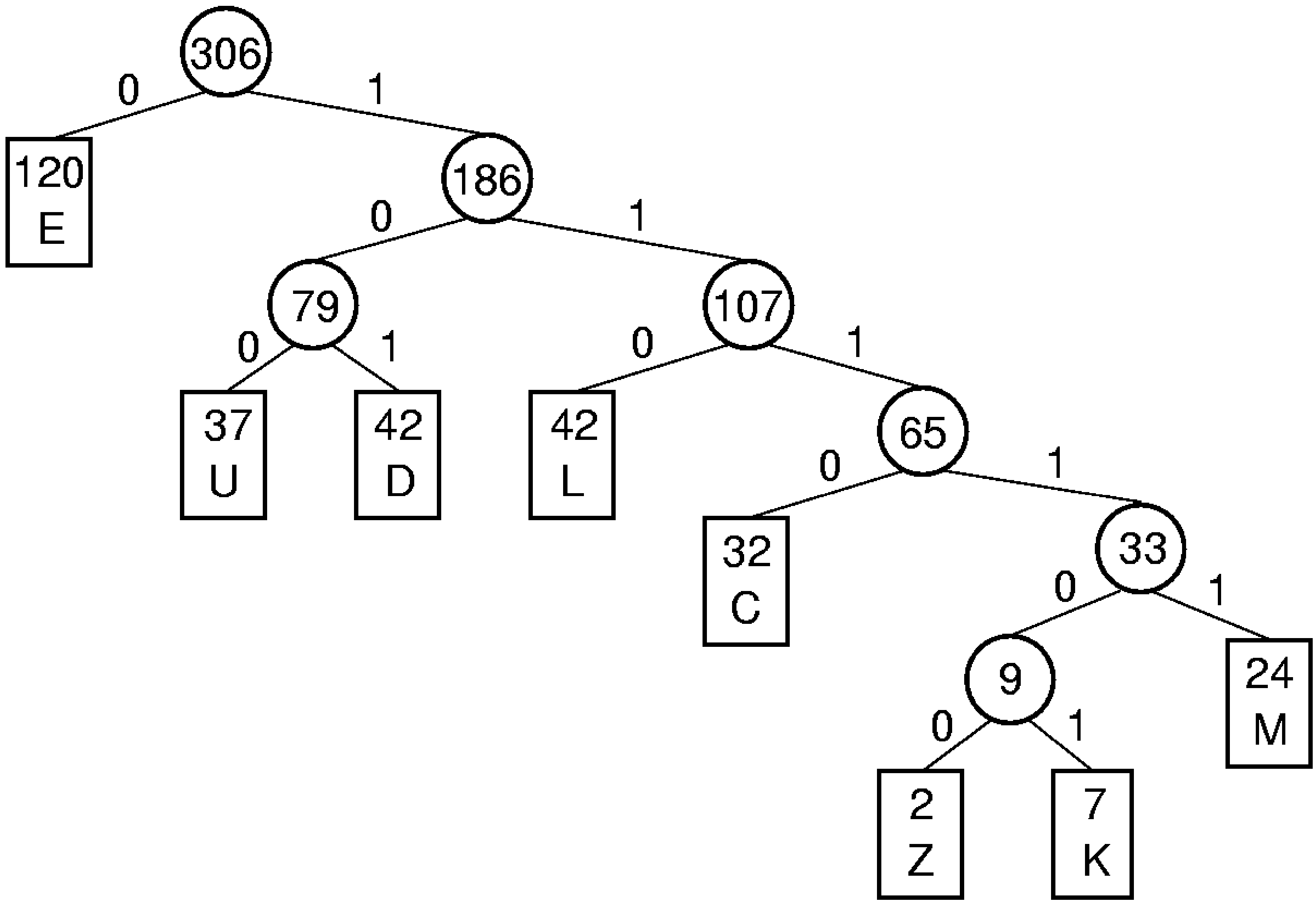
- 一个编码集合中，任何一个字符的编码都不是另外一个字符编码的前缀，这种编码叫作**前缀编码**
- 这种前缀特性保证了代码串被反编码时，不会有多种可能。例如，
  - 对于上面8个字符，编码为Z(111100)，K(111101)，F(11111)，C(1110)，U(100)，D(101)，L(110)，E(0)。
  - 这是一种前缀编码，对于代码“000110”，可以翻译出唯一的字符串“EEEL”

©版权所有，转载或翻印必究

前缀编码（续）

- 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”。

©版权所有，转载或翻印必究


306
0
1
120
E
186
0
1
79
107
0
1
0
1
37
U
42
D
42
L
65
0
1
32
C
33
0
1
9
24
M
0
1
2
Z
7
K

二叉树与前缀编码

- 利用二叉树来设计前缀编码
  - 叶结点表示字符
  - 从根结点到叶的对路径编码
    - 左分支表示‘0’
    - 右分支表示‘1’
- 从根到叶的路径分支所组成的字符串作为该叶结点字符的编码

©版权所有，转载或翻印必究

问题

- 是否前缀编码？
- 怎样保证这样的编码树所得到的编码总长度最小？

©版权所有，转载或翻印必究

Huffman树与前缀编码

- Huffman编码将代码与字符相联系
  - 不等长编码
  - 代码长度取决于对应字符的相对使用频率或“权”

©版权所有，转载或翻印必究

建立Huffman编码树

- 对于n个字符K_0，K_1，…，K_{n-1}，它们的使用频率分别为w_0, w_1，…，w_{n-1}，给出它们的前缀编码，使得总编码效率最高。
- 定义一个树叶的带权路径长度为权乘以它的路径长度(即树叶的深度)。

©版权所有，转载或翻印必究

扩充二叉树与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

怎么构造构造Huffman树?

- 如果从根开始安排
  - 假设权最大的某个w_x作为根的左子结点，而经过组合的子树可能比原来权最大的w_i还大。
  - 另外，w_i是外部结点的权，并不能一步定位。
- 适宜于从叶结点向根的方向来扩展二叉树

©版权所有，转载或翻印必究

建立Huffman编码树

- 首先，按照“权”(例如频率)将字符排为一列
  - 接着，拿走前两个字符(“权”最小的两个字符)
  - 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，使“权”的顺序保持
- 重复上述步骤直至序列处理完毕，则Huffman树建立完毕

©版权所有，转载或翻印必究

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)

©版权所有，转载或翻印必究

Huffman编码树的应用

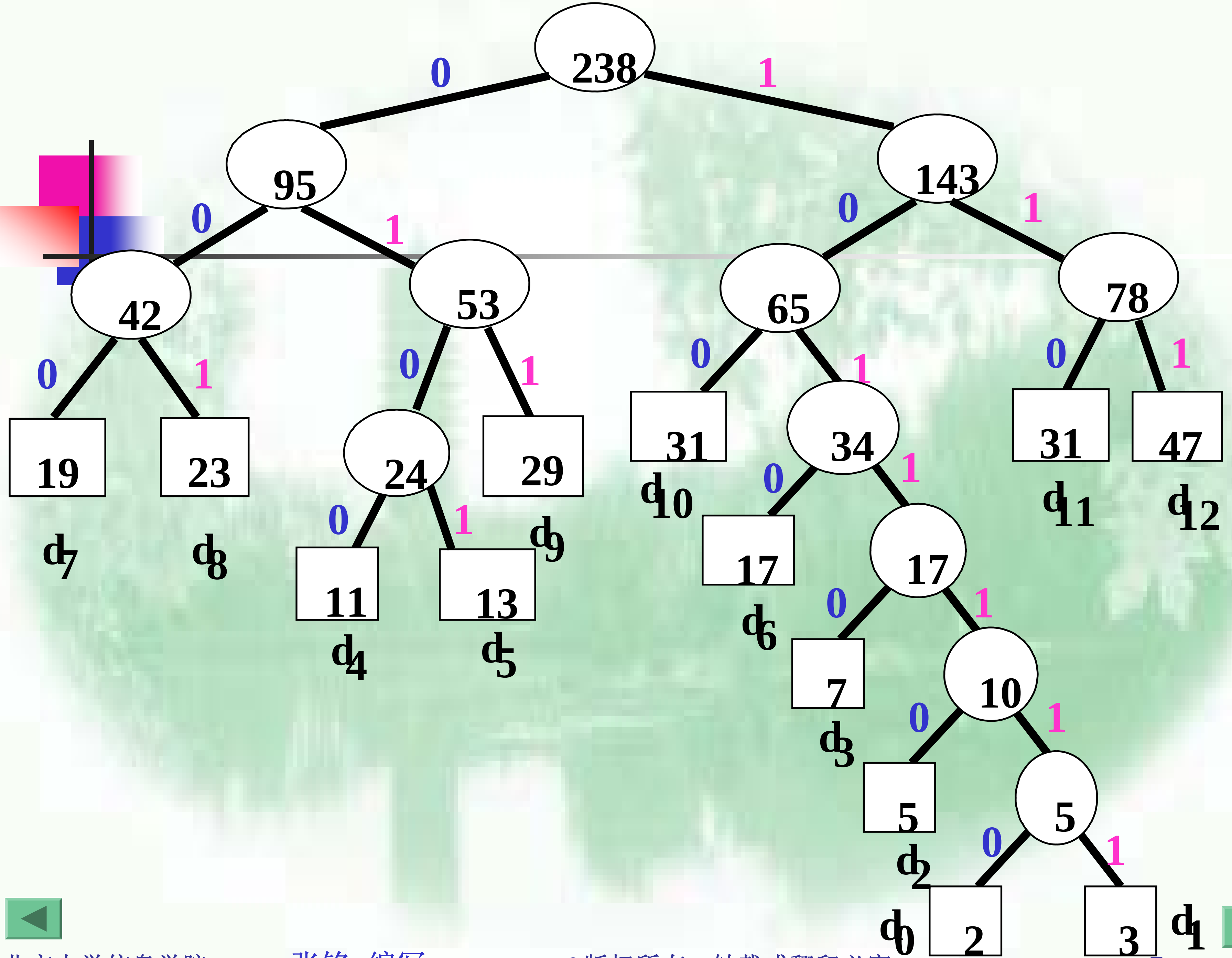
- 利用**Huffman**算法可以这样编码：用d_0，d_1，…，d_{n-1}作外部结点，W_0，W_1，…，W_{n-1}作外部结点的权，构造具有最小带权外部路径长度的扩充二叉树。
- 把从每个结点引向其左子女的边标上号码**0**，把从每个结点引向其右子女的边标上号码**1**。从根到每个叶子的路径上的号码连接起来就是这个叶子代表的字符的编码

©版权所有，转载或翻印必究

2	3	5	7	11	13	17	19	23	29	31	37	41
		5	7	11	13	17	19	23	29	31	37	41
			7	11	13	17	19	23	29	31	37	41
				11	13	17	19	23	29	31	37	41
						17	19	23	29	31	37	41
							19	23	29	31	37	41
									29	31	37	41
										31	37	41
											37	41

5											
	10										
		17									
		17	24								
			24	34							
			24	34	42						
				34	42	53					
					42	53	65				
					42	53	65	78			
							65	78	95		
									95	143	
											238

©版权所有，转载或翻印必究


238
0
1
95
143
0
1
0
1
42
53
65
78
0
1
0
1
0
1
0
1
19
23
24
29
31
34
31
47
d_7
d_8
d_9
d_{10}
d_{11}
d_{12}
0
1
0
1
11
13
17
17
d_4
d_5
d_6
0
1
7
10
d_3
0
1
5
5
d_2
0
1
2
3
d_0
d_1

编码：标记Huffman树中字符代码

- 从根结点开始
  - 左分支的边标记“0”
  - 右分支的边标记“1”
- 字符的Huffman编码:从根结点到该字符所在树叶的路径上的二进制代码串
- 把所有字符的二进制编码放入一个表中

对字符串进行编码时，通过查表来完成

©版权所有，转载或翻印必究

频率越大其编码越短

- 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:	010l
d_6:	1010	d_7:	000
d_8:	001	d_9:	011
d_{10}:	100	d_{11}:	110
d_{12}:	111		

©版权所有，转载或翻印必究

译码：从左至右逐位判别代码串，直至确定一个字符

- 与编码过程相逆
  - 从树的根结点开始
    - “0”下降到左分支
    - “1”下降到右分支
    - 到达一个树叶结点，对应的字符就是文本信息的字符
- 连续译码
  - 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

©版权所有，转载或翻印必究

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);
```


©版权所有，转载或翻印必究

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)
{
  MinHeap<HuffmanTreeNode<T>> heap;//定义最小值堆
  HuffmanTreeNode<T> *parent,&leftchild,&rightchild;
  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
```


©版权所有，转载或翻印必究

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 ();//选值最小的结点
        secondchild=heap. RemoveMin();//选值次小的结点
        //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;
}
```


©版权所有，转载或翻印必究

Huffman方法的正确性证明

- 贪心法的一个例子
  - Huffman树建立的每一步，“权”最小的两个子树被结合为一新子树
- 是否前缀编码？
- 是否最优解？

不存在对应于Huffman树中某分支结点的编码。

©版权所有，转载或翻印必究

Huffman性质

- 引理 含有两个以上结点的一棵**Huffman**树中，字符使用频率最小的两个字符是兄弟结点，而且其深度不比树中其他任何叶结点小。

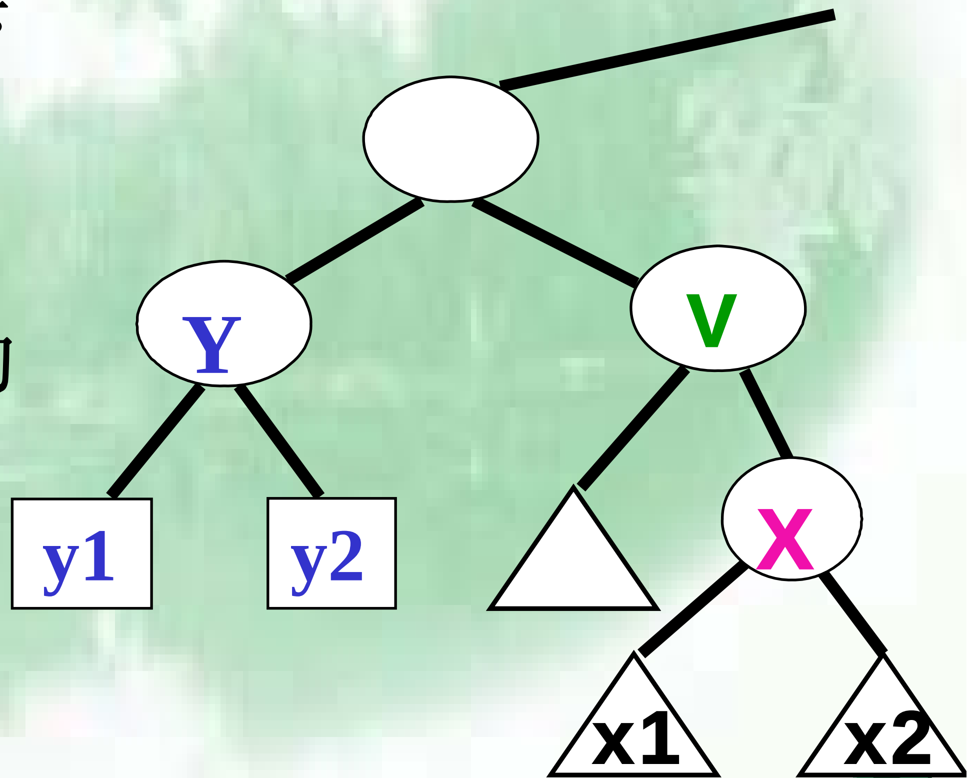
 ©版权所有，转载或翻印必究

证明

- 记使用频率最低的两个字符为**y1**和**y2**
- 由于**HuffmanTree()**函数在构造过程的第一步就选择了它们，所以它们一定是兄弟

©版权所有，转载或翻印必究

- 假设x1,x2是最深的结点
  - y1和y2的父结点Y一定会有比结点X更大的“权”
  - 否则，函数**HuffmanTree**会选择结点Y而不是X作为结点V的子结点
- 然而，由于y1和y2是频率最小的字符，这种情况不可能发生



- 定理 对于给定的一组字符，函数**HuffmanTree**实现了“最小外部路径权重”。
- 证明：对字符个数**n**作归纳进行证明
- 初始情况：令**n = 2, Huffman**树一定有最小外部路径权重
  - 只可能有成镜面对称的两种树
  - 两种树的叶结点加权路径长度相等
- 归纳假设：假设有**n-1**个叶结点的由函数**HuffmanTree**产生的**Huffman**树有最小外部路径权重

归纳步骤：

- 设一棵由函数**HuffmanTree**产生的树**T**有**n**个叶结点，**n>2**，并假设**w1≤w2≤…≤wn**，这里**w1**到**wn**代表字符的"权"
  - 记**V**是频率为**w1**和**w2**的两个字符的父结点。根据引理，它们已经是树**T**中最深的结点
  - **T**中结点**V**换为一个叶结点**V′**（权等于**w1 + w2**），得到另一棵树**T′**
- 根据归纳假设，**T′**具有最小的外部路径长度
- 把两个子结点**w1**和**w2**归还给**V′**，还原为**T**，则**T**也应该有最小的外部路径长度
- 因此，根据归纳原理，定理成立

©版权所有，转载或翻印必究

Huffman树编码效率

- 估计Huffman编码所节省的空间
  - 平均每个字符的代码长度等于每个代码的长度(c_i)乘以其出现的概率(p_i)，即：$c_1p_1 + c_2p_2 + \ldots + c_np_n$
  - 或$(c_1f_1 + c_2f_2 + \ldots + c_nf_n) / f_T$

    这里f_i为第i个字符的出现频率，而f_T为所有字符出现的总次数。

©版权所有，转载或翻印必究

Huffman树编码效率（续）

- 第127页图4.28
- 平均代码长度为：
  (3*(19+23+24+29+31+34+37+41)+4*(11+13+17)+5*7+6*5+7*(2+3))/238=

  $= 804/238 \approx 3.38$
- 对于这13个字符，等长编码每个字符需要 $\lceil \log 13 \rceil = 4$位而Huffman编码只需3.38位

©版权所有，转载或翻印必究

- 对于只由这13个字符所组成的238个字符的文章，Huffman编码所得到的编码长度为804位(bit)
- 而等长编码(每个字符4位)，则在同样的频率下，总共需要4*238=952位
- 因此对于这组字符，Huffman编码预计只需要等长编码804/952≈3.38/4≈84% 的空间
  - Huffman编码明显地缩短了编码长度

©版权所有，转载或翻印必究

Huffman树的应用

- 实际应用中，字符的使用频率的确随具体情况变化
- 如果预计字符频率与实际情况相符，那么所得代码长度将明显小于用等长编码获得的代码
- 如果字符频率的变化范围很大，则Huffman编码是很有效的，能得到很大的压缩比

©版权所有，转载或翻印必究

Huffman树的应用（续）

- 数据通信的二进制编码
  - 对于不同的频率分布组合，因而也会有不同的压缩比率。
  - 大多数的商业压缩程序都是采用2到3种编码方式以应付各种类型的文件。
- 归并法外排序，合并顺串
  - 短的先归并成较长的文件，最大的文件最后归并

©版权所有，转载或翻印必究

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
  - 穿线树
- 二叉树的遍历策略
- 二叉搜索树及其应用
  - 第**12**章**AVL**树——平衡的**BST**
- 堆的概念、性质与构造
  - 优先队列
- **Huffman**树的主要思想与具体应用
  - 编码

©版权所有，转载或翻印必究

The End

mzhang@db.pku.edu.cn

©版权所有，转载或翻印必究