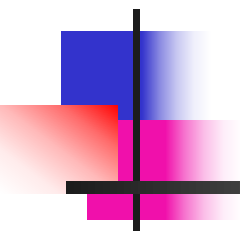


第五章 树





主要内容

- **5.1 树的概念**
- **5.2 树的链式存储**
- **5.3 树的顺序存储**
- **5.4 K叉树**



5.1 树的概念

- **5.1.1 树和森林**
- **5.1.2 森林与二叉树的等价转换**
- **5.1.3 树的抽象数据类型**
- **5.1.4 树的周游**



树的逻辑结构

■ 包含 n 个结点的有穷集合 K ($n>0$), 且在 K 上定义了一个关系 N , 关系 N 满足以下条件:

- 有且仅有一个结点 $k_0 \in K$, 它对于关系 N 来说没有前驱。结点 k_0 称作树的根
- 除结点 k_0 外, K 中的每个结点对于关系 N 来说都有且仅有一个前驱
- 除结点 k_0 外的任何结点 $k \in K$, 都存在一个结点序列 k_0, k_1, \dots, k_s , 使得 k_0 就是树根, 且 $k_s = k$, 其中有序对 $\langle k_{i-1}, k_i \rangle \in N (1 \leq i \leq s)$ 。这样的结点序列称为从根到结点 k 的一条路径



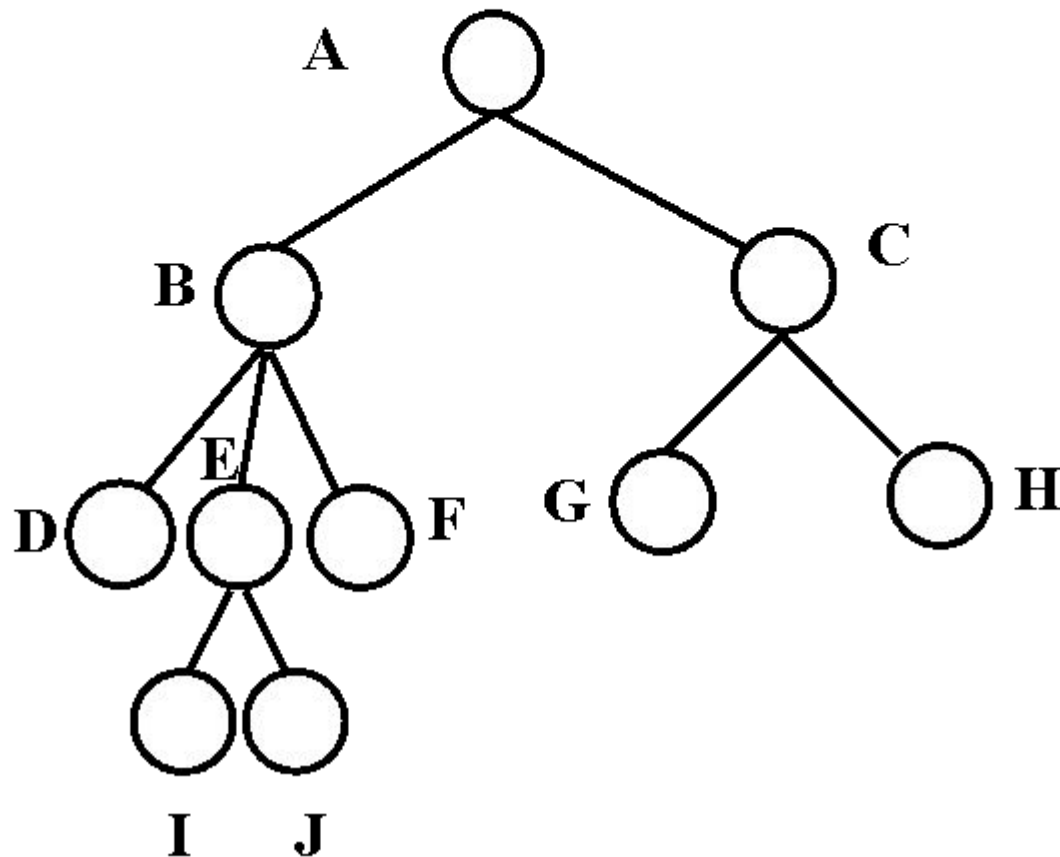
树形结构的各种表示法

树的逻辑结构是：

结点集合 $K = \{A, B, C, D, E, F, G, H, I, J\}$

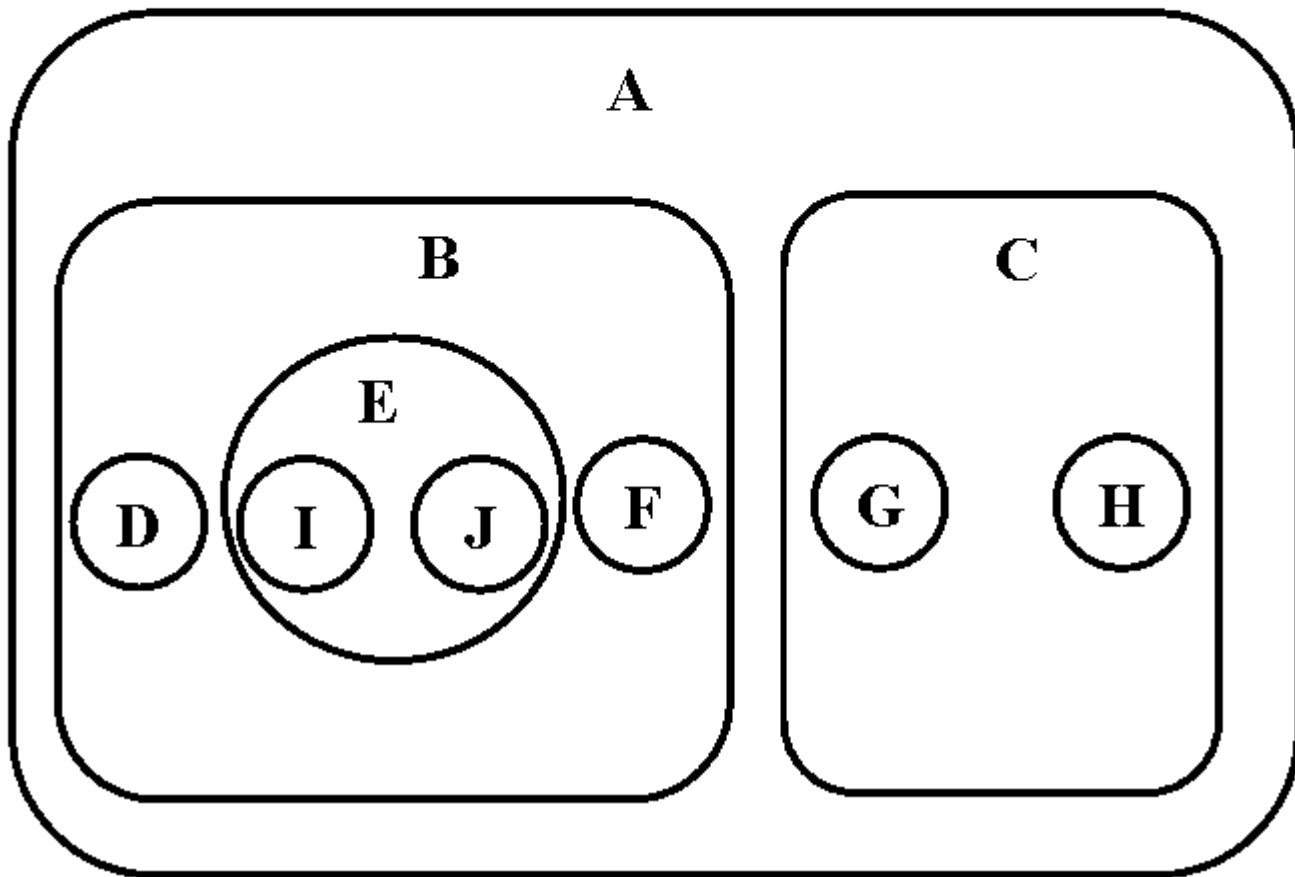
K 上的关系 $N = \{ \langle A, B \rangle, \langle A, C \rangle, \langle B, D \rangle, \langle B, E \rangle, \langle B, F \rangle, \langle C, G \rangle, \langle C, H \rangle, \langle E, I \rangle, \langle E, J \rangle \}$

树形结构的各种表示法



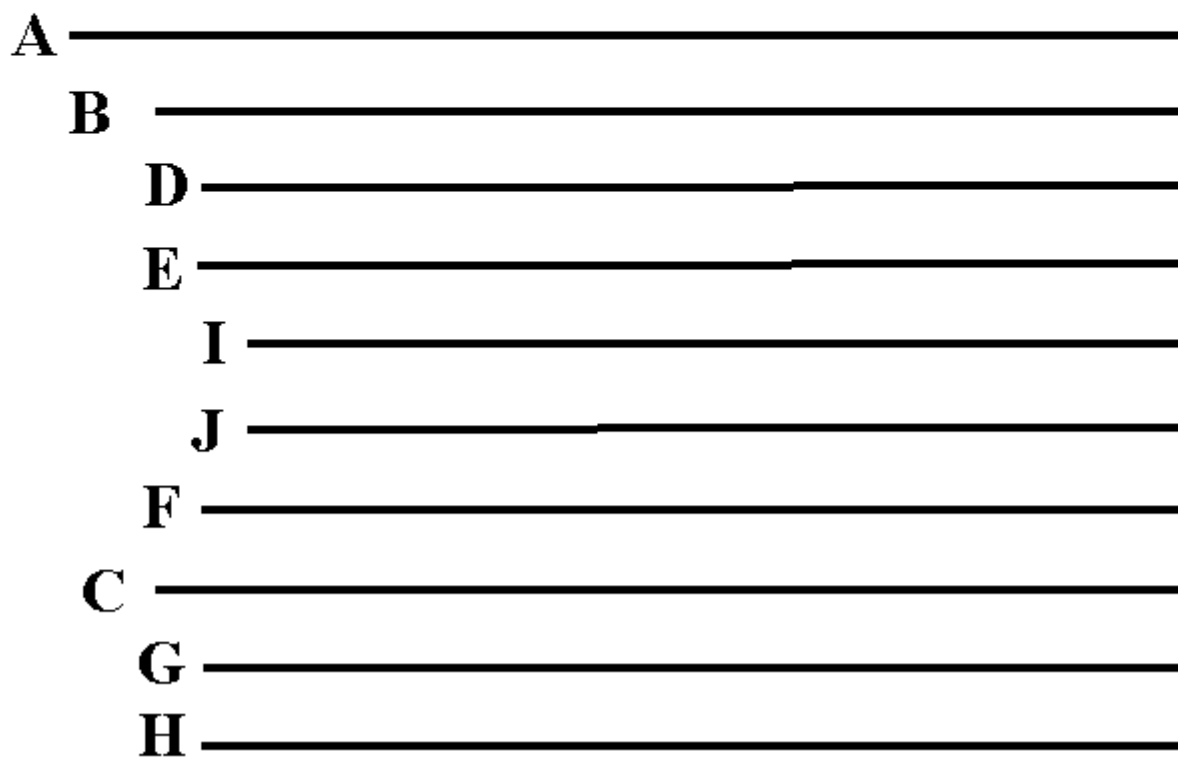
(a) 树形表示法

树形结构的各种表示法



(b) 文氏图表示法

树形结构的各种表示法



(c) 凹入表表示法



树形结构的各种表示法

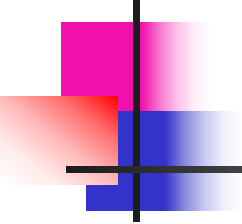
(A(B(D)(E(I)(J))(F))(C(G)(H)))

(d) 嵌套括号表示法



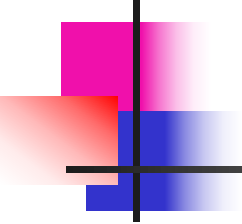
树的定义

- 树是包括 n 个结点的有限集合 T ($n \geq 1$)，使得：
 - 有一个特别标出的称作根的结点
 - 除根以外的其它结点被分成 m 个($m \geq 0$)不相交的集合 T_1, T_2, \dots, T_m ，而且这些集合的每一个又都是树。树 T_1, T_2, \dots, T_m 称作这个根的子树
- 这个定义是递归的，我们用子树来定义树：只包含一个结点的树必然仅由根组成，包含 $n > 1$ 个结点的树借助于少于 n 个结点的树来定义



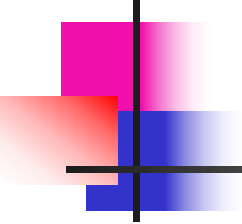
树结构中的概念

- 若 $\langle k, k' \rangle \in N$ ，则称 k 是 k' 的父结点（或称“父母”），而 k' 则是 k 的子结点（或“儿子”、“子女”）
- 若有序对 $\langle k, k' \rangle$ 及 $\langle k, k'' \rangle \in N$ ，则称 k' 和 k'' 互为兄弟
- 若有一条由 k 到达 k_s 的路径，则称 k 是 k_s 的祖先， k_s 是 k 的子孙
- 树形结构中，两个结点的有序对，称作连接这两结点的一条边



树结构中的概念

- 没有子树的结点称作**树叶**或**终端结点**
- 非终端结点称为**分支结点**
- 一个结点的子树的个数称为**度数**
- 根结点的**层数**为**0**，其它任何结点的层数等于它的父结点的层数加**1**



树结构中的概念

- **有序树** 在树 T 中如果子树 T_1, T_2, \dots, T_m 的相对次序是重要的, 则称树 T 为有向有序树, 简称**有序树**。
- 在有序树中可以称 T_1 是根的第一棵子树, T_2 是根的第二棵子树, 等等



森林与树

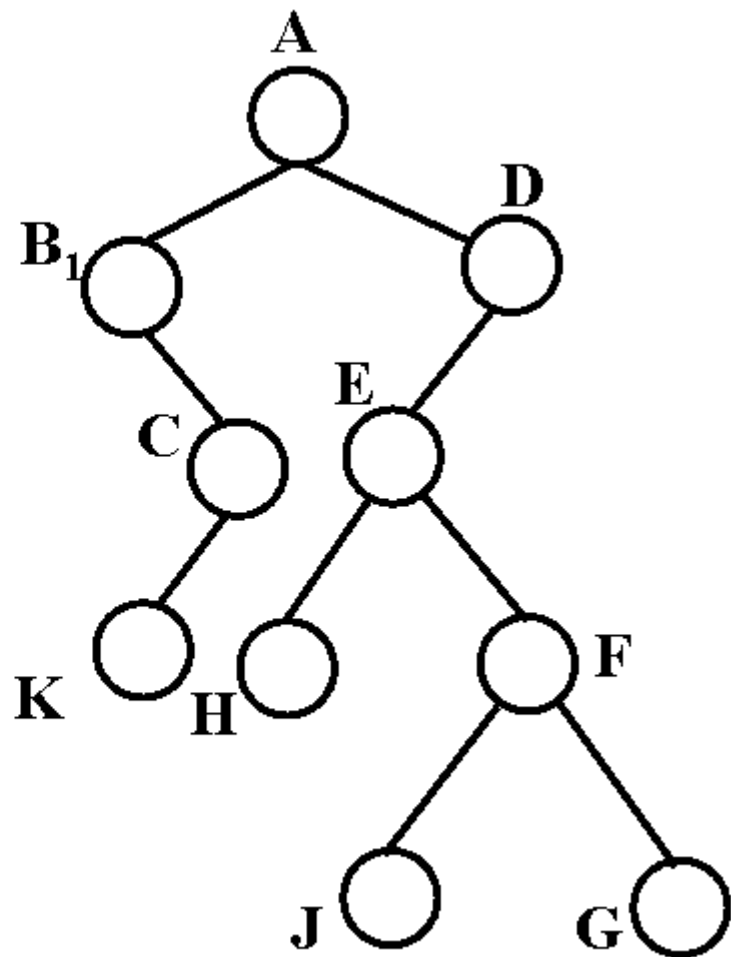
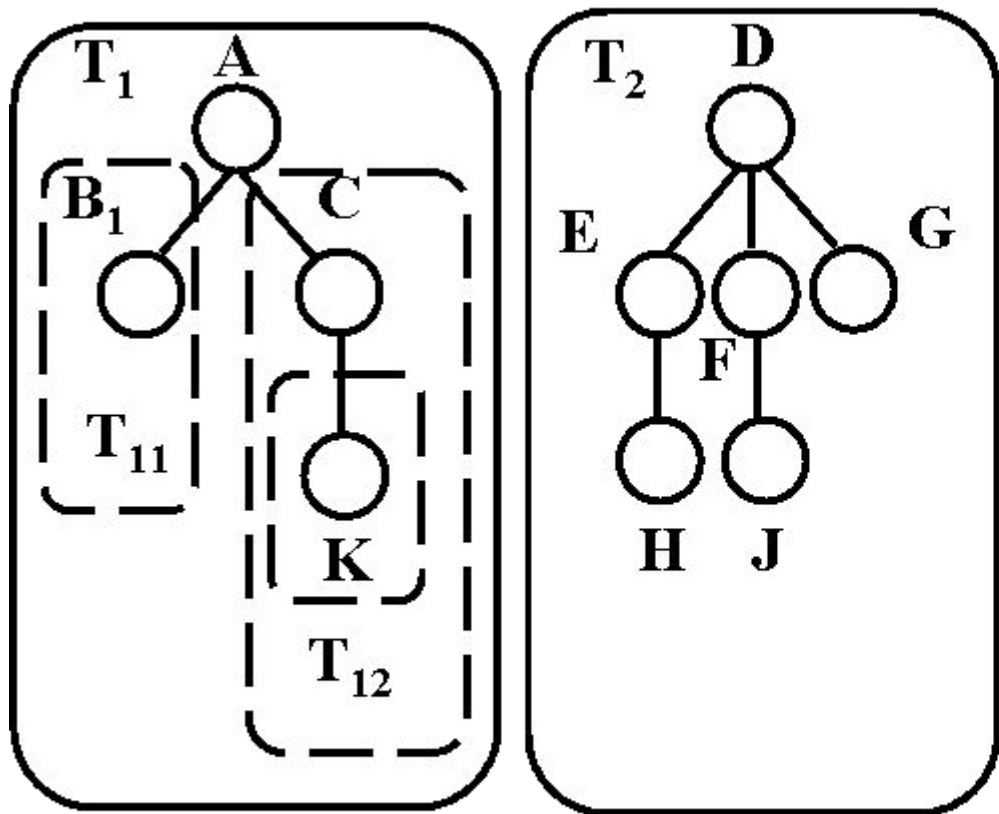
- 森林(**forest**) 森林是零棵或多棵不相交的树的集合(通常是有序集合)。
- 自然界的树和森林是不同的概念，而数据结构的树和森林只有微小的差别。删去树根，树就变成森林。加上一个结点作树根，森林就变成树



森林与二叉树的等价转换

- 在树或森林与二叉树之间有一个自然的一一对应的关系。
 - 任何森林都唯一地对应到一棵二叉树；反过来，任何二叉树也都唯一地对应到一个森林。
- 树所对应的二叉树里
 - 一个结点的左子结点是它在原来树里的第一个子结点
 - 右子结点是它在原来的树里的下一个兄弟

森林与二叉树的等价转换





森林到二叉树的等价转换

- 把森林 F 看作树的有序集合, $F=(T_1, T_2, \dots, T_n)$, 对应于 F 的二叉树 $B(F)$ 的定义是:
 - 若 $n=0$, 则 $B(F)$ 为空
 - 若 $n>0$, 则 $B(F)$ 的根是 T_1 的根 W_1 , $B(F)$ 的左子树是 $B(T_{11}, T_{12}, \dots, T_{1m})$, 其中 $T_{11}, T_{12}, \dots, T_{1m}$ 是 W_1 的子树; $B(F)$ 的右子树是 $B(T_2, \dots, T_n)$
- 此定义精确地确定了从森林到二叉树的转换



森林到二叉树的等价转换

- 设 \mathbf{B} 是一棵二叉树， \mathbf{rt} 是 \mathbf{B} 的根， \mathbf{L} 是 \mathbf{rt} 的左子树， \mathbf{R} 是 \mathbf{rt} 的右子树，则对应于 \mathbf{B} 的森林 $\mathbf{F}(\mathbf{B})$ 的定义是：
 - 若 \mathbf{B} 为空，则 $\mathbf{F}(\mathbf{B})$ 是空的森林。
 - 若 \mathbf{B} 不为空，则 $\mathbf{F}(\mathbf{B})$ 是一棵树 \mathbf{T}_1 加上森林 $\mathbf{F}(\mathbf{R})$ ，其中树 \mathbf{T}_1 的根为 \mathbf{rt} ， \mathbf{rt} 的子树为 $\mathbf{F}(\mathbf{L})$



树/森林的抽象数据类型

```
template<class T>
```

```
class TreeNode
```

```
{
```

```
  public:
```

```
    TreeNode(const T&); //拷贝构造函数
```

```
    virtual ~TreeNode(){}; //析构函数
```

```
    bool isLeaf(); //如果结点是叶，返回true
```

```
    T Value(); //返回结点的值
```

```
    TreeNode<T>* LeftMostChild(); //返回第一个左孩子
```

```
    TreeNode<T>* RightSibling(); //返回右兄弟
```



树/森林的抽象数据类型

```
void setValue(T&);// 设置结点的值
// 设置左子结点
void setChild(TreeNode<T>* pointer);
// 设置右兄弟
void setSibling(TreeNode<T>* pointer);
// 以第一个左子结点身份插入结点
void InsertFirst(TreeNode<T>* node);
// 以右兄弟的身份插入结点
void InsertNext(TreeNode<T>* node);
};
```



树/森林的抽象数据类型

```
template <class T>class Tree
{
public:
Tree();//构造函数
virtual ~Tree();//析构函数
//返回树中的根结点
TreeNode<T>* getRoot();
//创建树中的根结点，使根结点元素的值为rootValue
void CreateRoot(const T& rootValue);
//判断是否为空树，如果是则返回true
bool isEmpty();
```



树/森林的抽象数据类型

//返回**current**结点的父结点

TreeNode<T>* Parent(TreeNode<T>* current);

//返回**current**结点的前一个邻居结点

TreeNode<T>* PrevSibling(TreeNode<T>* current);

//删除以**subroot**为根的子树的所有结点

void DeleteSubTree(TreeNode<T>* subroot);

//先根深度优先周游树

void RootFirstTraverse(TreeNode<T>* root);

//后根深度优先周游树

void RootLastTraverse(TreeNode<T>* root);

//宽度优先周游树

void WidthTraverse(TreeNode<T>* root);

};



森林的周游

- 按深度的方向周游

- 先根次序:

- a)**访问头一棵树的根

- b)**在先根次序下周游头一棵树树根的子树

- c)**在先根次序下周游其他的树

- 后根次序:

- a)**在后根次序下周游头一棵树树根的子树

- b)**访问头一棵树的根

- c)**在后根次序下周游其他的树



先根深度优先周游森林

```
template <class T>
```

```
void Tree<T>::RootFirstTraverse(TreeNode<T>* root)
```

```
{
```

```
while(root!=NULL)
```

```
{
```

```
Visit(root->Value()); //访问当前结点
```

```
//周游头一棵树树根的子树
```

```
RootFirstTraverse(root->LeftMostChild());
```

```
root=root->RightSibling();//周游其他的树
```

```
}
```

```
}
```




后根深度优先周游森林

```
template <class T>
```

```
void Tree<T>::RootLastTraverse ( TreeNode<T>* root)
```

```
{
```

```
while (root !=NULL)
```

```
{
```

```
//周游头一棵树树根的子树
```

```
RootLastTraverse (root->LeftMostChild()); Visit
```

```
(root->Value());//访问当前结点
```

```
root=root->RightSibling();//周游其他的树
```

```
}
```

```
}
```



深度优先周游森林

- 按先根次序周游森林正好等同于按前序法周游对应的二叉树
- 按后根次序周游森林正好等同于按中序法周游对应的二叉树
- 不方便仿照中序法定义树的中根周游，因为当一个根多于两个子结点时无法明确给出根与这些子结点的次序



宽度优先周游森林

```
template <class T>  
void Tree<T>::WidthTraverse2(TreeNode<T>*  
    root){  
    using std::queue; // 使用STL队列  
    queue<TreeNode<T>*> aQueue;  
    TreeNode<T>* pointer=root;  
    if(pointer){  
        aQueue.push(pointer);  
        while(!aQueue.empty()){  
            pointer=aQueue.front(); // 取队列首结点指针
```



宽度优先周游森林

```
Visit(pointer->Value()); //访问当前结点  
while(pointer->RightSibling())  
{  
  if(pointer->LeftMostChild())//左子结点进入队列  
  aQueue.push(pointer->LeftMostChild());  
  pointer=pointer->RightSibling();  
  Visit(pointer->Value()); //访问右兄弟结点  
}
```



宽度优先周游森林

```
if(pointer->LeftMostChild())
```

```
aQueue.push(pointer
```

```
->LeftMostChild());
```

```
aQueue.pop();//出队列
```

```
}//end while
```

```
}//end if
```

```
}
```



5.2 森林的链式存储

- **5.2.1** 子结点表表示法
- **5.2.2** 左子结点/右兄弟结点表示法
- **5.2.3** 动态结点表示法
- **5.2.4** 动态“左子结点/右兄弟结点”二叉链表表示法
- **5.2.5** 父指针表示法



子结点表表示法

- 每个分支结点均存储其子结点信息，子结点按照从左至右的顺序形成一个链表
- “子结点表”表示法在数组中存储树的结点。每个结点包括结点值、一个父指针以及一个指向子结点链表的指针
- 结点的最左子结点可由链表的第一个表项直接找到
- 找到结点的右侧兄弟结点要困难一些



左子结点/右兄弟结点表示法

- 每个结点都存储结点的值，以及指向父结点、最左子结点和右侧兄弟结点的指针
- **ADT**的基本操作可通过读取结点中的一个值来实现。如果两棵树存储在同一个数组中，那么把其中一个添加为另一棵树的子树只需简单设置三个指针值即可。
- 这种表示法比子结点表表示法空间效率更高，而且结点数组中的每个结点仅需要固定大小的存储空间



动态结点表示法

- 为每个结点分配可变的存储空间
 - 将一个指向子结点的指针数组作为结点的一部分分配给结点。实质上，每个结点存储一个基于数组的子结点指针表。在子结点的数目不变时，这种方法效果最佳；如果子结点的数目发生变动（特别是增加），就必须提供一种专门的校正机制来改变子结点指针数组的大小
 - 每个结点存储一条子结点指针链表。本质上“子结点表”表示法相同，但是它动态地分配结点空间，而不是把结点分配在数组中

动态“左子结点/右兄弟结点” 二叉链表表示法

- 本质上，我们使用二叉树来替换树
- 新的结构中左子结点在树中是结点的最左子结点。右子结点是结点原来的右侧兄弟结点
- 可以很容易的把这种转化推广到森林，因为森林中每棵树的根结点可以看成互为兄弟结点
- 由于树的每个结点均包含固定数目的指针，而且树的**ADT**的每个函数均能有效实现，因此动态“左子结点/右兄弟结点”表示法比以上介绍的其他方法更为常用



树结点抽象数据类型的实现

//补充与具体实现相关的私有成员变量申明

private:

T m_Value; //树结点的值

TreeNode<T>*pChild; //左子结点

TreeNode<T>*pSibling; //右兄弟结点

//公有成员函数的具体实现

**template<class T>TreeNode<T>::TreeNode(const T&
value)**

{//拷贝构造函数

m_Value=value;

pChild=NULL;

pSibling=NULL;

}



树结点抽象数据类型的实现

```
template<class T> T TreeNode<T>::Value()  
{//返回结点的值  
    return m_Value;  
}  
template<class T> bool TreeNode<T>::isLeaf()  
{//如果结点是叶，返回true  
    if(pChild==NULL)  
        return true;  
        return false;  
}
```



树结点抽象数据类型的实现

```
template<class T>  
TreeNode<T>*TreeNode<T>::LeftMostChild()  
{ //返回第一个左子结点  
    return pChild;  
}  
  
template<class T>  
TreeNode<T>* TreeNode<T>::RightSibling()  
{ //返回右兄弟  
    return pSibling;  
}  
  
template<class T>void  
TreeNode<T>::setValue(T& value)  
{ //设置结点的值  
    m_Value=value;  
}
```



树结点抽象数据类型的实现

```
template<class T>  
void TreeNode<T>::setChild(TreeNode<T>* pointer)  
{//设置左子结点  
    pChild=pointer;  
}  
  
template<class T>  
void TreeNode<T>::setSibling(TreeNode<T>* pointer)  
{//设置右兄弟  
    pSibling=pointer;  
}
```



树结点抽象数据类型的实现

```
template<class T>
void TreeNode<T>::InsertFirst(TreeNode<T>* node)
{ // 以第一个子结点的身份插入结点
    if(!pChild)
        pChild=node;
    else
    {
        node->pSibling=pChild;
        pChild=n;
    }
}
```



树结点抽象数据类型的实现

```
template<class T>
void TreeNode<T>::InsertNext(TreeNode<T>* node)
{ // 以右兄弟的身份插入结点
    if(!pSibling)
        pSibling = node;
    else
    {
        node -> pSibling = pSibling;
        pSibling = node;
    }
}
```




树抽象数据类型的实现

//与具体实现相关的私有成员变量与成员函数的申明

private:

TreeNode<T>* root; //树根结点

//返回**current**的父节点，由函数**Parent**调用

TreeNode<T>* getParent

(TreeNode<T>* root,TreeNode<T>* current);

//删除以**root**为根的子树的所有结点

void DeleteNodes(TreeNode<T>*root);

//私有成员函数与公有成员函数的具体实现

template <class T>

Tree<T>::Tree()

{//构造函数

root=NULL;

}



树抽象数据类型的实现

```
template <class T> Tree<T>::~~Tree()
```

```
{//析构函数
```

```
    while ( root )
```

```
        DeleteSubTree( root );
```

```
}
```

```
template <class T>TreeNode<T>*
```

```
Tree<T>::getRoot()
```

```
{ //返回树中的根结点
```

```
    return root;
```

```
}
```



树抽象数据类型的实现

```
template <class T>
void Tree<T>::CreateRoot(const T& rootValue)
{ // 创建树中的根结点, 使根结点元素的值为rootValue
    if( !root )
        root=new TreeNode<T>(rootValue);
}
template <class T>
bool Tree<T>::isEmpty()
{ // 判断是否为空树, 如果是则返回true
    if(root)
        return false;
    return true;
}
```



树抽象数据类型的实现

```
template <class T>
```

```
TreeNode<T>* Tree<T>::PrevSibling(TreeNode<T>*  
current)
```

```
{//返回current结点的前一个邻居结点
```

```
using std::queue; //使用STL队列
```

```
queue<TreeNode<T>*> aQueue;
```

```
TreeNode<T>* pointer=root;//标识当前结点
```

```
//标识当前结点的前一个兄弟结点
```

```
TreeNode<T>* prev=NULL;
```

```
//当前结点为空，树为空或所求结点为根结点时，返回
```

```
//NULL
```



树抽象数据类型的实现

```
if((current==NULL) || (pointer==NULL) || (current==pointer))  
    return NULL;  
while(pointer){  
    if(pointer==current)  
        return prev; // 找到当前结点  
    aQueue.push(pointer);  
    prev=pointer;  
    pointer=pointer->pSibling; // 沿当前结点右兄弟结点链寻找  
}
```



树抽象数据类型的实现

```
while(!aQueue.empty()){  
prev=NULL;  
pointer=aQueue.front();  
aQueue.pop();// 出队列  
pointer=pointer->LeftMostChild();// 下降到左子结点  
while(pointer){  
if(pointer==current)  
return prev;
```



树抽象数据类型的实现

```
aQueue.push(pointer);  
prev=pointer;  
pointer=pointer->pSibling; //沿当前  
结点右兄弟结点链寻找  
} //end while  
} //end while  
return NULL;  
}
```



树抽象数据类型的实现

```
TreeNode<T>* Tree<T>::getParent(TreeNode<T>*  
    root,TreeNode<T>* current)  
{//私有成员函数，返回current的父节点，由函数Parent调用  
    TreeNode<T>* temp;  
    if(root==NULL)  
        return NULL;  
    if(root->LeftMostChild()==current) //找到父节点  
        return root;  
    //递归寻找父节点  
    if((temp=getParent(root->LeftMostChild(),current))  
        !=NULL)  
        return temp;  
    else return getParent(root->RightSibling(),current);  
}
```




树抽象数据类型的实现

```
template <class T>
```

```
TreeNode<T>*
```

```
Tree<T>::Parent(TreeNode<T>* current)
```

```
{//返回current结点的父结点
```

```
TreeNode<T>* pointer=current;
```

```
if(pointer !=NULL) return NULL;
```

```
TreeNode<T>* leftmostChild=NULL;
```



树抽象数据类型的实现

```
while((leftmostChild=PrevSibling(pointer))!=NULL)
    pointer=leftmostChild;
leftmostChild=pointer;
pointer=root;
if( leftmostChild ==root)
    return NULL;
else return getParent ( pointer , leftmostChild);
}
```

删除以root为根的子树的所有结点

```
template <class T>
void Tree<T>::DeleteNodes(TreeNode<T>* root)
{ // 删除以root为根的子树的所有结点
  if(root)
  {
    // 递归删除第一子树
    DeleteNodes(root->LeftMostChild());
    // 递归删除其他子树
    DeleteNodes(root->RightSibling()); delete root; // 删除根结点
  }
}
```



树抽象数据类型的实现

```
template <class T>  
void Tree<T>::DeleteSubTree(TreeNode<T>* subroot)  
{ // 删除以subroot为根的子树的所有结点  
TreeNode<T>* pointer=PrevSibling(subroot);  
if(pointer==NULL)  
{ // subroot为最左子结点的情况  
pointer=Parent(subroot);  
if(pointer)  
{  
pointer->pChild=subroot->RightSibling();  
subroot->pSibling=NULL;  
}
```



树抽象数据类型的实现

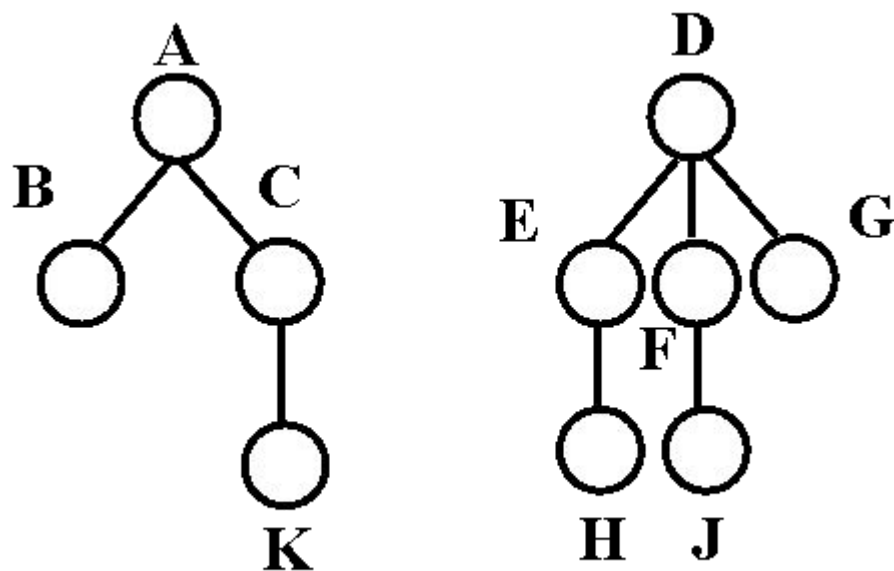
```
else{
    root=subroot->RightSibling();
    subroot->pSibling=NULL;
}
} //end if
else{//subroot不是最左子结点
    pointer->pSibling=subroot->
        RightSibling();
    subroot->pSibling=NULL;
}
DestroyNodes(subroot); //销毁以subroot为根的子树
}
```



父指针表示法

- 实现树的最简单方法是对每个结点只保存一个指针域指向其父结点，这种实现被称为**父指针(parent pointer)**表示法
- 如果两个结点到达同一根结点，它们一定在同一棵树中。如果找到的根结点是不同的，那么两个结点就不在同一棵树中

父指针数组表示法



父节点索引

标记

结点索引

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



等价类(equivalence classes)

- 等价关系具有自反性，对称性，传递性
- **UNION/FIND**算法可以很容易地解决等价类问题
 - 开始时，每个元素都在独立的只包含一个结点的树中，而它自己就是根结点
 - 通过使用函数**Different**可以检查一个等价对中的两个元素是否在同一棵树中。如果是，由于它们已经在同一个等价类中，不需要作变动。否则两个等价类可以用**UNION**函数归并



等价类的并查

(**UNION/FIND**) 算法

- 父指针表示法常常用来维护由一些不相交子集构成的集合。包含两种基本操作：
 - 判断两个结点是否在同一个集合中，查找一个给定结点的根结点的过程称为**FIND**
 - 归并两个集合，这个归并过程常常被称为**UNION**
 - 整个操作以“**UNION/FIND算法**”（也可以翻译为“并查算法”）命名



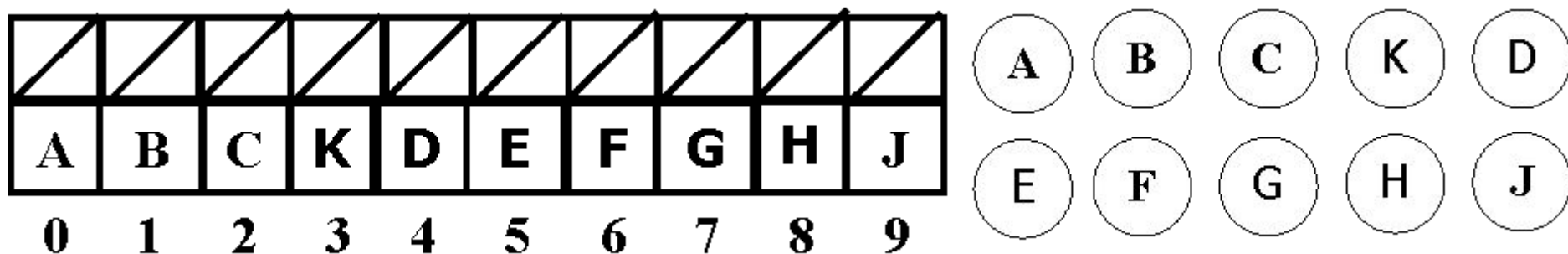
等价类的并查

(UNION/FIND) 算法

- “**UNION/FIND**” 算法用一棵树代表一个集合
- 如果两个结点在同一棵树中，则认为它们在同一个集合中。树中的每个结点（除根结点以外）有仅且有一个父结点
- 结点中仅需保存父指针信息，树本身可以存储为一个以其结点为元素的数组

UNION/FIND算法示例

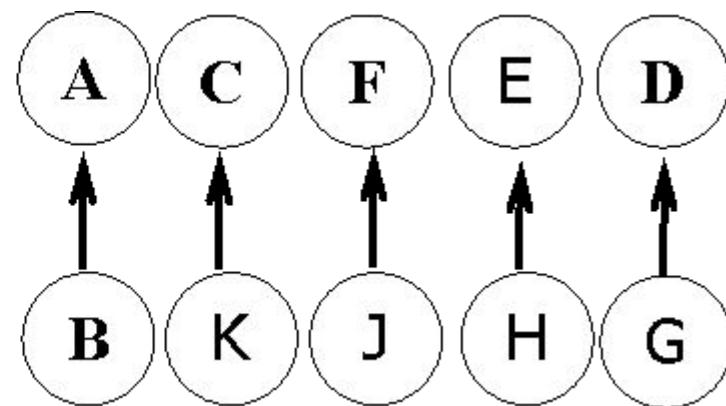
10个结点A、B、C、D、E、F、G、H、J、K和它们的等价关系 (A,B)、(C,K)、(J,F)、(H,E)、(D,G)、(K,A)、(E,G)、(H,J)



UNION/FIND算法示例

对这5个等价对的处理结果 (A,B)、(C,K)、(J,F)、(H,E)、(D,G)

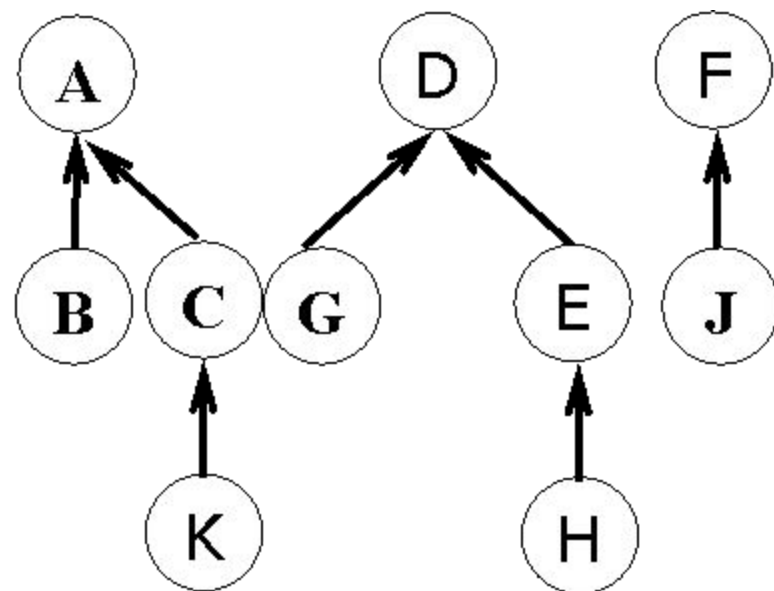
	0		2				4	5	6
A	B	C	K	D	E	F	G	H	J
0	1	2	3	4	5	6	7	8	9



UNION/FIND算法示例

对两个等价对 (**K**, **A**) 和 (**E**, **G**) 的处理结果

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





树的父指针表示与 **UNION/FIND**算法实现

```
template<class T>  
class ParTreeNode  
{ //树结点定义  
private:  
    Tvalue;//结点的值  
    ParTreeNode<T>* parent;//父结点指针  
    intnCount;//以此结点为根的子树的总结点个数  
public:  
    ParTreeNode();//构造函数  
    virtual ~ParTreeNode(){};//析构函数
```



树的父指针表示与 **UNION/FIND**算法实现

```
TgetValue(); //返回结点的值  
void setValue(const T& val); //设置结点的值  
//返回父结点指针  
ParTreeNode<T>* getParent();  
//设置父结点指针  
void setParent(ParTreeNode<T>* par);  
//返回结点数目  
int getCount();  
//设置结点数目  
void setCount(const int count);  
};
```



树的父指针表示与 **UNION/FIND**算法实现

```
template<class T> ParTreeNode<T>::ParTreeNode()  
{  
    parent=NULL;  
    nCount=1;  
}  
template<class T> T ParTreeNode<T>::getValue()  
{  
    return value;  
}
```




树的父指针表示与 **UNION/FIND**算法实现

```
template<class T>  
void ParTreeNode<T>::setValue(const T& val)  
{  
    value=val;  
}  
  
template<class T>  
ParTreeNode<T>* ParTreeNode<T>::getParent()  
{  
    return parent;  
}
```



树的父指针表示与 **UNION/FIND**算法实现

```
template<class T> void
ParTreeNode<T>::setParent(ParTreeNode<T>* par)
{
    parent=par;
}
template<class T>
int ParTreeNode<T>::getCount()
{
    return nCount;
}
```



树的父指针表示与 **UNION/FIND**算法实现

```
template<class T>
```

```
class ParTree
```

```
{ //树定义
```

```
public:
```

```
ParTreeNode<T>* array;//存储树结点的数组
```

```
intSize;           //数组大小
```

```
//查找node结点的根结点
```

```
ParTreeNode<T>* 
```

```
Find(ParTreeNode<T>* node) const;
```



树的父指针表示与 **UNION/FIND**算法实现

```
ParTree(const int size);//构造函数  
virtual ~ParTree();//析构函数  
//把下标为i, j的结点合并成一棵子树  
void Union(int i,int j);  
//判定下标为i, j的结点是否在一棵树中  
bool Different(int i,int j);  
};
```



树的父指针表示与 **UNION/FIND**算法实现

```
template <class T>
ParTree<T>::ParTree(const int size)
{
    Size=size;
    array=new ParTreeNode<T>[size];
}
template <class T>
ParTree<T>::~~ParTree()
{
    delete []array;
}
```



树的父指针表示与 **UNION/FIND**算法实现

```
template <class T>  
ParTreeNode<T>*  
ParTree<T>::Find(ParTreeNode<T>* node) const  
{  
    ParTreeNode<T>* pointer=node;  
    while ( pointer->getParent()!=NULL)  
        pointer=pointer->getParent();  
    return pointer;  
}
```



树的父指针表示与 **UNION/FIND**算法实现

```
template<class T>  
bool ParTree<T>::Different(int i,int j)  
{  
    ParTreeNode<T>*  
    pointeri=Find(&array[i]);///找到结点i的根  
    ParTreeNode<T>*  
    pointerj=Find(&array[j]);///找到结点j的根  
    return pointeri!=pointerj;  
}
```



树的父指针表示与 **UNION/FIND**算法实现

```
template<class T>
void ParTree<T>::Union(int i,int j)
{
    //找到结点i的根
    ParTreeNode<T>* pointeri=Find(&array[i]);
    //找到结点j的根
    ParTreeNode<T>* pointerj=Find(&array[j]);
    if(pointeri!=pointerj)
    {
        if(pointeri->getCount()>=pointerj->getCount())
        {
            pointerj->setParent(pointeri);
```




树的父指针表示与 **UNION/FIND**算法实现

```
    pointeri->setCount(pointeri->
        getCount()+pointerj->getCount());
}
else
{
    pointeri->setParent(pointerj);
    pointerj->setCount(pointeri->
        getCount()+pointerj->getCount());
}
} //end if
}
```



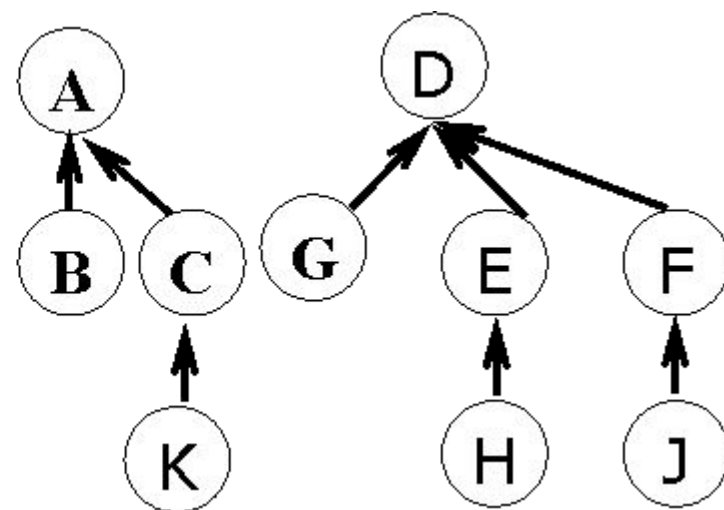
重量权衡合并规则

- 父指针表示法并不限制共享同一父结点的结点数目。为了使等价对的处理尽可能高效，每个结点到其相应的根结点的距离应尽可能小
- “重量权衡合并规则”（**weighted union rule**）。
 - 将结点较少树的根结点指向结点较多树的根结点。这可以把树的整体深度限制在 $O(\log n)$
 - 当处理完 n 个等价对后，任何结点的深度最多只会增加 $\log n$ 次

UNION/FIND算法示例

使用标准重量权衡合并规则处理等价对 (H,J) 的结果

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





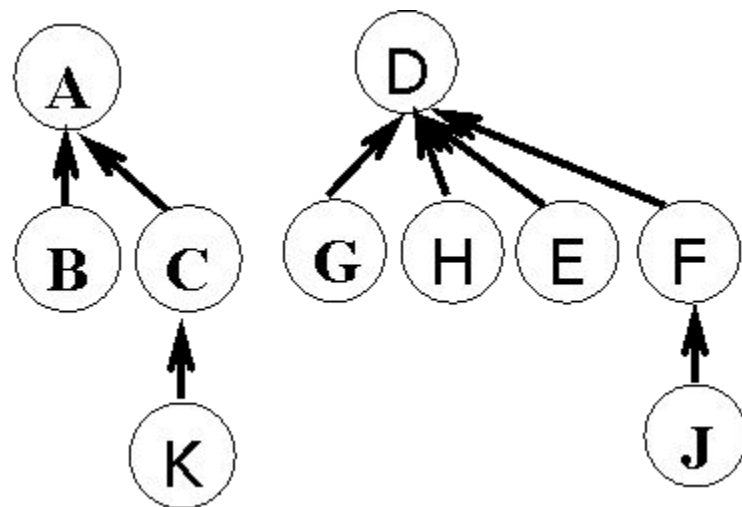
路径压缩

- 一种可以产生极浅树的方法。当查找一个结点**X**的根结点时可以采用路径压缩
- 设根结点为**R**，则路径压缩把由**X**到**R**的路径上每个结点的父指针均设置为直接指向**R**。
- 首先要查找**R**，然后顺着由**X**到**R**的路径把每个结点的父指针域均设置为指向**R**

UNION/FIND算法示例

使用路径压缩规则处理等价对 (H,E) 的结果

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





路径压缩

```
template <class T>  
ParTreeNode<T>*  
ParTree<T>::FindPC(ParTreeNode<T>* node) const  
{  
    if(node->getParent() == NULL)  
        return node;  
    node->setParent(FindPC(node->getParent()));  
    return node->getParent();  
}
```



5.3 树的顺序存储

- **5.3.1 带右链的先根次序表示法**
- **5.3.2 带双标记位的先根次序表示法**
- **5.3.3 带左链的层次次序表示法**
- **5.3.4 带度数的后根次序表示法**

带右链的先根次序表示法

- 在带右链的先根次序表示中，结点按先根次序顺序存储在一片连续的存储单元中。每个结点除包括结点本身数据外，还附加两个表示结构的信息字段，结点的形式为：

ltag	info	rlink
------	------	-------

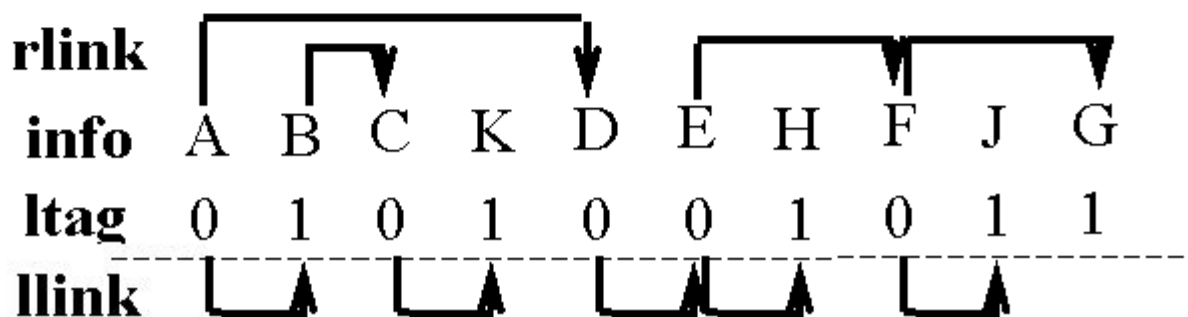
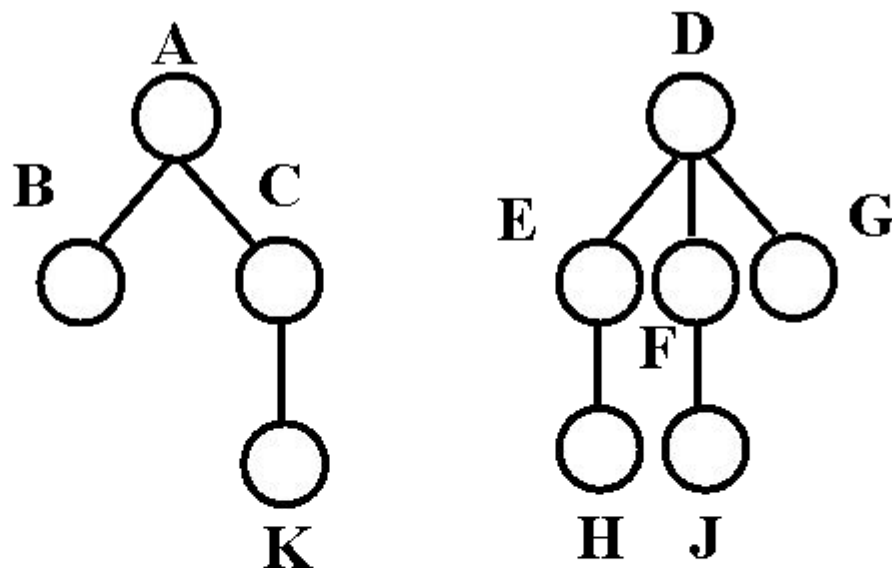
- **info**是结点的数据，
- **rlink**是右指针，指向结点的下一个兄弟、即对应的二叉树中结点的右子结点
- **ltag**是一个一位的左标记，当结点没有子结点，即对应的二叉树中结点没有左子结点时，**ltag**为 **1**，否则为**0**



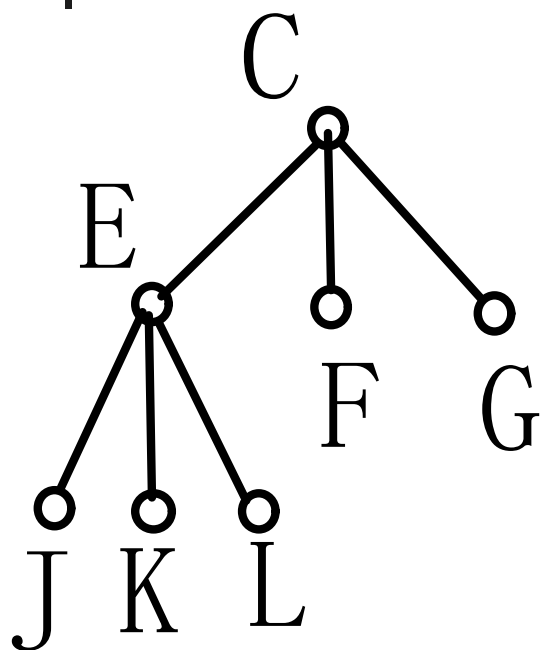
带右链的先根次序表示法

- 这种表示法与**llink—rlink**表示法相比，用**ltag**代替了**llink**，占用的存储单元要少些，但并不丢失信息
- 可以从结点的次序和**ltag**的值完全可以推知**llink**
 - **ltag**为0的结点有左子结点，它的**llink**指向存储区中该结点顺序的下一个结点
 - **ltag**为1的结点没有左子结点，它的**llink**为空

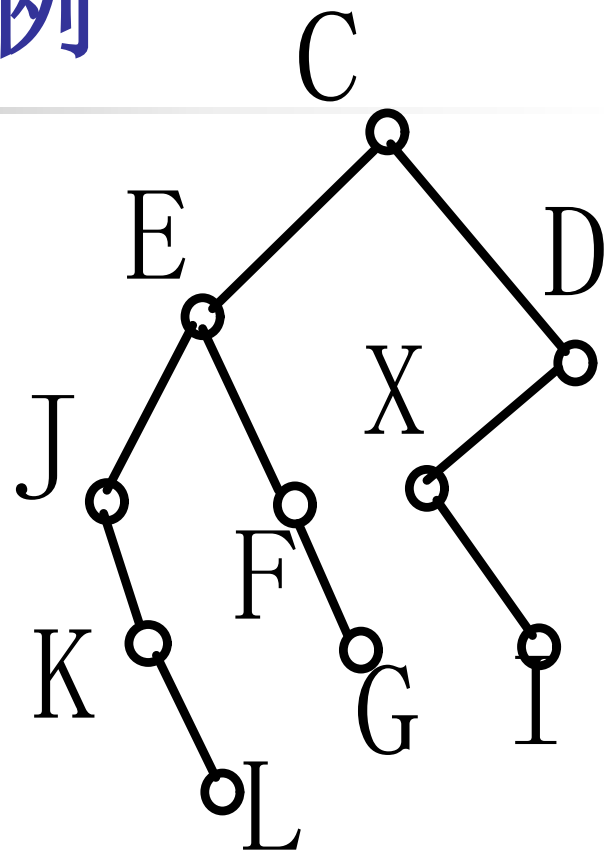
帶右鏈的先根次序表示法



森林图例

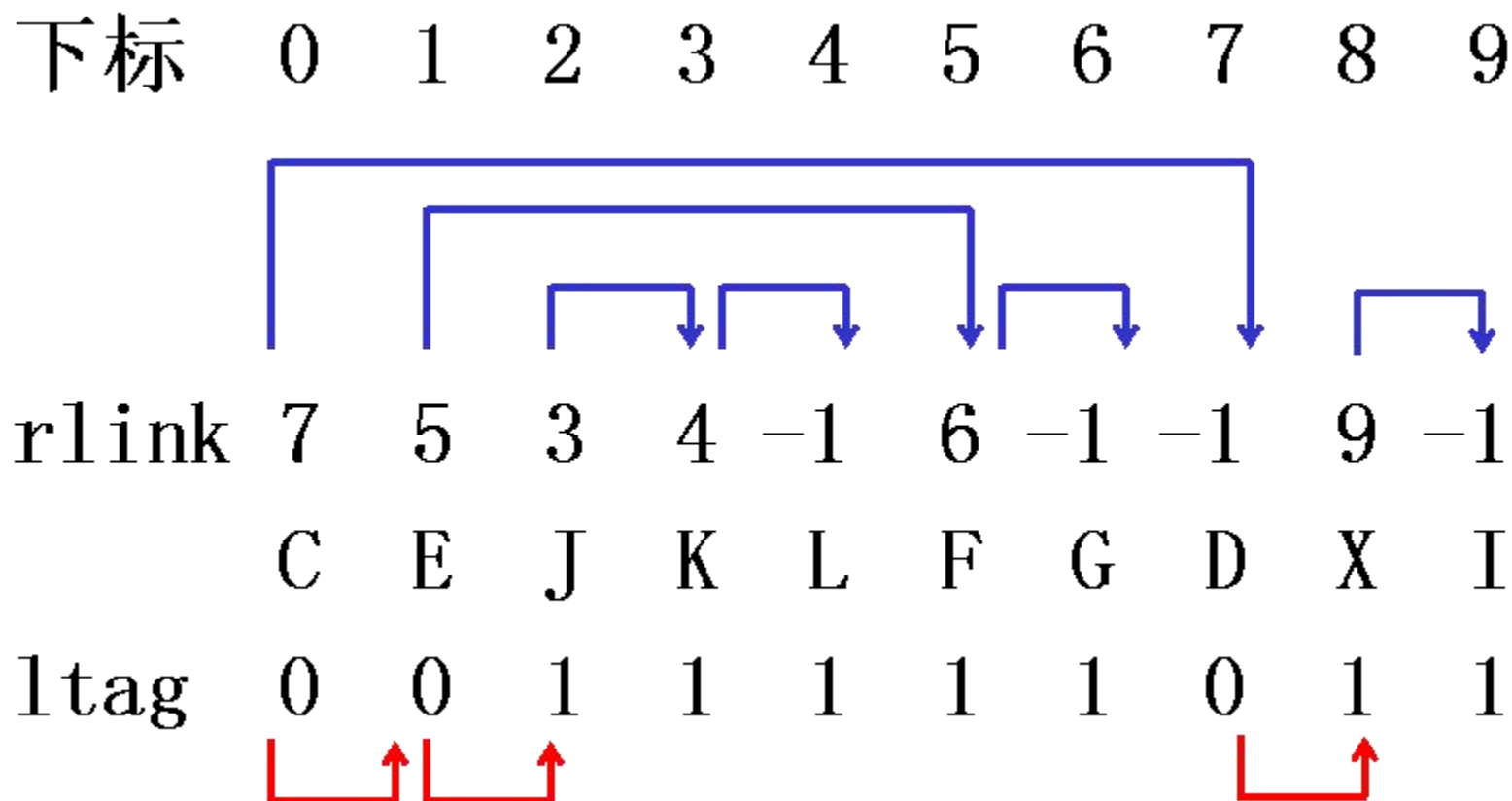


森林



等价的二叉树

带右链的先根次序rlink-ltag





带双标记位的先根次序表示法

- 带右链的先根次序表示法中每个结点都包括**ltag**和**rlink**字段，事实上**rlink**也不是必需的。代之以一位的**rtag**就足以表示出整个森林的结构信息
- 规定当结点没有下一个兄弟，即对应的二叉树中结点没有右子结点时**rtag**为**1**，否则为**0**



带双标记位的先根次序表示法

- 由结点的排列次序和**ltag**，**rtag**的值就可推知**rlink**的值。
 - 当一个结点**x**的**rtag**为**1**时，它的**rlink**显然应为空
 - 当一个结点**x**的**rtag**为**0**时，它的**rlink**应指向结点序列中排在结点**x**的子树结点后面的那个结点**y**
- 如何确定这个结点**y**呢？



带双标记位的先根次序表示法

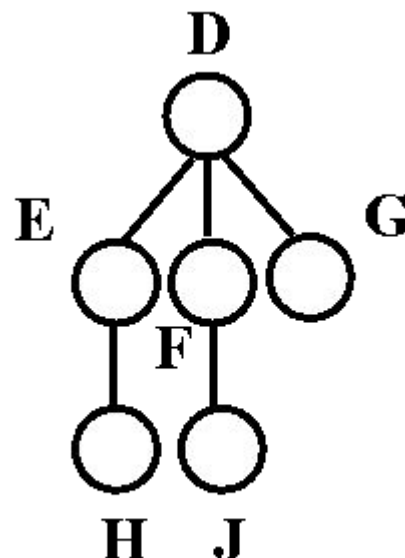
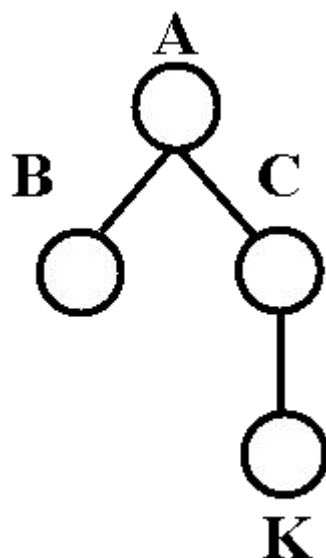
- 任何结点的子树结点在先根序列中都排在该结点之后；并且，任何结点的子树结点，在先根序列中排在最后的一个结点一定没有子结点，所以它的**ltag**为**1**
- 在顺序扫描带双标记位的先根次序表示，试图确定各结点的**rlink**值的过程中，当遇到一个**rtag**为**0**的结点**x**时，要继续往后扫描，在结点**x**的子树结点中找**ltag**为**1**的、该子树的最后一个结点，序列中排在这个结点后面的那个结点就是结点**x**的**rlink**应该指向的结点**y**



带双标记位的先根次序表示法

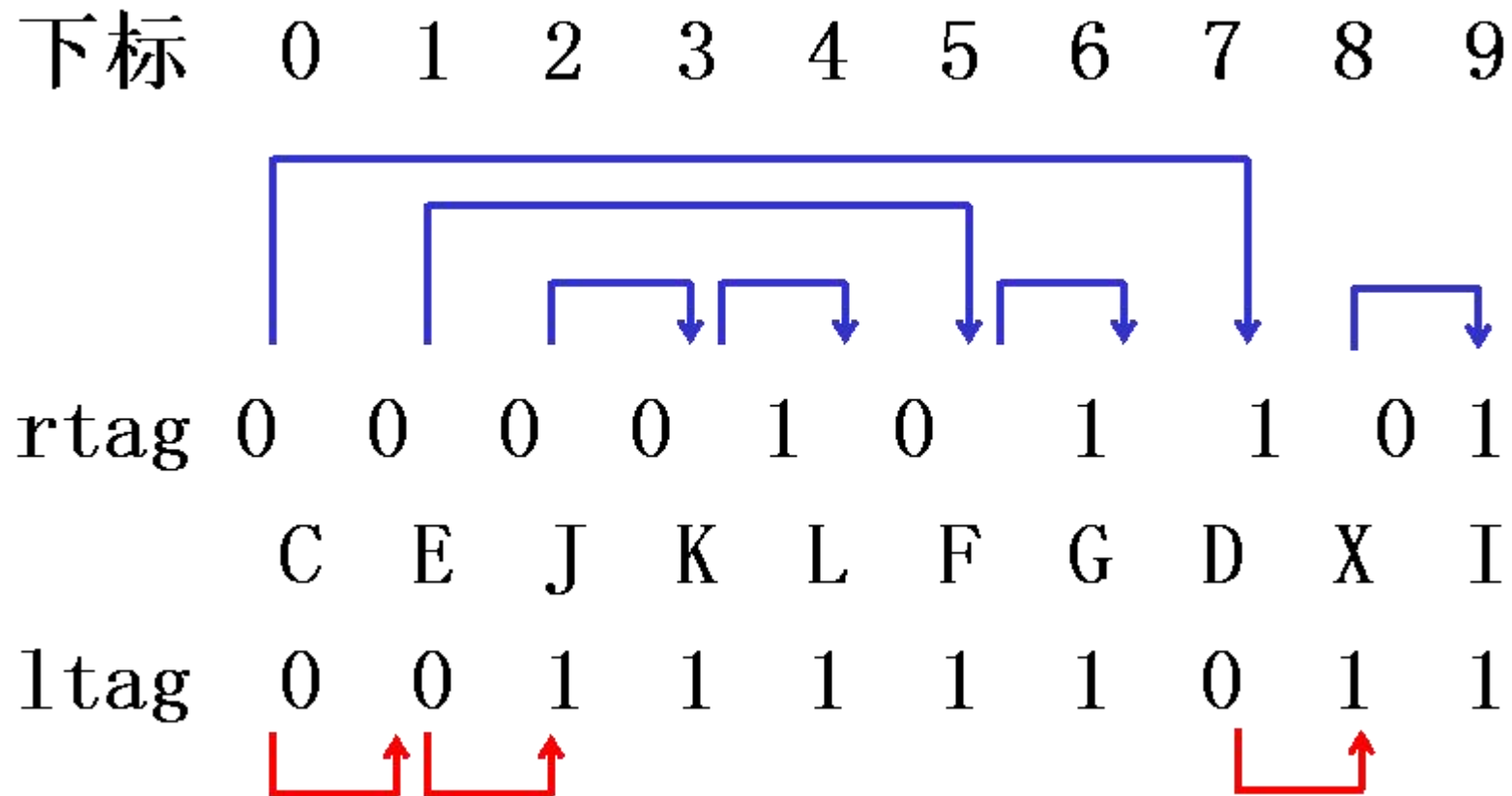
- 由于先根次序表示的树结构是嵌套的，因此在扫描结点 x 的子树结点序列找最后一个结点的过程中，极有可能遇到一棵更小的子树的根结点 x' ，它的 $rtag$ 也等于0
- 处理过程中需要用一個栈来记录待配置 $rlink$ 的结点
- 带双标记位的先根次序表示法虽然比带右键的先报次序表示法进一步节省了存储空间，但由于需要额外的处理来推导失去的信息，所以采用得更多的还是带右键的先根次序表示法

带双标记位的先根次序表示法



rtag	0	0	1	1	1	0	1	0	1	1
info	A	B	C	K	D	E	H	F	J	G
ltag	0	1	0	1	0	0	1	0	1	1

带双标记位的先根次序rtag-ltag





带双标记位先根次序树构造算法

```
template<class T>  
class DualTagTreeNode  
{//双标记位先根次序树结点类  
    public:  
        Tinfo;//树结点信息  
        int ltag;//左标记  
        int rtag;//右标记  
        DualTagTreeNode();  
        virtual ~DualTagTreeNode();  
};
```



带双标记位先根次序树构造算法

```
template<class T>
```

```
DualTagTreeNode<T>::DualTagTreeNode()
```

```
{
```

```
    ltag=1;
```

```
    rtag=1;
```

```
}
```

```
template<class T>
```

```
DualTagTreeNode<T>::~~DualTagTreeNode()
```

```
{}
```



带双标记位先根次序树构造算法

```
template <class T>
```

```
Tree<T>::Tree(DualTagTreeNode<T>* nodeArray,  
int count)
```

```
{//利用带双标记位的先根次序表示的树构造左子结点右兄弟方  
式
```

```
//表示的树
```

```
using std::stack; // 使用STL中的stack
```

```
stack<TreeNode<T>* > aStack;
```

```
//准备建立根结点
```

```
TreeNode<T>* pointer=new TreeNode<T>;
```

```
root=pointer;
```



带双标记位先根次序树构造算法

//处理一个结点

for(int i=0;i<count-1;i++)

{

pointer->setValue(nodeArray[i].info);

if(nodeArray[i].rtag==0)

aStack.push(pointer); //将结点压栈

else

pointer->setSibling(NULL);//右兄弟设为空

TreeNode<T>* temppointer=new TreeNode<T>;



带双标记位先根次序树构造算法

```
if(nodeArray[i].ltag==0)
pointer->setChild(temppointer);
else{
    pointer->setChild(NULL);/左子结点设为空
    pointer=aStack.top();
    aStack.pop();
    pointer->setSibling(temppointer);
}
```

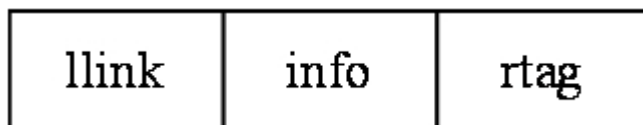


带双标记位先根次序树构造算法

```
pointer=temppointer;  
}//end for  
//处理最后一个结点  
pointer->setValue(nodeArray  
[count-1].info);  
pointer->setChild(NULL);  
pointer->setSibling(NULL);  
}
```


带左链的层次次序表示法

- 在带左链的层次次序表示中，结点按层次次序顺序存储在一片连续的存储单元中，每个结点除包括结点本身数据外，还附加两个表示结构的信息字段



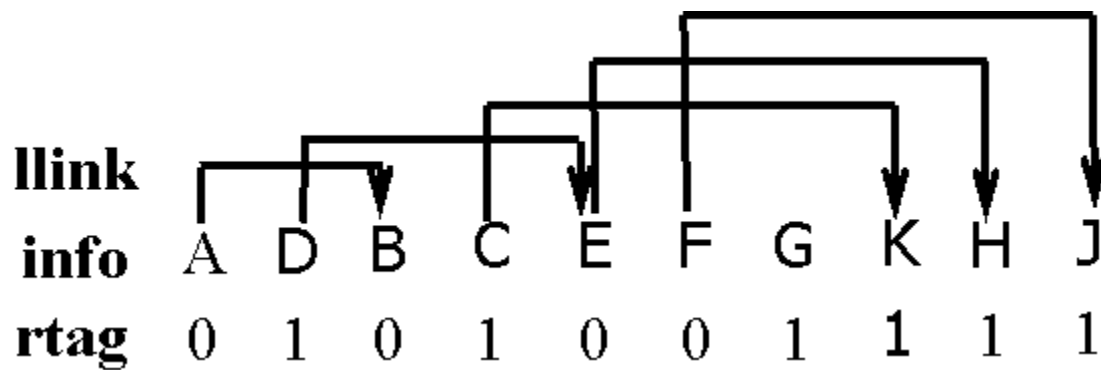
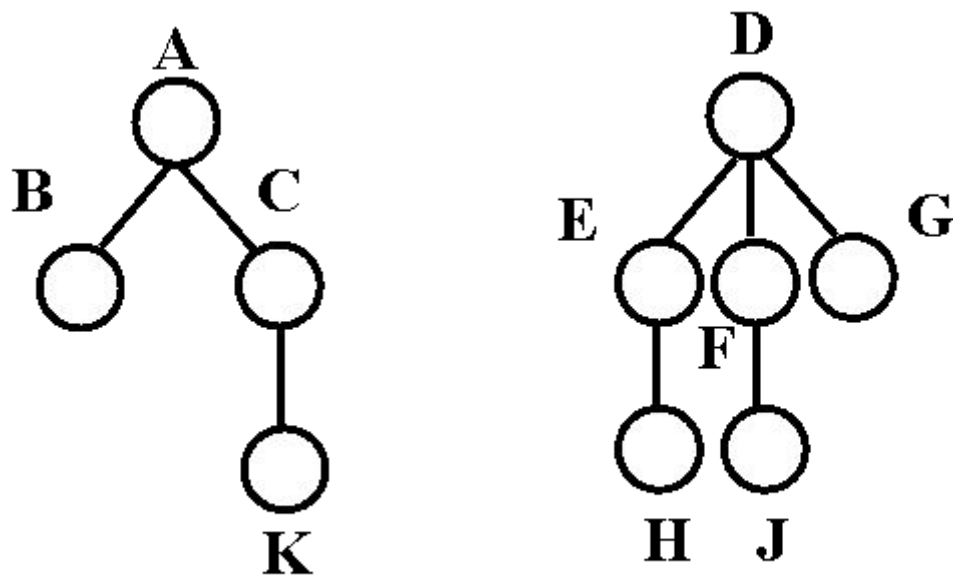
- **info**是结点的数据
- **llink**是左指针，指向结点的第一个子结点，即在对应的二叉树中的左子结点
- **rtag**是一个一位的右标记，当结点没有下一个兄弟，即对应的二叉树中结点没有右子结点时，**rtag**为**1**，否则为**0**



带左链的层次次序表示法

- 这个表示法里没有包括结点的**rlink**，但由结点的次序和**rtag**的值完全可以推知**rlink**
 - 当结点的**rtag**为**1**时，**rlink**为空
 - 当结点的**rtag**为**0**时，**rlink**指向顺序的下一个结点

带左链的层次次序表示法





带左链层次次序树构造算法

```
template<class T>  
class LeftLinkTreeNode  
{  
    //带左链层次次序树结点类  
    public:  
    Tinfo; //树结点信息  
    LeftLinkTreeNode<T>*llink; //左指针  
    int rtag; //右标记  
    LeftLinkTreeNode();  
    virtual ~LeftLinkTreeNode(){};  
};
```



带左链层次次序树构造算法

```
template <class T>
```

```
LeftLinkTreeNode<T>::LeftLinkTreeNode()
```

```
{//构造函数
```

```
llink=NULL;
```

```
rtag=1;
```

```
}
```



带左链层次次序树构造算法

```
template <class T>
```

```
Tree<T>::Tree(LeftLinkTreeNode<T>* nodeArray,  
int count)
```

```
{//构造函数,利用左链层次顺序构造树
```

```
using std::queue;//使用STL中的队列
```

```
queue<TreeNode<T>*> aQueue;
```

```
TreeNode<T>* pointer=new TreeNode<T>;
```

```
//为根结点做准备
```

```
root=pointer;
```

```
int currentIndex= - 1;//数组标识初始化为负1
```



带左链层次次序树构造算法

```
while(pointer || !aQueue.empty())
{
    if(pointer){ // 当前访问结点不空
        currentIndex++;
        pointer->
        setValue(nodeArray[currentIndex].info);
        if(nodeArray[currentIndex].llink){ // 设置当前结点的左
            子结点指针，并将左子结点指针入 // 队列
            TreeNode<T>* leftpointer=new TreeNode<T>;
```



带左链层次次序树构造算法

```
pointer->setChild(leftpointer);  
aQueue.push(leftpointer);  
}  
else pointer->setChild(NULL);  
if(nodeArray[currentIndex].rtag==0)  
{//设置当前结点的右兄弟指针  
    TreeNode<T>* rightpointer=new TreeNode<T>;  
    pointer->setSibling(rightpointer);  
}  
else pointer->setSibling(NULL);
```




带左链层次次序树构造算法

```
//沿当前结点的右兄弟结点向下遍历
pointer=pointer->RightSibling();
}
else
{
    //当前结点为空，从队列中出结点
    pointer=aQueue.front();
    aQueue.pop();
}
} //end while
}
```



带度数的后根次序表示法

- 在带度数的后根次序表示中，结点按**后根次序顺序**存储在一片连续的存储单元中，结点的形式为



- 其中**info**是结点的数据，**degree**是结点的度数



带度数的后根次序表示法

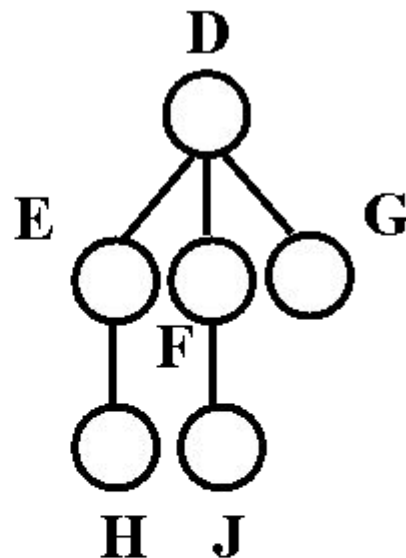
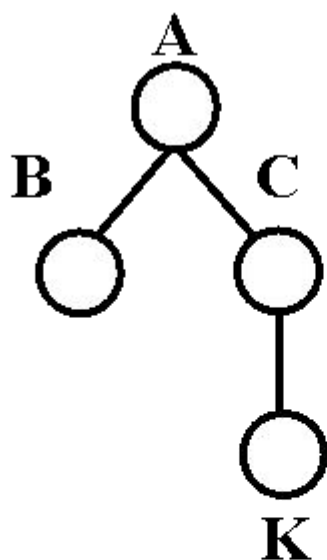
- 这种表示法不包括指针，但它仍能完全反映树的结构
- 若某结点的**degree**值为**m**，则该结点有**m**个子结点，最右的子结点就是后根次序序列中该结点的前驱，最后第二个子结点是以最右子结点为根的子树在后根次序序列中的前驱，...



带度数的后根次序表示法

- 以结点 x 为根的子树在后根次序序列中的前驱可以这样求：令 $s=0$ ，在后根次序序列中，从 x 开始从后往前走，每经过一个结点 z ，执行 $s \leftarrow s+1-\text{degree}(z)$ ；到 $s=1$ 时，则再往前走的一个结点就是以 x 为根的子树在后根次序序列中的前驱

带度数的后根次序表示法



degree	0	0	1	2	0	1	0	1	0	3
info	B	K	C	A	H	E	J	F	G	D



森林的链式存储讨论

- 带度数的先根次序？
- 带度数的层次次序？
- 冗余问题



5.4 K叉树

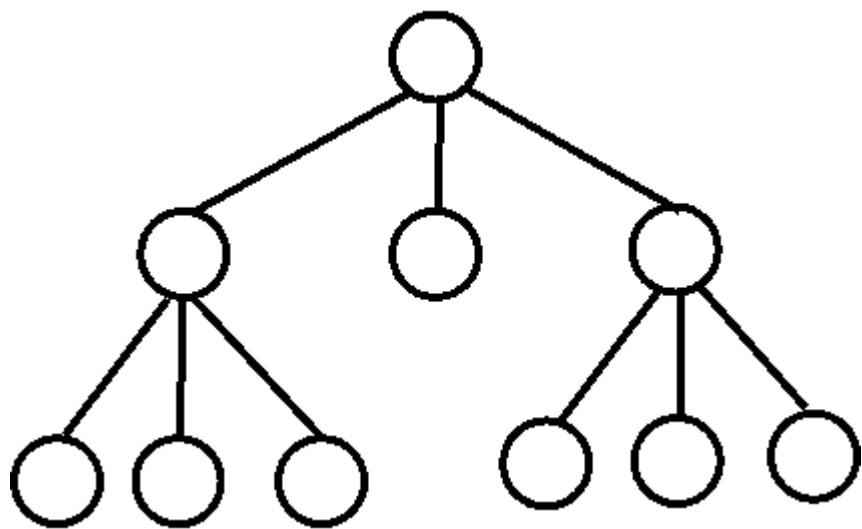
- **K叉树(K-ary Tree)**的结点有**K**个子结点
- 不同于树，**K叉树**的结点有**K**个子结点，子结点数目是固定的，因此相对来说容易实现。
- 注意当**K**变大时，空指针的潜在数目会增加，并且叶结点与分支结点在所需空间大小上的差异也会更为显著。这样，当**K**增加时，对叶结点与分支结点采用不同实现的需要就变得更加迫切。



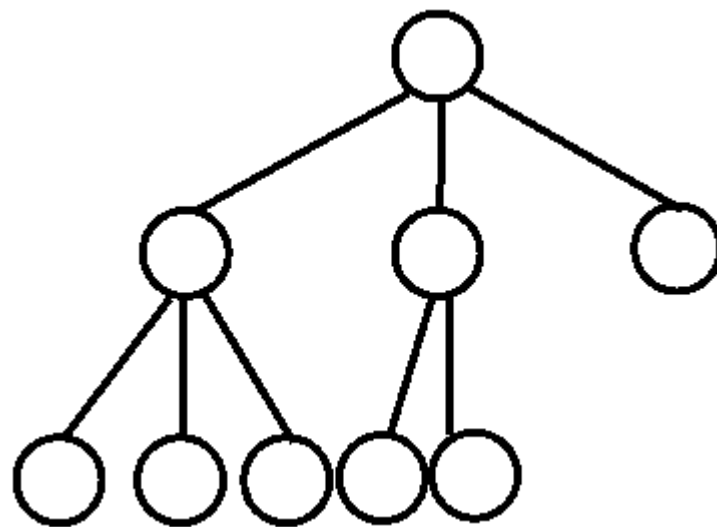
5.4 K叉树

- 满K叉树和完全K叉树与满二叉树和完全二叉树是类似的
- 二叉树的许多性质可以推广到K叉树
- 也可以把完全K叉树存储在一个数组中

5.4 K叉树



满3叉树

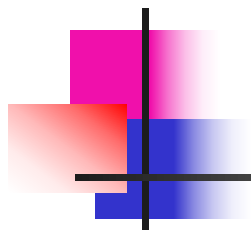


完全3叉树



总结

- 树和森林的概念
- 树与二叉树的联系、区别与转换
- 树的链式存储
 - “左子结点/右兄弟结点” 二叉链表
 - 父指针表示法
- 树的顺序存储
- **K**叉树



The End