北京大学

第六章 树

宋国杰

北京大学信息科学技术学院

课程内容

- 6.1 树的概念
- 6.2 树的链式存储
- 6.3 树的顺序存储
- 6.4 K叉树

6.1.1 树的定义

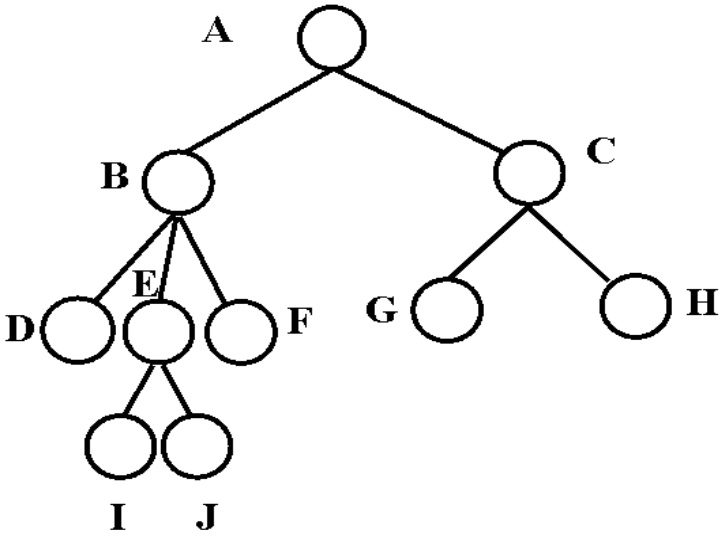
- 树是包括n个结点的有限集合T（n≥1），使得：
  - 一个根结点
  - 除根以外的其它结点被分成m个(m≥0)不相交的集合T1，T2，…，Tm，而且这些集合的每一个又都是树。 树T1，T2，…，Tm称作这个根的子树
- 定义是递归的

逻辑结构

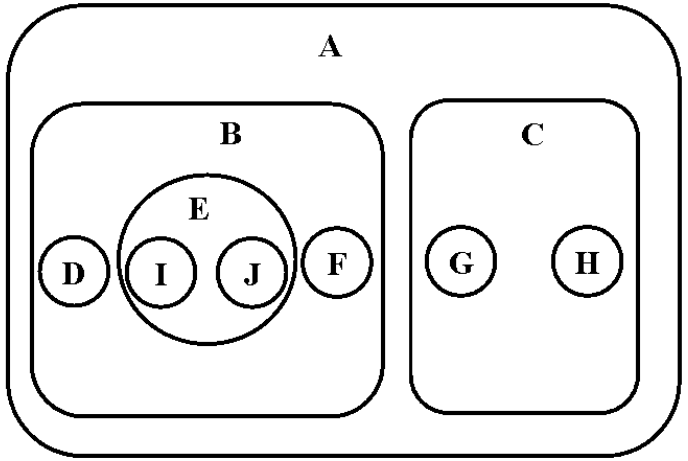
- 包含n个结点的有穷集合K (n>0)，且在K上定义了一个关系N，关系N满足以下条件：
  - 有且仅有一个结点$k_0 \in K$，它对于关系N来说没有前驱。结点k_0称作树的根
  - 除k_0外，K中每个结点对于关系N来说都有且仅有一个前驱
  - 除k_0外，任何结点$k \in K$，存在一结点序列k_0，k_1，…，k_s，使得k_0就是树根，且$k_s=k$，其中有序对$<k_{i-1}, k_i> \in N(1 \le i \le s)$。这样的结点序列称为从根到结点k的一条路径

树形结构的表示法

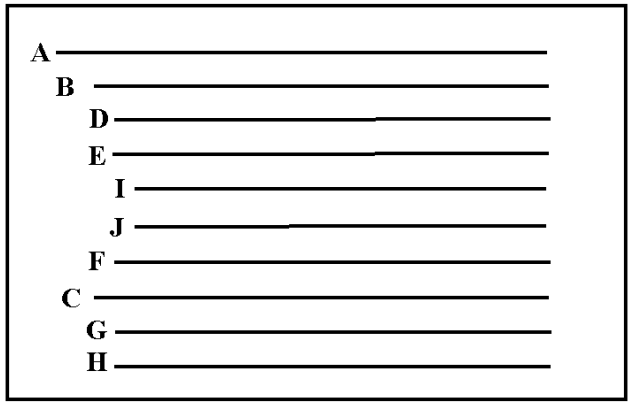
- 树的逻辑结构
  - 结点集合K={A，B，C，D，E，F，G，H，I，J}
  - K上的关系N={ <A，B>，<A，C>，<B，D>，
    <B，E>，<B，F>，<C，G>，
    <C，H>，<E，I>，<E，J> }



（a）树形表示法



（b）文氏图表示法

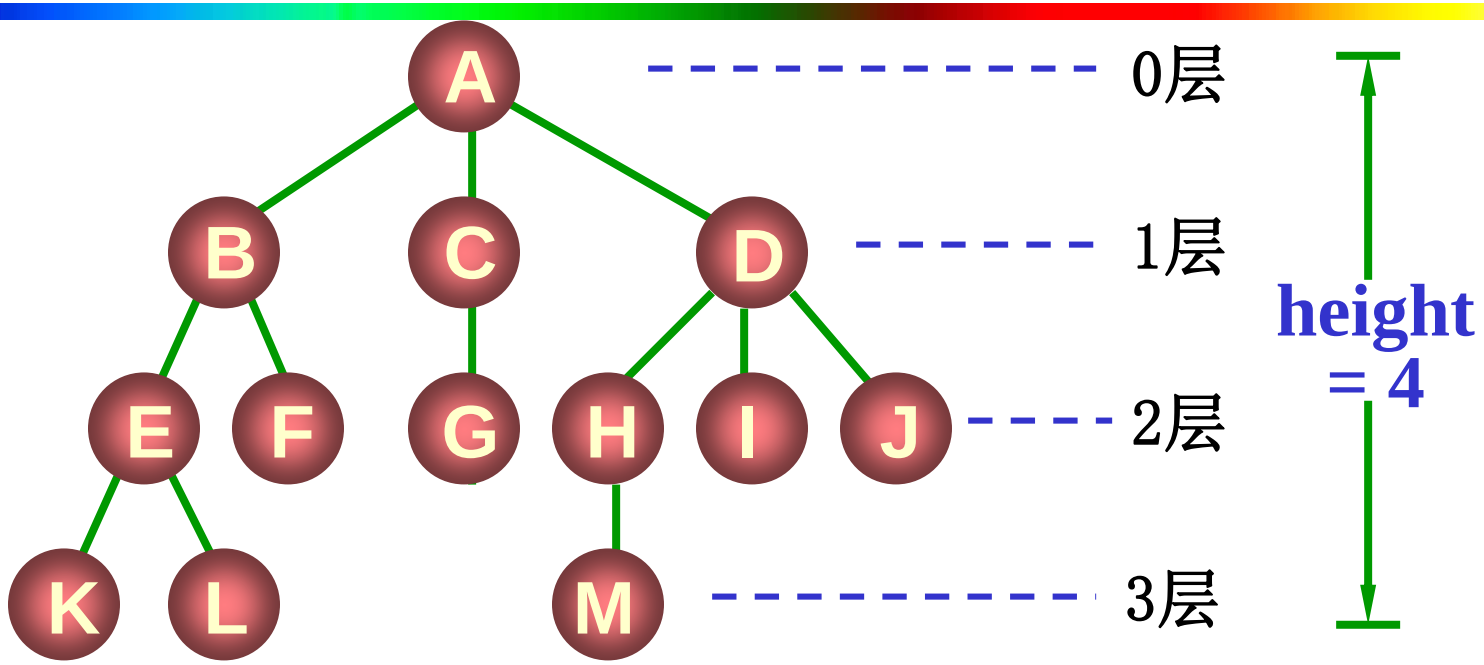


（c）凹入表表示法

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

（d）嵌套括号表示法

树的基本术语



结点
结点的度
叶结点
分支结点

子女
双亲
兄弟

祖先
子孙
结点层次

树的深（高）度
树的度
有序树
无序树
森林

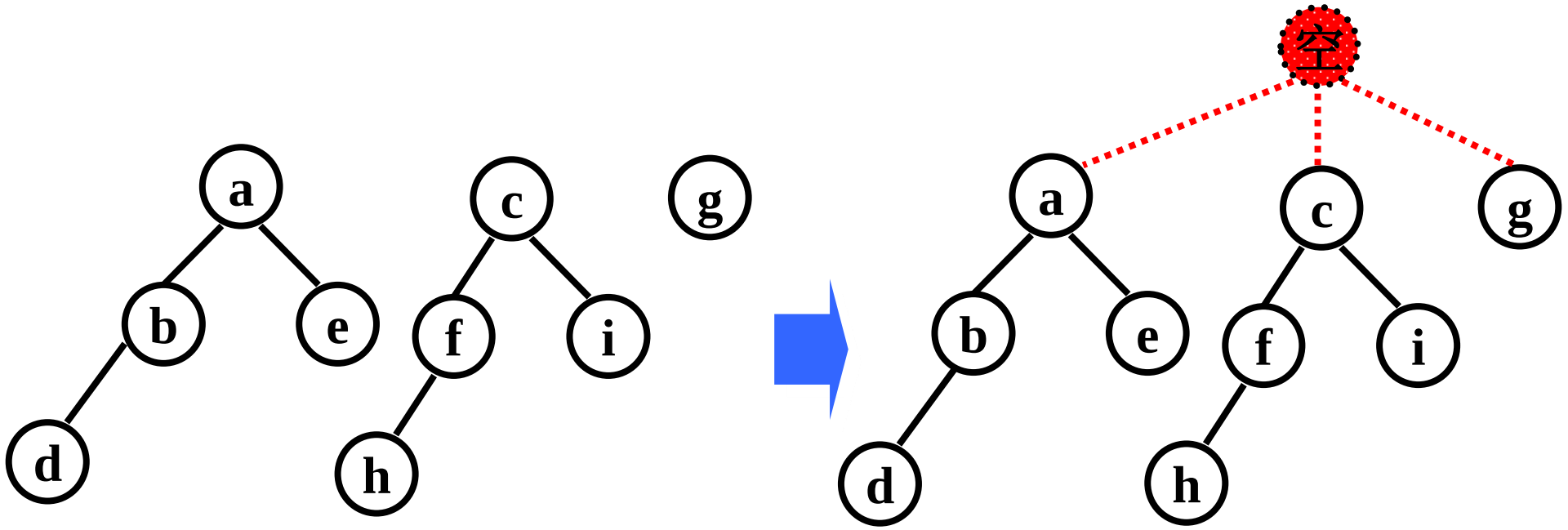
树结构中的概念

- 有序树
  - 把树结点的子结点按从左到右的次序顺序编号
- **度为2的有序树并不是二叉树**
  - 第一子结点被删除后，第二子结点自然顶替成为第1子结点。
  - 度为2并且严格区分左右两个子结点的有序树才是二叉树。

森林

- 森林是零棵或多棵不相交的树集合（通常是有序集合）
- 对于树中的每个结点，其子树组成的集合就是森林；

  而加入一个结点作为根，森林就可以转化成一棵树了



6.1.2 森林与二叉树的等价转换

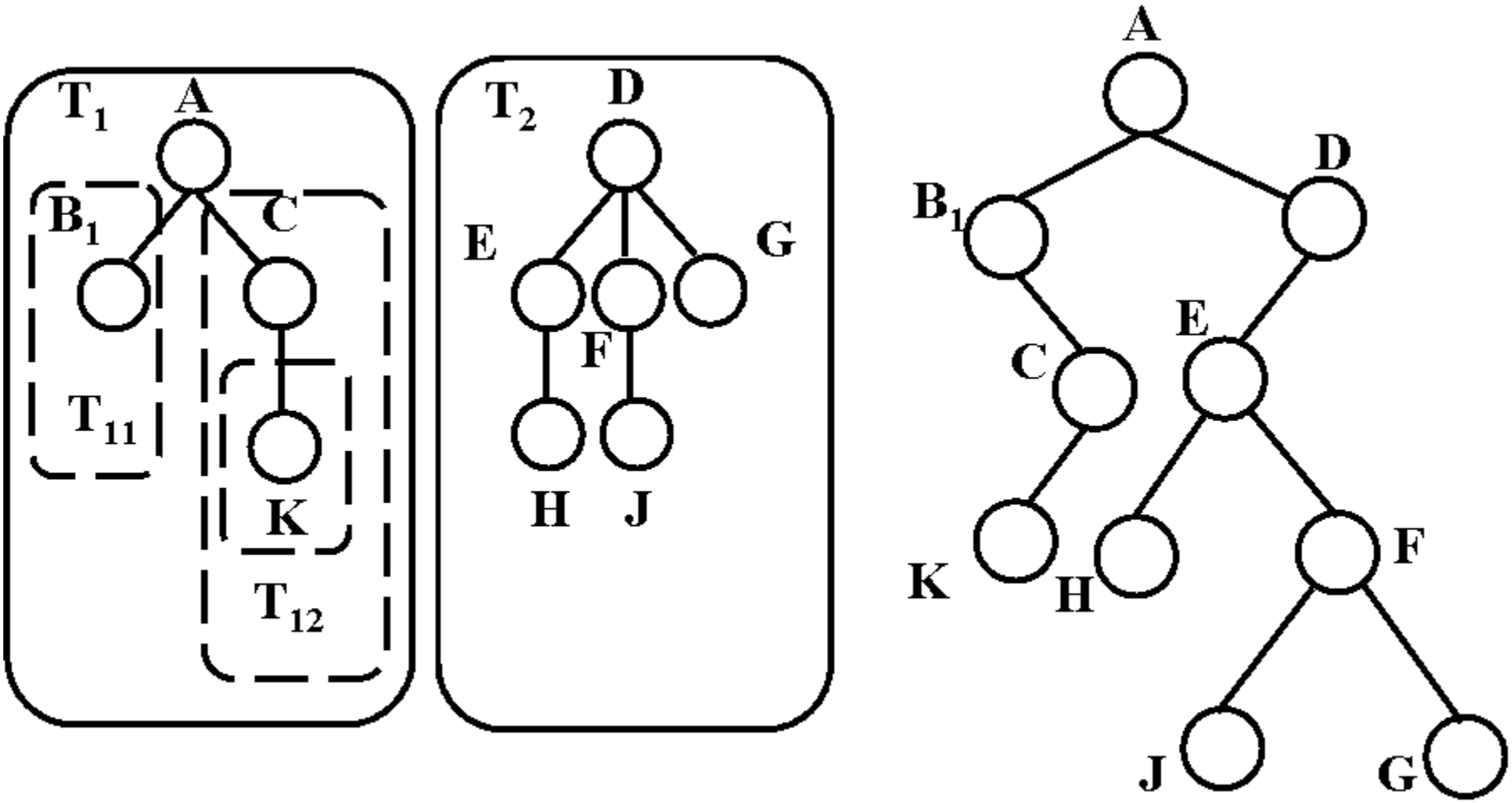
- 树或森林与二叉树一一对应
  - 任何森林都可以用一棵二叉树唯一表达
  - 任何二叉树也都唯一对应到一个森林
- 树所对应的二叉树中
  - 一个结点的左子结点是它在原来树里的第一个子结点
  - 右子结点是它在原来的树里的下一个兄弟

左孩子,右兄弟

森林到二叉树的等价转换

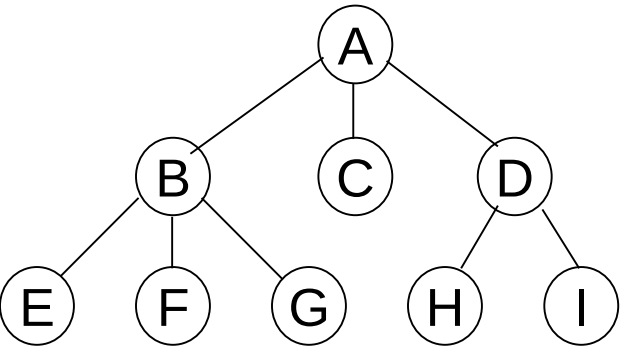
- 把森林F看作树的有序集合，$F=(T_1, T_2, \ldots, T_n)$，对应于F的二叉树B(F)的定义是：
  - 若n=0，则B(F)为空
  - 若n>0，则B(F)的根是T_1的根R_1，B(F)的左子树是$B(T_{11}, T_{12}, \ldots, T_{1m})$，其中$T_{11}, T_{12}, \ldots, T_{1m}$是$R_1$的子树；B(F)的右子树是$B(T_2, \ldots, T_n)$
- 此定义精确地确定了从森林到二叉树的转换

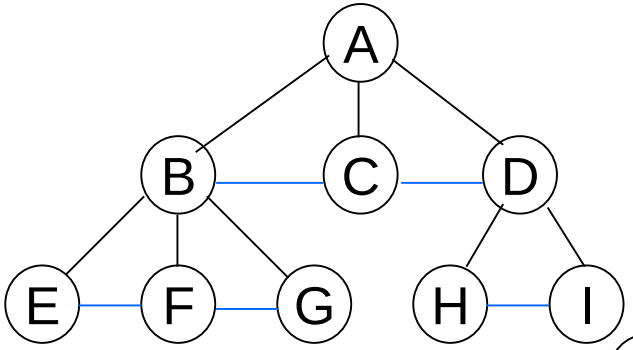
示例1

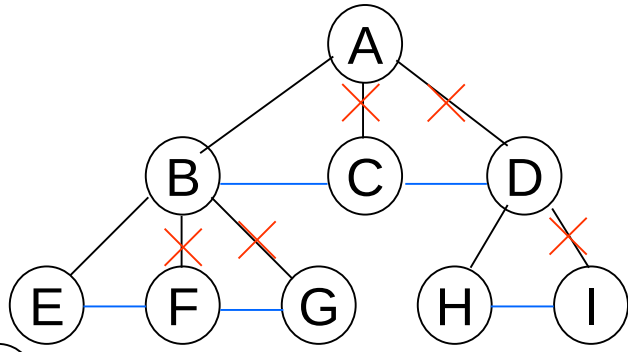


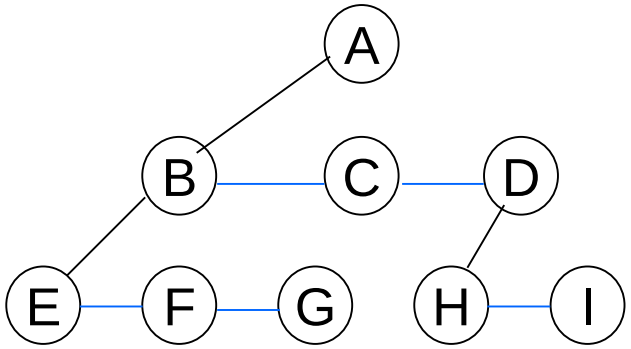
示例2

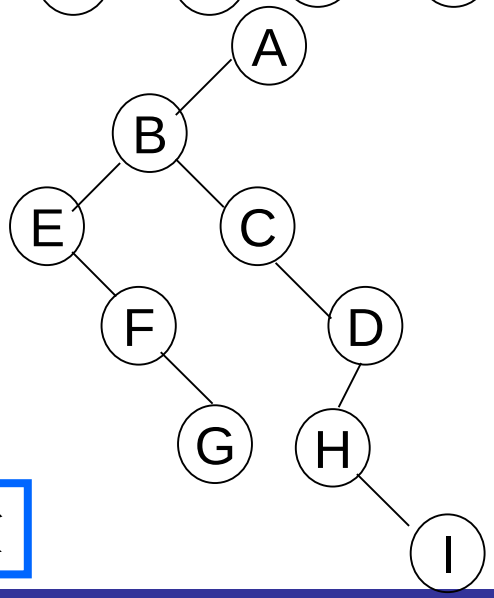
- **加线**：在树中所有相邻的兄弟之间加一连线
- **抹线**：对树中每个结点，除其最左孩子外，抹去该结点与其余孩子间的连线
- **整理**：以树的根结点为轴心，将整树顺时针转45°







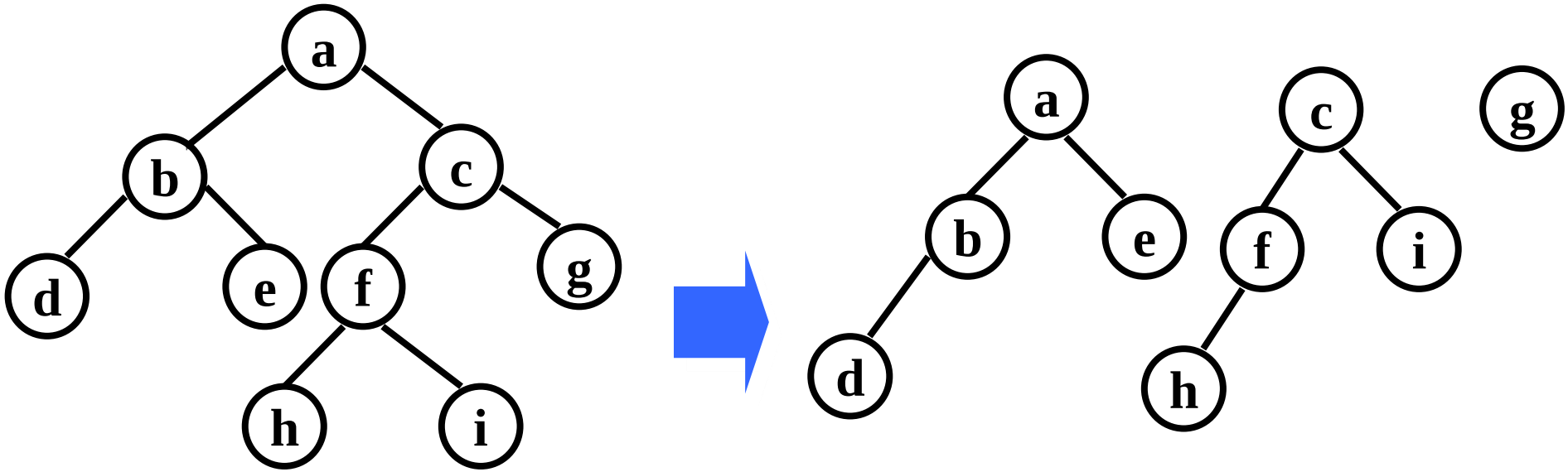




树转换成的二叉树其右子树一定为空

二叉树到森林的等价转换

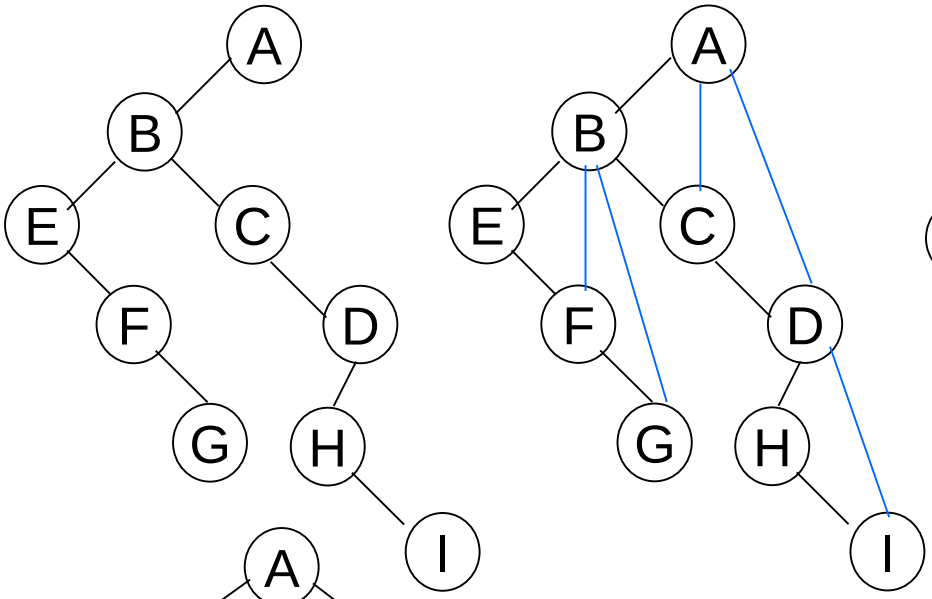
- 设B是一棵二叉树，root是B的根，L和R分别是root的左子树和右子树，则森林 F(B)的定义是：
  - 若B为空，则F(B)是空的森林。
  - 若B不为空，则F(B)是一棵树T_1加上森林F(R)，其中树T_1的根为root，root的子树为F(L)

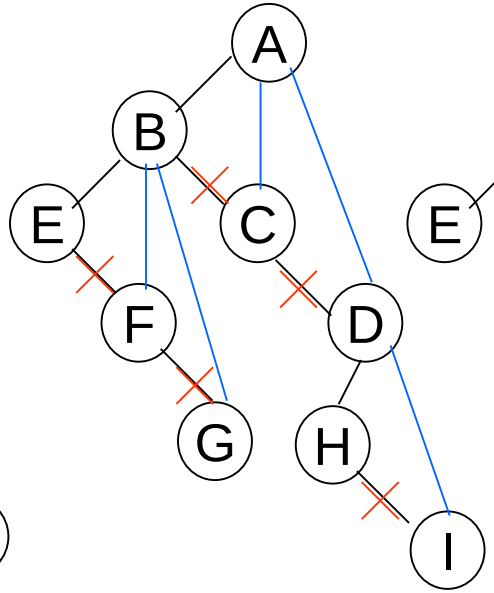


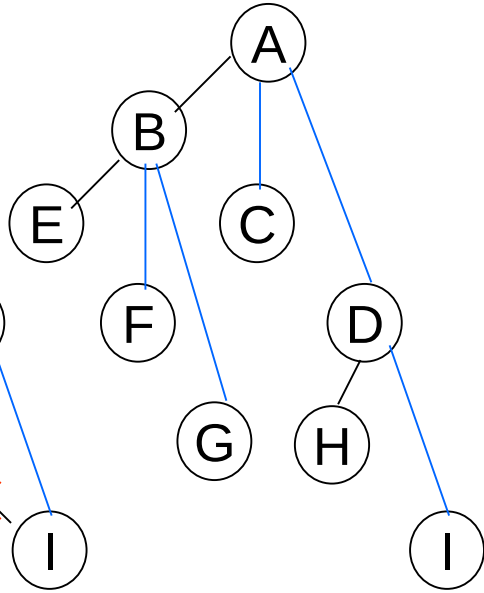
（a）二叉树　　（b）森林

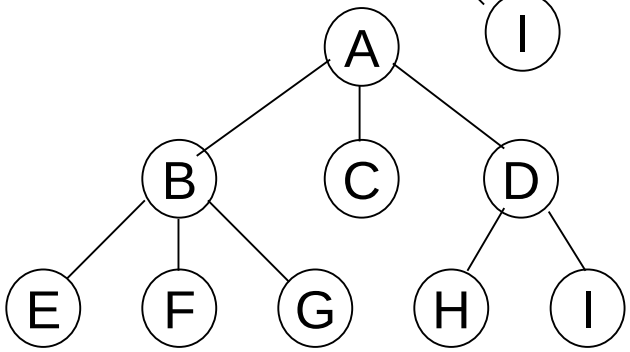
示例1：二叉树转换为树

- **加线**：若p结点是父结点的左孩子，则将p的右孩子，右孩子的右孩子，……沿分支找到的所有右孩子，都与p的双亲用线连起来
- **抹线**：抹掉原二叉树中双亲与右孩子之间的连线
- **调整**：将结点按层次排列，形成树结构



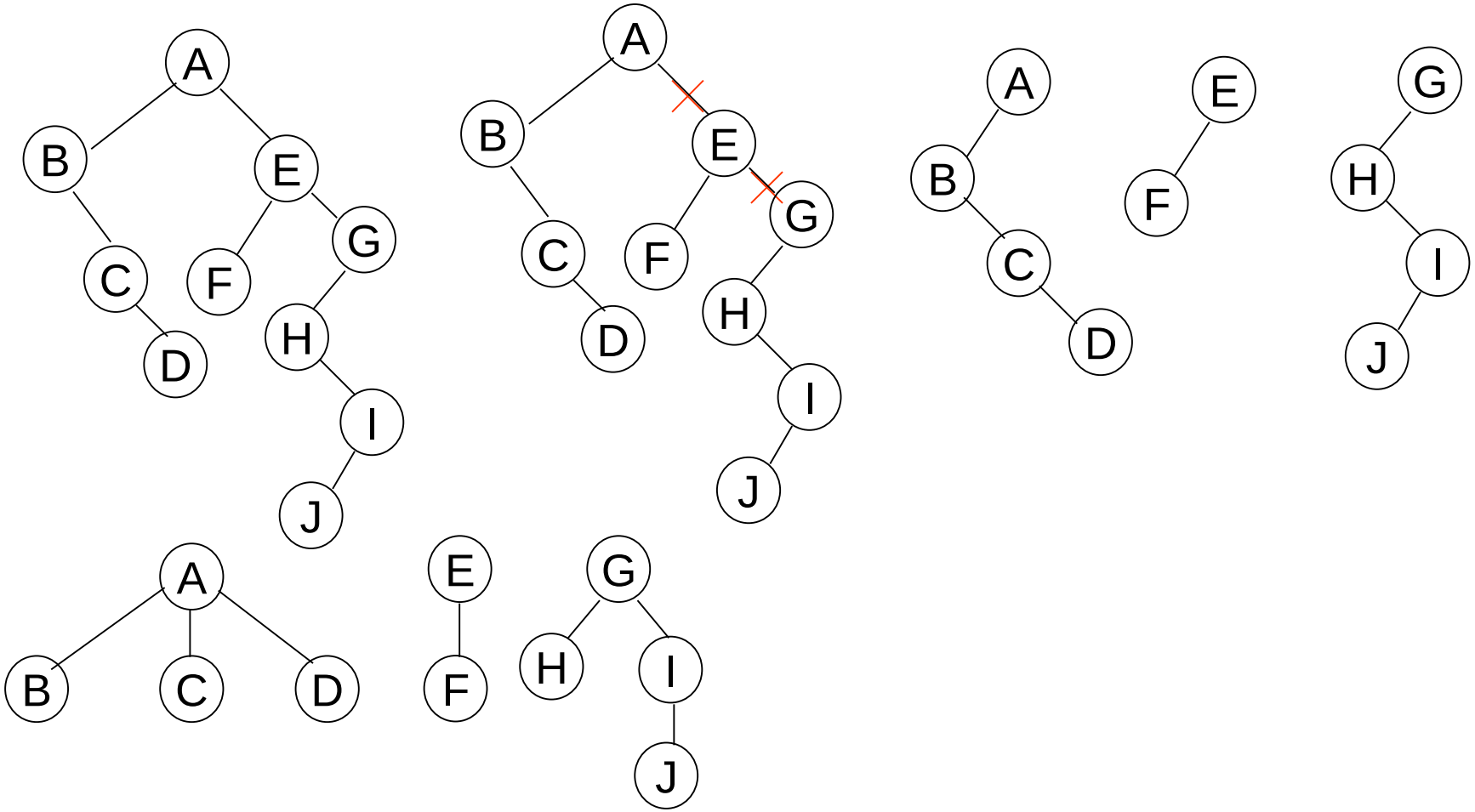






示例2：二叉树转换成森林

- 抹线：将二叉树中根结点与其右孩子连线，及沿右分支搜索到的所有右孩子间连线全部抹掉，使之变成孤立的二叉树
- 还原：将孤立的二叉树还原成树



6.1.3 树的抽象数据类型

➢ 树节点的抽象数据类型

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

树的抽象数据类型

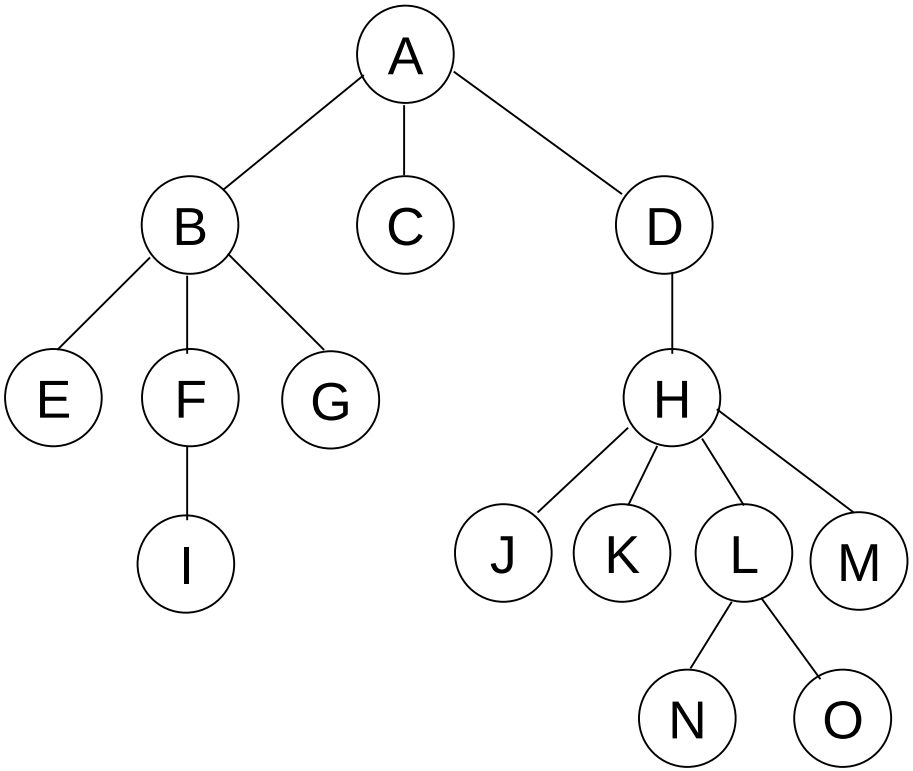
```
template <class T>
    class Tree  {
    public:
        Tree();                                  //构造函数
        virtual ~Tree();                         //析构函数
        TreeNode<T>* getRoot();                  //返回树中的根结点
        //创建树中的根结点，使根结点元素的值为rootValue
        void CreateRoot(const T& rootValue);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

```
    //返回current结点的父结点
    TreeNode<T>* Parent(TreeNode<T>* current);
    //返回current结点的前一个兄弟结点
    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);
};
```

6.1.4 树(森林)的周游

- 按深度方向周游
  - 先根次序
    - 若树非空，则遍历方法为：
      - 访问根结点
      - 从左到右，依次先根遍历根结点的每一棵子树
  - 后根次序
    - 若树非空，则遍历方法为：
      - 从左到右，依次后根遍历根结点的每一棵子树
      - 访问根结点

无中根次序周游



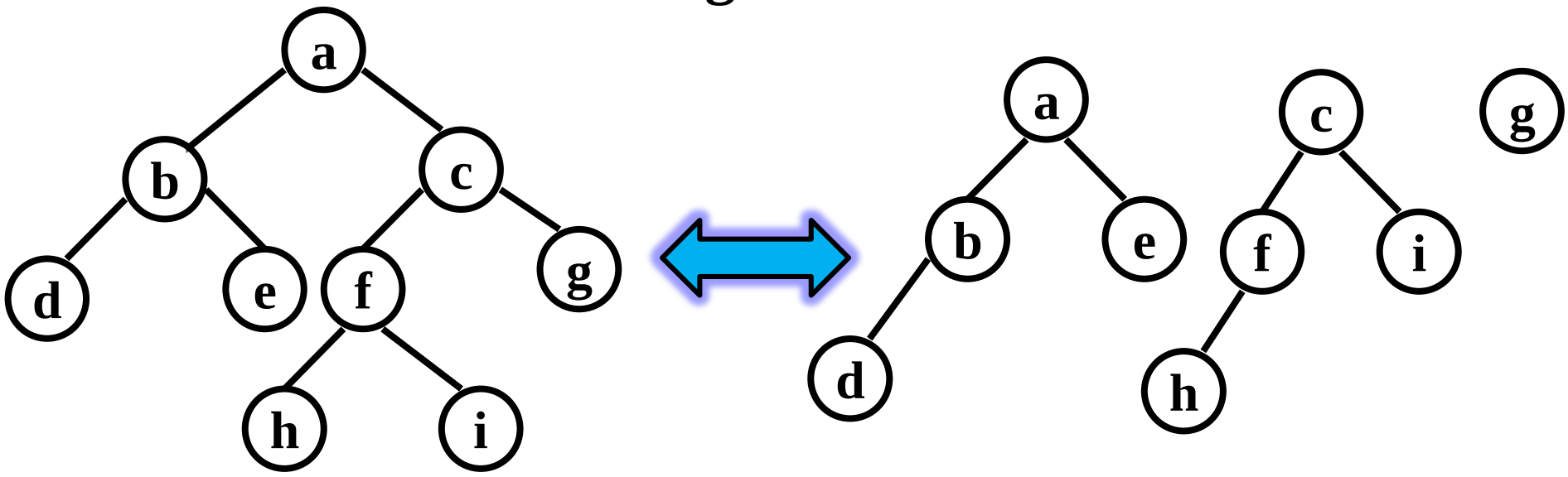
先根遍历：**ABEFIGCDHJKLNOM**

后根遍历：**EIFGBCJKNOLMHDA**

层次遍历：**ABCDEFGHIJKLMNO**

周游性质

- 按先根次序周游树正好等于对应二叉树的前序周游
  - 周游结果：abdecfhig
- 按后根次序周游树正好等于对应二叉树的中序周游
  - 周游结果：dbeahficg



（a）二叉树　　（b）森林

先根深度优先周游树(森林)

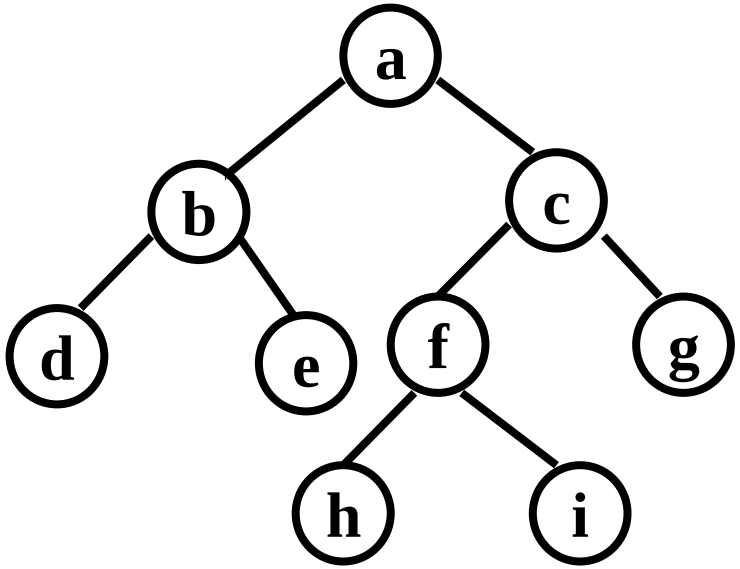
```
template <class T>
void Tree<T>::RootFirstTraverse(TreeNode<T>* root){
    while(root!=NULL) {
        Visit(root->Value()); //访问当前结点
        //周游头一棵树树根的子树
        RootFirstTraverse(root->LeftMostChild());
        root=root->RightSibling();//周游其他的树
    }
}
```

后根深度优先周游树(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
    }
}
```

广度优先周游树(森林)

- 先访问层数为0的结点；然后从左到右逐个访问层数为1的结点；……；依此类推，直到访问完树中的全部结点



- 实现
  - 可以利用一个队列实现其算法
  - 首先把被周游的树根送入队列
  - 其后，每当从队首取出一棵树，访问其根结点之后，马上把它的子树按从左到右的次序送入队列尾端
  - 重复此过程直到队列为空

广度优先周游树(森林)算法

```
template<class T>
void Tree<T>::WidthTraverse(TreeNode<T> * root)  {
    using std::queue;                                    // 使用STL队列
    queue<TreeNode<T>*> aQueue;
    TreeNode<T> * pointer = root;
    while (pointer != NULL) {
        aQueue.push(pointer);                    // 当前结点进入队列
        pointer = pointer->RightSibling(); //指向结点的右兄弟
    }
```

```
    while (!aQueue.empty()) {
        pointer = aQueue.front();          // 获得队首元素
        aQueue.pop();                      // 当前结点出队列
        Visit(pointer->Value());           // 访问当前结点
        pointer = pointer-> LeftMostChild();
        // pointer指向当前结点的最左孩子
        while (pointer != NULL) {   //当前结点的子结点进队列
            aQueue.push(pointer);
            pointer = pointer->RightSibling();
        }
    }
}
```

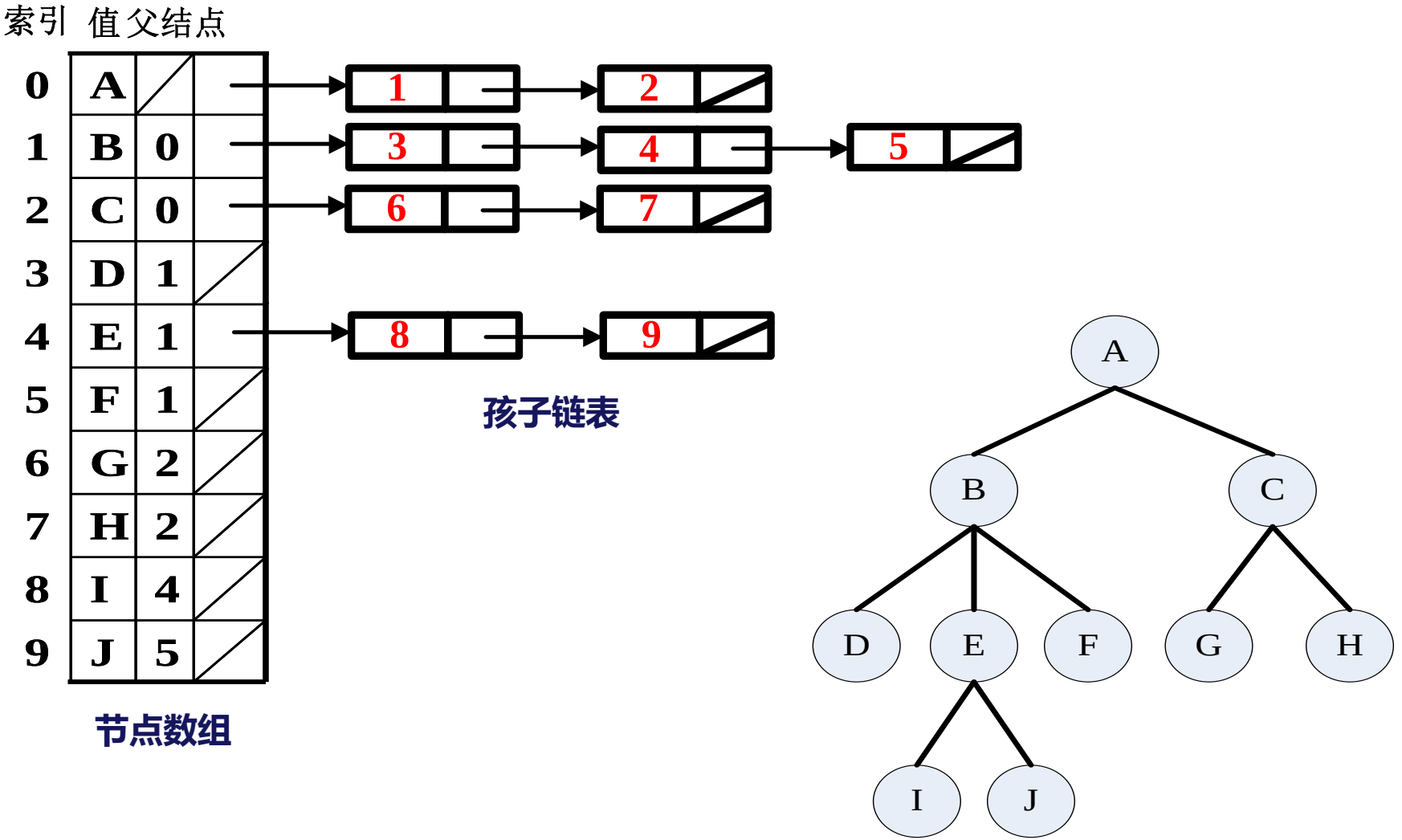
课程内容

- 6.1 基本概念
- 6.2 链式存储
- 6.3 顺序存储
- 6.4 K叉树

6.2 树的链式存储

- 子结点表表示法
- 动态结点表示法
- 静态“左子结点/右兄弟结点”表示法
- 动态“左子结点/右兄弟结点”表示法
- 父指针表示法及在并查集中的应用

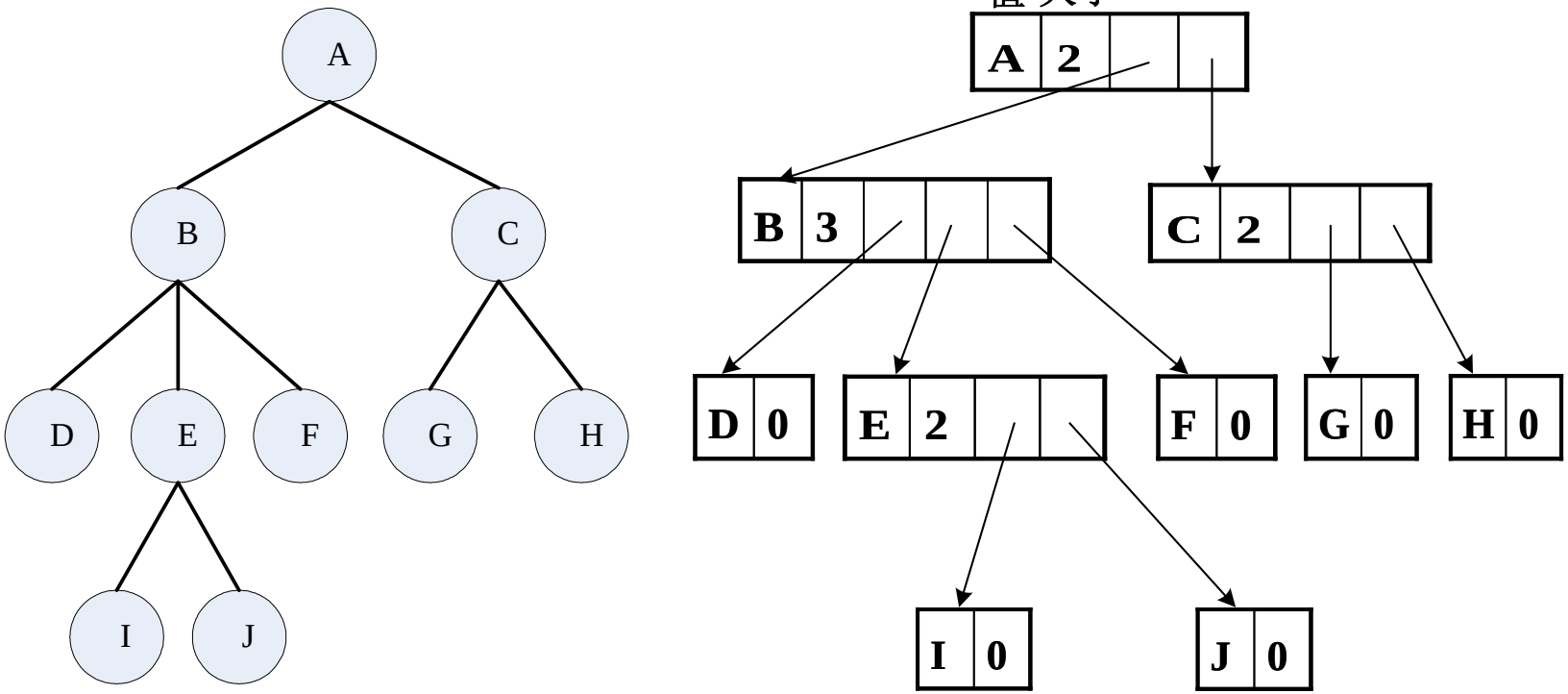
1、子节点表示法


索引 值 父结点
0 A
1 B 0
2 C 0
3 D 1
4 E 1
5 F 1
6 G 2
7 H 2
8 I 4
9 J 5
1
2
3
4
5
6
7
8
9
孩子链表
节点数组
A
B
C
D
E
F
G
H
I
J

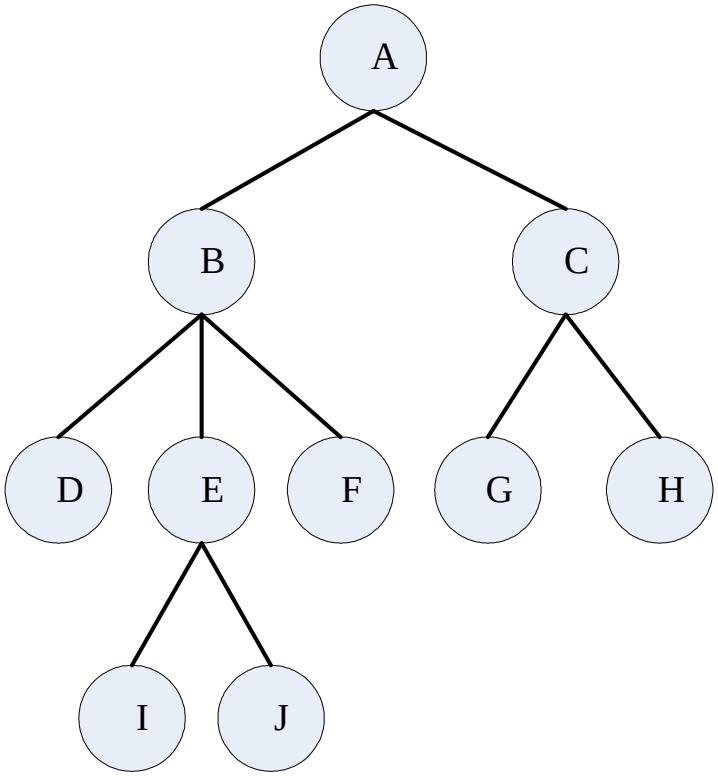
分析

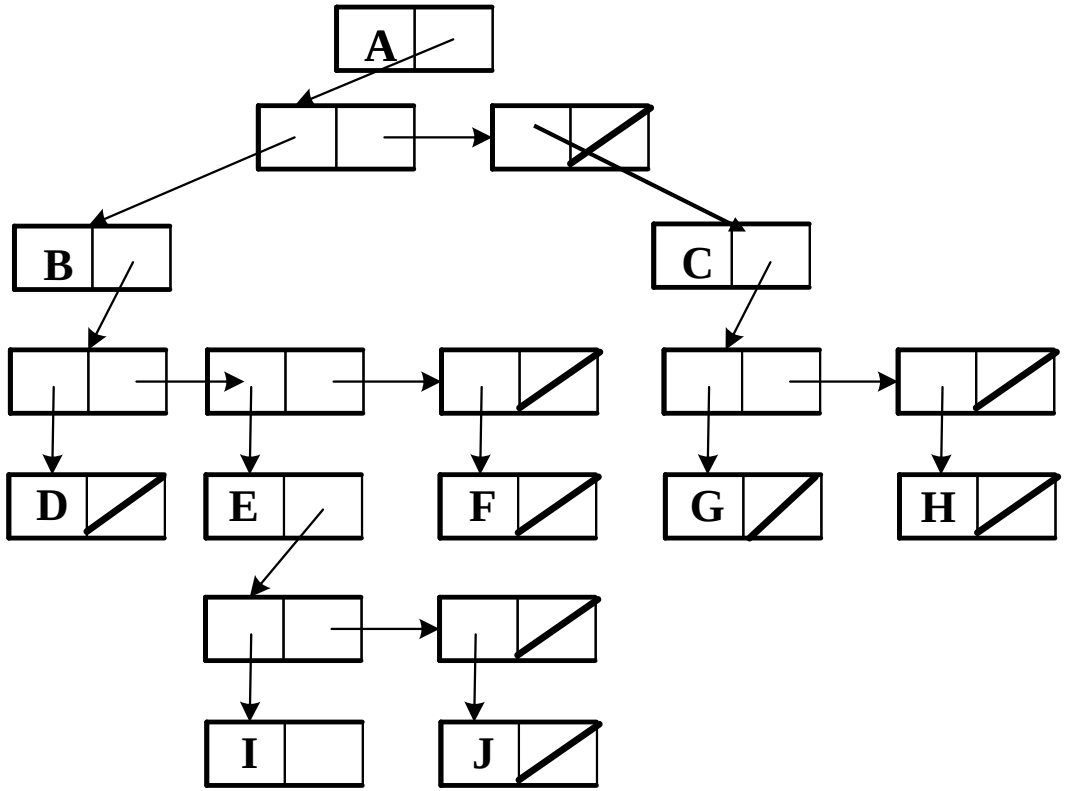
- 优点
  - 查孩子个数和结点的值容易
- 缺点
  - 找兄弟结点困难
- 树的归并容易
  - 只需一棵树的根添到另一棵树的孩子结点表中即可

2、动态结点表示法：指针数组法

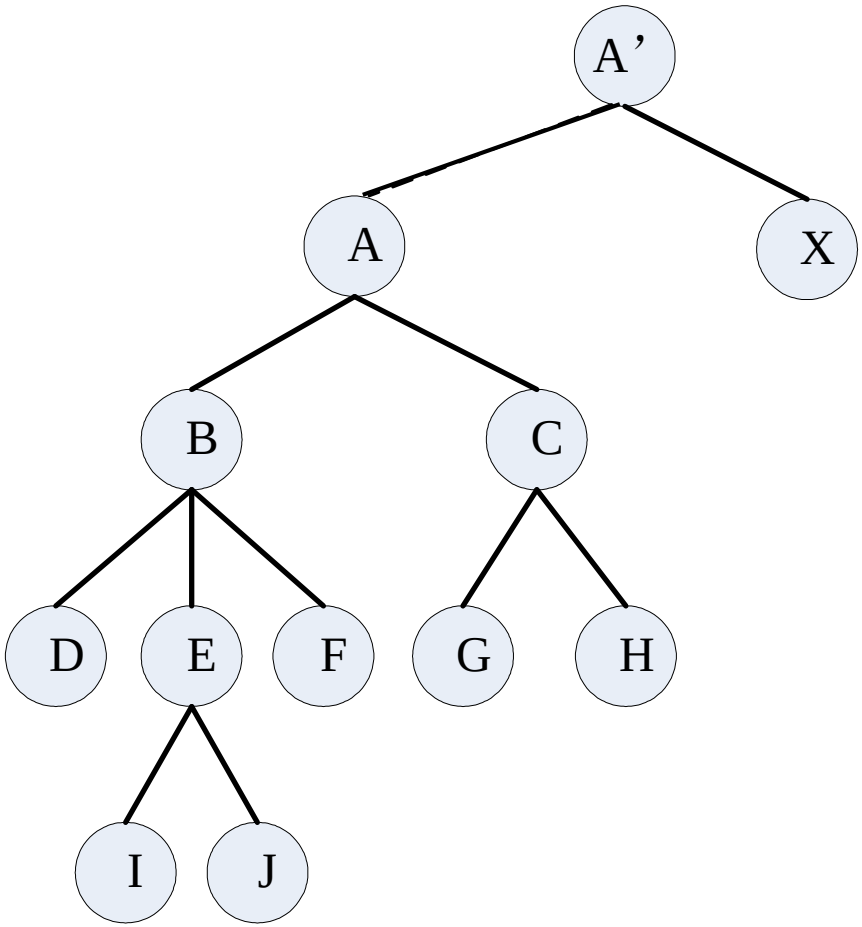


指针链表法





3、静态“左孩子/右兄弟”表示法



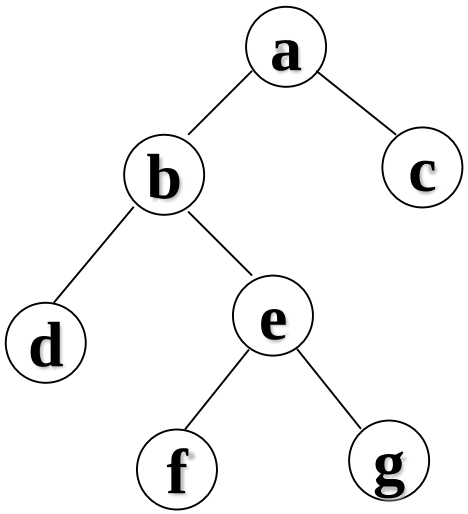
	左子结点	值	父结点	左兄弟结点
1	1	A	10	11
2	3	B	0	2
3	6	C	0	
4		D	1	4
5	8	E	1	5
6		F	1	
7		G	2	7
8		H	2	
9		I	4	9
10		J	4	
11	0	A’		
12		X	10	

分析

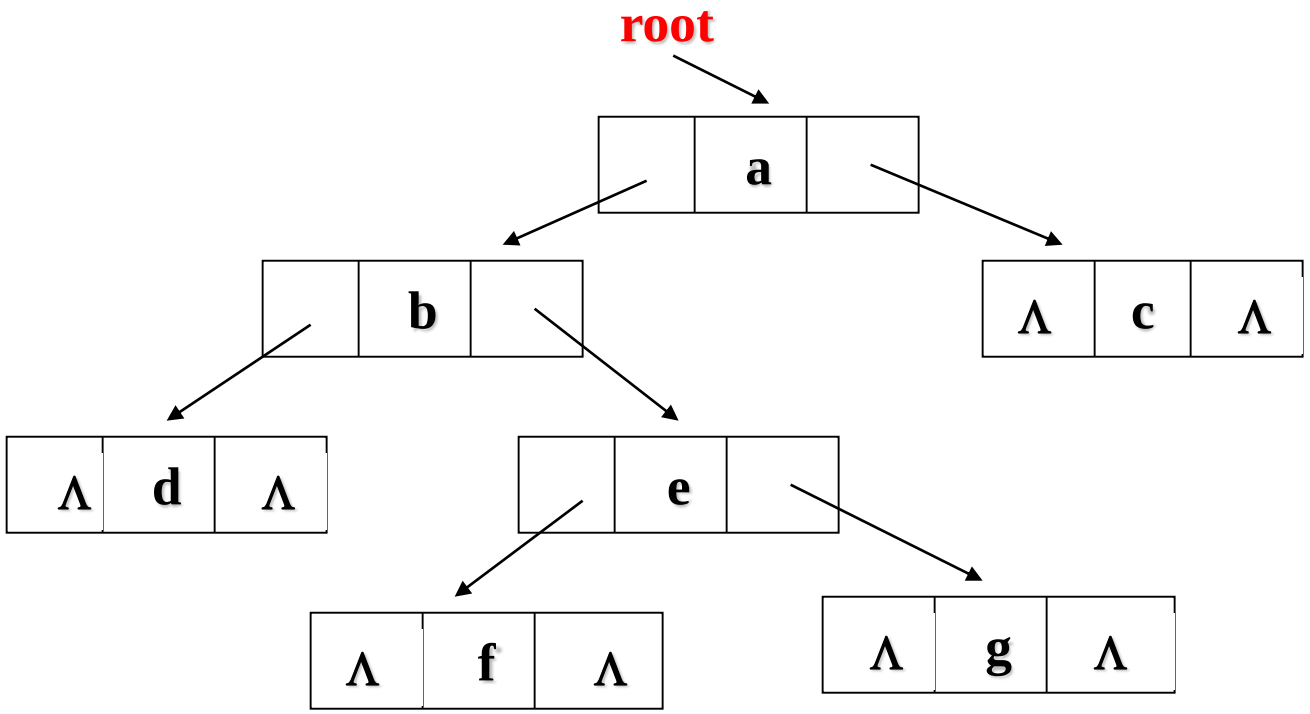
- 树的合并
  - 如果两棵树存储在同一个数组中，那么把其中一个添加为另一棵树的子树只需简单设置指针值即可
- 优点
  - 比子结点表表示法空间效率更高
  - 结点数组中的每个结点仅需要固定大小的存储空间

4、动态“左孩子/右兄弟”二叉链表表示法

- 本质上，使用二叉树来替换树



(a)二叉树



(b) 二叉树的链式存储

基本思想

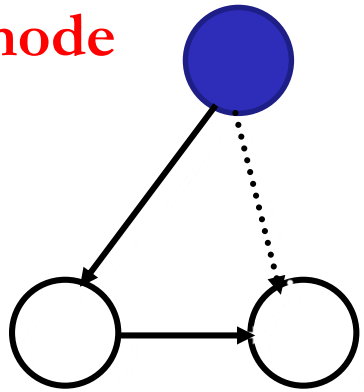
- 左子结点在树中是结点的最左子结点，右子结点是结点原来的右侧兄弟结点
- 树的每个结点均包含固定数目的指针，ADT的每个函数均能有效实现

树结点抽象数据类型的实现

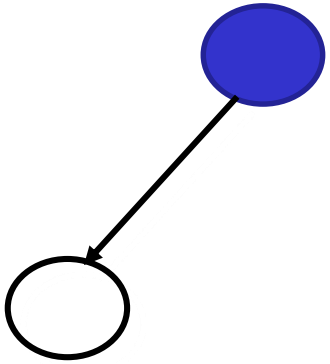
```
private: //补充与具体实现相关的私有成员变量申明
    T m_Value;                          //树结点的值
    TreeNode<T>*    pChild;             //左孩子
    TreeNode<T>*    pSibling;           //右兄弟


//公有成员函数的具体实现
template<class T> bool TreeNode<T>::isLeaf()
{//如果结点是叶，返回true
    if(pChild==NULL)
        return true;
    return false;
}
```

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



node



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
    TreeNode<T>* Parent(TreeNode<T>*current);
     //删除以root为根的子树的所有结点
    void Tree<T>::DestroyNodes(TreeNode<T> *root)
     //删除以subroot为根的子树的所有结点
    void Tree<T>::DeleteSubTree(TreeNode<T> *subroot)
```

```
template  <class T>   //利用广度优先遍历的方法来进行判定！
TreeNode<T>* Tree<T>::Parent(TreeNode<T> *current)  {
    using std::queue;  // 使用STL队列
    queue<TreeNode<T>*> aQueue;
    TreeNode<T> *pointer = root;
    TreeNode<T> *upperlevelpointer = NULL;      // 用于记录parent结点
    if (current != NULL && pointer != current) {
        while (pointer) {                              // 森林中所有根结点进队列
            if (current == pointer)
                return NULL;                  // 根的父结点指针为空，返回
            aQueue.push(pointer);
            pointer=pointer-> RightSibling();
        }
```

```
        while (!aQueue.empty()) {
                pointer = aQueue.pop();          // 出队列
                upperlevelpointer = pointer;     // 指向上一层的结点
                pointer = pointer-> LeftMostChild(); // 指向最左孩子
                while (pointer) {                // 当前结点的子结点进队列
                        if (current == pointer)
                                return upperlevelpointer; // 返回父结点指针
                        else {
                                aQueue.push(pointer);
                                pointer = pointer->RightSibling();
                        }
                }//end while
        }//end while
    }//end if
    return NULL;
}
```

```
template <class T>
void Tree<T>::DestroyNodes(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
    }
}//后续遍历方式进行树的删除！
```

```
template <class T>
void Tree<T>::DeleteSubTree(TreeNode<T>* subroot){
//将以subroot为根的子树从树中分离出来，调用DestroyNodes(subroot)
    TreeNode<T>* pointer=PrevSibling(subroot); //如何实现？
    if(pointer==NULL) {
        pointer=Parent(subroot);
        if(pointer) {       //父结点pointer存在的情况下
            pointer->pChild=subroot->RightSibling();
            subroot->pSibling=NULL;
        }
        else {              //父结点pointer不存在的情况下
            root=subroot->RightSibling();
            subroot->pSibling=NULL;
        }
    }//end if
```

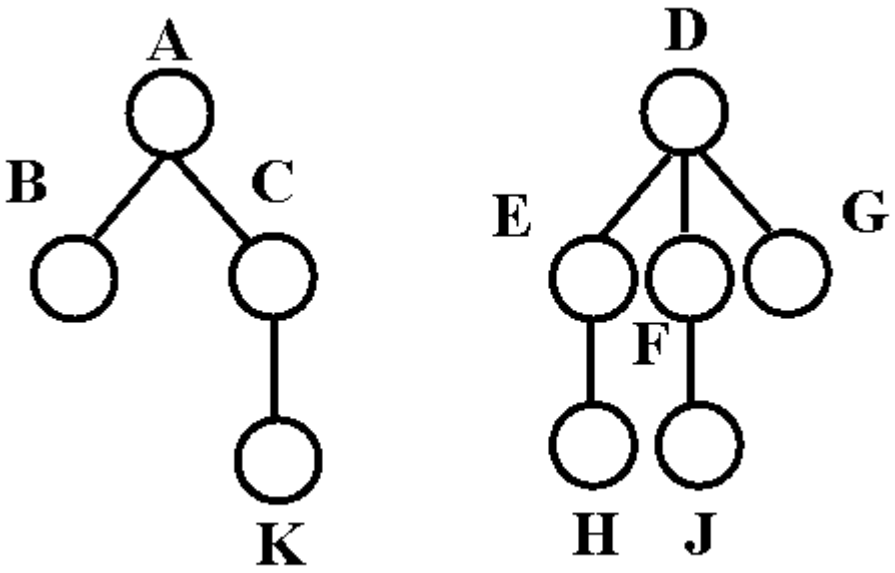
subroot为最左子结点的情况

```
        else{//subroot不是最左子结点
                //链接到删除结点后的结点
               pointer->pSibling=subroot->rightSibling();
               subroot->pSibling=NULL;
        }
      DestroyNodes(subroot);   //销毁以subroot为根的子树
}
```

6.2.5 父指针表示法及在并查集中的应用

- 在某些应用中，只需要知道父结点情况，因此每个结点只需要保存一个指向其父结点的指针域，这种实现被称为父指针(parent pointer)表示法。
- 用数组存储树所有结点，同时在每个结点中附设一个“指针”指示其父结点的位置。
- 由于树中每一个结点的父指针是唯一的，所以父指针表示法可以唯一表示一棵树。

父指针数组表示法



结点索引	0	1	2	3	4	5	6	7	8	9
标记	A	B	C	K	D	E	F	G	H	J
父节点索引	/	0	0	2	/	4	4	4	5	6

分析

- 优点
  - 寻找父结点只需O(1)时间
  - 求树根结点非常方便
- 缺点
  - 寻兄弟节点麻烦，需要查询整个树结构
  - 没有标识节点的左右次序，适合无序树的情况

并查集

- 并查集是一种特殊集合，由不相交子集构成
- 基本操作
  - Find：判断两个结点是否在同一个集合中
  - Union：归并两个集合
- 并查集可用于求解等价类问题

等价类的求解问题

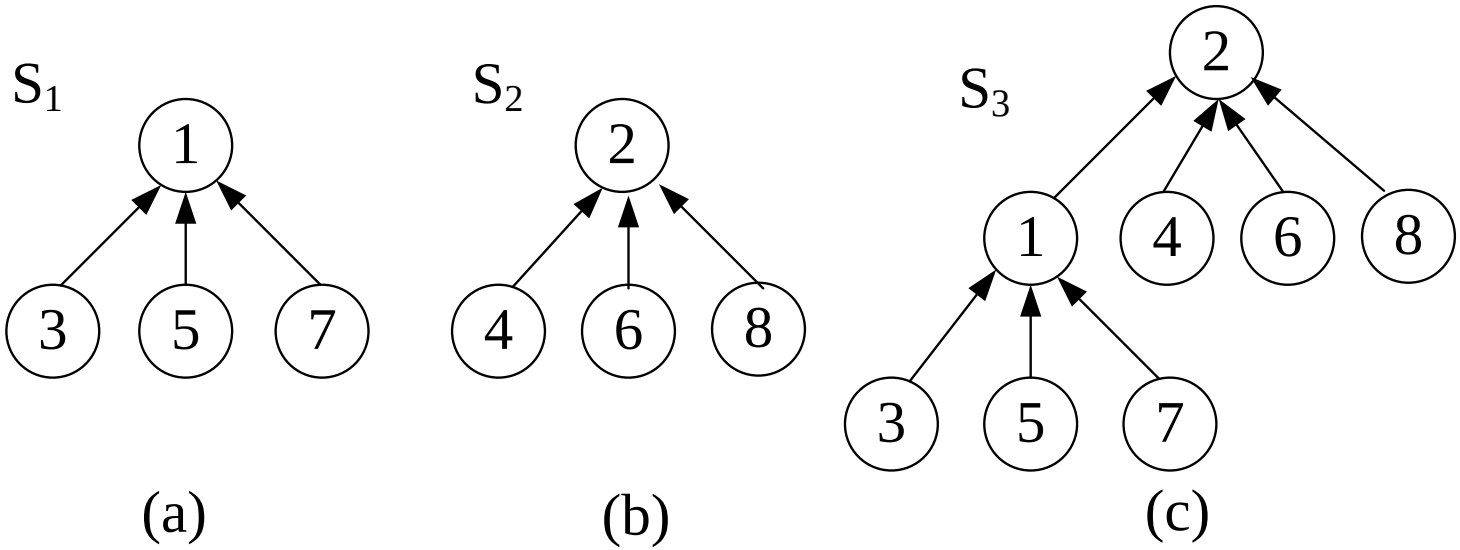
- 等价关系具有自反性、对称性和传递性
- 并查算法可以很容易地解决等价类问题
  - 开始时，每个元素都在独立的只包含一个结点的树中，而它自己就是根结点
  - 通过使用函数Different，一个等价对中的两个元素是否在同一棵树中
    - 如果是，由于它们已经在同一个等价类中，不需要作变动
    - 否则两个等价类可以用UNION函数归并

并查算法

- 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
- 用一棵树代表一个集合
- 两种基本操作
  - 判断两个结点是否在同一个集合中，使用FIND算法
  - 归并两个集合，使用UNION算法
- 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

等价类



集合的表示方法

树结点的ADT

```
template<class T>
    class ParTreeNode    {                          //树结点定义
private:
    T      value;                                   //结点的值
    ParTreeNode<T>* parent;                         //父结点指针
    int              nCount;          //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
    T      getValue();                              //返回结点的值
    void  setValue(const T& val);                   //设置结点的值
    ParTreeNode<T>*  getParent();                   //返回父结点指针
    void  setParent(ParTreeNode<T>* par);           //设置父结点指针
    int     getCount();                             //返回结点数目
    void  setCount(const int count);                //设置结点数目
}
```

树的ADT

```
template<class T>
    class ParTree {//树定义
public:
    ParTreeNode<T>* array;          //存储树结点的数组
    int    Size;                    //数组大小
    ParTree(const int size);        //构造函数
    virtual ~ParTree();             //析构函数
    //查找node结点所属子树的根结点
    ParTreeNode<T> * Find(ParTreeNode<T>* node) const;
    void Union(int i,int j);        //把下标为i，j的结点所属子树合并
    bool Different(int i,int j);    //判定下标为i，j的结点是否在一棵树中
};
```

成员函数

```
template <class T>
ParTree<T>::ParTree(const int size)  //构造函数
{
    Size=size;
    array=new ParTreeNode<T>[size];
}

template <class T>    //析构函数
ParTree<T>::~ParTree()
{
    delete []array;
}
```

```
template <class T>                          //查找node结点所属子树的根结点
ParTreeNode<T>*ParTree<T>::Find(ParTreeNode<T>* node) const{
    ParTreeNode<T>* pointer=node;
    while ( pointer->getParent()!=NULL)
        pointer=pointer->getParent();
    return pointer;
}

template<class T>                           //判断两个结点是不是属于同一棵子树
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;
}
```

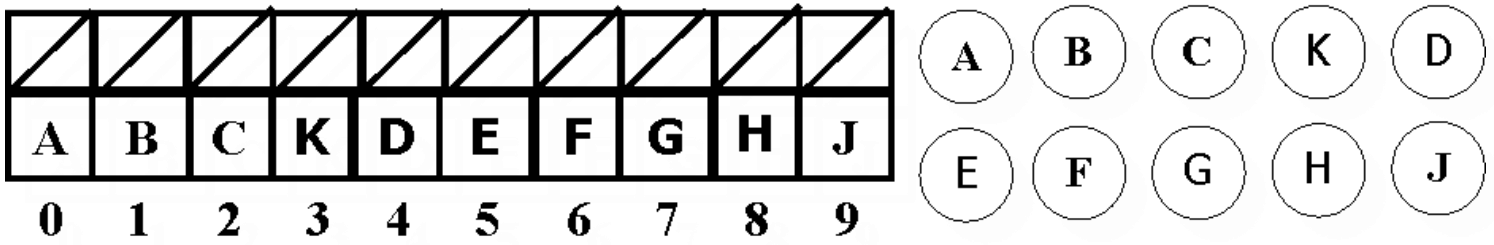
```
template<class T> void ParTree<T>::Union(int i,int j){
    //将结点i和结点j所属的树合并成为一棵树
    ParTreeNode<T>* pointeri=Find(&array[i]);   //找到结点i的根
    ParTreeNode<T>* pointerj=Find(&array[j]);   //找到结点j的根
    if(pointeri!=pointerj){
        if(pointeri->getCount()>=pointerj->getCount()){
            pointerj->setParent(pointeri);
            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

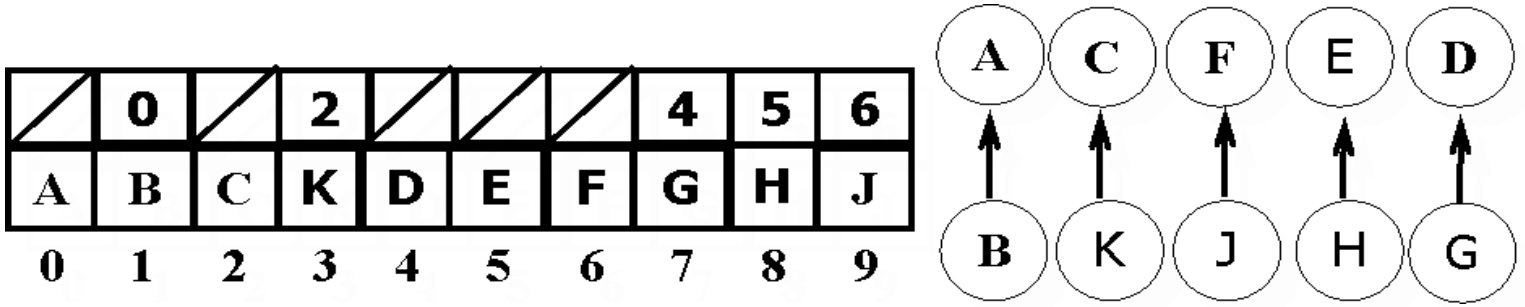
- “重量权衡合并规则”（weighted union rule）
  - 将结点较少树的根结点指向结点较多树的根结点
  - 这可以把树的整体深度限制在O(log n)
  - 当处理完n个等价对后，任何结点的深度最多只会增加logn次
    - 每次归并，最大高度最多增加1
    - 而结点个数成倍增加

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)、 F,J)、(E,H)、(D,G)、(K,A)、(E,G)、(H,J)

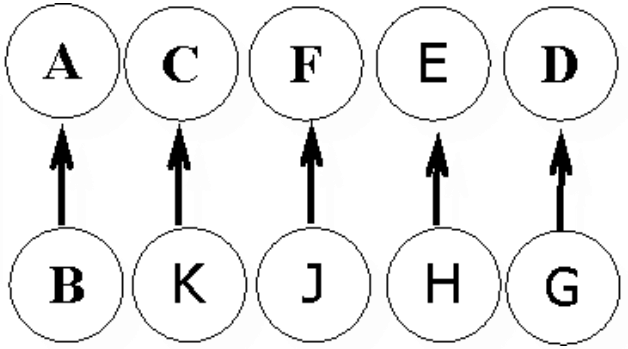


- 对等价对（A,B）、(C,K)、(F,J)、(E,H)、(D,G)的处理结果

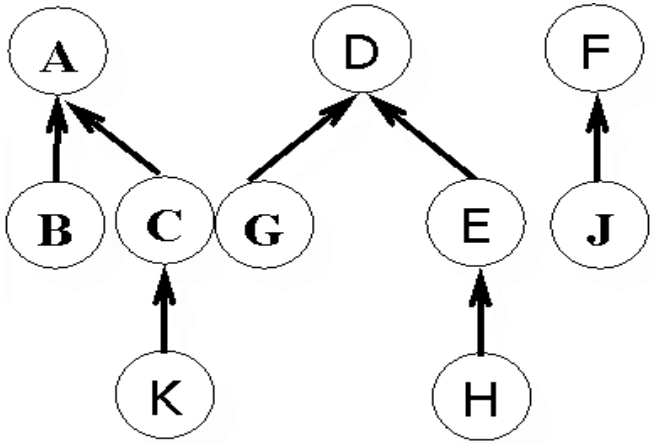


对两个等价对（K，A）和（E，G）的处理结果

	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

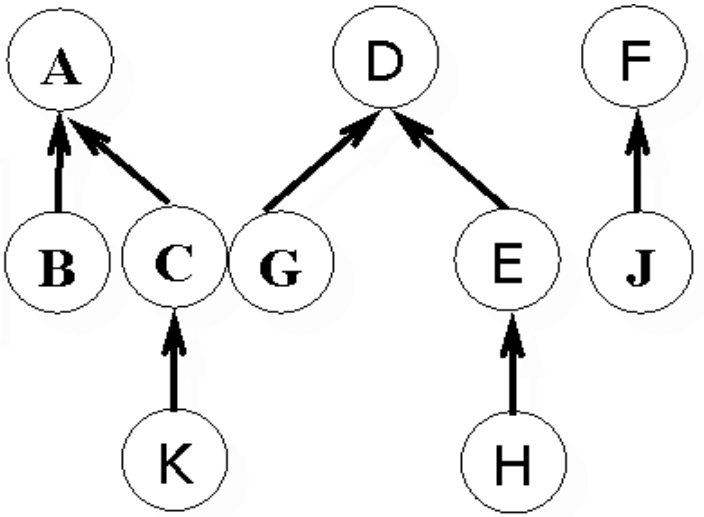


	0	0	2		4		4	5	6
A	B	C	K	D	E	F	G	H	J
0	1	2	3	4	5	6	7	8	9

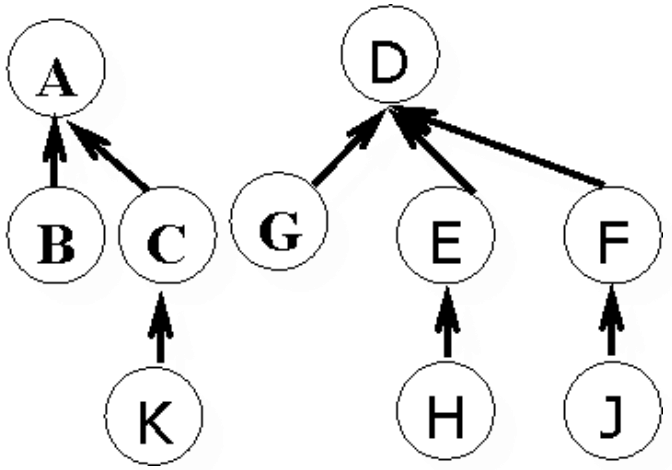


使用标准重量权衡合并规则处理等价对（H,J）的结果

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



	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



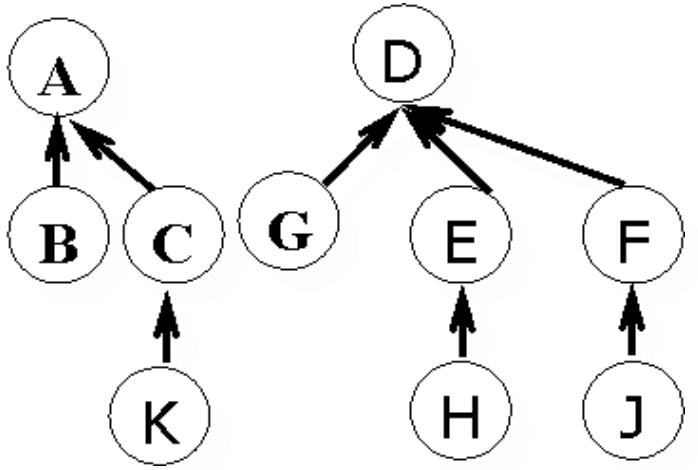
路径压缩算法

```
template <class T>   //带路径压缩的find算法
ParTreeNode<T> *ParTree<T>::FindPC(ParTreeNode<T>*
    node) const  {
    if(node->getParent()==NULL)
          return node;
    node->setParent(FindPC(node->getParent()));
    return node->getParent();
}
```

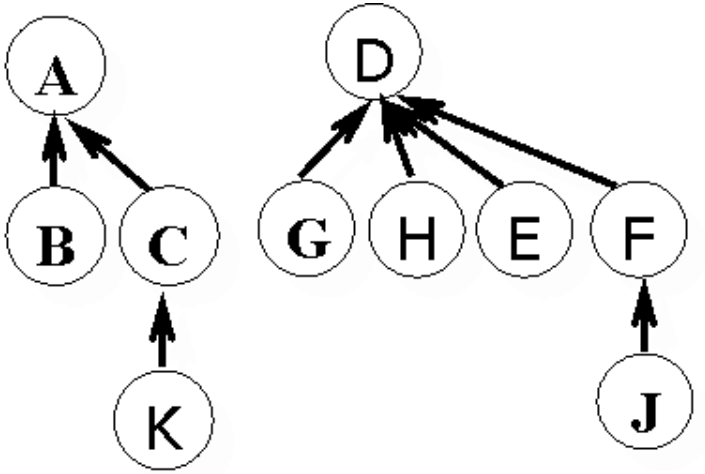
路径压缩示例

- 使用路径压缩规则处理等价对（H,E）的结果

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



/	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



课程内容

- 6.1 基本概念
- 6.2 链式存储
- **6.3 顺序存储**
- **6.4 K叉树**

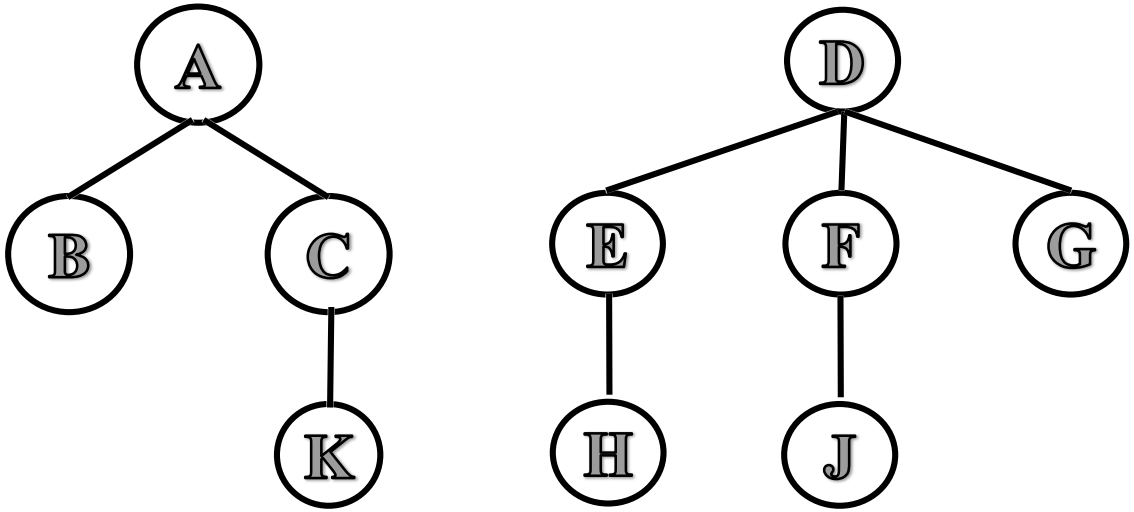
顺序存储方法

- 按照树的某种遍历结果序列进行存储
  - 带右链的先根次序表示法
  - 带双标记位的先根次序表示法
  - 带度数的后根次序表示法
  - 带双标记的层次次序表示
- **如何还原出树的结构是关键！**

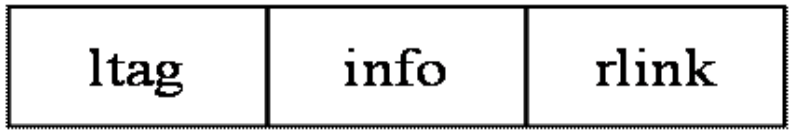
6.3.1 带右链的先根次序表示法

- 先根遍历树，结果有如下特点
  - 任何结点的子树的所有结点都直接跟在该结点之后
  - 每棵子树的所有结点都聚集在一起，中间不会插入别的结点
  - 任何一个分支结点后面跟的都是它的第一个子结点（如果存在的话）
- 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
- 但还需如下信息支持
  - 孩子关系如何确定？
  - 兄弟关系如何确定？

A B C K D E H F J G

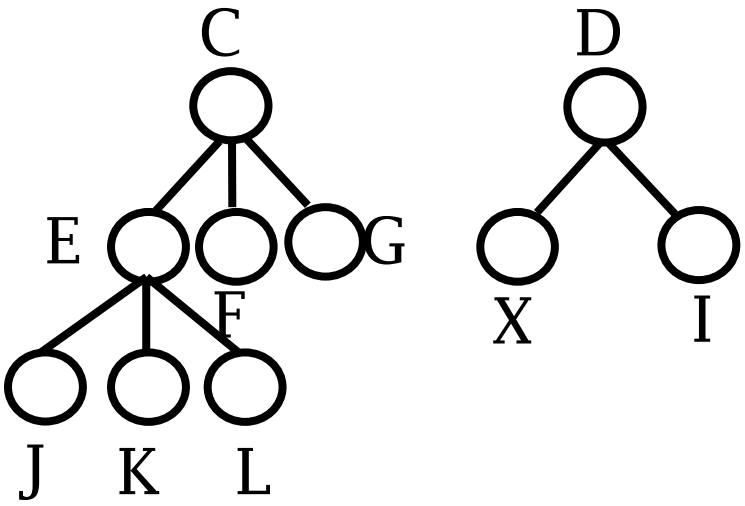


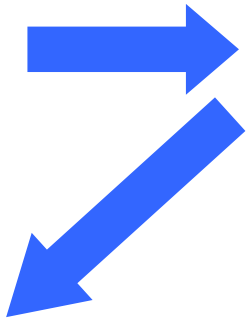
- 结点除包含本身数据外，还附加两个表示结构的信息字段
  - **rlink**：是右指针，指向下一个兄弟
  - **ltag**是一个1位的左标记
    - **0**：有子结点
    - **1**：没有子结点

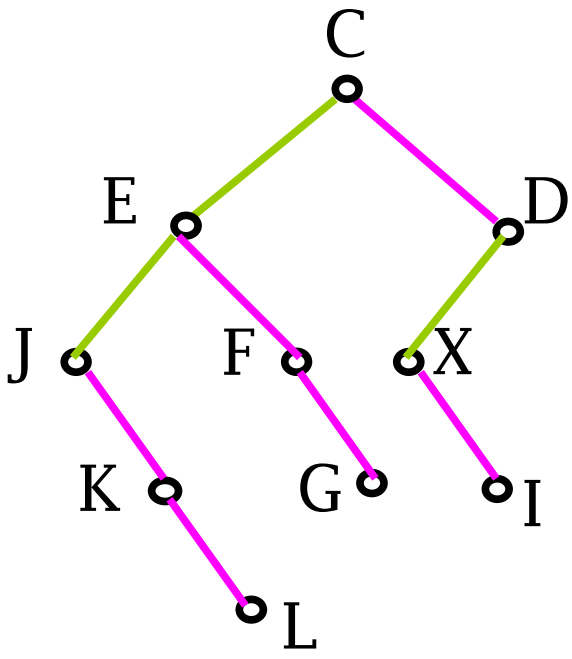


- 与二叉链表相比，用**ltag**代替**llink**，占用存储单元少，但并不丢失信息
- 可以从结点的次序和**ltag**的值完全可以推知**llink**
  - **ltag**为**0**：结点有左子结点，其**llink**指向存储区中该结点顺序的下一个结点
  - **ltag**为**1**：结点没有左子结点，它的**llink**为空

图示



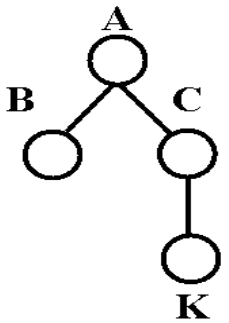


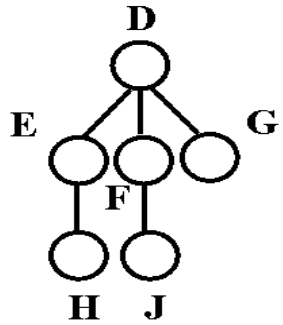


下标	0	1	2	3	4	5	6	7	8	9
rlink	7	5	3	4	-1	6	-1	-1	9	-1
info	C	E	J	K	L	F	G	D	X	I
ltag	0	0	1	1	1	1	1	0	1	1

6.3.2 带双标记位的先根次序表示法

- 事实上，带右链的先根次序表示法中rlink也不是必需的
- 以1位的rtag就足以表示出整个森林的结构信息
  - 当结点无兄弟时，rtag为1，否则为0
- 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以结点x为根的子树中最后结点的后面的那个结点y

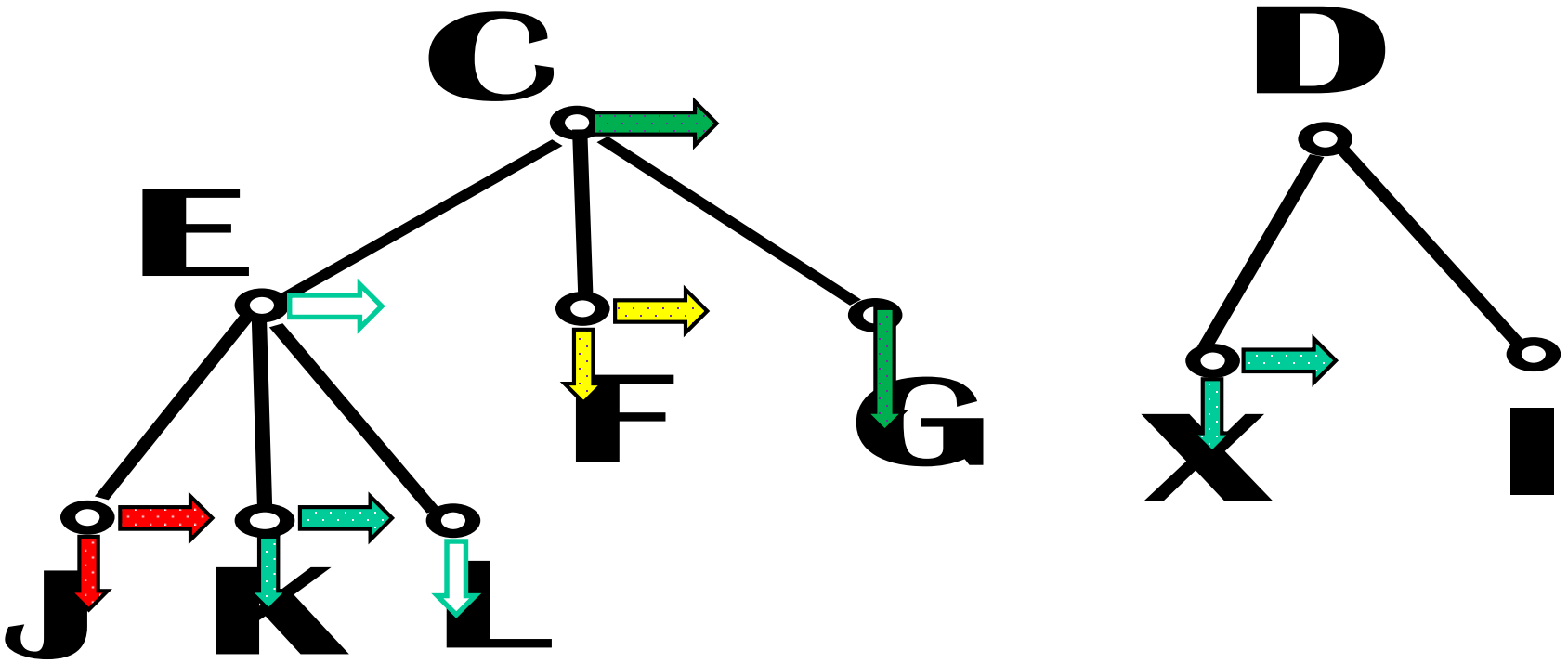




- 确定结点y成为解决问题的关键！

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

有兄弟节点与无孩子节点一一对应，满足栈特性



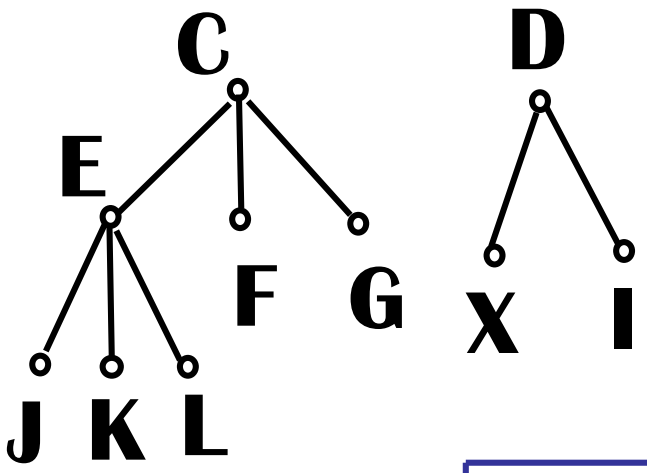
rtag	0	0	0	0	1	0	1	1	0	1
info	C	E	J	K	L	F	G	D	X	I
ltag	0	0	1	1	1	1	1	0	1	1

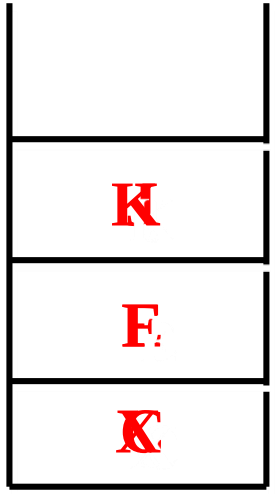
结点x的兄弟结点y的确定方法

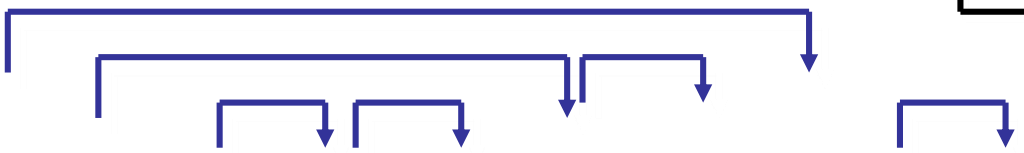
- 由排列次序和ltag，rtag的值推知rlink的值
  - 先根次序中子树结点嵌套出现，在顺序搜索中要嵌套处理x的所有子树，因此确定y要用到栈结构
  - **有兄弟的结点(rtag=0)，都唯一对应一个无孩子的结点(ltag=1)，成对出现，满足栈特性，即**
    - 扫描到一个rtag为0的结点就将它进栈
    - 扫描到一个ltag为1的结点时，就从栈顶弹出一个结点，并设置其rlink指向下一个要读出的节点，即其兄弟节点

“有兄弟就入栈，无孩子就出栈”

- 处理过程中需要用一个栈来记录待配置rlink的结点







rtag	0	0	0	0	1	0	1	1	0	1
info	C	E	J	K	L	F	G	D	X	I
ltag	0	0	1	1	1	1	1	0	1	1

树结点类定义

```
template<class T>
class DualTagTreeNode
{//双标记位先根次序树结点类
    public:
        T       info;                   //树结点信息
        int     ltag;                   //左标记
        int     rtag;                   //右标记
        DualTagTreeNode();              //构造函数
        virtual ~DualTagTreeNode();     //析构函数
};
template<class T> DualTagTreeNode<T>::DualTagTreeNode(){
        ltag=1;  rtag=1;
}
template<class T>DualTagTreeNode<T>::~DualTagTreeNode(){}
```

构造左子结点右兄弟树算法

```
template <class T>
Tree<T>::Tree(DualTagTreeNode<T>* nodeArray, int  count){
    using std::stack;                                   // 使用STL中的stack
    stack<TreeNode<T>* >        aStack;
    TreeNode<T>* pointer=new TreeNode<T>;               //建立根结点
    root=pointer;
    for(int i=0;i<count-1;i++) {                        //处理一个结点
        pointer->setValue(nodeArray[i].info);
        if(nodeArray[i].rtag = =0)   //有兄弟，则压栈～
            aStack.push(pointer);
        else                                            //无兄弟，兄弟域设为空～
            pointer->setSibling(NULL);
```

处理右兄弟标志位

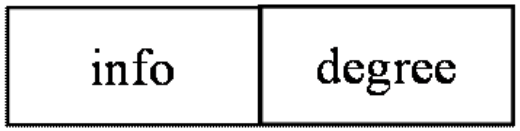
```
        TreeNode<T>* temppointer=new TreeNode<T>;
        if(nodeArray[i].ltag==0)                    //有孩子，则设为孩子～
            pointer->setChild(temppointer);
        else{                                       //无孩子则出栈~
            pointer->setChild(NULL);                //左子结点设为空
            pointer=aStack.pop();
            pointer->setSibling(temppointer);
        }
        pointer=temppointer;
    } //end for
        pointer->setValue(nodeArray[count-1].info);
        pointer->setChild(NULL);
        pointer->setSibling(NULL);
}
```

处理左孩子标志位

处理最后一个结点

6.3.3 带度数的后根次序表示法

- 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



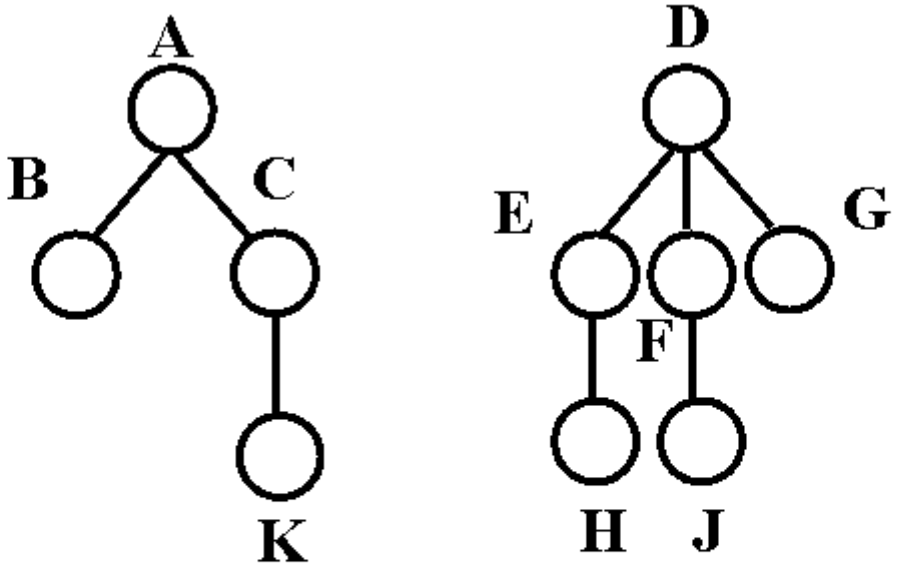
  - 其中，**info**是结点的数据，**degree**是结点的度数

- **这种表示法不包括指针，但它仍能反映树结构**

- 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
  - 最右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
  - ….

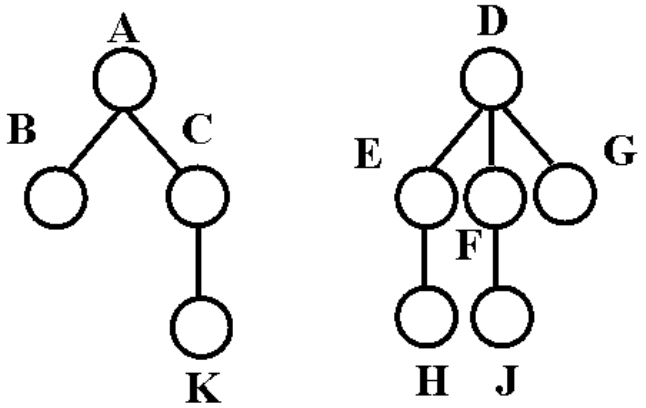
树(森林)的图示



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

后根次序序列到森林的转换

- 以结点*x*为根的子树在后根次序序列中的前驱求解：
  - 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←s+1（结点数）-degree(z)（边数）；
  - 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(**原因：子树的结点数等边数加1**)



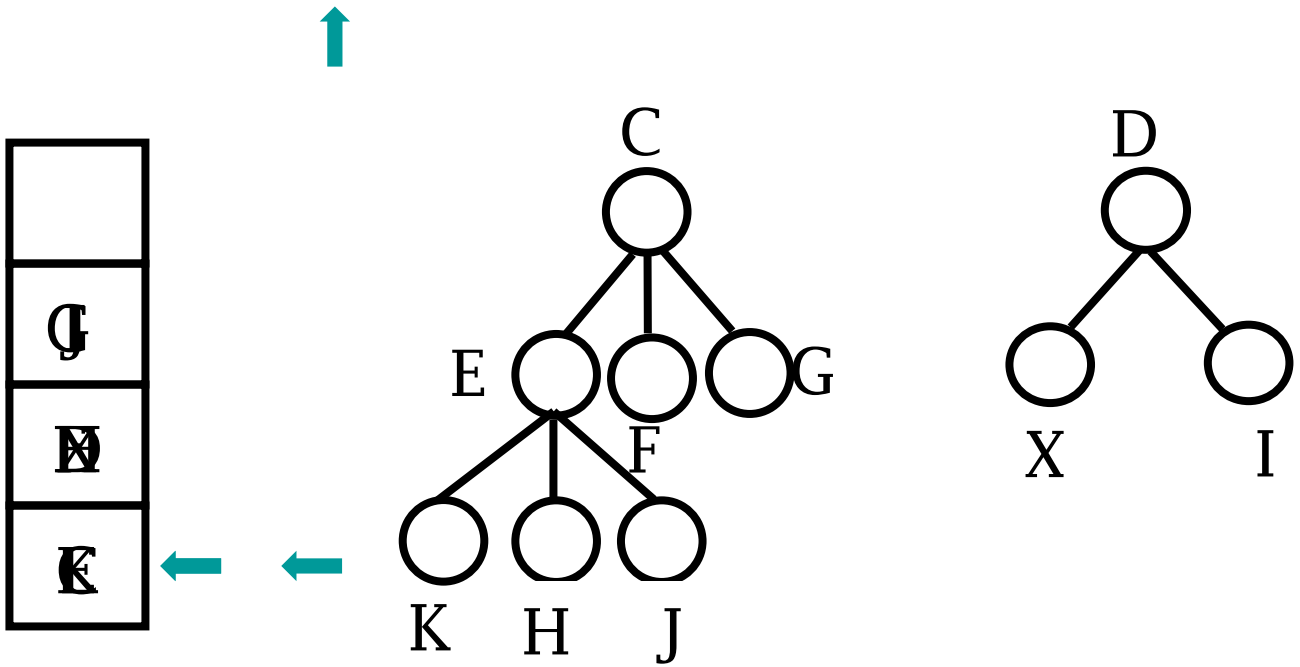
degree	0	0	1	2	0	1	0	1	0	3
info	B	K	C	A	H	E	J	F	G	D

换一种思路

- 将带度数的后根次序表示转化成森林时
  - 从左至右进行扫描，度为零的结点是叶子结点（也可看作一棵子树）
  - 当遇到度数非零（设为k）的结点时，则排在该结点之前且离它最近的k个子树的根就是该结点的k个子结点
- 利用栈实现
  - 遇到零度顶点就入栈；
  - 遇到非零k度顶点就从栈中弹出k个节点作为其子节点，然后将该非零顶点入栈
  - 持续扫描， 直至序列扫描完毕

示例

degree	0	0	0	3	0	0	3	0	0	2
info	K	H	J	E	F	G	C	X	I	D

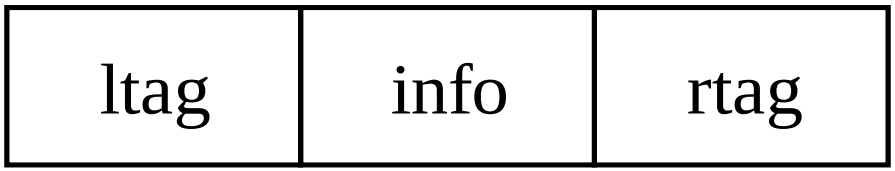


讨论

- 带度数的先根次序？
- 带度数的层次次序？

6.3.4 带双标记的层次次序表示

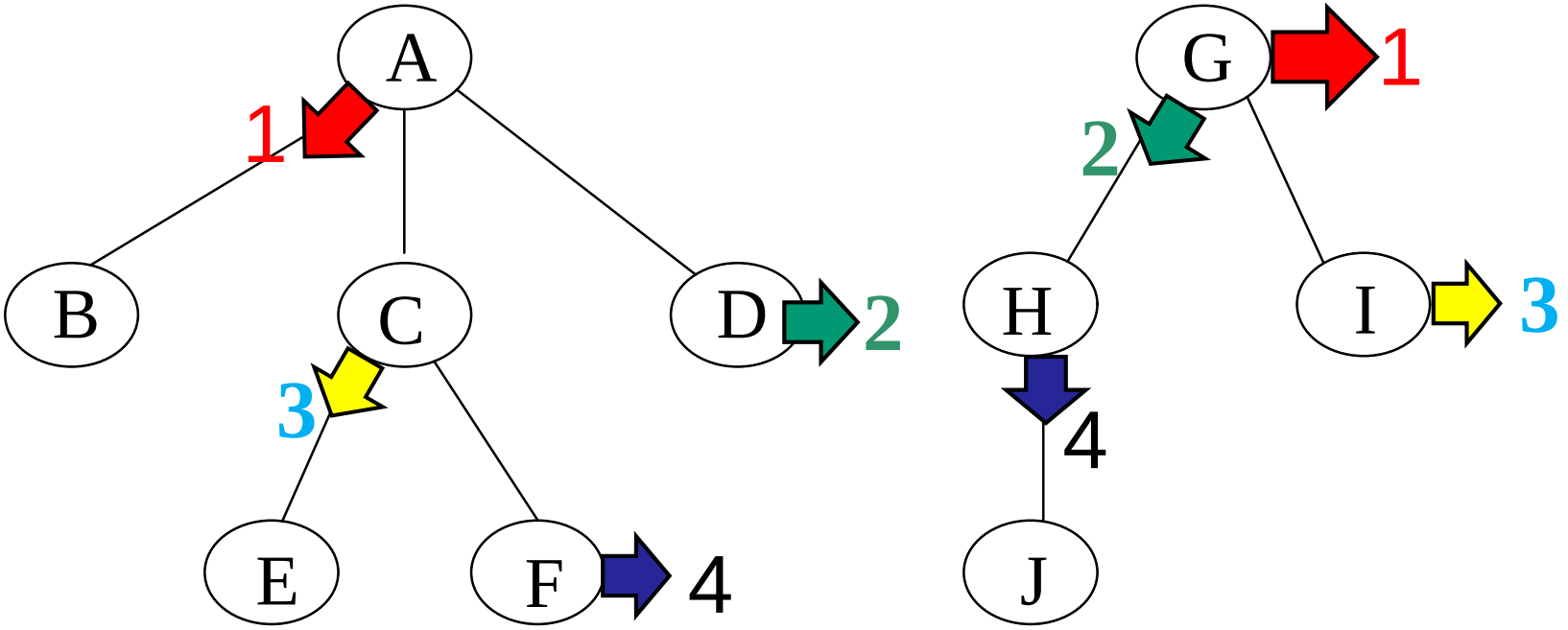
- 类似于带双标记的先根次序表示，引入带双标记的层次次序表示法，其结点的形式为：



  - 当结点没有下一个兄弟时标记位**rtag**为**1**，否则为**0**
    - **rtag**为**0**时，下一个节点即为其兄弟节点
    - **rtag**为**1**时，无兄弟节点
  - 当结点没有左孩子时标记位**ltag**为**1**，否则为**0**

关键：处理ltag为0的情况

有孩子节点与无兄弟节点一一对应，满足队列特性



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

有孩子则入队列，无兄弟出队列！

- 顺序扫描带双标记的层次次序序列
  - 如果结点的ltag值为1，则置其llink为空；当结点的ltag为0时，该结点入队列；
  - 如果结点的rtag值为0，那么其后的结点y就是其右兄弟；
  - 否则，如果结点的rtag值为1，则rlink为空，此时出队列x，并将x的llink指向序列中后续结点 y即可。

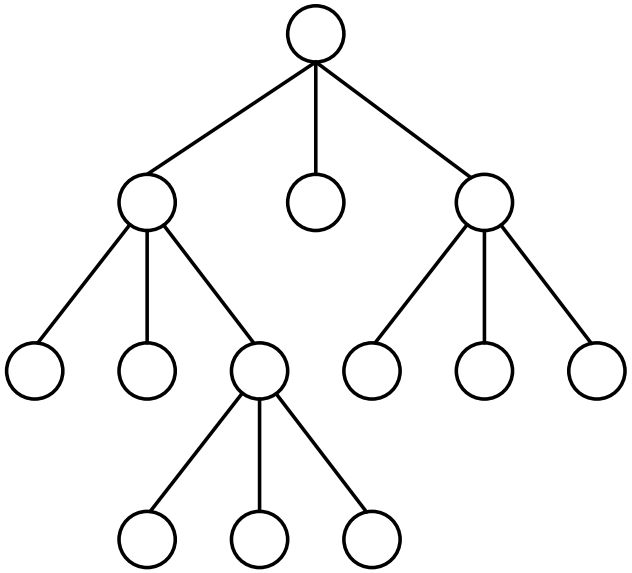
与双标记先根遍历代码几乎一样

请自己予以实现！

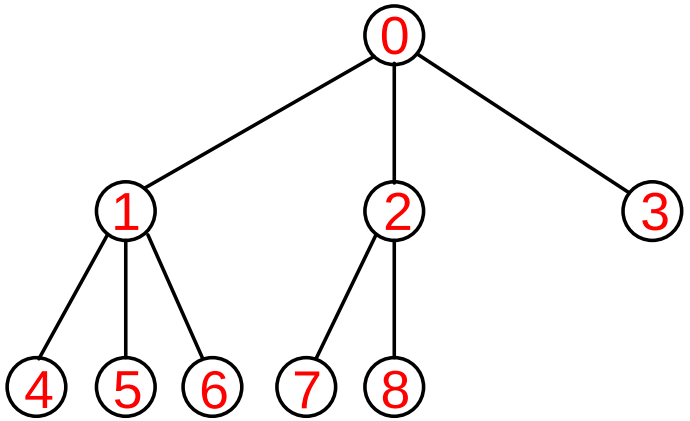
6.4 K叉树

- K叉树(K-ary Tree)的结点有K个有序子结点
- 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
- 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

示例



（a）满3叉树



（b）完全3叉树

知识点总结

- 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

再见…

联系信息：

电子邮件：gjsong@pku.edu.cn

电　　话：62754785

办公地点：理科2号楼2307室